Zbigniew Michalewicz
Martin Schmidt
Matthew Michalewicz
Constantin Chiriac

0100 0001
0100 0010
0100 1001

Adaptive Business Intelligence

Springer

Adaptive Business Intelligence

Zbigniew Michalewicz · Martin Schmidt
Matthew Michalewicz · Constantin Chiriac

Adaptive Business Intelligence

Springer

Authors

Zbigniew Michalewicz

School of Computer Science
University of Adelaide
Adelaide, SA 5005
Australia

zbyszek@cs.adelaide.edu.au
www.cs.adelaide.edu.au/~zbyszek

Martin Schmidt
Matthew Michalewicz
Constantin Chiriac

SolveIT Software Pty Ltd
P.O. Box 3161
Adelaide, SA 5000
Australia

martin.schmidt@SolveITSoftware.com
matthew.michalewicz@SolveITSoftware.com
constantin.chiriac@SolveITSoftware.com
www.SolveITSoftware.com

Library of Congress Control Number: 2006930103

ACM Computing Classification (1998): I.2.8, F.2.2, G.1.6, I.2.1, I.5.1, J.1, H.4

ISBN-10 3-540-32928-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-32928-2 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Cover Design: KünkelLopka, Heidelberg
Typesetting and Production: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig

Printed on acid-free paper 45/3100 YL - 5 4 3 2 1 0

To Adam, Ewa, and Arthur.
Z. M.

To Ana, my family and my friends:
May Love, Knowledge and Good Luck guide your way!
M. S.

To my parents, children, and loving wife Luiza.
M. M.

To my parents and my wife Larisa.
C. C.

Preface

> "My name is Sherlock Holmes. It is my business to know what other people do not know."
> *The Adventure of the Blue Carbuncle*
>
> "What do you think of it, Watson?"
> "A masterpiece. You have never risen to a greater height."
> *The Adventure of the Bruce-Partington Plans*

Since the computer age dawned on mankind, one of the most important areas in information technology has been that of "decision support." Today, this area is more important than ever. Working in dynamic and ever-changing environments, modern-day managers are responsible for an assortment of far-reaching decisions: *Should the company increase or decrease its workforce? Enter new markets? Develop new products? Invest in research and development?* The list goes on. But despite the inherent complexity of these issues and the ever-increasing load of information that business managers must deal with, all these decisions boil down to two fundamental questions:

- What is likely to happen in the future?
- What is the best decision right now?

Whether we realize it or not, these two questions pervade our everyday lives – both on a personal and professional level. When driving to work, for instance, we have to make a traffic prediction before we can choose the quickest driving route. At work, we need to predict the demand for our product before we can decide how much to produce. And before investing in a foreign market, we need to predict future exchange rates and economic variables. It seems that regardless of the decision being made or its complexity, we first need to make a prediction of what is likely to happen in the future, and then make the best decision based on that prediction. This fundamental process underpins the basic premise of *Adaptive Business Intelligence.*

Simply put, Adaptive Business Intelligence is the discipline of combining prediction, optimization, and adaptability into a system capable of answering these two fundamental questions: *What is likely to happen in the future?* and *What is the best decision right now?* To build such a system, we first need to understand the methods and techniques that enable prediction, optimization, and adaptability. At first blush, this subject matter is nothing new, as hundreds of books have already been written on business intelligence, data mining and prediction methods, optimization techniques, and so forth. However, none of these books has explained how to combine these various technologies into a software system that is capable of predicting, optimizing, and adapting. This text is the first on the subject.

When we set out to write *Adaptive Business Intelligence*, we had three important objectives in mind: First of all, we wanted to explain why the future of the business intelligence industry lies in systems that can make decisions, rather than tools that produce detailed reports. As most business managers now realize, there is a world of difference between having good knowledge and detailed reports, and making smart decisions. Michael Kahn, a technology reporter for Reuters in San Francisco, makes a valid point in the January 16, 2006 story entitled "Business intelligence software looks to future":

> *"But analysts say applications that actually answer questions rather than just present mounds of data is the key driver of a market set to grow 10 per cent in 2006 or about twice the rate of the business software industry in general.*
>
> *'Increasingly you are seeing applications being developed that will result in some sort of action,' said Brendan Barnacle, an analyst at Pacific Crest Equities. 'It is a relatively small part now, but it is clearly where the future is. That is the next stage of business intelligence.'"*

We could not agree more.

Second, we wanted to explain the principles behind many prediction methods and optimization techniques in simple terms, so that any business manager could grasp them. Even though most business managers have a limited technology background, they should not be intimidated by terms such as "artificial neural networks," "fuzzy logic," "evolutionary algorithms," "ant systems," or "agent-based modeling." They should understand the strengths and weaknesses of these methods and techniques, their operating principles, and applicability. Armed with such knowledge, business managers will be in a better position to control the application of these methods and techniques in their respective organizations.

And, third, we wanted to underscore the enormous applicability of Adaptive Business Intelligence to many real-world business problems, ranging from demand forecasting and scheduling, to fraud detection and investment strategies. From a high-level perspective, most of these business problems have similar characteristics, and the application of Adaptive Business Intelligence can provide significant benefits and savings.

To facilitate the discussion in this book, we have divided the chapters into three parts that correspond to the three objectives listed above. In Part I, we present the fundamental ideas behind Adaptive Business Intelligence, and explain the different roles that prediction, optimization, and adaptability play in producing near-optimal decisions. We also discuss the characteristics that many business problems have in common, and why these characteristics increase the complexity of the problem-solving exercise. Furthermore, we introduce a particular distribution problem that is used throughout the text as a running example. Given that the prediction and optimization issues in this example are common to most business problems, it should be relatively easy for the reader to extrapolate this example to many other business domains.

Because countless texts have already been written on the subject of database technologies, data warehousing, online analytical processing, reporting, and the like, we saw little point in rehashing the tools and techniques that are routinely used to access, view, and manipulate organizational data. Instead, Part II of the book discusses the various prediction methods and optimization techniques that can be used to develop an Adaptive Business Intelligence system. The distribution example is continued throughout these chapters, effectively highlighting the strengths and weaknesses of each method and technique. Each chapter in Part II is concluded by a *Recommended Reading* section that provides suggestions for readers who want to learn more about particular methods or techniques.

Part III begins with a chapter on hybrid systems and adaptability, explaining how to "combine" the various methods and techniques discussed in Part II, and how the component of adaptability can be added to the final design. In the remaining chapters of the book, we discuss the definitive solution to the distribution problem that was used throughout the text, as well as the application of Adaptive Business Intelligence to several other complex business problems.

Without a doubt, we believe that business managers of all levels would benefit from this text. Anyone who makes operational and strategic decisions – whether on the factory floor or in the boardroom – will find this book invaluable for understanding the science and technology behind better predictions and decisions. We hope that readers will enjoy reading the book as much as we enjoyed writing it, and that they will profit from it.

We would like to thank everyone who made this book possible, and who took the time to share their thoughts and comments on the subject of Adaptive Business Intelligence. In particular, we would like to express our gratitude to SolveIT Software's scientific advisory board, which includes Hussein Abbass, Valerio Aimale, Jürgen Branke, Mike Brooks, Carlos Coello, Ernesto Costa, Kalyanmoy Deb, Gusz Eiben, Fred Glover, Philip Hingston, Jong-Hwan Kim, Bob McKay, Marek Michalewicz, Masoud Mohammadian, Pablo Moscato, Michael Rumsewicz, Marc Schoenauer, Alice Smith, Russel Stonier, Lyndon While, Xin Yao, and Jacek Zurada. Our special appreciation also goes to two anonymous reviewers who provided us with many insights and useful suggestions, and to Ronan Nugent, who did a wonderful job of editing this book and helping us make the entire project a success.

Lastly, we would like to thank the most famous fictional detective of all time, Sherlock Holmes, for providing us with the entertaining quotes at the beginning of each chapter. Mr. Holmes remains one of the most famous problem-solvers of all time, and his methodology is based on prediction *("It is a capital mistake to theorize before you have all the evidence")*, optimization ("*… I had best proceed on my own lines, and then clear the whole matter up once and for all"*), and adaptability ("*I have devised seven separate explanations … But which of these is correct can only be determined by the fresh information, which we shall no doubt find waiting for us*"). Needless to say, his methodology bears a striking resemblance to Adaptive Business Intelligence! Enjoy.

Adelaide, Australia
October 2006

Zbigniew Michalewicz, Martin Schmidt
Matthew Michalewicz, Constantin Chiriac

Contents

Part I: Complex Business Problems

Part II: Prediction and Optimization

Part III: Adaptive Business Intelligence

Part I:
Complex Business Problems

1 Introduction

> "Like all Holmes's reasoning, the thing seemed simplicity itself when it was once explained."
> *The Stock-Broker's Clerk*

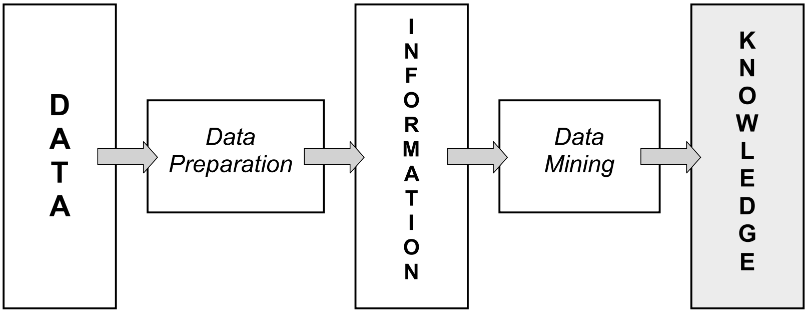
> "Before turning to … matter which present the greatest difficulties, let the inquirer begin by mastering more elementary problems."
> *A Study in Scarlet*

"The answer to my problem is hidden in my data… but I cannot dig it up!" This popular statement has been around for years as business managers gathered and stored massive amounts of data in the belief that they contain some valuable insight. But business managers eventually discovered that raw data are rarely of any benefit, and that their real value depends on an organization's ability to analyze them. Hence, the need emerged for software systems capable of retrieving, summarizing, and interpreting data for end-users.

This need fueled the emergence of hundreds of *business intelligence* companies that specialized in providing software systems and services for extracting *knowledge* from raw data. These software systems would analyze a company's operational data and provide knowledge in the form of tables, graphs, pies, charts, and other statistics. For example, a business intelligence report may state that 57% of customers are between the ages of 40 and 50, or that product X sells much better in Florida than in Georgia.[1]

Consequently, the general goal of most business intelligence systems was to: (1) access data from a variety of different sources; (2) transform these data into information, and then into knowledge; and (3) provide an easy-to-use graphical interface to display this knowledge. In other words, a business intelligence system was responsible for collecting and digesting data, and presenting knowledge in a friendly way (thus enhancing the end-user's ability to make good decisions). The following diagram illustrates the processes that underpin a traditional business intelligence system:

[1] Note that *business intelligence* can be defined both as a "state" (a report that contains knowledge) and a "process" (software responsible for converting data into knowledge).



Although different texts have illustrated the relationship between data and knowledge in different ways, the distinction between data, information, and knowledge is quite clear:

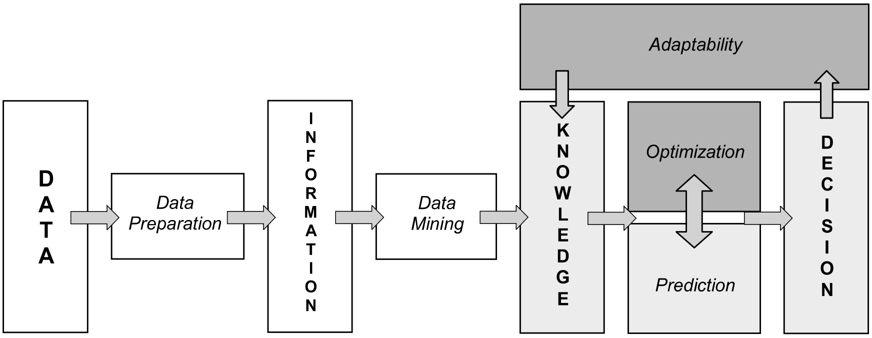
- *Data* are collected on a daily basis in the form of bits, numbers, symbols, and "objects."
- *Information* is "organized data," which are preprocessed, cleaned, arranged into structures, and stripped of redundancy.
- *Knowledge* is "integrated information," which includes facts and relationships that have been perceived, discovered, or learned.

Because knowledge is such an essential component of any decision-making process (as the old saying goes, *"Knowledge is power!")*, many businesses have viewed knowledge as the final objective. But it seems that knowledge is no longer enough. A business may "know" a lot about its customers – it may have hundreds of charts and graphs that organize its customers by age, preferences, geographical location, and sales history – but management may still be unsure of what decision to make! And here lies the difference between "decision support" and "decision making": all the knowledge in the world will not guarantee the right or best decision.

Moreover, recent research in psychology indicates that widely held beliefs can actually hamper the decision-making process. For example, common beliefs like "the more knowledge we have, the better our decisions will be," or "we can distinguish between useful and irrelevant knowledge," are not supported by empirical evidence. Having more knowledge merely increases our confidence, but it does not improve the accuracy of our decisions. Similarly, people supplied with "good" and "bad" knowledge often have trouble distinguishing between the two, proving that irrelevant knowledge decreases our decision-making effectiveness.

Today, most business managers realize that a gap exists between having the right knowledge and making the right decision. Because this gap affects management's ability to answer fundamental business questions (such as "What should be done to increase profits? Reduce costs? Or increase market share?"), the future of business intelligence lies in systems that can provide answers and recommendations, rather than mounds of knowledge in the form of reports. *The future of business intelligence lies in systems that can make decisions!* As a result, there is a new trend emerging in the marketplace called *Adaptive Business Intelligence.*

In addition to performing the role of traditional business intelligence (transforming data into knowledge), Adaptive Business Intelligence also includes the decision-making process, which is based on prediction and optimization:



While *business intelligence* is often defined as "a broad category of application programs and technologies for gathering, storing, analyzing, and providing access to data," the term *Adaptive Business Intelligence* can be defined as "the discipline of using prediction and optimization techniques to build self-learning 'decisioning' systems" (as the above diagram shows). Adaptive Business Intelligence systems include elements of data mining, predictive modeling, forecasting, optimization, and adaptability, and are used by business managers to make better decisions.

This relatively new approach to business intelligence is capable of recommending the best course of action (based on past data), but it does so in a very special way: An Adaptive Business Intelligence system incorporates prediction and optimization modules to recommend near-optimal decisions, and an "adaptability module" for improving future recommendations. Such systems can help business managers make decisions that increase efficiency, productivity, and competitiveness. Furthermore, the importance of *adaptability* cannot be overemphasized. After all, what is the point of using a software system that produces sub par schedules, inaccurate demand forecasts, and inferior logistic plans, time after time? Would it not be wonderful to use a software system that could *adapt* to changes in the marketplace? A software system that could *improve* with time?

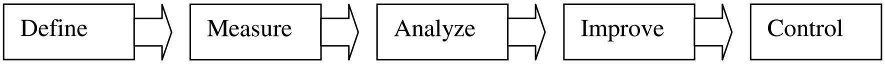
The concept of adaptability is certainly gaining popularity, and not just in the software sector. Adaptability has already been introduced in everything from automatic car transmissions (which adapt their gear-change patterns to a driver's driving style), to running shoes (which adapt their cushioning level to a runner's size and stride), to Internet search engines (which adapt their search results to a user's preferences and prior search history). These products are very appealing for individual consumers, because, despite their mass production, they are capable of adapting to the preferences of each unique owner after some period of time.

The growing popularity of adaptability is also underscored by a recent publication of the US Department of Defense. This lists 19 important research topics for the next decade and many of them include the term "adaptive": *Adaptive* Coordinated Control in the Multi-agent 3D Dynamic Battlefield, Control for *Adaptive* and Cooperative Systems, *Adaptive* System Interoperability, *Adaptive* Materials for Energy-Absorbing Structures, and Complex *Adaptive* Networks for Cooperative Control.

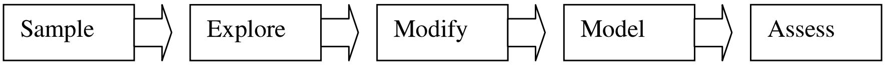
For sure, adaptability is here to stay. It is a vital component of any intelligent system, as it is hard to argue that a system is "intelligent" if it does not have the capacity to adapt. Moreover, modern definitions of natural and artificial intelligence include the term "adaptive." For humans, the importance of adaptability is obvious: our ability to adapt was a key element in the evolutionary process. In the case of artificial intelligence, consider a chess program capable of beating the world chess master: Should we call this program intelligent? Probably not. We can attribute the program's performance to its ability to evaluate the current board situation against a multitude of possible "future boards" before selecting the best move. However, because the program cannot learn or adapt to new rules, the program will lose its effectiveness if the rules of the game are changed or modified. Consequently, because the program is incapable of learning or adapting to new rules, the program is not intelligent.

The same holds true for any expert system. No one questions the usefulness of expert systems in some environments (which are usually well defined and static), but expert systems that are incapable of learning and adapting should not be called "intelligent"! Some expert knowledge was programmed in, that is all.

It is not surprising that the fundamental components of Adaptive Business Intelligence are already emerging in other areas of business. For example, the *Six Sigma* methodology is a great example of a well-structured, data-driven methodology for eliminating defects, waste, and quality-control problems in many industries. This methodology recommends the following sequence of steps:



Note that the above sequence is very close "in spirit" to part of the previous diagram, as it describes (in more detail) the adaptability control loop. Clearly, we have to "measure," "analyze," and "improve," as we operate in a dynamic environment, so the process of improvement is continuous. The SAS Institute proposes another methodology, which is more oriented towards data mining activities. Their methodology recommends the following sequence of steps:



Again, note that the above sequence is very close to another part of our diagram, as it describes (in more detail) the transformation from data to knowledge. It is not surprising that businesses are placing considerable emphasis on these areas, because better decisions usually translate into better financial performance. And better financial performance is what Adaptive Business Intelligence is all about. Systems based on Adaptive Business Intelligence aim at solving real-world business problems that have complex constraints, are set in time-changing environments, have several (possibly conflicting) objectives, and where the number of possible solutions is too large to enumerate. Solving these problems requires a system that incorporates modules for prediction, optimization, and adaptability. In the following chapters of this book, we will discuss these modules in detail, and see how they are combined to create an Adaptive Business Intelligence system.

2 Characteristics of Complex Business Problems

> "See the foxhound with hanging ears and drooping tail as it lolls about the kennels, and compare it with the same hound as, with gleaming eyes and straining muscles, it runs upon a breast-high scent – such was the change in Holmes since the morning."
> *The Adventure of the Bruce-Partington Plans*

> " 'I had,' said he, 'come to an entirely erroneous conclusion which shows, my dear Watson, how dangerous it always is to reason from insufficient data.' "
> *The Adventure of the Speckled Band*

The statement "complex business problems are difficult to solve" is so obvious that it does not require any justification. A closer look at any real-world business problem, whether in distribution, customer retention, or fraud detection, will bear witness to this obvious truth. Most complex business problems share the following characteristics, which is the reason they are so difficult to solve:

- *The number of possible solutions is so large that it precludes a complete search for the best answer.* In other words, the number of possible distributions, routes, fraud rules, or transportation plans is so large, that examining all the possibilities would take many centuries of supercomputing time.
- *The problem exists in a time-changing environment.* This means that yesterday's decision, however optimal, may be far from optimal today.
- *The problem is heavily constrained.* For most problems, the final solution should satisfy many restrictions imposed by internal regulations, capacities, laws, and/or preferences. Sometimes finding even one feasible solution (i. e., a solution that satisfies all problem-specific constraints) is quite difficult.
- *There are many (possibly conflicting) objectives.* For example, the goal of many scheduling problems is to minimize both time and cost, but these two objectives work against each other (as a decrease in time usually results in an increase in cost, and vice versa). To allow business managers to effectively control these tradeoffs, such problems may require an entire set of solutions (rather than just a single solution).

Of course, the above list can be extended to include many other characteristics, such as incomplete information (e. g., the necessary data were not recorded), noisy data (e. g., the data contain rounded figures and estimates), and uncertainly (e. g., the data are not reliable). However, the four primary characteristics listed above are sufficient for our purposes, so let us discuss each of them in turn.

2.1 Number of Possible Solutions

Let us assume we want to find the best solution to a problem with 100 decision variables. To keep this example simple, let us also assume that each of these decision variables is binary (i. e., each decision variable takes one of two possible values, such as "yes" or "no"). Each possible combination of these 100 variables produces some result, which we can evaluate and label with a "quality measure score." Assume, for example, that a sequence

"yes" & "yes" & "no" & "no" & "no" & "yes" & "no" & … & "yes"

produces a quality measure score of 17.3, whereas the sequence

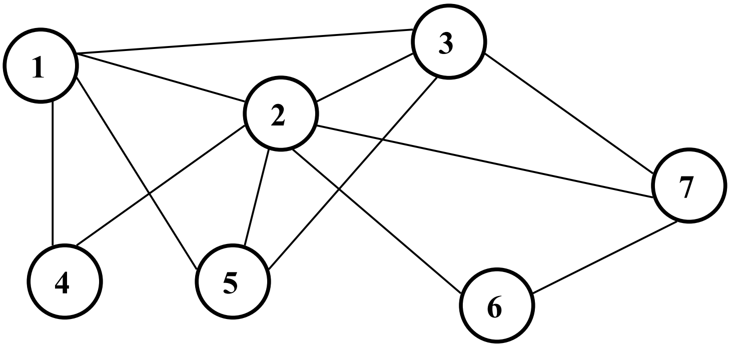
"yes" & "no" & "no" & "yes" & "no" & "yes" & "no" & … & "no"

produces a quality measure score of 18.1. The higher the quality measure score, the better the solution, hence the latter solution is better than the former. Our task is to find the combination of values for the 100 variables that produces *the highest possible quality measure score*. In other words, we would like to find a solution that cannot be improved.

If we do not have any additional problem-specific knowledge, our approach might be to evaluate all possible combinations. However, the number of possible combinations is enormous. Although each variable can only take one of two values ("yes" or "no"), the number of possible solutions grows at an exponential rate: there are four combinations (2 × 2) for two variables, eight combinations (2 × 2 × 2) for three variables, and so on. With 100 variables, there are 2 × 2 × … × 2 (100 times) combinations – a number that corresponds to 10^{30}. Evaluating all of these combinations is simply impossible. Even if we had a computer that could test 1,000 combinations per second, and we began using this computer one billion years ago, we would have evaluated less than 1% of the possible solutions by today!

Let us consider another example, the famous traveling salesman problem. Conceptually, the problem is very simple: traveling the shortest possible distance, the salesman must visit every city in his territory (exactly once) and then return home.[2] The diagram below represents a seven-city version of this problem:

[2] Some closely related problems require slightly different criteria, such as finding a tour of the cities that yields the minimum travel time, minimum fuel cost, or a number of other possibilities, but the underlying principle is the same.



With seven cities, the problem has 360 possible solutions,[3] making it relatively easy to solve. However, by adding a few more cities, this number also grows exponentially. To see the maddening growth of possible solutions, consider the following:

- A 10-city problem has 181,440 possible solutions.
- A 20-city problem has about 10^{16} possible solutions.
- A 50-city problem has about 10^{62} possible solutions.

By way of comparison, our planet holds approximately 10^{21} liters of water, so a 50-city problem has an unimaginably large number of solutions. The number of possible solutions to a 100-city problem exceeds (by many orders of magnitude) the estimated number of atoms in the whole Universe! It is so large that we cannot even conceive of sets with that many elements. Note also, that most real-world business problems are far more complex than this (in terms of possible solutions). They are defined by a much larger number of variables, and these variables usually take on more values than just "yes" or "no." In such cases, the number of possible solutions is truly astronomical!

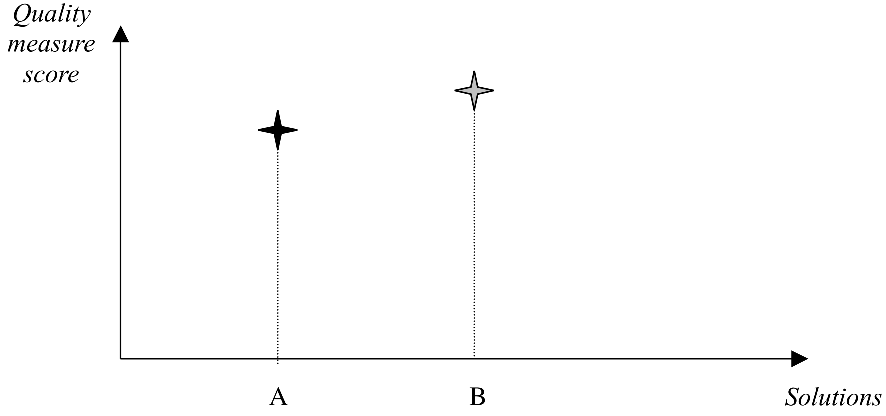
So, how can business managers find optimal solutions to such problems? Because the numbers of possible distributions, routes, fraud rules, or transportation plans might be so large that to examine all possibilities (with even the fastest supercomputers) would take many centuries at the best, an exhaustive search that relies on computing power is clearly not the answer. In the following chapters, we will discuss a real-world business problem where the number of possible solutions is *much* larger than the numbers presented here, and show how such problems can be solved using an Adaptive Business Intelligence system.

[3] We assume that the problem is symmetric (i.e., the distance between cities *i* and *j* is the same as the distance between *j* and *i*). Note also, that solution 1–2–3–4–5–6–7 is the same as solution 3–4–5–6–7–1–2, as both these solutions have a different starting city but represent the same cycle.

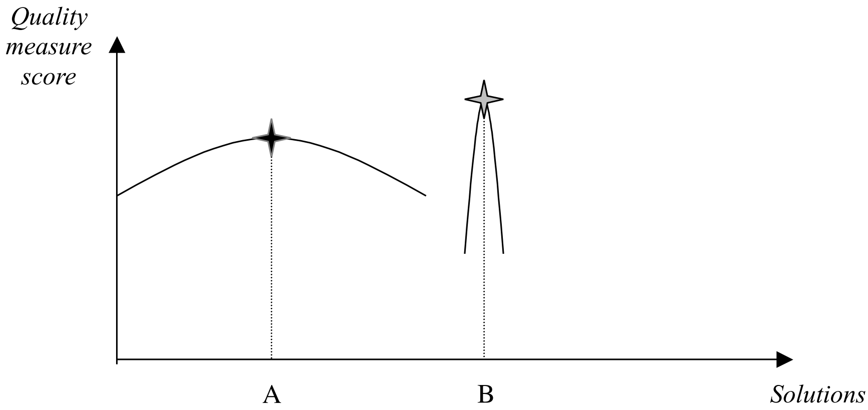
2.2 Time-Changing Environment

Business managers know that the marketplace is not static, and yet they take static snapshots of the problems they are trying to solve. A snapshot is a good starting point for analyzing and understanding a problem, but on its own, it paints a false picture. Because real-world business problems are set in time-changing environments, it is important to address the time factor explicitly. To illustrate this point, let us consider a real-world version of the traveling salesman problem with many delivery trucks. If the problem is carefully analyzed and a set of delivery routes found, the quality of these routes will be affected by many cyclical factors (such as rush-hour and weekend traffic, weather and road conditions, and so forth), and by random events (such as labor strikes or delivery truck accidents). Because the problem is influenced by so many environmental factors, any solution to a static snapshot of this problem might prove useless.

There are some additional issues related to time-changing environments that are worth noting. For example, imagine that we are considering the implementation of solution A or solution B:



Which of these two solutions would we select? Well, the question seems trivial: Because solution B has a higher quality measure score, solution B is better than solution A. Although this statement is true – solution B *is* better than solution A – the answer might not be that straightforward. It may happen that solution A "sits" on a relatively flat peak, whereas solution B "sits" on a very narrow peak:



We can interpret the above graph as follows: Solution B is better than solution A (there is no doubt about that), but if we are forced to modify solution B for any reason (because of equipment failure, bad weather, labor strike, etc.), then the quality of solution B will deteriorate very quickly. Solution A, on the other hand, is much more "stable" in the sense that it can tolerate changes and modifications without a sharp drop in quality. Given that solution A is less risky than solution B, should we still select the "better" solution B?

2.3 Problem-Specific Constraints

All real-world business problems have constraints of some sort, and if a particular solution does not satisfy these constraints then we cannot consider this solution. Scheduling problems serve as a good example of real-world constraints. For example, consider the problem of scheduling airline crews for various flights. First, we have to make a list of all the flights that require service, along with all the requirements for these flights (such as skill levels, number of crewmembers, and so forth). Next, we need a database of all available crewmembers, together with their preferences and characteristics. Once we have all this information, we need to find the "best" assignment of individual crewmembers to different flights – but what does the "best" mean? Well, in this case it may mean the cost of implementing the schedule, the flexibility of making changes/replacements, the degree to which personnel preferences are satisfied, etc. Note, however, that the final schedule must satisfy a few *hard constraints,* such as:

- A crewmember cannot serve on two flights at the same time.
- The crewmembers scheduled for a specific flight must satisfy some requirements (e. g., the captain must have clearance to operate that particular plane, the number of stewardesses should correspond to the capacity of the plane).
- Various laws and regulations, which might call for some minimum layover time for crewmembers in between flights.

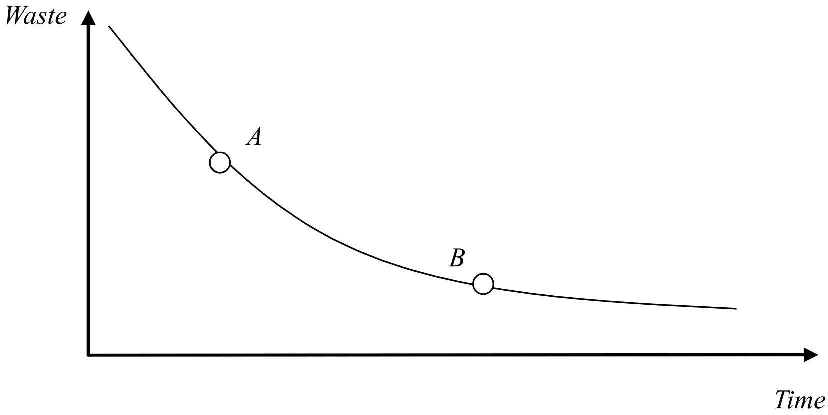
These are examples of hard constraints, which a feasible solution cannot violate. In addition to hard constraints, there are also many *soft constraints*. These represent requirements that are not mandatory, but "nice to have," such as:

- Providing crewmembers with five consecutive rest days each month.
- Not scheduling some crewmembers together on the same flight (for personal reasons).
- Minimizing layover time.

Similar considerations apply to the traveling salesman problem: constraints include capacity limits, delivery time windows, maximum driving time, etc. Some of these constraints are hard (e. g., not transporting chemicals and food together on the same truck), while others are soft (e. g., personnel preferences). But regardless of whether we are routing trucks or scheduling crewmembers, it is necessary to assert the relative importance of each soft constraint by assigning numeric weights to it. When solving the problem, we can then use these weights to calculate a final quality measure score for each possible solution. Without first doing this, it would be extremely difficult to evaluate the various solutions.

2.4 Multi-objective Problems

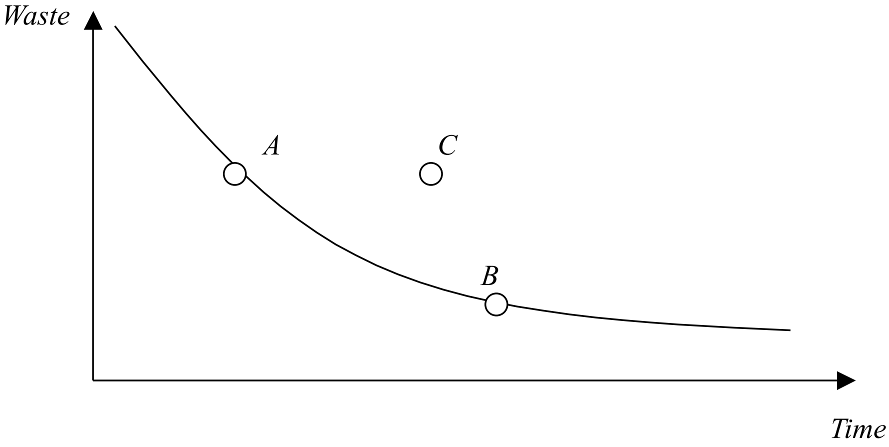
It is quite unusual for any real-world business problem to have only one objective. Consider, for example, what objectives are important for optimizing production? The objectives may include the minimization of production time and the minimization of material waste. Note, however, that these objectives might "work" against each other, as the minimization of production time may trigger an increase in material waste, and vice versa. We can illustrate this tradeoff with the following diagram:



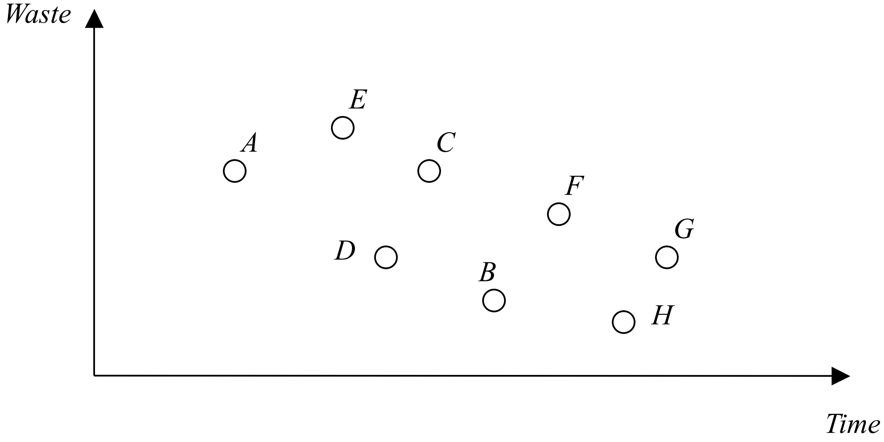
Clearly, the shorter the production time, the greater the waste. The above curve gives us the approximate relationship between these two objectives, and we can estimate the amount of additional time required to reduce waste by a specific amount. Let us consider solutions A and B: Which of them is better? Solution A is faster, but the amount of material waste is higher, and vice versa. In problems with multiple objectives, it is possible to find a solution that is best with respect to the first objective, but not the second, and a different solution that is best with respect to the second objective, but not the first. Multiple-objective (multi-objective) problems pose the challenge of defining the quality of a solution in terms of several (possibly conflicting) parameters.

So, which solution should we select? A or B? It is impossible to answer this question without first agreeing on a common denominator for time and waste. For example, we can translate both objectives into dollars by calculating that five minutes of production time is worth $100, and each pound of material waste is worth $180. We can then calculate the merits (expressed in dollars) of both solutions, compare the numbers, and select the solution with the lowest dollar figure. This approach can be used with a larger number of objectives, but it might be tricky to find the common denominator for some of these objectives if they include criteria such as “workplace health and safety.”

The real-world version of the traveling salesman problem also has multiple objectives: Besides trying to minimize the total travel distance, we are also trying to make as many “on time” deliveries as possible, balance the travel time among all the delivery trucks, and so on. If we have a set of possible solutions to this multiple-objective problem, then it might be convenient to classify these solutions into *dominated* solutions and *non-dominated* solutions. A solution is dominated if a feasible solution exists (i. e., a solution that satisfies all problem-specific constraints) that is (1) at least as good with respect to every objective, and (2) strictly better with respect to at least one objective. The figure below illustrates the case: Solution C is dominated because solution A is as good as solution C on the *waste* dimension, and better on the *time* dimension:



A solution that is not dominated by any other feasible solution is called a *non-dominated* solution. Again, the following figure illustrates the case, where solutions E, C, F, and G are dominated:



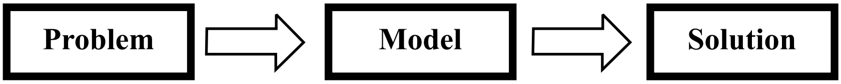
Solution E is dominated because A is better on all objectives; solution C is dominated because D is better on all objectives; and solutions F and G are dominated because B is better on all objectives. On the other hand, solutions A, D, B, and H are non-dominated, because no solutions exist that are as good or better than any of them on all objectives.

Of course, the non-dominated solutions are of interest to us. Ideally, any system that deals with multiple-objective problems should return several diverse solutions (preferably all non-dominated). Each of these solutions might be of interest to us, but in most cases, we can only implement one solution. To decide between these various non-dominated solutions, either human expertise is used to find common denominators (e. g., to express time and waste in dollars) or some higher-level knowledge.

For example, we may assess the relative importance of each objective by assigning numeric weights (in the same manner they were applied to soft constraints in the previous section), or by imposing a ranking for all objectives and then selecting solutions that follow this ranking. Another option would be to select the most important objective and then convert the remaining objectives into constraints that must be satisfied. Very often, the selected approach is dependent on the problem.

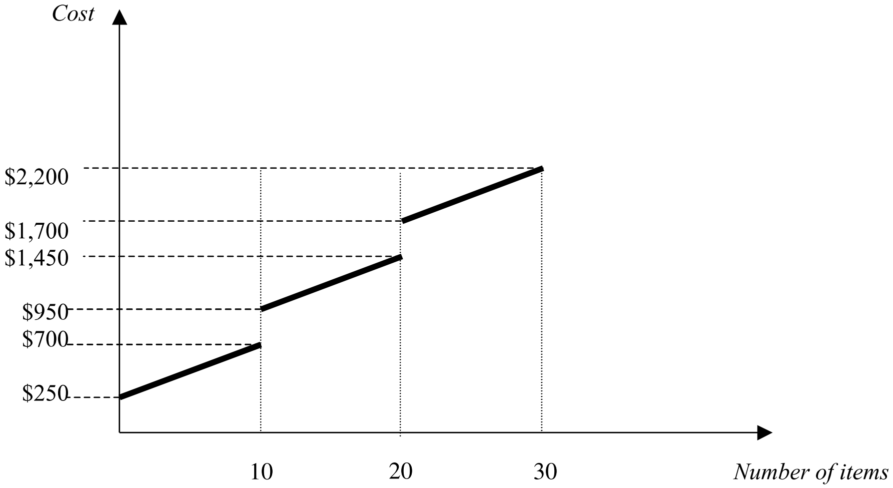
2.5 Modeling the Problem

The problem-solving process consists of two separate steps: (1) creating a model of the problem, and (2) using that model to generate a solution:



Because of this two-step process, we must realize that we are only finding a solution to the *model* of a problem. If the model is accurate, then the solution will be meaningful. But if the model has too many vague assumptions and approximations, the solution may be meaningless, or worse.

Consider the following example: Suppose a company has 80 warehouses and 5 distribution centers, and every possible route between each warehouse and distribution center has a measurable transportation cost. The shape of this cost function depends on a variety of factors, including the distance between the warehouse and the distribution center, the quality of the road, the traffic density, the number of required stops, the average speed limit, and so forth. The transportation model between warehouse 22 and distribution center 4 might be:

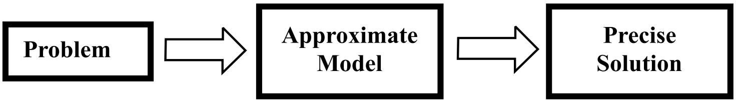


In this model, the cost is zero when there is no delivery. If up to 10 items are delivered, then we incur a fixed cost of $250 and an additional cost of $50 per item shipped (thus, the cost for shipping six items would be $250 + (6 × $50) = $550). However, if we transport 11 or more items (but not more than 20), we have to use two trucks. In this case, the cost is $700 for 10 items, an additional $250 for the second truck, and $50 per each additional item (of course, the real situation could be more complex than this, with trucks having different capacities, timetables, etc.). Given the above assumptions, we can construct a model of the entire problem where we specify:

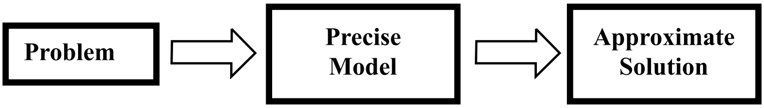
- *All the problem variables.* In this case, there are 400 variables (80 warehouses × 5 distribution centers), with each variable indicating the number of items to be sent from any warehouse to any distribution center.
- *The constraints that define a feasible solution.* These could be (1) no transport from any warehouse can exceed the number of available items in that warehouse, and (2) the total shipment to any distribution center must satisfy the demand (i. e., the total shipment is at least equal to the number of ordered items).
- *The objective.* In this particular example, we might be concerned with minimizing the total transportation cost.

If our model for transporting items between warehouse 22 and distribution center 4 accurately describes the real-world situation, then we can construct similar models for the other warehouses and distribution centers. But such precise models may prove to be of limited utility. Why? For starters, the cost function in this particular model is discontinuous, and discontinuities present severe difficulties for traditional optimization techniques. Hence, the results that we would obtain from using traditional (e. g., gradient-based[4]) techniques on these functions are likely to be quite poor. And if we cannot derive a solution based on this model, the model – as perfect as it may be – is useless for deciding what to do!

So what options do we have? Well, we can try to simplify the model so that traditional optimization techniques produce better solutions, or we can keep the model unchanged and use a non-traditional approach for finding a near-optimum solution. Put a different way, the first approach uses an approximate model of a problem, and then finds the precise solution for this approximate model:

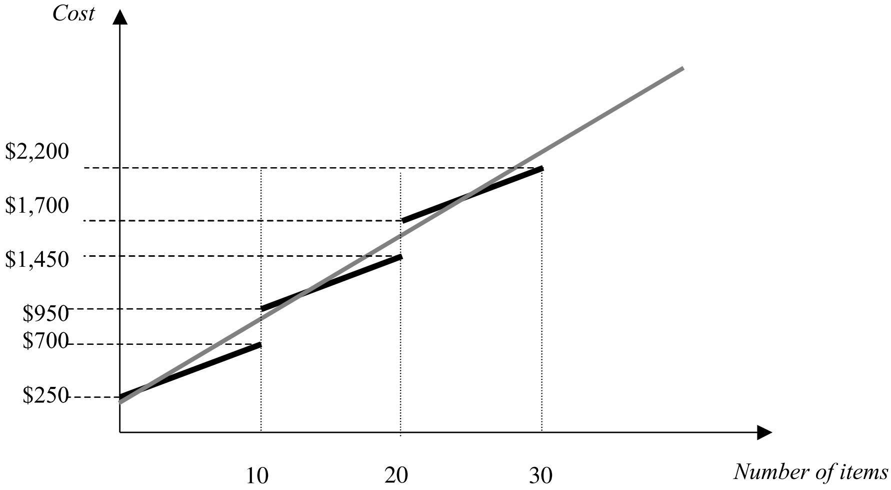


and the second approach uses a precise model of the problem, and then finds an approximate solution for this precise model:



The first approach is quite tempting. For example, we can *approximate* the transportation model between warehouse 22 and distribution center 4 as follows:

[4] *Gradient* is a rate of change with respect to distance of a variable quantity.



We can simplify the other models in a similar manner, and by making all the models linear we can then use a linear programming method[5] to find a precise solution. However, this precise solution would be a solution for the simplified model, and not for the real problem!

The second approach is to leave the precise model unchanged – with all its discontinuities and irregularities – and use a non-traditional method to find a near-optimal solution. Of these two approaches, the latter one is often superior (i. e., an approximate solution to a precise model is "better" than a precise solution to an approximate model). To understand why, look at this situation in the following way: An approximate model usually hides the "irregularities" of a problem, thus allowing some traditional method (e. g., linear programming) to provide a precise solution. However, by hiding the irregularities of a problem, we lose much of the information needed to find the optimal solution! For example, in our simplified model above, note that the difference in transportation cost between 20 and 21 items is now the same as the difference between 19 and 20 items (which is not the case in the precise model). Consequently, the simplified model does not "see" the thresholds that play a major role in identifying the optimal solution. Thus, the "optimal" solution for an approximate model is usually more appropriate for the wastebasket than for implementation!

2.6 A Real-World Example

Let us consider a real-world example to put these business problem characteristics into context. This example is based on a "pollution control" research project that

[5] *Linear programming* is a problem-solving method in which a linear function of a number of variables is subject to a number of constraints in the form of linear inequalities.

was completed during the late 1990s by a team of scientists from Poland and the United States. Today, we consider this project to be a forerunner of Adaptive Business Intelligence.

The main purpose of the project was to reduce ecological damage in Poland through better energy production and distribution. However, the problem was not that straightforward: The proposed solution had to reduce ecological damage *without increasing aggregate operational costs of the power stations* or *failing to meet consumer demand* (i. e., reducing the country's energy output in order to reduce ecological damage was not an option). Since the country's demand for electricity had to be satisfied, there were numerous constraints for energy production in each region. Although this problem is quite specific, it illustrates many characteristics that are common to almost all real-world business problems (e. g., an enormous number of possible solutions, many complex constraints, a time-changing environment). The similarities between this problem and other complex business problems will become apparent in the following chapters.

To solve this challenging problem, an experimental Adaptive Business Intelligence system was developed. The goal of the system was to reduce ecological damage in Poland by optimizing the energy output (and consequent pollution output) of 132 government-owned power stations. However, before we discuss the system, let us briefly address the complexity of the problem.

First, because the system was tasked with optimizing the energy production of 132 power stations, the number of possible solutions was enormous. If an integer between 1 and 10 represented the production level of each power station (i. e., 1 = 10% production, 2 = 20% production, and so forth), the number of possible solutions would be 10^{132} (10 possible solutions for one power station, 10 × 10 possible solutions for two power stations, 10 × 10 × 10 possible solutions for three power stations, etc.). A supercomputer capable of evaluating 1,000 solutions per second would require billions of years to examine less than 1% of the search space! The fact that these power stations operated in a time-changing environment only compounded the problem, as the search space of possible solutions changed from one day to the next.

Second, the hard and soft constraints of this problem were far from trivial. Since enough energy had to be produced to supply the entire country, it was necessary to take into account the capacity and layout of the power grid in Poland. Also, although the demand for energy was constantly in flux, it was not possible to change the production levels of a particular power station too drastically. On top of everything, the system had to "protect" certain areas of Poland from long exposures to sulphur dioxide (SO_2), as some areas of the country were significantly more sensitive to SO_2 pollution than other areas.

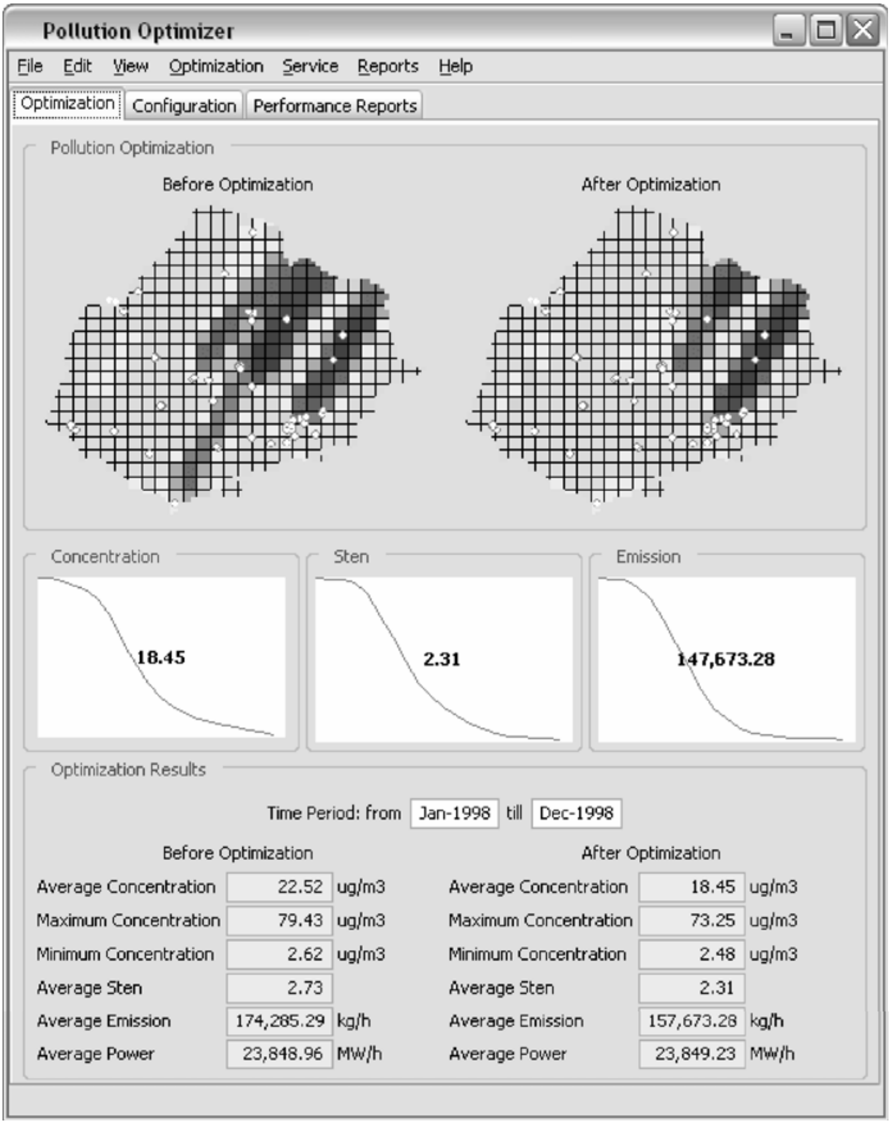
The first step in solving this problem lay in identifying the relevant problem variables and all the relationships between them. This was accomplished by building a computational "grid model" of Poland's land area and corresponding power grid. Each square in the grid corresponded to 30 km × 30 km, which collectively covered Poland's approximate 900 km × 750 km land area. The locations of the 132 government-owned power stations were plotted along with their energy output and emission levels, and then the pollution tolerances for each 30 km × 30 km square were computed. It was also necessary to take into account the pollution caused by private enterprises in Poland, as well as many foreign sources (mainly from Germany and the Czech Republic). Because it was not possible to influence these sources, they were defined as *background concentrations*. Hence, the resulting pollution concentrations in each square were the sum of both the primary and background concentrations.

Because weather played a major role in determining how pollution affected the environment in Poland (e. g., gusty winds could easily spread high levels of pollution to remote areas of the country), it was necessary to use sophisticated weather forecasts for predicting the ecological damage that each 30 km × 30 km square would sustain during the next 48 hours. Thus, by using data on the amount and type of pollution emitted from each power station, along with a weather forecast for the speed and direction of winds, a prediction model was developed to predict how much pollution would be created, how it would be dispersed, and how much damage it would cause in the affected squares.

To minimize the overall ecological damage in Poland (in light of the predictions made by the prediction module), an evolutionary algorithm was developed to find the optimal production level for each power station (evolutionary algorithms are discussed in Sect. 6.6). Although it was not possible to decrease the overall energy produced in Poland, it was possible to make local or regional adjustments to take advantage of various weather conditions. Furthermore, the evolutionary algorithm strived to *continuously* optimize the pollution in Poland on a daily basis. When a new weather forecast became available (along with the corresponding prediction for ecological damage), the system did not start the search from scratch; instead, it incorporated this new knowledge and continued the optimization process. Lastly, the system incorporated a module for adaptability, which compared the predicted outcome with the actual outcome. If significant prediction errors were discovered, the adaptability module would adapt the parameters of the prediction module (more on this subject in Sect. 10.3). Hence, the system was able to continuously improve its performance.

Using historical data for weather forecasts and emission levels, several experiments were conducted using this new system. The optimization process was applied to this historical data, and then the optimized results were compared with the actual, non-optimized results. From these experiments, the following conclusions were drawn: The amount of energy produced was equal in both cases, the aggregate operating costs were also equal, but the ecological damage in Poland was 15% to 18% less when the optimization process was applied. A “saving” of 15 to 18% in ecological damage is a tremendous result, especially when we consider that human lives are at risk in this particular problem. The system also contained numerous input screens, so that individual parameters could be modified to alter regional pollution tolerances (in relation to cost and service).

The sample screen below shows the optimization results. Both the “before” and “after” maps in this screen represent the country of Poland, which is divided into a grid model of 30 km × 30 km squares (which we discussed earlier). The darker



Time Period: from Jan-1998 till Dec-1998

Before Optimization			After Optimization		
Average Concentration	22.52	ug/m3	Average Concentration	18.45	ug/m3
Maximum Concentration	79.43	ug/m3	Maximum Concentration	73.25	ug/m3
Minimum Concentration	2.62	ug/m3	Minimum Concentration	2.48	ug/m3
Average Sten	2.73		Average Sten	2.31	
Average Emission	174,285.29	kg/h	Average Emission	157,673.28	kg/h
Average Power	23,848.96	MW/h	Average Power	23,849.23	MW/h

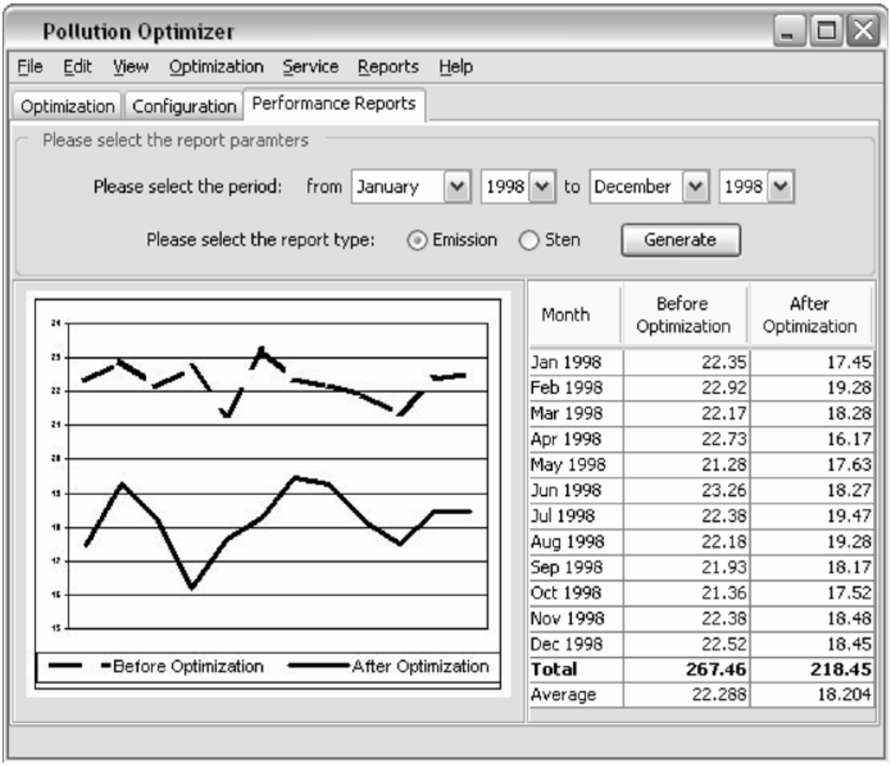
squares represent higher concentrations of pollution, and the white dots represent the 132 power stations:[6]

The left-hand-side map shows the non-optimized results for ecological damage in Poland, while the right-hand-side map shows the optimized results for the same time period. Clearly, there are fewer dark squares on the right-hand-side (optimized) map. This is reflected in the concentration and emission numbers shown at the bottom of

[6] Some dots represent multiple power stations.

the screen. It is important to note that the system achieved these lower pollution levels while maintaining the same power production output of approximately 23,849 megawatts per hour.[7]

The report below shows the potential pollution reduction that would have been achieved in 1998:


Pollution Optimizer
File Edit View Optimization Service Reports Help
Optimization Configuration Performance Reports
Generate
24
23
22
21
20
19
18
17
16
15
Before Optimization
After Optimization

Please select the report paramters

Please select the period: from January 1998 to December 1998

Please select the report type: ● Emission ○ Sten

Month	Before Optimization	After Optimization
Jan 1998	22.35	17.45
Feb 1998	22.92	19.28
Mar 1998	22.17	18.28
Apr 1998	22.73	16.17
May 1998	21.28	17.63
Jun 1998	23.26	18.27
Jul 1998	22.38	19.47
Aug 1998	22.18	19.28
Sep 1998	21.93	18.17
Oct 1998	21.36	17.52
Nov 1998	22.38	18.48
Dec 1998	22.52	18.45
Total	**267.46**	**218.45**
Average	22.288	18.204

The right-hand-side graph displays two levels of emitted toxic materials: The *Before Optimization* column displays the average monthly emissions for the default energy production scheme; and the *After Optimization* column displays the average monthly emissions for the optimized energy production scheme. The difference between the non-optimized and optimized schemes is approximately 18% (267.46 – 218.45 = 49.01 / 267.48 = 18.32%), and this was accomplished without increasing the aggregate operational cost of the power stations or decreasing the amount of energy produced. By allowing the system some flexibility in these constraints, further ecological savings could be realized.

[7] The *sten* values in the sample screen represent a scientific estimate of a region's sensitivity to pollution (mainly defined by the level of sulphur dioxide present).

As indicated at the beginning of this section, similar “savings” can be realized in other business domains, even if the particular problem is substantially different. In Chap. 4, we will discuss the overall structure of an Adaptive Business Intelligence system by outlining its key components. At that stage, it will be easy to see the similarities between this “pollution control” research project, and the core components of an Adaptive Business Intelligence system (prediction, optimization, and adaptability).

3 An Extended Example: Car Distribution

> "You must drop everything, Sherlock. Never mind your usual petty puzzles of the police-court. It's a vital international problem that you have to solve."
> *The Adventure of the Bruce-Partington Plans*

> "I have devised seven separate explanations, each of which would cover the facts as far as we know them. But which of these is correct can only be determined by the fresh information which we shall no doubt find waiting for us."
> *The Adventure of the Copper Beeches*

Let us consider an extended example that most of us can relate to, which involves used cars. Specifically, let us investigate what leasing companies do with the cars that are returned at the end of a lease agreement (the so-called "off-lease" cars). We will refer to this extended example in many of the following chapters, so it is worthwhile to study this real-world business problem in detail.

3.1 Basic Terminology

However, before we discuss this example, let us first cover some basic terminology:[8]

- *Database*: A (historical) collection of data, which is the starting point for data mining and model building. Databases are usually updated on a regular basis, thereby increasing the number of stored cases. It is convenient to represent a database as a table, or a collection of tables. We can visualize the necessary data as a two-dimensional table; in this example, the table represents used cars that were sold at auction. One dimension of the table represents the number of cases (each case is a particular car), and the other dimension represents the number of variables (the characteristics of each car):

[8] Note, however, that we will not discuss the physical organization of data (which is important from a performance perspective), but rather the data structures that support the modeling of information.

VIN	Type	Make	Model	Miles	Year	Color	Transmission	Body/Doors	Damage
2G1FP22P1P2100001	Rental	Chevy	S-10	34,983	2002	Silver	Manual	2D	$0
WB3PF43X8X9000331	Lease	Chevy	Cavalier	59,402	2001	Red	Automatic	2D Coupe	$0
4BBG38FJF04JDK000	Lease	Chrysler	Sebring	74,039	2000	Gray	Automatic	2D Coupe	$500
DJOW03FFU990SJ206	Lease	Ford	Escape	37,984	2001	Green	Manual	4D Sport	$250
JD8320DJ2094GK2X3	Rental	Ford	Focus	30,842	2001	Green	Manual	4D Sedan	$0
2JE9F0284JD0213M3	Lease	Isuzu	Rodeo	59,044	1999	White	Automatic	4D Sport	$250
4380JDDD9W02MD001	Rental	Jeep	Cherokee	48,954	2000	Black	Automatic	4D Sport	$500
490DK20285JF0209D	Rental	Mazda	626	38,943	2000	White	Automatic	4D Sedan	$0
10D92JD920KD00002	Lease	Nissan	Altima	39,488	2000	Black	Automatic	4D Sedan	$0
D920DKJ0284JJ9990	Rental	Nissan	Altima	23,584	1999	White	Manual	4D Sedan	$0
JD88D92JJD02K3361	Rental	Saturn	L	21,048	2001	White	Automatic	4D Sedan	$750
10DS0JJ20DXI00093	Lease	Suzuki	Vitara	15,849	2003	Yellow	Automatic	2D Sport	$0
21KD02KD0DJ920M27	Lease	BMW	Z3	49,858	2000	Blue	Manual	2.3 RSTR	$250
389DJ2DD298JWQ082	Lease	Ford	Explorer	42,893	2002	Green	Automatic	XLT 4WD	$0
108DJ2048FJJ20043	Rental	Ford	Mustang	20,384	2002	Red	Manual	GT	$0
DJC82002009DD2J04	Rental	Mercury	Frontier	27,849	2001	Silver	Automatic	SE-V6 Crew	$500
830DMM3029XMW0092	Lease	Honda	Accord	26,849	2002	Yellow	Automatic	EX V6	$0
CNEU200220CCI2202	Rental	Toyota	4Runner	33,483	2000	Silver	Automatic	SR5	$0
CNDJ2940JD88D2JD0	Lease	VW	Beetle	5,459	2003	Blue	Manual	GLS 1.8T	$0
1VC0CMEJ200V9EJJ1	Lease	Toyota	4Runner	81,837	2001	Silver	Automatic	LMTD 4WD	$250

- *Case:* A data structure that represents a single occurrence of recorded phenomenon (e. g., a sale, transaction). Referring to our database of sold cars, each case provides a description of a car (VIN,[9] make, model, mileage, etc.) together with transaction details (sale price, date, location). Cases (also called *records, instances,* and *examples*) correspond to the horizontal rows in database tables.
- *Variable*: The smallest data unit in a case. Variables (also called *attributes, fields, features, dimensions,* and *characteristics*) correspond to the vertical columns in a table. For example, in our database of sold cars, there are many variables (e. g., "color," "mileage"), and each variable takes its value from a predefined set. For example, the variable "color" can take any of the following values: "white," "silver," "beige," etc. The variable "mileage" (marked in the above table as "miles") can take any integer as a value (e. g., 34,789).
- *Prediction method:* A particular method that is used to build a prediction model from a data set. For example, we can use the *linear regression* method to develop a linear regression model (covered in Sect. 5.2.1).
- *Prediction model:* The output of a prediction method. Different prediction models have different characteristics, advantages, and disadvantages (e. g., response time, precision, ease of updating the model).
- *Parameter:* A number that defines the relative "merit" of a variable that is used in a prediction model. Every model has a few parameters (sometimes called *coefficients* or *weights*). For example, a linear regression model for predicting the sale price of a particular car at a particular location may be defined by a function:

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

  which provides the predicted price for a new case when supplied with the numeric values of the other variables ("mileage," "year," "color," etc.). The numbers *a, b, c, d, …* are the parameters of the model, and they define how much "weight" the model gives to each variable (in this example, parameter *a* is a constant). For example, if the parameters *b* and *c* are set to –0.6 and –0.2 in the above function, then this means that the model will put three times more weight on the variable "mileage" than on the variable "year" when predicting the sale price. The parameters of a model require careful tuning during the training phase.

Because everything begins with data, these definitions are a prerequisite to our discussions. The other relevant terms will be provided in the course of the text.

3.2 Off-lease Cars

When a lease agreement expires, the off-lease car is either returned to the leasing company or purchased by the leasee. The leasing company does not worry about the cars that are purchased, but the cars that are returned have to be sold at auction.

[9] *VIN* is an acronym for "vehicle identification number," which is a string of 17 digits and letters that contains considerable information about a specific vehicle, (including country of origin, manufacturer, and model year).

Each returned car is different in its make, model, body style, trim, color, year, mileage, and damage level.

A leasing company maintains this information in an inventory database, as the one displayed in the previous section. Because a car's characteristics will strongly influence its sale price, leasing companies also keep careful track of this information. The sales data are typically kept in a database that contains the VIN, ZIP code of the auction site (there are hundreds of different auction sites in the United States), date, and the sale price of each car:[10]

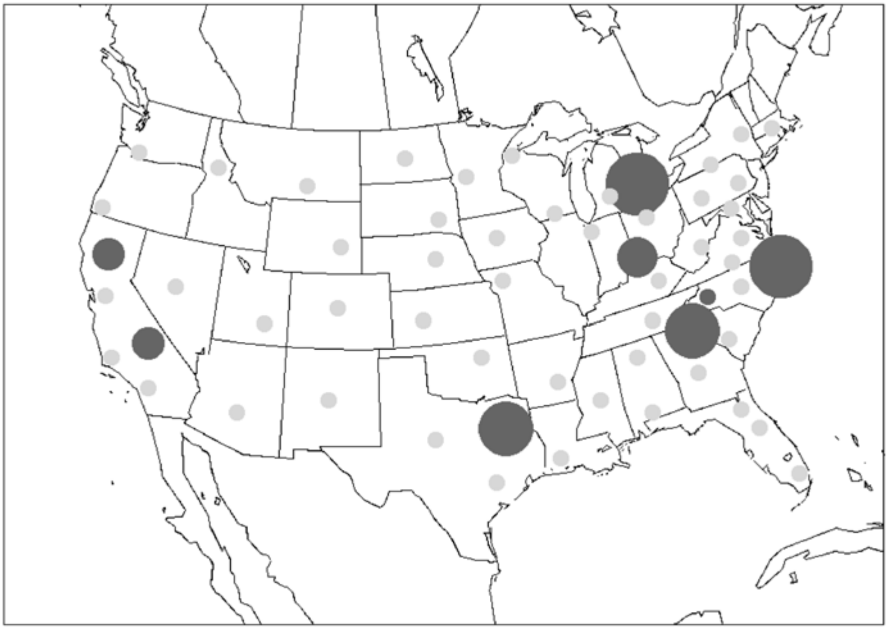
VIN	ZIP	Date	Price
39WWK93309KJ33012	28262	2.11.2004	$12,035
UDJ2293M99DL0K220	30334	2.11.2004	$15,600
4D09WJD92JE93H990	30334	2.11.2004	$10,590
KD37D92JF83NF8822	90012	3.11.2004	$9,265
NKI2389DD974F2235	28262	3.11.2004	$13,450
K29DH38FHW02HD923	48243	3.11.2004	$13,955
MDK293HFDWH299305	90012	4.11.2004	$12,495
28DN39FNDJW2N0024	90012	4.11.2004	$11,925
29H93NFI3HJF93F04	48243	4.11.2004	$11,396
ND920ENF1NAD02834	48243	5.11.2004	$9,835
D39DJ39EHQ8HH9335	28262	5.11.2004	$8,965
02UFIMF03JF9SH935	90012	5.11.2004	$13,960
D932NF93HG9057362	48243	5.11.2004	$8,830
00F8EB3IDNB293758	48243	8.11.2004	$7,920
IE038THJ203TH0234	28262	8.11.2004	$19,250
39FH324MV092HGM39	48243	8.11.2004	$22,640
F92N9F389FH120458	90012	8.11.2004	$13,580
F9485JG03H25495J5	30334	9.11.2004	$16,970
08GN94HJH03J49327	30334	9.11.2004	$14,320
F04JH402KG4509G45	48243	9.11.2004	$9,110

Given all of this information, the question is: Where should leasing companies send their off-lease cars to get the best price? This is a difficult question to answer, especially when we consider that some leasing companies lease more than *one million* cars each year. That is more than 3,000 cars per day – a truly staggering figure!

3.3 The Problem

Now, imagine that we are in charge of selling off-lease cars for a fictitious leasing company. Let us assume that the company receives 3,000 off-lease cars each day and does business with 50 designated auction sites. The following figure illustrates the case (for a particular day of operation):

[10] We could easily obtain the characteristics of each car by merging it with the previous table.



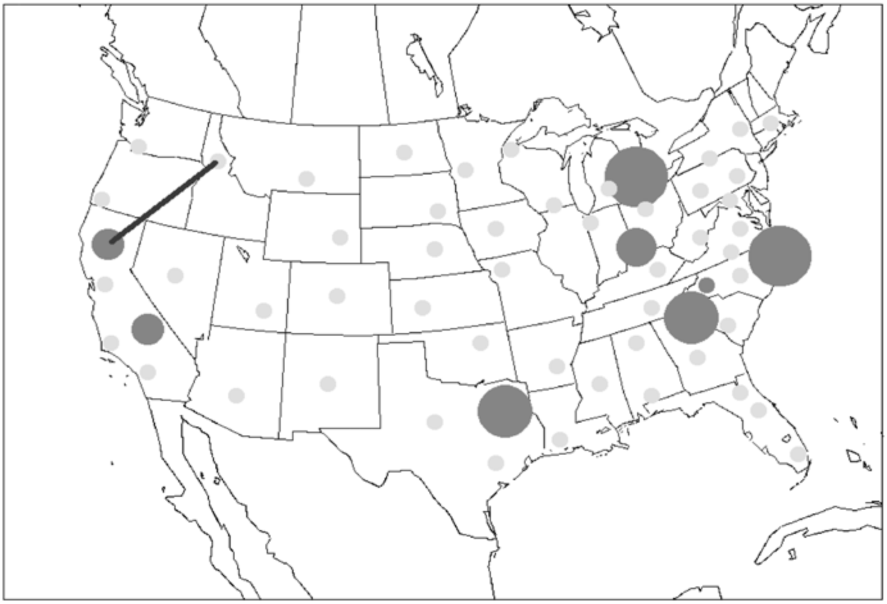
The darker circles represent the locations of the returned cars; the larger the circle, the more cars were returned in that location (clearly, most of the cars were returned on the east coast of the United States). Note that the sizes and locations of the darker circles will vary from one day to the next, as different people and organizations will return their cars at different locations. The lighter circles of the same size, on the other hand, represent the 50 auction sites that are available for selling the returned off-lease cars. The locations of these auction sites are fixed.[11]

Our task is to distribute a daily load of 3,000 cars to these 50 auction sites. In other words, we have to assign an auction site to each car. For example, the first car may be located at a dealership in Northern California, and we have to make a decision where to ship it. At first glance, this looks easy. We might be tempted to take one car at a time, consult some report[12] on what the average sale price for that particular car is at each auction site (after adjusting for mileage, trim, damage level, etc.), and then ship the car to the auction site with the highest average sale

[11] Although the locations of the 50 auction sites are fixed, the company may, from time to time, change the auctions it does business with by dropping some sites and adding new ones (thereby changing the layout of the 50 lighter circles). This may happen if cars are routinely damaged at some sites, auction fees go up, or some other reason. However, these decisions raise several additional questions, such as: *How do we evaluate the monetary impact of dropping some sites and adding others?* and, *Can we increase profits by replacing some auction sites with others?* We will address these important questions later in the text.

[12] Many reports are available for estimating the auction price of cars, including *Black Book*, *Kelley Blue Book*, the *Manheim Market Report*, and others.

price. Of course, we would also have to estimate the transportation cost (the longer the distance, the higher the cost) to each auction site, but these calculations are quite manageable. Using this method, our decision for the first car can be visualized in the following way:

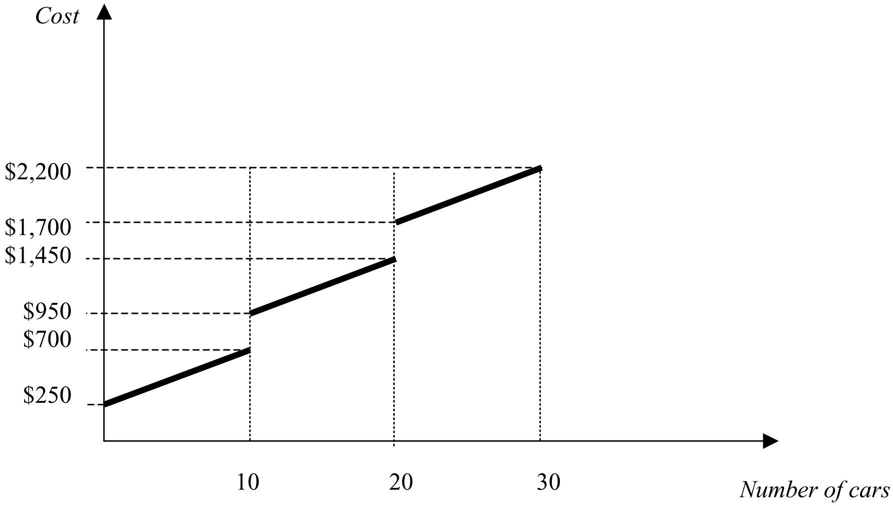


The line represents our decision to ship the car from Northern California (larger circle) to an auction site in Idaho (smaller circle). We would have to repeat this decision process for each car, and, although tedious, it is definitely doable. To minimize the amount of work involved, we can also write a simple program to do this tedious work for us.

So, what is the problem? Well, simply put, the problem is that the above approach does not work very well. To see why, we have to delve a little deeper into the problem.

3.4 Transportation

When we ship an entire truckload of cars from one place to another, we get a cheaper rate per car than if we send only one car (or a few cars) at a time. This occurs because the major component of the transportation cost is the truck, and it is of lesser importance how many cars are actually on the truck. Hence, the relationship between the transportation cost (between two fixed locations) and the number of transported cars may look something like this model (recall the discussion on transportation cost models from the previous chapter):



Given the above model, the cost for sending a single car from one location to another is $250, but the cost of sending two cars is "only" $300 (note that the cost per car is then $150). Then there are $50 increments per car up to 10 cars (as we can load all of them onto a single truck). Thus, shipping 10 cars would cost $700, or just $70 per car. Note that the increments from one car to the next for the first 10 cars need not be linear. For example, the truck might have two levels and loading a car onto the upper level might be slightly more expensive than loading a car onto the lower level. For this reason, a straight line need not represent the "1 to 10 cars" part of the graph. Furthermore, if we want to transport 11 cars, then we will experience a "jump" in cost: We will pay $700 for 10 cars on the first truck, plus $250 for a single car on the second truck (for a total of $950).

Because most transportation companies have trucks of different shapes and sizes (which can hold different amounts of cars), a more realistic transportation cost model may look something like (again, we assume fixed locations for "to" and "from"):

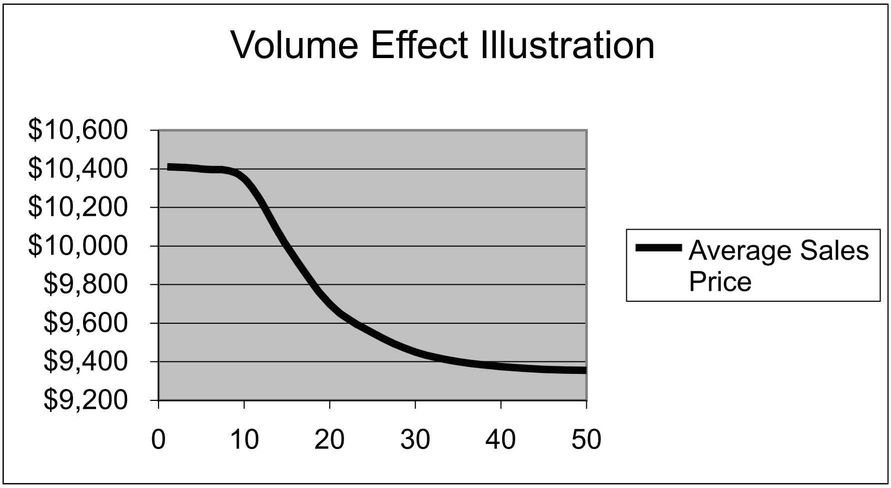
- 1–6 cars: $120 per car.
- 7–10 cars: $95 per car.
- 11–14 cars: $85 per car.
- The cost of transporting more than 14 cars is calculated as follows: $85 per car for multiples of 14 cars (such as 28, 42, 56, etc.), $120 per car for the remaining 1–6 cars, $95 per car for the remaining 7–10 cars, and $120 per car for the remaining 10–13 cars. Transporting 20 cars, for example, would cost $85 per car for the first 14 cars and then $120 per car for the remaining 6 cars, for a total of $1,910 (14 × $85 + 6 × $120).

In addition to transportation cost, we also have to deal with numerous other issues, such as depreciation, auction calendars (auctions are not organized every day), and risk (cars can fall down from trucks and get damaged, cars might get stolen, etc.). We will discuss some of these later in this chapter.

3.5 Volume Effect

Although our goal is to ship each car to the auction site that offers the best price, we might inadvertently trigger the "volume effect" by sending too many cars of same color, make, and mileage to the same auction site. To understand the volume effect, imagine an auction site that holds car auctions every second Wednesday at 10 am. Used-car buyers come to this auction site in the morning to inspect the cars, choose the ones they want, and decide how much to bid. Now imagine what would happen if we sent 45 white Ford Mustangs to this auction site: In all probability, these cars would sell for the minimum opening price. With 45 identical cars for sale, most buyers would be reluctant to bid up the price. On the other hand, imagine if we sent only one white Ford Mustang to this auction site: There would be several interested buyers, and they would bid up the price of the car. This volume effect is extremely important, as the higher the auction sale price, the more money the company will make.

The volume effect for a particular car at a particular auction site might look something like:



This graph illustrates the volume effect phenomenon: We get more money per car by selling *fewer* similar cars. For example, the current average sale price for a particular car at a particular auction site might be $10,400. We can get the same price if we ship up to seven cars to this location. However, if we ship 30 similar cars, then the average sale price per car will drop to $9,450. Note that the term "similar" can mean more than just the same make/model or color. For example, many white compact cars of different makes and models would still compete for the same buyers, thereby reducing the average sale price per car.

Consequently, we cannot consider one car at a time – we have to consider the collective effect of all the cars. This means that for one car we have 50 possible solutions (as we can ship this car to one of 50 auction sites), for two cars we have 2,500 possible solutions (50 × 50), for three cars we have 125,000 possible solutions (50 × 50 × 50), etc. For 3,000 cars, we have approximately 50^{3000} possible solutions (50 multiplied by itself 3,000 times)! This is an overwhelming number (larger than the estimated number of atoms in the Universe) and no supercomputer can evaluate all the possible combinations in a billion human lifetimes. Nevertheless, we have to make a decision for all of these cars *today!*

3.6 Price Depreciation and Inventory

To further complicate matters, every auction site has a set day for selling cars (e. g., every second Friday at 10 am). Because of this, if we ship 100 cars to an auction site and the delivery arrives one or two days *after* the auction day, then the cars will have to wait until the next sale day. This is bad for the leasing company, because on top of other reasons (such as paying the auction sites to keep the cars on their premises for up to two weeks), the cars will also lose value on a daily basis. This phenomenon is called "price depreciation," and it amounts to approximately $10 per day, per average car.

Also, we might already have 1,000 cars at a particular auction site, but only 250 of our cars are sold (on average) at each sale. Hence, it will take approximately four auction sales to sell our current inventory. This means that if we ship an additional 100 cars to this particular auction, they would be sold some two months later. Therefore, we have to take into account our current inventory at each auction site, as it will affect the price depreciation on future shipments. Of course, our inventory should also include cars that are in transit to the auction, and before we can make any distribution decision, it would be necessary to check our inventory levels at *all* 50 auction sites. Otherwise, it would be next to impossible to make a smart decision on where to ship today's load of cars (this is especially important for large leasing companies, which may have 50,000 or more cars sitting in inventory).

3.7 Dynamic Market Changes

Another issue that makes our job more difficult is the fluctuation in used car prices. These fluctuations may be slow and subtle, or sudden and dramatic (as was the case after the September 11th terrorist attacks), and they are often region specific (e. g., it is better to sell convertibles in sunny Florida during the wintertime than in snowy Boston – something called the "seasonality effect"). These changing market conditions force us to stay on top of price changes at each auction site – something that is very difficult to do. We also have to deal with next year's models entering the marketplace during the months of August and September, as the older models will immediately drop in price when this happens (also part of the "seasonality effect"). This is an important consideration when we approach this time of year, as it might be better to ship cars nearby and sell them "in time," rather than sending them far away to fetch a better price. Additionally, new body style models are introduced every few years, which cause an even bigger drop in price for the older body styles.

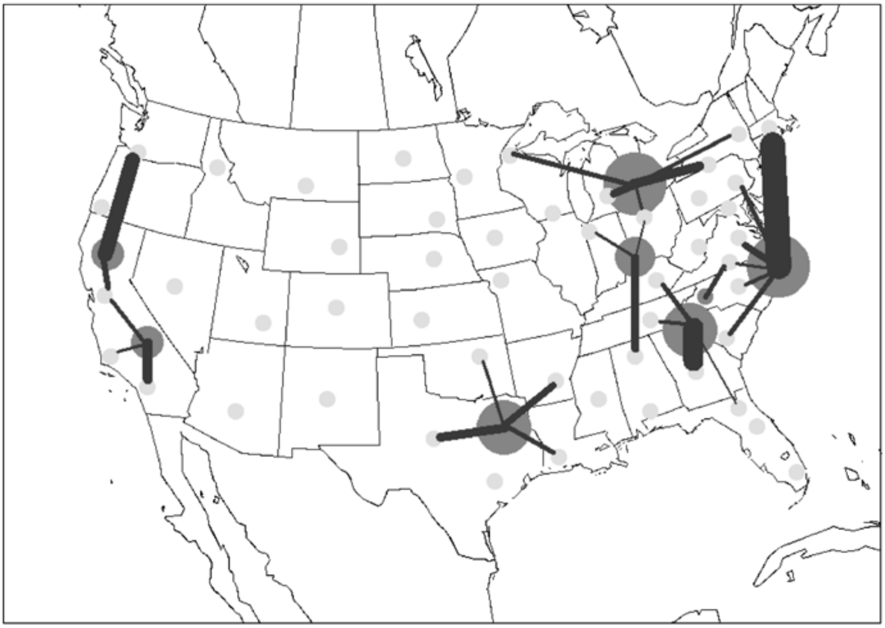
And that is not all. Note that it takes some time to transport a car to a specific auction site. The truck has to drive to the pick-up location, load the car, pick up some additional cars (possibly somewhere close by), and then, finally, drive the cars to the assigned auction site. This process can take two weeks (or longer for shipments from the East Coast to the West Coast), and during this time the prices might change at the auction site. Hence, we have to predict the sale price for each car a couple of weeks ahead of time, and take into account seasonality and other changes in the market. Again, let us illustrate this by an example.

Say we are considering several possible auction sites for a car located in Jacksonville, Florida. Specifically, we are considering sending the car to an auction site in Georgia, Pennsylvania, or California. The price estimations for these three auction sites are different, because we are interested in an estimated sale price for 5 days from today for the Georgia auction site, an estimated sale price for 10 days from today for the Pennsylvania auction site, and an estimated sale price for 15 days from today for the California auction site. The differences in time are due to the transportation distance. However, to estimate these prices, we have to take into account factors such as the seasonality effect and price depreciation, and these additional factors make our decision a bit tricky.

In making the decision of Georgia vs. Pennsylvania vs. California, we also have to take into account transportation costs, the volume effect, and current inventories (as discussed earlier). We also have to weigh the possibility of a better price in California against higher transportation costs, higher price depreciation, higher risk, etc. The decision is not an easy one.

3.8 The Solution

Without a doubt, this is not a trivial problem. In order to make a decision, we have to consider the characteristics of each car, many different possible auction sites, complex transportation costs, volume effects, countrywide inventory of cars (or cars in transit to the auction), price depreciation, and market-driven changes in price. At the end of the day, we do not know whether our decisions are "optimal" – whether they maximize the potential profit for the company, or whether there is a better solution – but we can visualize the overall distribution for a particular day as follows:



In the figure above, the lines represent the transportation connections between the car locations and auction sites, and the thicknesses of these lines represent the transported volume (the thicker the line, the more cars are shipped). Because the problem is so complex, every leasing company has a dedicated team for assigning off-lease cars to auctions sites. These teams face the formidable task of recommending the best possible solution for each daily load of cars. A small mistake, an inferior recommendation that results in a net loss of “only” $150 per car, may cost the company hundreds of thousands of dollars in a single day!

On the other hand, if an Adaptive Business Intelligence system was used to improve the daily distribution of cars to the tune of $200 per car (note here that this corresponds to an increase of only 1.33% in the price of an average car), then the increased annual profits might translate into hundreds of millions of dollars! Hence, to maximize the overall net profit, an Adaptive Business Intelligence system must *decide* where to ship each car. As discussed earlier, the problem is extremely complex, and the system would have to address the following issues:

- *Transportation.* When a whole truckload of cars is shipped from one place to another, the company is charged a cheaper rate. Therefore, it is important to take into account the number of cars that are transported on each truck.
- *Risk factors.* Cars can be damaged, stolen, or the transportation truck might be involved in an accident. Longer trips also increase the probability of a delay.
- *Volume effect.* If many cars of the same type are sent to the same auction site, then the volume effect will kick in and an oversupply of a particular car will decrease its price.

- *Size of the search space.* The distribution of 3,000 cars to 50 auction sites gives us 50^{3000} possible solutions, which is much larger than the estimated number of atoms in the Universe.
- *Price depreciation.* Every auction site has a typical sale day (e. g., every Wednesday at 10 am, or every second Thursday at 11 am). If some cars arrive one day after the sale date, then they will sit at the auction site for one or two weeks (until the next sale day), and the price depreciation is often around $10 per day, per average car.
- *Recent history.* When making a recommendation, all decisions made during the past few weeks must be taken into account. Many of those cars may still be in transit, and if they are going to the same auction site, then they might be sold on the same day.
- *Inventory.* It is important to monitor the inventory level of cars at all auction sites, as each site has a particular throughput. If an auction can handle 250 cars per sale day, and the current inventory is larger than 250, then additional time must be added to the estimated sale date.
- *Dynamic market conditions.* Market prices for cars change quite frequently, sometimes slowly and sometimes very quickly. Leasing companies (like most businesses) operate in a non-stationary environment that is influenced by many external factors, such as: (1) seasonality (e. g., it is not easy to sell convertibles in New York during the wintertime), (2) the arrival of new models (e. g., new models enter the marketplace in August, influencing the price of older models), and (3) weather (which influences the number of dealers present at an auction, which in turn influences the sale price).
- *Business rules.* It is essential to accommodate various business rules that can be added or dropped at any time (e. g., “do not send any red cars to South-East auctions”). This is important for analyzing what-if scenarios.

Let us conclude this chapter with an observation that we have already discussed (in general terms) in Chap. 1: namely, *that all the knowledge in the world will not guarantee the right or best decision.* Hence, even if we possessed “perfect knowledge” and were capable of accurately predicting the price of *any* car at *any* auction site for *any* day, we still would not know how to optimally distribute 3,000 cars on any given day because of all the issues mentioned above. The number of possible distributions is simply too large to be processed in any reasonable amount of time. Hence, even though it is true that data mining can extract useful knowledge from data, it is a myth that the extraction of such knowledge will make a business run better! We may have perfect knowledge of all cars, prices, and locations, and still our distribution decisions might be very bad!

4 Adaptive Business Intelligence

"By Jove, Mr. Holmes, I think you have hit it."
The Adventure of the Lion's Mane

" 'Elementary,' said he."
The Adventure of the Crooked Man

In Chap. 2, we discussed the characteristics of real-world business problems, which included:

- The number of possible solutions to the problem is so large that it prohibits any systematic (complete) search for the best solution.
- The problem exists in a time-changing environment, and therefore requires a set of solutions (rather than a single solution).
- The problem is so heavily constrained that finding even one feasible solution is often difficult, let alone searching for the optimum solution (or set of solutions).
- The problem includes many (possibly conflicting) objectives.

In the previous chapter, we gave an example of a real-world business problem in the car leasing industry. There is no question that the number of possible solutions to the car distribution problem is too large for any systematic search (we indicated that the distribution of 3,000 cars to 50 auction sites gives us 50^{3000} possible solutions). We may also want to impose many constraints. For example, we may wish to restrict the maximum transportation distance (e. g., "cars should not be shipped more than 1,000 miles from their original location"), exclude some types of cars from some auction sites (e. g., "red cars should not be sent to auction sites in Texas" or "cars with more than 100,000 miles should not be sent to auctions in the South-East"), etc. Note also, that in our example:

- The number of returned cars changes every day.
- The locations where the cars are returned changes every day.
- The prices at each auction site change over time.
- The number of available auction sites may change over time.
- The volume effect changes over time.
- Transportation costs change over time, etc.

Because all organizations operate in a time-changing environment, they are forced to constantly adapt and adjust. Consequently, an Adaptive Business Intelligence system must include three major components: (1) a component for making predictions (in our case, sale price predictions), (2) a component for making near-optimal decisions (in our case, the distribution of cars), and (3) a component for adapting the prediction module to changes in the environment. To create such a system, the following four steps should be followed:[13]

- The available data must be prepared and thoroughly analyzed (the data mining stage).
- A prediction model must be developed based on the data mining results.
- An optimization module must be developed that uses the prediction model to recommend the best solution.
- An adaptability module must be developed that is responsible for adapting the prediction module to the time-changing environment.

Let us discuss the details of these seemingly simple steps, as understanding them will enable us to realize the powerful benefits of Adaptive Business Intelligence!

4.1 Data Mining

Data typically reside in one or more databases, and we first have to understand the structure of each database and its tables. To illustrate this further, let us use a table from Chap. 3 that has basic data about cars and their sale prices:

[13] Before you can solve a problem, you have to understand it. In software development, this means gaining a complete understanding of the user requirements. Unfortunately, this process can be quite tricky, because business managers understand business processes while software developers understand software development. To bridge this gap, software developers should talk directly with business managers to understand the relevant business process and user requirements. With a complete understanding of the problem, software developers can then create a user requirements document that is verified by the business managers. Because software development is covered many specialized texts, we will instead focus on the conceptual issues involved in developing an Adaptive Business Intelligence system.

VIN	ZIP	Date	Price
39WWK93309KJ33012	28262	2.11.2004	$12,035
UDJ2293M99DL0K220	30334	2.11.2004	$15,600
4D09WJD92JE93H990	30334	2.11.2004	$10,590
KD37D92JF83NF8822	90012	3.11.2004	$9,265
NKI2389DD974F2235	28262	3.11.2004	$13,450
K29DH38FHW02HD923	48243	3.11.2004	$13,955
MDK293HFDWH299305	90012	4.11.2004	$12,495
28DN39FNDJW2N0024	90012	4.11.2004	$11,925
29H93NFI3HJF93F04	48243	4.11.2004	$11,396
ND920ENF1NAD02834	48243	5.11.2004	$9,835
D39DJ39EHQ8HH9335	28262	5.11.2004	$8,965
02UFIMF03JF9SH935	90012	5.11.2004	$13,960
D932NF93HG9057362	48243	5.11.2004	$8,830
00F8EB3IDNB293758	48243	8.11.2004	$7,920
IE038THJ203TH0234	28262	8.11.2004	$19,250
39FH324MV092HGM39	48243	8.11.2004	$22,640
F92N9F389FH120458	90012	8.11.2004	$13,580
F9485JG03H25495J5	30334	9.11.2004	$16,970
08GN94HJH03J49327	30334	9.11.2004	$14,320
F04JH402KG4509G45	48243	9.11.2004	$9,110

And another table that has more detailed data about each car:

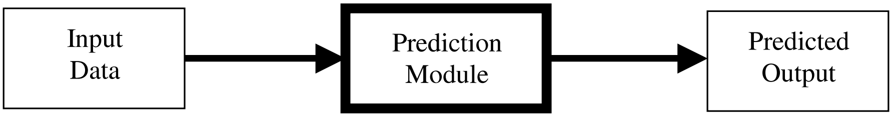
VIN	Type	Make	Model	Miles	Year	Color	Transmission	Body/Doors	Damage
2G1FP22P1P2100001	Rental	Chevy	S-10	34,983	2002	Silver	Manual	2D	$0
WB3PF43X8X9000331	Lease	Chevy	Cavalier	59,402	2001	Red	Automatic	2D Coupe	$0
4BBG38FJF04JDK000	Lease	Chrysler	Sebring	74,039	2000	Gray	Automatic	2D Coupe	$500
DJOW03FFU990SJ206	Lease	Ford	Escape	37,984	2001	Green	Manual	4D Sport	$250
JD8320DJ2094GK2X3	Rental	Ford	Focus	30,842	2001	Green	Manual	4D Sedan	$0
2JE9F0284JD0213M3	Lease	Isuzu	Rodeo	59,044	1999	White	Automatic	4D Sport	$250
4380JDDD9W02MD001	Rental	Jeep	Cherokee	48,954	2000	Black	Automatic	4D Sport	$500
490DK20285JF0209D	Rental	Mazda	626	38,943	2000	White	Automatic	4D Sedan	$0
10D92JD920KD00002	Lease	Nissan	Altima	39,488	2000	Black	Automatic	4D Sedan	$0
D920DKJ0284JJ9990	Rental	Nissan	Altima	23,584	1999	White	Manual	4D Sedan	$0
JD88D92JJD02K3361	Rental	Saturn	L	21,048	2001	White	Automatic	4D Sedan	$750
10DS0JJ20DXI00093	Lease	Suzuki	Vitara	15,849	2003	Yellow	Automatic	2D Sport	$0
21KD02KD0DJ920M27	Lease	BMW	Z3	49,858	2000	Blue	Manual	2.3 RSTR	$250
389DJ2DD298JWQ082	Lease	Ford	Explorer	42,893	2002	Green	Automatic	XLT 4WD	$0
108DJ2048FJJ20043	Rental	Ford	Mustang	20,384	2002	Red	Manual	GT	$0
DJC82002009DD2J04	Rental	Mercury	Frontier	27,849	2001	Silver	Automatic	SE-V6 Crew	$500
830DMM3029XMW0092	Lease	Honda	Accord	26,849	2002	Yellow	Automatic	EX V6	$0
CNEU200220CCI2202	Rental	Toyota	4Runner	33,483	2000	Silver	Automatic	SR5	$0
CNDJ2940JD88D2JD0	Lease	VW	Beetle	5,459	2003	Blue	Manual	GLS 1.8T	$0
1VC0CMEJ200V9EJJ1	Lease	Toyota	4Runner	81,837	2001	Silver	Automatic	LMTD 4WD	$250

Since we have explained what each column means in the previous chapter,[14] we should already have a good understanding of the data. Next, we have to clean the data. "Clean the data?" Yes, data can be "dirty"! This terminology refers to situations that involve missing data, inconsistent data, incorrect data, etc. Manually entered data usually has to be cleaned since humans make typing errors and often use different words to describe the same thing (e. g., "Chevrolet" and "Chevy"). Cleaning the data means identifying what data might be missing, what data might have been manually entered, what data might be inconsistent or incorrect, and so on (we will discuss the process of preparing and cleaning data in more detail in the next chapter).

In this small sample, there are no missing values or irregularities, so we do not have to worry about dirty data. We can now proceed with the *data mining* activity, which is an analytical process for finding relationships and patterns among variables (e. g., finding the relationship between "sale price," "make," "model," and "mileage"). Although *knowledge discovery* (e. g., text mining, discovery of association rules) is oftentimes an important goal of data mining, we are more interested in using the data mining results to build a model (or set of models) for predicting some response (e. g., the outcome of a loan application, the probability that a credit card transaction is fraudulent). The reason for this is that predictions are directly applicable to decision making, whereas knowledge discovery is closer to decision support. In the car distribution example, we are interested in predicting the sale price for particular off-lease cars at particular auction sites at particular dates in the future. Therefore, we should focus our data mining efforts on finding the relationships and patterns that exist among the various variables in the data set, and then use this knowledge to develop an effective model for predicting the sale price.

4.2 Prediction

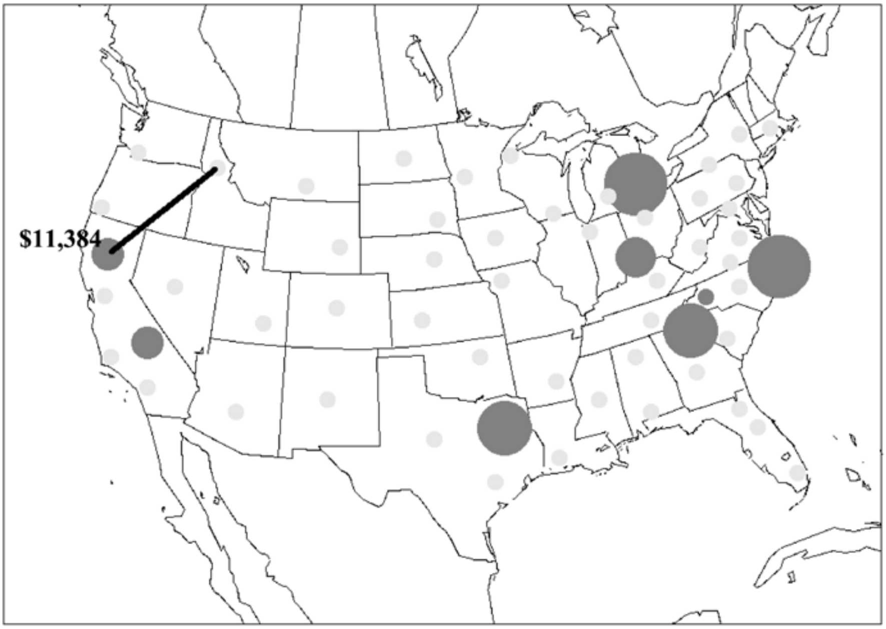
The basic function of a prediction module (which can made up of one or more prediction models) is to produce an output based on some input:



To make the prediction module functional, it is necessary to "train" the various underlying models using historical data. During this process, the prediction module "learns" how to predict the output given the input (we explain this process in detail in Chap. 5). As an example, imagine predicting what a particular Honda Accord (with known features and known mileage) would sell for at a particular auction site in Northern California at a particular point in time (e. g., in two weeks). In this case, the input would be the auction site in Northern California, the

[14] In many data mining activities, understanding what the columns mean and how they are related is not an easy task!

car Honda Accord (with all specified features), and the estimated sale date (say, May 21st, as today is May 7th). The output, on the other hand, is the predicted sale price for this car. In the illustration below, the prediction module predicts a sale price of $11,384:



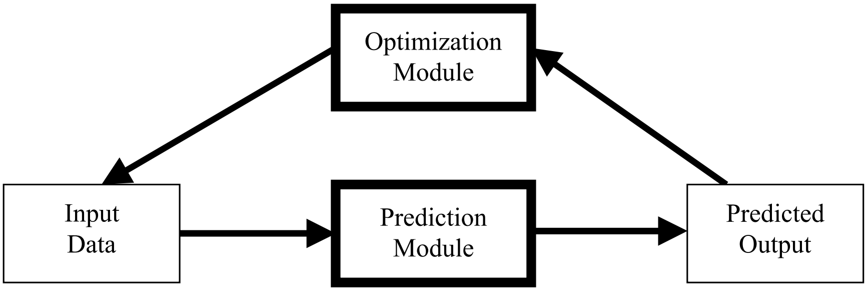
But how does a prediction module “know” what the sale price of a particular Honda Accord will be on a particular auction site at a particular point in time? Well, the prediction module would have to “learn” to predict the sale price based on the historical sales data for all Honda Accords at the auction site in Northern California. For the sake of simplicity, let us assume we have the following Honda Accord sales data from last week’s auction in Northern California:

Make / Model	ZIP	Date	Price
KD37D92JF83NF8822	94102	8.11.2004	$9,265
MDK293HFDWH299305	94102	8.11.2004	$12,495
28DN39FNDJW2N0024	94102	8.11.2004	$11,925
29H93NFI3HJF93F04	94102	8.11.2004	$11,396
ND920ENF1NAD02834	94102	8.11.2004	$9,835
D39DJ39EHQ8HH9335	94102	8.11.2004	$8,965
02UFIMF03JF9SH935	94102	8.11.2004	$13,960
D932NF93HG9057362	94102	8.11.2004	$8,830
00F8EB3IDNB293758	94102	8.11.2004	$7,920
IE038THJ203TH0234	94102	8.11.2004	$19,250

Let us also assume that our (very basic) prediction module "predicts" the sale price for each make/model at each auction site by looking at the average sale prices for the previous week (hence, it does not take into account the actual mileage, color, or other variables). It is interesting to note, however, that creating even a basic prediction module such as this presents many difficulties. For instance, a separate prediction model is needed for each make, model, and location. Hence, the prediction module might need to contain around 10,000 separate prediction models (20 makes × 10 models × 50 locations). Furthermore, if our database contains approximately three million cases (data collection spanning 30 months), then each prediction model will have an average of 300 cases. The distribution of these cases will be non-uniform. For example, we might have 2,800 cases per location for some makes/models (e. g., Toyota Camry), but only one or two cases per location for some other makes/models (e. g., Porsche 911)! Consequently, it would be impossible to create a prediction model for some makes/models. Regardless of these complications, let us assume for the moment that our basic prediction module makes a prediction of $11,384 (for the Honda Accord being sent to an auction site in Northern California). Of course, this "prediction" would not be very accurate, but it serves as a good starting point for further discussion in Sect. 4.4 below.

4.3 Optimization

Next comes the development of an optimization module capable of recommending the best answer. Note that the "best answer" is based on the prediction module's output. For example, to evaluate the merit of a particular distribution of cars, we need to predict their sale prices. The relationship between the prediction and optimization modules can be displayed as follows:

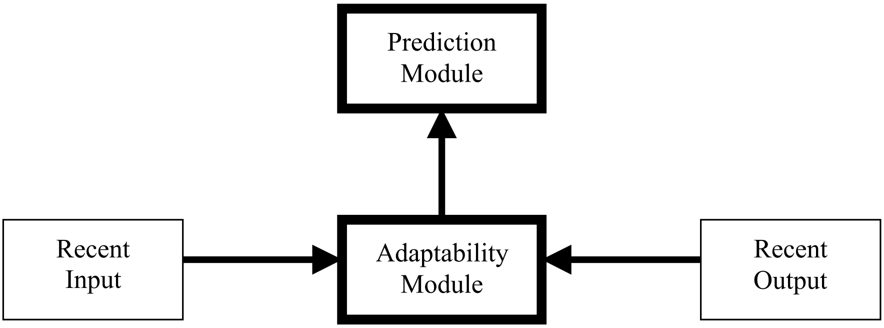


In the above diagram, the optimization module generates a distribution solution that serves as *input data* for the prediction module. This input data provides a destination assignment (i. e., auction site) for each car, which the prediction module uses to generate the predicted sale prices. The optimization module then uses the sum of all the predicted sale prices (i. e., the *output data*) to gauge the quality of the input data: The higher the sum of the predicted car sale prices, the better the input data. To maximize the sum of all the predicted sale prices, the optimization module tries many different input data combinations and then evaluates the output data (of course, the optimization module must modify the output data for transportation costs and other adjustments). Many different techniques can used to construct an efficient optimization module, which we will discuss in Chap. 6.

4.4 Adaptability

Developing effective prediction and optimization modules is a great start, but by themselves these modules are insufficient for today's ever-changing environments. Because today's accurate prediction might be inaccurate tomorrow, the prediction module must be capable of "learning from" and "adapting to" changes in the environment. The concept of adaptability has far-reaching consequences: Imagine a system that would improve over time by learning from its own prediction errors. Such a system would truly be adaptive!

The adaptability process can be illustrated as follows:



To detect errors between the predicted result and the actual result, an adaptability module compares the predicted sale prices (i. e., recent input) with the actual prices for each car (i. e., recent output). If errors exist, the adaptability module will "tune" the prediction module to decrease the prediction error.

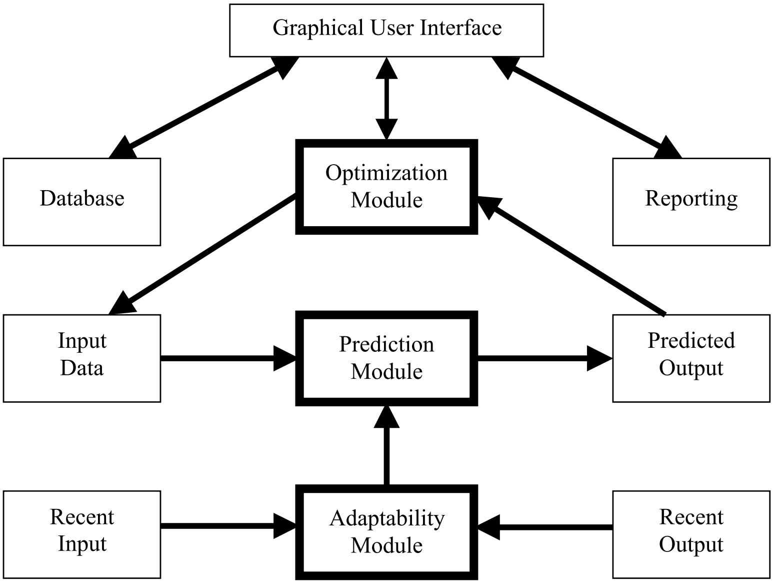
As an example of this process, let us return to the basic prediction module discussed in Sect. 4.2, which "learns" the average sale price for each make/model at each auction site by looking at the sale prices for the previous week. If the adaptability module updates the prediction module every week by using a rolling time window, then the prediction module can adapt to changes in the sale prices. Alternatively, imagine that the prediction module has certain rules that can be expressed as follows:

if [Make = Honda] **and** [Model = Accord] **and** [Color = white] **and** [40,000 < Mileage < 50,000] **and** [Year = 2000] **and** [Damage Level = $0], **then** Sale Price = $11,384.

Each of these rules has a weight,[15] and the weights of rules can be modified (say, on a weekly basis) to tune the predictions in a changing environment. In other words, the adaptability module can "adapt" to environmental changes by updating these rules, with the update frequency depending on how fast the environment changes. Of course, a *really* good adaptability module can make its own decision on the update frequency by continuously measuring its own prediction errors. Hence, a *really* good adaptability module can adapt its own speed of adaptation! It is important to note, however, that the adaptability module's effectiveness in updating the prediction module is influenced by the type of prediction methods that were used to build the underlying models. In the following chapter, we will discuss some of these methods in detail.

4.5 The Structure of an Adaptive Business Intelligence System

The prediction, optimization, and adaptability modules are the core components of an Adaptive Business Intelligence system. However, this does not mean that other components are not important (e. g., an easy-to-use graphical user interface, a database for storing information). Thus, the overall structure of an Adaptive Business Intelligence system resembles the following diagram:



[15] The "weight" of a rule is just a parameter that provides a relative measure of the rule's merit. Rules that are more important are assigned higher values (recall the definition for *parameter* in Sect. 3.1).

In Sect. 2.6, we discussed how prediction, optimization, and adaptability were used in the “pollution control” research project in Poland. The prediction module was responsible for predicting the ecological damage that each 30 km × 30 km square would sustain during the next 48 hours; the optimization module (consisting of an evolutionary algorithm) was responsible for recommending the ideal energy production level for each power station; and the adaptability module was responsible for tuning the system’s performance. Furthermore, there were a few reporting and visualization modules incorporated into the system, which interfaced with the databases. In Part III of this book, we will discuss a modern Adaptive Business Intelligence system with all of these components.

Part II:
Prediction and Optimization

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 is it 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:

Mileage / (Current Year – 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 covert 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:

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

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:

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

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:

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

which simply means that the prediction for the next (future) case is calculated as a total of two values: the actual last case (*Actual(t)*) with parameter *a* and the last prediction (*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:

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

so *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:

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

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.

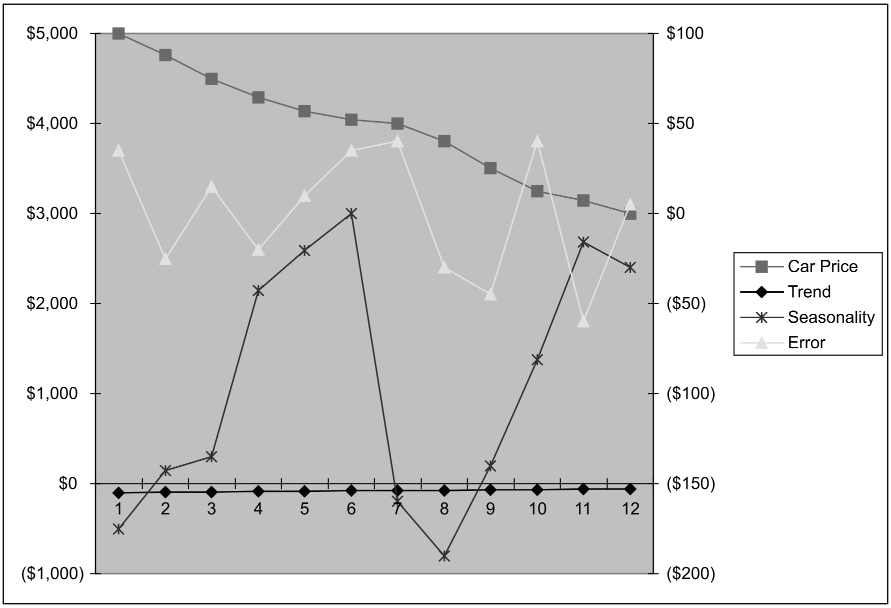
Although there are several approaches to decomposing a time series problem into separate components, the basic concept is based on experience: First the trend-cycle is removed, then the seasonal components are addressed. Any remaining error is attributed to randomness; thus:

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

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*:

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

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 revaluated and the parameter values readjusted (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 distance[6] 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).

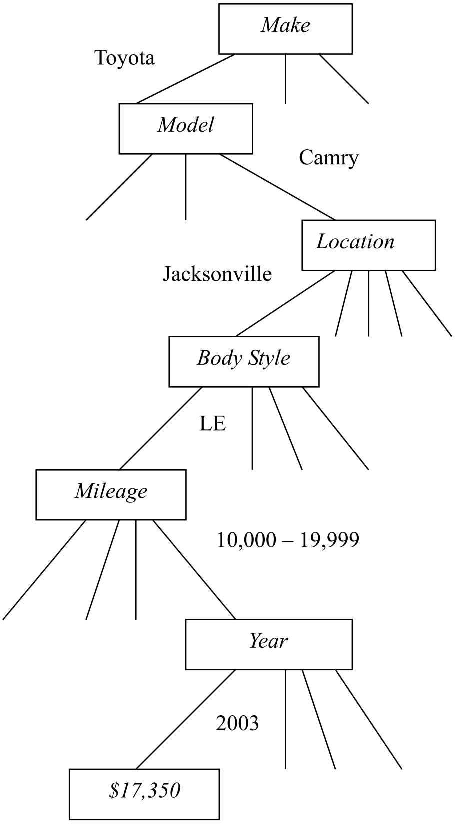
The most widely used logic method, on the other hand, is the *decision tree*. Because the structure of a decision tree is relatively easy to follow and understand (especially for smaller trees), its popularity is widespread. To make a prediction for a new case, the root of a tree is examined, a test 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).

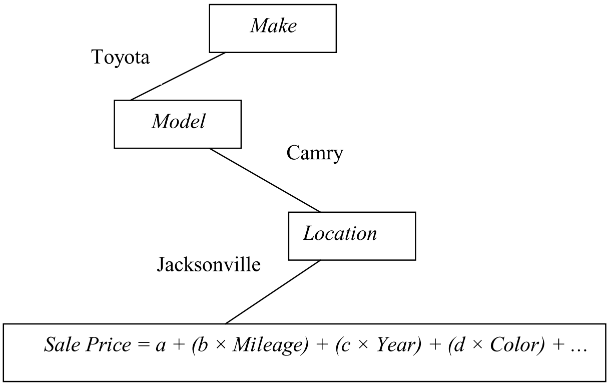
We can easily picture a decision tree for used car prices. At the root of the tree a decision is made on "make": if there are 30 different makes, then the root node would have 30 branches. On the second level of the tree, a decision is made on "model," then the third level provides branches for "location." 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:


Make
Toyota
Model
Camry
Location
Jacksonville
Sale Price = a + (b × Mileage) + (c × Year) + (d × Color) + …

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]

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

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:

$$\begin{aligned}\text{Sale Price} = {} & \text{a} + (\text{b} \times \text{Model}) + (\text{c} \times \text{Location}) + (\text{d} \times \text{Mileage}) \\ & + (\text{e} \times \text{Year}) + (\text{f} \times \text{Color}) + \ldots\end{aligned}$$

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

$$\begin{aligned}\text{Sale Price} = {} & \text{a} + (\text{b} \times \text{Location}) + (\text{c} \times \text{Mileage}) + (\text{d} \times \text{Year}) \\ & + (\text{e} \times \text{Color}) + \ldots\end{aligned}$$

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:

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

[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 ≤ Mileage ≤ 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 ≤ Mileage ≤ 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 rules[11] 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 ≤ 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 ≤ Mileage ≤ 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 applied to a particular problem. To answer this question, it is necessary to evaluate and compare different models. Because the comparisons have to be unbiased, the evaluation methodology should be fair and just. At first blush, this may seem easy. After all, after we complete and train a few models, we can test them on the data and measure the prediction error. The best model would then be selected for implementation.

Unfortunately, it is not that simple.

First of all, the amount of available data might not be that large. Even in the car distribution example, if we take into account all the different makes and models sold at all the different auction sites, our "large" data set of three million cases is actually quite small. With 20 makes that each have 10 models, sold at 50 different locations, we are talking about 10,000 variables. Our data set of three million cases suddenly turns into just a few hundred cases for each make/model at each location. Also, because some makes/models are common while others are rare, the rare makes/models will only have a few cases per location sold at irregular intervals during the past three years.

Secondly, the performance of a prediction model on the training data might be very different from the performance of the same model on an independent set of data. This might be due to overfitting, which is a common phenomenon. In short, a model tunes itself during the training stage to such an extent that all predictions on the training data set are perfect! However, the point is to develop a prediction model that performs well on *new* data! Overfitting is a serious problem, as most prediction methods are capable of generating 100% accurate predictions on the training data set.

Thirdly, prediction models that provide different outcomes require different techniques for error measurement. For instance, a prediction model may indicate whether a new case belongs to class A or B, or, if we have a larger number of classes, the probability that a new case belongs to each class. Alternately, a prediction model may predict a number (e. g., sale price) or a sequence of numbers (e. g., sale price *and* sale date). In each of these examples, we have to carefully consider what we are predicting (i. e., what the outcome of the prediction model is), and apply the appropriate error measurement technique.

Finally, we have to take into account the cost of a potential error. When classifying cases into two categories ("yes" or "no," "fraudulent" or "legitimate," etc.), there are two types of errors: (a) *false-positive*, where the outcome is incorrectly predicted as "yes," when in fact it is "no," and (b) *false-negative*, where the outcome is incorrectly predicted as "no," when in fact it is "yes" (this issue is discussed in more detail in Sect. 12.5). Clearly, the cost of these errors is very different. By classifying a legitimate transaction as fraudulent (false-positive), there is a small cost to check the transaction. On the other hand, classifying a fraudulent transaction as legitimate (false-negative) usually carries a much higher cost, especially if the transaction is significant (e. g., approving a fraudulent transaction for $5,000).[13]

Because different models may generate a different number of false-positives and false-negatives on test data, the costs of these two types of errors must be taken into

[13] A more detailed example of fraud detection is provided in Sect. 12.5.

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 disjoined 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 al 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 ≈ 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 Indurkhya (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.

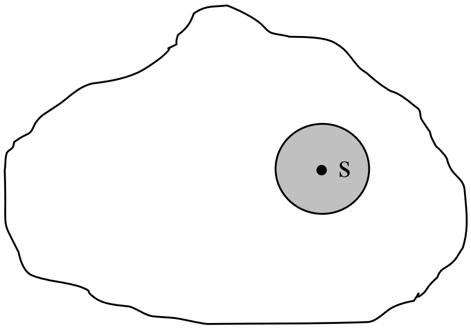
Because the evaluation function is not provided with a problem, how should we go about choosing the correct evaluation function? Oftentimes, the objective can suggest a particular evaluation function. In the traveling salesman problem, for instance, we considered using distance as the evaluation function. 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 technique[18] 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:

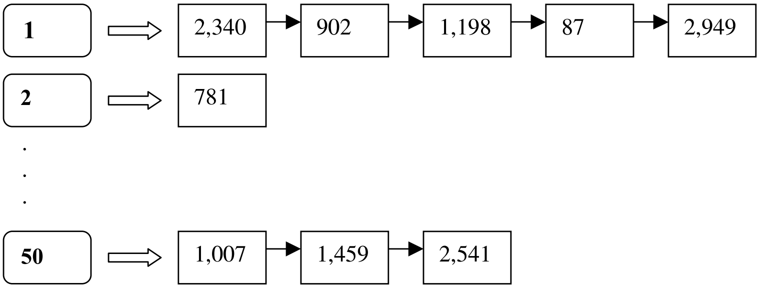
23	41	5	…	19	41

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 × 50 × 50 × … × 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:

1	3	1	...	1	2

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:

1	1	1	...	1	1

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]

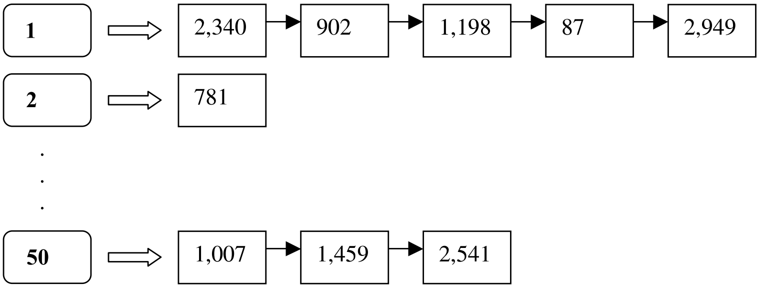
Of course, there are many other possibilities.[22] For example, if an auction site were allowed to differ by five (rather than just one), then the neighborhood would be much larger. Each auction site would define 10 possible neighbors (e. g., auction 23 could be replaced by any of the following auctions: 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 × 3 × 3 × … × 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.

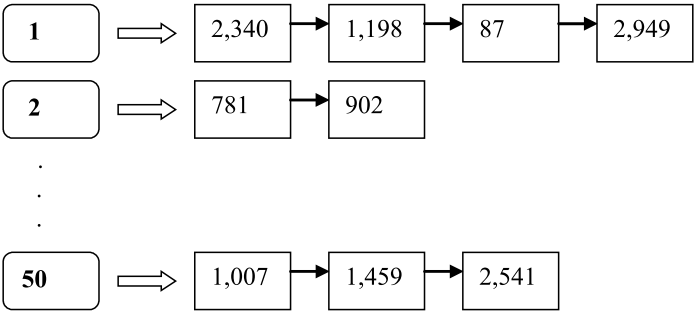
[21] For some vectors, however, the neighborhood size is slightly less than 6,000 (e.g., auction sites 1 and 50 can only be replaced by 2 and 49 respectively).

[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 × 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 × 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:

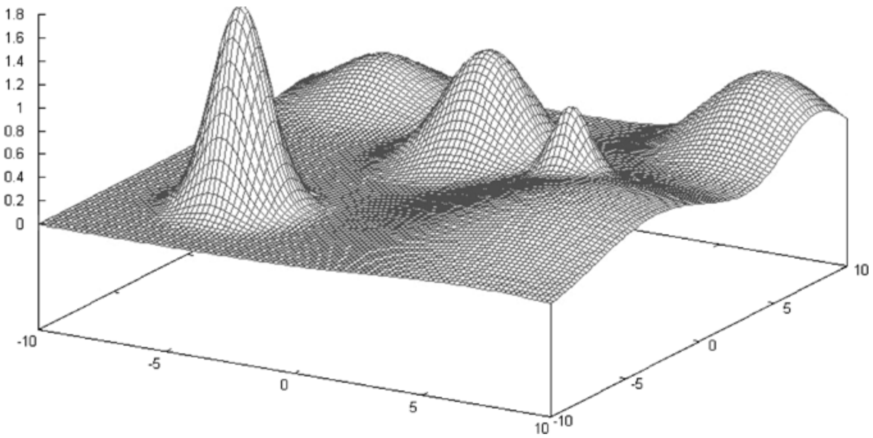
23	41	5	...	19	41

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

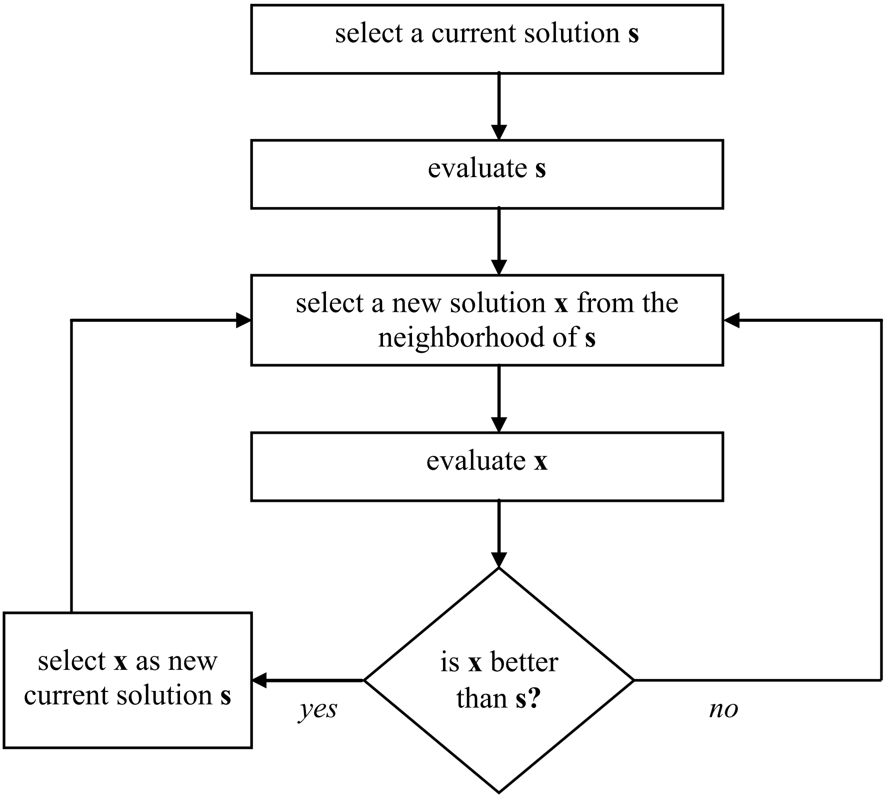
The evaluation function defines a *quality measure score landscape* (also known as a *response surface* or *fitness landscape*) that is much like a topography of hills and valleys. Within this three-dimensional landscape, the problem of finding a solution with the highest quality measure score is similar 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 techniqu'es, 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 techniques 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:


select a current solution **s**
evaluate **s**
select a new solution **x** from the neighborhood of **s**
evaluate **x**
is **x** better than **s?**
yes
no
select **x** as new current solution **s**

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 …

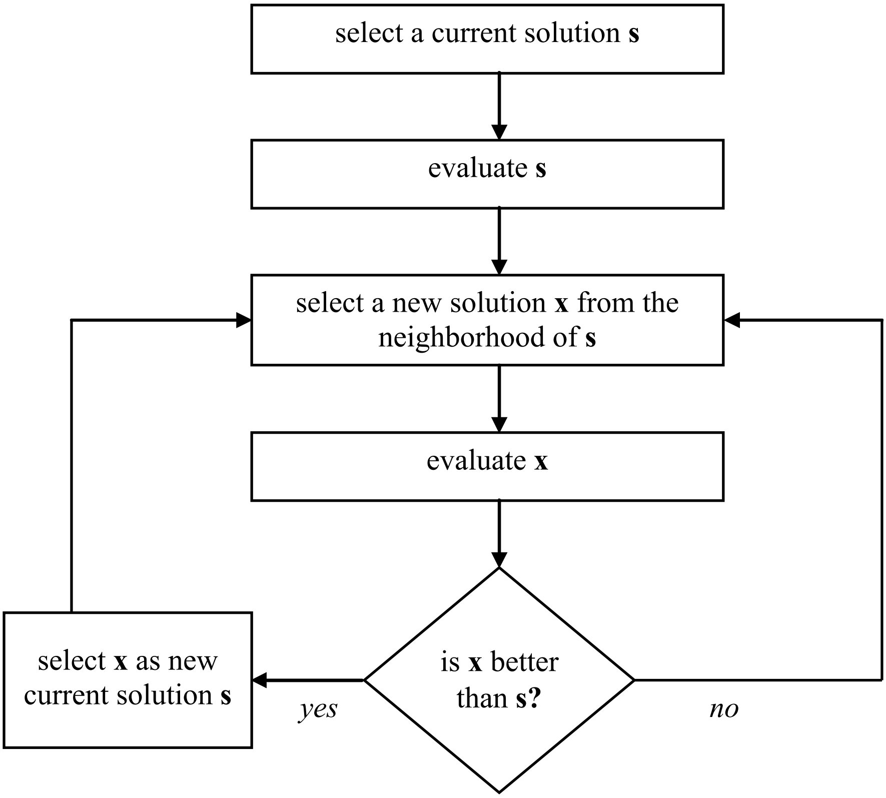
So, what should we do? We can restart out hill-climbing algorithm several times, each time from a different (possibly random) location, and hope that one of these runs will provide us with the global optimum solution (which may or may not happen).

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.

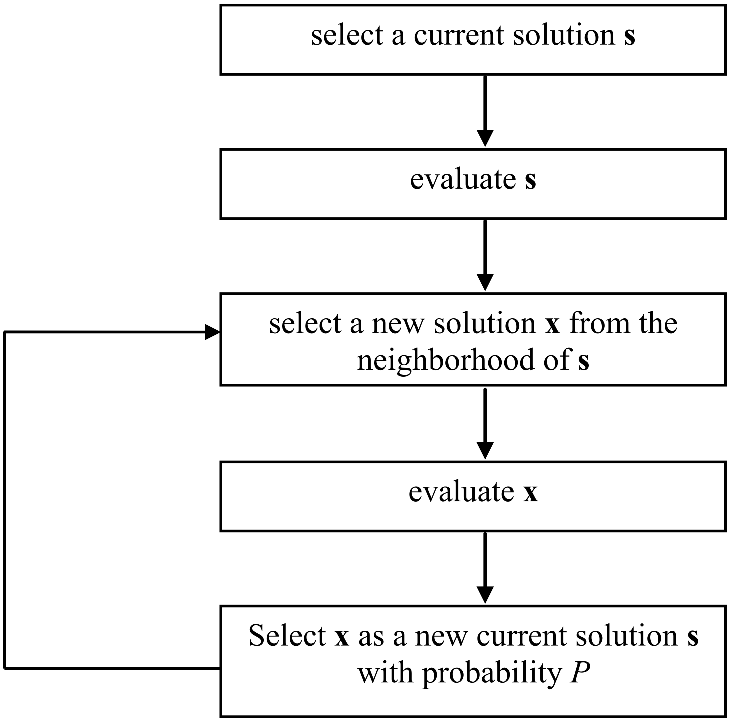
So what can we do about it? How can we design an optimization technique that has a chance to escape local optima, to balance exploration and exploitation, and to make the search independent from the initial configuration? There are a few possibilities, and we will discuss some of them in this chapter, 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:


select a current solution **s**
evaluate **s**
select a new solution **x** from the neighborhood of **s**
evaluate **x**
is **x** better than **s?**
yes
no
select **x** as new current solution **s**

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*:


select a current solution **s**
evaluate **s**
select a new solution **x** from the neighborhood of **s**
evaluate **x**
Select **x** as a new current solution **s** with probability *P*

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

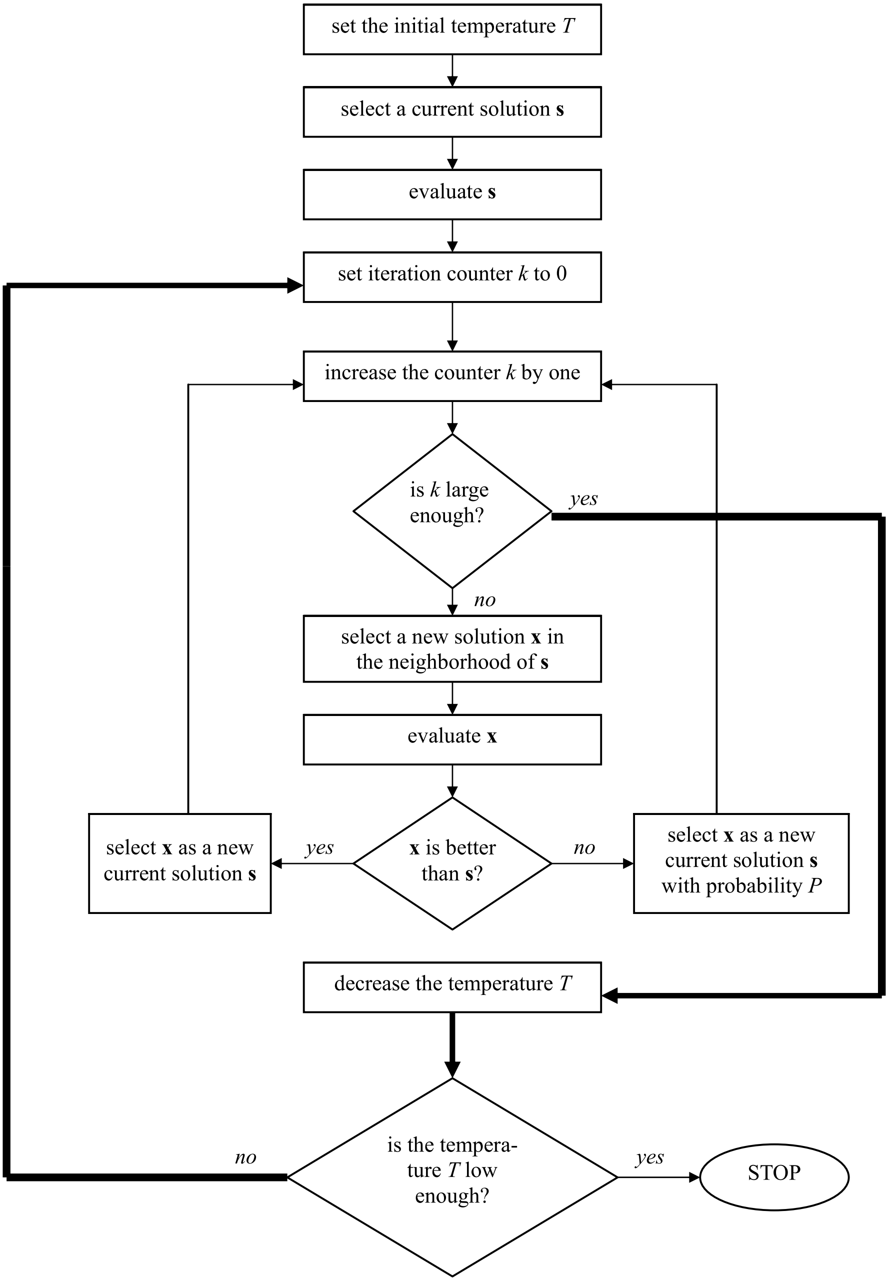
The simulated annealing technique (also known as Monte Carlo annealing, statistical cooling, probabilistic hill-climbing, stochastic relaxation, and the probabilistic exchange algorithm) is based on an analogy taken from thermodynamics. To grow a crystal, we begin by turning the raw material into a molten 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.


set the initial temperature *T*
select a current solution **s**
evaluate **s**
set iteration counter *k* to 0
increase the counter *k* by one
is *k* large enough?
yes
no
select a new solution **x** in the neighborhood of **s**
evaluate **x**
x is better than **s**?
yes
no
select **x** as a new current solution **s**
select **x** as a new current solution **s** with probability *P*
decrease the temperature *T*
is the temperature *T* low enough?
no
yes
STOP

The internal loop (represented by the thin line) generates a new solution in the neighborhood of the current solution, accepts superior solutions, accepts or declines inferior solutions (according to probability *P*, which depends on the quality measure score difference between two solutions and the current value of parameter *T*), and repeats this process some (relatively large) number of times. When this iterative cycle is complete, the system drops the temperature a bit (outer loop, represented by the thick line) and then switches back to the internal loop, repeating the process of creating, evaluating, and possibly accepting the neighboring solutions. If the temperature is low enough (i. e., when it reaches the freezing point), the algorithm stops.

As this flowchart illustrates, simulated annealing applies a modified version of the stochastic hill climber where the value of parameter *T* is gradually decreased during a run. As mentioned earlier, the behavior of the algorithm resembles a random search at higher temperatures (i. e., at the beginning of the run) and a classic hill climber at lower temperatures (i. e., toward the end of the run). We can parody this process with a drunken explorer searching for the highest peak on a quality measure score landscape. Initially, the explorer does not care whether he goes up or down, and this directional indifference corresponds to the early stages of simulated annealing when the temperature is high and the probability of accepting or rejecting an inferior solution is close to 50%. However, the explorer would sober up over time, and this corresponds to dropping the temperature and making more "reasonable" decisions on where to go next. Toward the end of the search, the explorer is completely sober and always walking uphill ...

Let us take a closer look at simulated annealing by applying it to the car distribution example. Any implementation of simulated annealing would require us to answer some general questions that are necessary for any optimization technique (i. e., *What is the representation? How are neighbors defined? What is the evaluation function?* And so on), and we will assume that the answers to these questions are the same as for the hill climber example discussed in Sect. 6.1.[25] There are, however, some questions that are specific to simulated annealing, such as:

- How to determine the initial value of parameter *T*?
- How to determine the number of iterations (i. e., how to define the "is *k* large enough?" statement in the flowchart)?
- How to "cool" the system (i. e., how to define the "decrease the temperature *T*" statement in the flowchart)?
- When to stop (i. e., how to define the "is the temperature *T* low enough?" statement in the flowchart)?

The first question deals with the temperature parameter, which must be set before starting the algorithm. Should we start with $T = 100$, $T = 500$, or something else? Well, let us think about that. At the beginning of the simulated annealing

[25] The same representation is used (i.e., where a vector of 3,000 values provides indices of auction sites from 1 to 60). The definition of the neighborhood remains the same: a new solution **x** is a neighbor of current solution **s** if it differs (at most) by 1 on any position. The initial solution might be generated randomly (i.e., for each entry, a random number from the range of 1 to 60 is produced), or we can use some heuristic-based solution (e.g., an initial solution that assigns each car to the nearest auction site). The evaluation function also remains the same.

run, we would like to make almost random decisions for accepting or rejecting inferior solutions (remember that superior decisions are always accepted). That means that the answer to “$T=$?” depends on the “average difference” in quality measure scores between two randomly generated solutions from the same neighborhood. This is important, as the probability of accepting a new solution is based on this difference and the value of the parameter T. The initial temperature should be high enough to generate probabilities of acceptance close to 100% for inferior (new) solutions that have an average difference in the quality measure score. In our case, say we generated 1,000 random pairs of solutions plus their neighbors, evaluated them, and then discovered that the average difference in the quality measure score is 5.05. Based on this information, we should start with an initial temperature of $T=100$, because at this temperature the probability of accepting an inferior solution (worse than the current solution by 5.05) is over 95% (whereas for temperature $T=50$, this probability would only be around 90%).[26]

Note that the next two questions (“‘How to determine the number of iterations?” and “How to cool the system?”) are really about the number of temperature levels and the number of iterations performed at each level. Assuming limited computing resources (which is usually the case), these questions represent a typical trade-off present in most implementations of simulated annealing: Is it better to make more temperature levels or to search each level more thoroughly? Unfortunately, there are no easy answers here as these issues are problem dependent. In the car distribution example, we can decide on 10,000 attempts per temperature level, with a cooling scheme that multiplies the current value of parameter T by 0.95 every time the temperature drops …

The final question (“When to stop?”) can be answered as follows: Think about a freezing temperature for which it would be almost impossible to accept any inferior solution. For instance, if the temperature is $T=0.001$, then a new solution that is inferior to the current solution by just 0.01 would have a probability of less than 0.005% of being accepted. Using these numbers, there would be 225 different temperature levels: $T=100$ at the first iteration, $T=95$ at the second (100 × 0.95), $T=90.25$ at the third (95 × 0.95), and so forth, until iteration 225, where the temperature value drops for the last time to 0.001. This happens for $T=0.001023$; the next drop, from $T=0.001023$ to $T=0.000972$ (0.001023 × 0.95), would trigger the termination condition. In total, the simulated annealing algorithm would generate and evaluate 2,250,000 solutions, as 10,000 solutions are generated and evaluated at each temperature level.

Implementing the rest of our simulated annealing algorithm is straightforward. Note that the algorithm would start with some initial solution **s**, such as:

23	41	5	…	19	41

[26] These numbers are derived from standard functions used in simulated annealing for calculating such probabilities.

The solution is evaluated and draws a quality measure score of 171.49, and then a new solution **x** is generated from its neighborhood:

24	41	6	…	19	40

If the new solution **x** is better than solution **s**, then it is selected as the new current solution. If solution **x** is worse, it still might be selected: everything depends on the difference in the quality measure score between the current solution **s** and the new solution **x,** as well as the current value of parameter T. Since the temperature is relatively high at the beginning of the run, the probability of accepting an inferior solution would be also high. For instance, if solution **x** generated a quality measure score of 166.44, then the probability of acceptance would be around 95% (recall that parameter $T=100$ at this stage, and solution **x** is worse than solution **s** by 5.05, which is the exact "average difference" in quality measure scores computed earlier in this section).

This process continues for 10,000 iterations, with the algorithm generating, evaluating, and accepting or rejecting a neighboring solution at each iteration. Because the temperature stays very high ($T=100$) during these 10,000 iterations, the probability of accepting inferior solutions is also very high. Thus, this phase of the search resembles a random search.

Assume that after 10,000 iterations the current solution is:

33	42	12	…	43	21

which evaluates to 176.78. Then we drop the temperature a bit (from 100 to 95) and repeat the sequence of 10,000 iterations. However, we start with the current solution (above) and a temperature of $T=95$. Although the probability of accepting an inferior solution is not as high as before (because the value of parameter T is lower), it is still relatively high. Assume that 10,000 iterations later we arrive at:

28	43	31	…	12	24

which evaluates to 184.95. Then we repeat the process again: we drop the temperature (this time from 95 to 90.25) and continue for another 10,000 iterations. The probability of accepting an inferior solution is lower than at the previous temperature level, but still relatively high.

After a while, we arrive at the final temperature level, where the value of parameter $T=0.001023$. Accepting an inferior solution is very unlikely at this low temperature, and the algorithm (for its final 10,000 iterations) acts like a classic hill climber. It would not be surprising to get a very good solution, say:

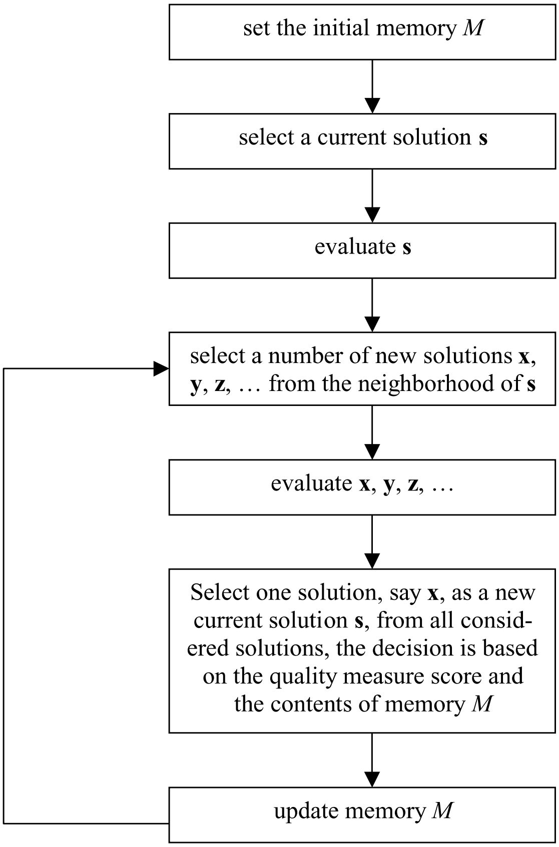
43	41	18	…	17	22

which evaluates to 1,444.87. Clearly, during the 2,250,000 total iterations at different temperature levels, the algorithm climbed and escaped from many "local" hills. Finally, during the hill-climbing stage at the end of the run, it made the final climb hoping to arrive at the global optimum!

6.5 Tabu Search

The main idea behind *tabu search* is very simple: "memory" forces the search to explore new areas of the search space to escape from local optima. We can memorize some recently examined solutions and these become "tabu" (forbidden) when selecting the next solution. Note that tabu search is deterministic (as opposed to simulated annealing, which is probabilistic), but it is possible to add some probabilistic elements to it.

The following flowchart outlines the basic steps of tabu search (without a termination condition):


set the initial memory M
select a current solution $\mathbf{s}$
evaluate $\mathbf{s}$
select a number of new solutions $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, … from the neighborhood of $\mathbf{s}$
evaluate $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, …
Select one solution, say $\mathbf{x}$, as a new current solution $\mathbf{s}$, from all considered solutions, the decision is based on the quality measure score and the contents of memory M
update memory M

So, what is so special about tabu search? Well, there are a few interesting features that require a more detailed description. Most of the boxes in the above flowchart are self-explanatory, and after discussing hill climbers and simulated annealing, we now know how to define a neighbor and select and evaluate a solution (or several solutions) from a neighborhood. What remains unclear is the "memory" component of tabu search. The best way to explain this concept is by returning to the car distribution example, using the same representation, neighbor definition, initial solution, and evaluation function as before.[27]

Let us begin by tracing some steps of the tabu search algorithm, starting with our usual initial solution **s:**

23	41	5	…	19	41

From the neighborhood of solution **s**, we generate several solutions (solutions **x**, **y**, **z**, etc.) and select the best one [28] (say it was solution **x**):

24	41	6	…	19	41

Now let us introduce the special feature of tabu search: *memory*. In order to differentiate between older and more recent changes in the solution vector, we need to remember the index of variables that were changed, as well as the "time" when these changes were made. In the case of the car distribution problem, we need to keep a time stamp for each position in the solution vector that provides information on the recency of the change. Because memory is the key feature of tabu search, let us illustrate this concept by continuing our example.

Vector M (of length 3,000) will serve as our memory. This vector is initially set to all 0s, and at any stage of the search if the i-th position of this vector has a value j, it means that j is the number of the most recent iteration when the i-th position of the solution vector was changed.[29] Hence, our memory vector (after selecting solution **x** at the first iteration, which replaces **s** as a current solution) is:

[27] The same representation is used (i.e., where a vector of 3,000 values provides indices of auction sites from 1 to 60). The definition of the neighborhood remains the same: a new solution **x** is a neighbor of current solution **s** if it differs (at most) by 1 on any position. The initial solution might be generated randomly (i.e., for each entry, a random number from the range of 1 to 60 is produced), or we can use some heuristic-based solution (e.g., an initial solution that assigns each car to the nearest auction site). The evaluation function also remains the same.

[28] Note that current solution **s** and new solution **x** differ in several positions. Later, choosing the "best" solution is influenced by memory (this is discussed later in this section).

[29] Of course, $j = 0$ implies that the i-th position of the solution vector has never been changed. Only for $j > 0$ does the value of j indicate the iteration number.

1	0	1	…	0	0

Note that the entries in the solution vector that did not change have corresponding values of 0 in this memory vector. Continuing our example, assume that a new solution vector selected in the second iteration is:

24	42	6	…	19	40

In that case, the memory vector would have the following values:

1	2	1	…	0	2

as the last change of the first position happened in the first iteration, the last change of the second position happened in the second iteration, the last change of the third position happened in the first iteration, and so on. Note that the second-to-last position has not changed yet (this is indicated by the value 0 in the memory vector).

The general idea behind memory is that if some positions in the solution vector have changed, then the algorithm should leave these positions alone for some number of future iterations (i. e., they would be tabu for some number of iterations). This forces the algorithm to explore other parts of the search space, and after the required number of iterations has elapsed, these positions would become available again.

It might also be useful to alter the definition of memory in such a way that the information stored in memory is erased after some number of iterations. Assuming that the information can stay in memory for 50 iterations, then if the i-th position of vector M has a value j, a new interpretation of it can be that the i-th position of the solution vector was changed $50 - j$ iterations ago. Under this interpretation, vector M might have the following values after several iterations:

9	17	50	…	0	34

The numbers in this memory vector provide the following information.

- Position 1 is not available for the next 9 iterations.
- Position 2 is not available for the next 17 iterations.
- Position 3 is not available for the next 50 iterations (i. e., this position in the solution vector was just changed).
- Position 2,999 is available.
- Position 3,000 is not available for the next 34 iterations.

In other words, the most recent change took place in the third position, and all non-zero positions in the memory vector are considered tabu.

It might be interesting to point out that the main difference between these two interpretations of memory is simply a matter of implementation. The latter approach interprets the values as the number of iterations for which a given position is not available, while the former interpretation simply stores the iteration number of the most recent change at a particular position. In the above example, if the difference between the iteration counter and the i-th memory value is greater than 50 (our memory horizon), it should be forgotten. Hence, this interpretation only requires updating a single position in the memory per iteration, and increasing the iteration counter. In either case, tabu search utilizes the memory vector to force the search to explore new areas of the search space in an effort to escape from local optima. The recent changes that were made in the solution vector are tabu for the next iteration (i. e., entries with corresponding non-zero values in the memory vector).

Suppose that at some stage of the tabu search process, the quality measure score of the current solution **s** is 189.03 and the *best available* neighbor is solution **y**, with a quality measure score of 187.77. Note that this value represents a *decrease* in quality between the current and new solution. Note also that the available neighborhood is much smaller than the entire neighborhood, as many positions in the solution vector are tabu (i. e., their corresponding values in the memory vector are non-zero). On the other hand, imagine that a tabu neighbor, solution **q**, yields a quality measure score of 197.83. Assume further that the score of 197.83 is the best score from the beginning of the search, but because solution **q** is tabu we must ignore it!

Upon reflection, this policy might be too restrictive. It might happen that one of the tabu neighbors of current solution **s** produces a quality measure score that is *much* better than that of any previous solution. Perhaps we should make the search more flexible, and bend the rules somewhat if we find an outstanding solution. In normal circumstances, the tabu search should evaluate the *entire* neighborhood, and select a non-tabu solution as the next current solution, whether or not this non-tabu solution has a better quality measure score than the current solution. But in circumstances that are not “normal” – i. e., an outstanding tabu solution is found in the neighborhood – the superior solution should be selected as the next current solution.

Of course, other possibilities exist for increasing the flexibility of the search. For example, we could change the previous deterministic selection procedure into a probabilistic method, where better solutions have an increased chance of being selected. In addition, we could change the memory horizon during the search: sometimes it might be worthwhile to remember “more,” and at other times to remember “less” (e. g., when the algorithm climbs a promising hill in the search space).

Another option is even more interesting: The memory structure discussed so far can be labeled as *recency-based* memory, because it only records the last few iterations. This structure might be extended by a *frequency-based* memory, which operates over a much longer time horizon h (by time horizon, we mean the number of past iterations taken into account) and measures the frequency of change at each position. For example, an additional vector H may serve as a long-term memory. This vector is initially set to all 0s, and at any stage of the search the value j at the i-th position of this vector is interpreted as "during the last h iterations of the algorithm, the i-th entry of the solution vector was changed j number of times." Usually, the value of time horizon h (i. e., the number of past iterations we consider) is quite large, at least in comparison with the horizon of the recency-based memory. Thus after many iterations with $h = 50{,}000$, the long-term memory H might have the following values:

11	15	22	…	20	17

These frequencies (the total of which should equal 50,000) show the distribution of changes at each position of the solution vector during the last 50,000 iterations. The principles of tabu search indicate that this type of memory might be useful for diversifying the search. For example, the frequency-based memory provides information on changes in the solution vector that have been infrequent, and we can diversify the search by exploring these positions.

The use of long-term memory in tabu search is usually restricted to special circumstances. For example, we might encounter a situation where all non-tabu neighbors produce inferior quality measure scores. Thus, to make a meaningful decision about which direction to explore next, it might be worthwhile to consult the long-term memory. There are many possibilities for incorporating this information into the decision-making process, but the most typical approach makes the most frequent changes less attractive by penalizing the quality measure score. As an example, assume that the quality measure score of current solution **s** is 235.33, all non-tabu neighbors produce inferior values (230.11, 233.45, 231.47, etc.), and none of the tabu neighbors provides a value greater than 237.77 (the highest value found so far), so we cannot apply the aspiration criterion. This is a typical situation for consulting the frequency-based memory. The evaluation function used in such circumstances (for a new solution **x**) is the original evaluation function minus some penalty. This penalty, on the other hand, is calculated as a product of some parameter (say it is 0.1) and the total of all entries in the memory vector that correspond to changed entries of the solution vector. Let us illustrate this by an example.

Assume that neighbor solution **x** (with a quality measure score of 230.11) differs from current solution **s** in several positions (position 7, 65, 298, etc.). By referring to the memory vector H and adding together all the values in these positions, we arrive at a total of 304, which is multiplied by the penalty parameter 0.1 to produce a penalty of 30.4. After repeating this procedure for all the other neighbors under consideration, let us assume that neighbor solution **y** (with a quality measure score of 233.45) generated a penalty of 36.1, and that neighbor solution **z** (with a quality measure score of 231.47) generated a penalty of 30.9. From this it is clear that we should select neighbor solution **z**, because it has the highest final quality measure score:

- Neighbor solution **x** has a final evaluation of 199.71 (230.11 – 30.4).
- Neighbor solution **y** has a final evaluation of 197.35 (233.45 – 36.1).
- Neighbor solution **z** has a final evaluation of 200.57 (231.47 – 30.9).

Although the above example of using frequency values to create a penalty measure diversifies the search, we can also consider some other options.

Over the years, tabu search has become increasingly complex as different scientists have modified the classic technique by incorporating additional rules. We have already seen one such rule, called *aspiration by objective*, which overrides the tabu search when a neighbor yields a solution that is the best found so far. We can also use an additional rule, called *aspiration by default*, to select a neighbor that is the "oldest" of all those considered. It might also be a good idea to memorize not only the recent neighbors, but also whether or not these neighbors generated any improvement. This information can be incorporated into search decisions (called *aspiration by search direction*). We can also apply the concept of "influence," which measures the degree of change of a solution, either in terms of distance between the current and new solution, or the change in the solution's feasibility if we are dealing with a constrained problem. A neighbor has larger influence if a "larger" step was made from the current solution to the new, and this information can be incorporated into the search (so-called *aspiration by influence*). Of course, there are many possible ways of implementing memory structures, aspiration criteria, etc.

6.6 Evolutionary Algorithms

In the previous sections of this chapter, we discussed the hill climber, stochastic hill climber, simulated annealing, and tabu search. All of these optimization techniques represent the approach of processing a single solution (i. e., holding on to the best solution found so far and trying to improve it). This is intuitively sound, remarkably simple, and often quite efficient. The only decision to make during the execution of the algorithm is whether to "accept" or "reject" a newly generated neighbor solution. To make this decision, we can use many different rules. For example, hill climbers use deterministic rules: if an examined neighbor solution is superior, then proceed to that neighbor and continue searching from there; otherwise, continue searching in the current neighborhood. Simulated annealing uses probabilistic rules: if a neighbor solution is superior, accept this as the new current solution; otherwise, either probabilistically accept this new inferior solution anyway or continue to search in the current neighborhood. Tabu search uses the history of the search: take the best available neighbor, which need not be better than the current solution, but which is not listed in memory as a restricted or "tabu" neighbor.

Rather than processing a single solution, evolutionary algorithms[30] process a “population ” of competing solutions. In other words, evolutionary algorithms simulate the evolutionary process of competition and natural selection, where the solutions in the population fight for room in future generations. Additionally, new solutions are generated by means of genetically inspired operators (e. g., mutation or crossover) in a manner similar to natural evolution.

So, how do evolutionary algorithms work? Suppose, as before, that we are searching for the best solution to a difficult problem. Instead of generating an initial solution (as we did for other methods), we start with a population of initial solutions[31] (perhaps generated by random samples from the search space). The evaluation function then determines the quality measure score of each initial solution. Superior solutions, as determined by the evaluation function, are favored to become “parent” solutions for the next generation of “offspring” solutions. As before, new solutions can be generated probabilistically in the neighborhood of old solutions. However, evolutionary algorithms provide an additional twist: we can also examine the neighborhoods of pairs of solutions. That is, we can use more than one parent solution to generate a new offspring solution. One way to do this is by taking different “parts” of two parent solutions and then putting them together to form an offspring solution. For example, we might:

- Take the first half of one parent together with the second half of another.
- Take the “middle” segment from one parent and implant it as the new “middle” segment of the second parent.
- Take the numbers present in the solution vector of both parents and create some (possibly weighted) average of numbers for the offspring solution.[32]

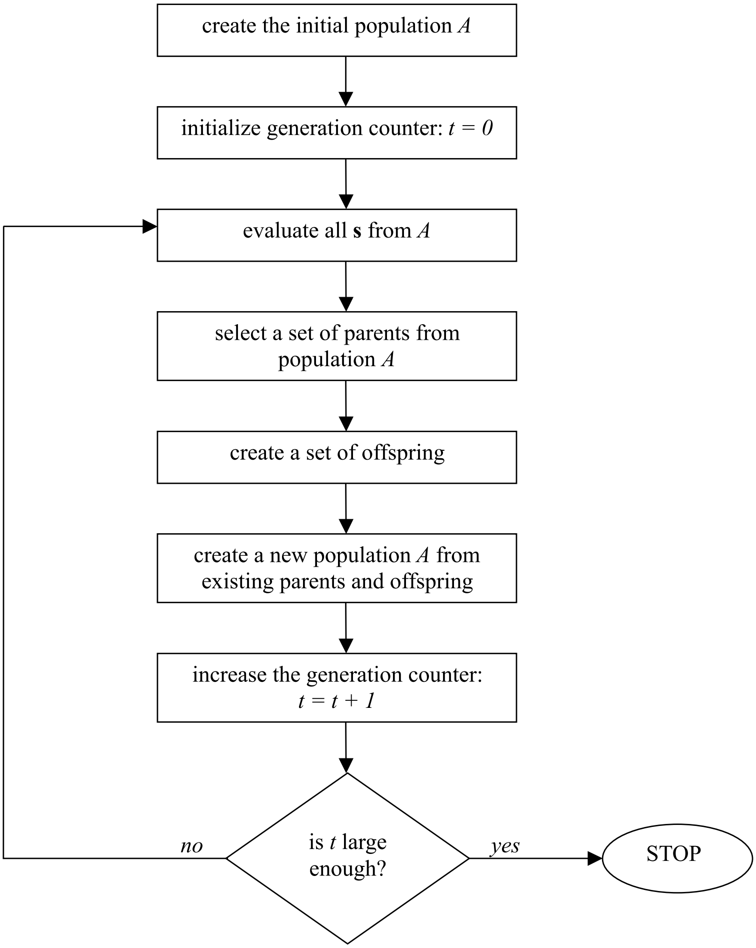
With each generation, the individual solutions compete against themselves (or also against their parents) for inclusion in the next generation of solutions. After many generations (i. e., iterations), we can often observe a succession of improvements in the quality of solutions and convergence toward the neighborhood of a near-optimum solution.

[30] There are many terms related to evolutionary algorithms. The most popular are: *genetic algorithms, evolutionary programming,* and *evolution strategies.* In this book, we use just one term, “evolutionary algorithms” without going into deeper details on similarities and differences between these techniques. Many people use the term “genetic algorithms” in the same manner as we use “evolutionary algorithms,” because the term “genetic algorithms” is better known than “evolutionary algorithms” in the business community. Also, in Sect. 9.1, we will discuss a special class of evolutionary algorithms called *genetic programming.*

[31] In evolutionary algorithms, these solutions are called *chromosomes* (to emphasize a link with genetics).

[32] While designing such recombination operators, it is sometimes difficult to escape the temptation of “improving nature”… For example, should we consider more than two parents to generate offspring? In evolutionary algorithms, this is relatively easy: we can build an offspring solution by taking the first segment of the first parent, merge it with the second segment of the second parent, and extend it with the third segment of the third parent.

The following flowchart outlines the basic steps of an evolution algorithm:


create the initial population *A*
initialize generation counter: *t = 0*
evaluate all **s** from *A*
select a set of parents from population *A*
create a set of offspring
create a new population *A* from existing parents and offspring
increase the generation counter: *t = t + 1*
is *t* large enough?
no
yes
STOP

This flowchart is very simple to follow: First, we generate some number of (possibly random) solutions. Since the population size is one of the parameters of the evolutionary algorithms, let us say we created 100 solutions as members of population *A*. At the same time, we set our generation counter *t* to zero. At this stage we will be entering an "evolutionary loop" that is repeated many times, and the process will terminate when *t* is "large enough" (many other termination conditions are possible, such as lack of progress after some number of generations). During this evolutionary loop, a few activities are performed. First of all, all the solutions in population *A* are evaluated, as we would like to know which solutions are superior and which are inferior. Second, a subset of parent solutions is selected from the population, favoring superior solutions. Third, the selected parent solutions produce offspring solutions by means of some variation operators; quite often terms like *crossover* or *mutation* are used. Fourth, a new population is selected (usually of the same size as the original population) from the existing parent and offspring solutions. And lastly, the generation counter *t* is increased and the loop is repeated.

Without a doubt, this is an appealing approach for solving complex problems! Why should we labor to solve a problem by calculating difficult mathematical expressions or developing complicated computer programs to create approximate models of a problem, when we can discover near-optimum solutions using models of much greater fidelity. However, it is appropriate to ask some important questions: How much work does it take to implement these concepts into an algorithm? Is this cost effective? And can we solve *real* problems using evolutionary algorithms? Well, let us explore these issues by continuing the car distribution example.

Say we started from an initial population *A*, which consists of 100 randomly generated solutions (the first 10 are displayed below). We have created our "universe," set our generation counter *t* to zero, and are ready to enter the evolutionary loop:

23	41	5	…	19	41
49	3	25	…	12	34
4	12	15	…	20	3
17	17	16	…	19	22
44	25	43	…	36	19
1	29	42	…	6	29
39	41	15	…	40	4
21	22	24	…	27	38
19	13	16	…	33	41
47	2	53	…	10	50
	⋮			⋮	

We begin the evolutionary loop by evaluating all the initial solutions and identifying the parent solutions. From the 100 solutions present in the population, say we select 80 parents. One of the more popular parent selection strategies is called *tournament selection*, where we select two random solutions from our population, compare their quality measure scores, and select the better one as a parent. We repeat this process 80 times (i. e., until we select 80 parents). For example, say we selected:

24	41	6	...	19	41

and:

17	17	16	...	19	22

for tournament competition. If the first solution has a quality measure score of 168.34 and the other 172.41, then the latter solution is the "winner" and is placed in a temporary population of parents. Note, however, one interesting twist: As we select pairs of random solutions, it is quite possible that some (relatively strong) solution is selected more than once for a tournament, and it wins such a tournament more than once. In that case, this solution would be represented in the population of parents several times (e. g., the population of parents may contain several identical solutions). This is a desired side effect, as we would like to promote good "genetic traits" during the evolutionary process. If a solution is capable of winning several tournaments, then the chances are that its quality measure score is above average, and that it will also produce "above average" offspring solutions.

Now we are ready for the most important step of the evolutionary loop: the creation of offspring solutions. As mentioned earlier in this section, we can consider several possibilities here. For example, we can use a *mutation operator*, which would change some positions in a parent solution to produce an offspring solution. For example, a parent:

17	17	16	...	19	22

can be "mutated" to produce the following offspring solution:

43	17	16	...	3	22

Mutation occurred in the first position of the solution (where the value 17 was replaced by a randomly generated value of 43), and on the second-to-last position (where the value 19 was replaced by a randomly generated value of 3). When it comes to the process of mutation, we have some flexibility. For example:

- The probability of mutation may be very low (e. g., 1%). This would mean that on average 1% of the positions in the parent solution would be changed to produce a new offspring solution. As there are 3,000 positions in the solution vector, on average 30 positions would receive new values. Of course, the probability of mutation (whether it be 1%, 0.1% or 10%) is a parameter of the algorithm that we are responsible for adjusting.
- Regardless of whether the probability of mutation is 0.1% or 10%, we also have to decide "how" to mutate. One possibility is to replace a selected position by a random value from the range from 1 to 50 (as there are 50 auction sites to choose from). However, we may also consider some "intelligent" mutations. If we have auction sites sorted by growing distance from each car (so that close numbers correspond to close auction sites in a geographical sense), then the mutation operator might be allowed to "slightly" modify a position. In particular, the value 17 from the first position of the solution would be allowed to change into 15, 16, 18, or 19, but not any other number.
- We might also have to consider more than one mutation operator. One mutation operator, for example, would be responsible for "small" changes, and the other for "big" changes (to introduce some additional diversity into our search). Therefore, even if the auction sites are sorted in some meaningful way, we may still allow random changes of values in the solution vector.

As indicated earlier, we can explore some other operators for creating offspring. For example, we can use a *crossover* operator, which produces two offspring solutions by mixing some positions from two parent solutions. For example, the parents:

23	41	5	...	19	41
49	3	25	...	12	34

are cut after the 1,357th position, and two offspring are generated by putting together the first part of the first parent (i. e., all 1,357 initial entries) with the second part of the second parent (i. e., the remaining 3,000 – 1,357 = 1,643 entries) for the first offspring, and by gluing together the first part of the second parent (the initial 1,357 entries) with the second part of the first parent (1,643 last entries) for the second offspring:

23	41	5	...	12	34
49	3	25	...	19	41

Of course, it is possible to have more than one cutting point. In particular, we may consider a crossover where we cut after every position, and a random decision is made whether we should use a value from the first or second parent. For example, the pair of parents we discussed earlier may produce the following offspring (i. e., where all positions in the offspring are selected randomly from the first or second parent):

23	41	25	…	19	34

The role of the crossover operator is straightforward: it might be that at some stage of the search process, some solutions contain individual decisions of high quality (e. g., sending the first car to auction 43). Crossover can speed up the search process, as these quality building blocks can be mixed together to create a solution with a larger number of quality components.

Anyway, after applying several variation operators (possibly different types of mutations and crossovers), a set of 200 offspring solutions is created for the car distribution example. Now we are ready for the final step in the evolutionary loop: creation of the next generation (we also increase our generation counter t at this stage, so it takes on a value of 1). As the population size is 100, our task is to create a new population of 100 solutions out of 80 selected parents and 200 created offspring. Again, there are many ways to accomplish this task. For example, we can:

- Apply a tournament selection process to all 280 solutions (80 parents and 200 offspring).
- Exclude the parents and build a new generation from the offspring only.
- Design a tournament selection where more than two individuals compete to be selected.
- Rank all the solutions by their quality measure score and then allocate the probabilities of selection accordingly.

In most implementations, however, we use the *elitist* strategy, which selects the best solutions from one generation to the next. The cycle is then complete. We arrive at $t=1$ generation with a new population of 100 solutions. We then repeat the evolutionary loop by evaluating these 100 solutions, selecting the parents, applying crossover and mutation operators to create offspring, selecting the next generation, and so on, until generation counter t hits a threshold (e. g., 100,000). At this stage, we expect to have many quality solutions in our final population, and the best solution is selected as the final solution.

6.7 Constraint Handling

As we discussed in Chap. 3, the car distribution problem has many problem-specific constraints: inventory levels, transportation costs, volume effect, and so on. Hence, the application of modern optimization techniques probably will not be so straightforward (where we just evaluate new solutions and decide which ones to keep or discard). For instance, suppose that we want to introduce some hard constraints on the distribution of cars (e. g., "red cars should not be sent to auction sites in Texas" or "cars with more than 100,000 miles should not be sent to South-East auction sites"). But how can we find feasible solutions to the car distribution problem when such hard constraints are imposed? Well, there are generally three ways to influence the search toward feasible solutions: through the evaluation function, the representation of the solution, and the operators (and, of course, through some combination of the three). These three approaches are independent of the optimization technique used.

As indicated earlier, selecting the representation of a solution is a major design decision. Some representations will make handling constraints easier, while others will make it harder. First, let us summarize the constraints present in the car distribution problem:

- Transportation distance limit.
- Acceptable transportation routes.
- Inventory level limits.
- Exclusions based on car mileage.
- Exclusions based on car color.
- Specified auction dates.

Additional issues to take into account that are related more to the evaluation function include:

- Transportation costs.
- Volume effect.
- Reduction of "stragglers" (i. e., cars that stand many days at the original location without being moved to an auction).

Many constraint-handling methods have been proposed over the last few decades, but the simplest approach is the *death penalty*, where solutions that violate a constraint are immediately removed from further consideration. However, in many cases, the death penalty approach does not work very well. For instance, in highly constrained problems, the system generates a new solution, checks its feasibility, and then discards it because it is not feasible. And if 99% of the system's effort is wasted on generating and removing infeasible solutions, then 99% of the system's effort is unproductive! Moreover, the first feasible solution found (regardless of quality) can drive the whole system to converge, and the final result might be a solution of poor quality.

Therefore, some leniency might be warranted toward infeasible solutions, as the death penalty might be too much medicine. What about some less drastic penalties?

Indeed, one of the most popular group of methods is based on various penalty functions. The idea is that if a potential solution (whether in evolutionary algorithms, simulated annealing, or tabu search) violates some problem-specific constraint, then the solution is "penalized" by making its quality measure score smaller (i. e., less attractive). In other words, the quality measure score of a solution consists of two parts: the output of the evaluation function and a penalty score for violating constraints.

For example, if a particular solution (which recommends to ship the first car to auction 23, the second car to auction 12, and so forth):

23	12	5	…	19	41

violates a constraint that limits the transportation distance to 800 miles (e. g., the distance between the location of the second car and auction 12 is 915 miles), we should penalize it. Let us assume that the quality measure score of this solution is 564.34, which is based on the predicted sale prices of these cars after taking into account depreciation and transportation costs, auction fees, and the volume effect. However, since the second car exceeds the transportation distance limit of 800 miles, we should make this score less attractive. One way of setting such penalties is to assign a penalty weight for each violated constraint.

Now we have to design a penalty function for this type of constraint violation. For example, we can assign 10 penalty points for every 100 miles in excess of the transportation distance limit. In our case, the excess is 115 miles, so the penalty weight would be 11.5. Thus, due to this single violation, the quality measure score of this solution would be reduced to 552.84 (564.34 – 11.5). But that is not all! There are a variety of other constraints, such as excluding some cars from various auction sites (these exclusions are based on the mileage and/or color of the car). It may happen, that the above solution contains a few violations of this type. For example, it has assigned the first car (which is red) to auction 23, which has a constraint against red cars … Again, we would have to assign a penalty weight for this violation, and let us assume that we subtract an additional 7 points. We need to deal with all the constraints in a similar fashion, and only after all the penalties have been totaled can we calculate the final quality measure score.

This penalty approach is relatively straightforward and easy to implement. However, it has many disadvantages. First of all, it is tricky to assign meaningful penalty weights, as it might be difficult to weigh the relative penalty of violating different constraints: in the above example, the color violation is equivalent (in penalty points) to a transportation distance violation of 70 miles – which may or may not be accurate. It is even harder to tune all these penalty weights to the original evaluation function: If the solutions are evaluated in the range from 0 to 100, then the above penalty weights would clearly be too high. And if the penalty weights are too high, then the effect is similar to that of the death penalty, as heavily penalized solutions would not stand much chance of survival. However, if the penalty weights are too low, then the final solution may violate too many constraints …

These issues can be addressed in a number of ways, one of which is to design *dynamic penalties* that change (usually increase) with each iteration. The purpose of dynamic penalties is to allow early solutions to sample a variety of points in the search space without paying too much attention to feasibility (as the penalty weights are low). Then, as the penalty weights gradually increase, greater emphasis is placed on feasible solutions.[33] The drawback of this approach is that we have to specify the penalty weight changes in advance, and, again, this can be quite tricky. Another approach deals with this difficulty in the following way: For population-based techniques, we can assume some "healthy" ratio between the number of feasible and infeasible solutions in the population, and during the run of the algorithm we maintain this ratio by increasing or decreasing the penalty weights. Thus, there is no need to specify the penalty weight changes in advance; the system adapts these weights by itself. Again, the drawback lies in having to define what the "healthy" ratio is for a particular problem.

Another way of dealing with constraints is based on the idea of *repair algorithms*. Let us take the same solution as before, with the same constraint violations of transportation distance, color, and so forth:

23	12	5	…	19	41

Instead of penalizing this solution, we may try to "repair" it. As before, the distance between the location of the second car and auction 12 is 915 miles, while the transportation distance limit is 800 miles. Because auction 12 is not appropriate for this car, we can try to repair this single position by using some other auction. Here we may have to consider several possibilities on "how to repair"? We can select auctions at random and check if they satisfy this constraint, or we may consider auctions in some predefined order (e. g., based on proximity). Either way, we can repair the solution by replacing auction 12 with auction 47:

23	47	5	…	19	41

We can handle the other constraint violations in the same way: If a constraint is violated, we can search for a suitable replacement. However, this process is not so straightforward. Complex constraints can involve many variables, and so repairing one segment of the solution might trigger a constraint violation in some other segment. In complicated cases such as these, we should use a problem-dependent repair algorithm.

[33] Note the similarity of this approach to simulated annealing: the algorithm initially allows up and down movements, but, with time, it begins to favor moves that lead to superior solutions.

There is an additional (very interesting) twist present in the use of repair algorithms. When a solution is repaired, the repair might be temporary (i. e., the solution is changed only for the purpose of evaluation, as it is much easier to evaluate a feasible solution than an infeasible one). The original (infeasible) solution stays in the population for further processing. This is called the *Baldwin effect,* where solutions are evaluated on the basis of their potential, rather than their current state.

The other approach would be to make the repair permanent (i. e., the solution is changed permanently), which is called *Lamarckian evolution.* This would be equivalent to improving our genes so that they are passed onto our offspring. Both of these approaches – Baldwin effect and Lamarckian evolution – have their advantages and disadvantages. Many practitioners apply them in some ratio (e. g., only 10% of repairs are done on a permanent basis, so that the process is 10% Lamarckian and 90% Baldwinian) and claim that a mixture produces better results than a pure Baldwinian or Lamarckian approach.

Yet another approach for constraint handling is based on *decoders.* The idea is that data present in a solution vector are interpreted (i. e., decoded) in such a way that they correspond to a feasible solution. This would require indirect representation of solutions and some preprocessing. In the case of distributing cars, this means sorting all the available auction sites by their distance from each car's current location (as discussed at the beginning of this chapter). For the first car (say it is located at a dealership in Kansas) auction 1 would be the closest (distance-wise), auction 2 would be the second closest, and so forth. The representation of a solution may look something like:

1	3	1	…	1	2

The above vector indicates that the first car should be sent to the closest auction site, the second car should be sent to the third-closest auction, and so on. Using the decoder approach, we can maintain a sorted list of *available* (i. e., feasible) auctions for each car. By doing so, many constraints (e. g., exclusions based on distance, mileage, or color) can be taken care of during the preprocessing stage. For example, if auction 13 is too far for a given car, then we can exclude auction 13 from the sorted list of available auctions for this car. We can extend this "exclusion" idea even further, by adding other constraints: if a car is red and auction 17 (which is within 800 miles radius from the current location of the car) does not admit red cars, then, again, we can exclude auction 17 from the sorted list of available auctions for this car. Thus, there is no need to penalize or repair infeasible solutions.

These three main approaches – penalties, repairs, and decoders – do not constitute the complete list of constraint-handling methods. Sometimes it is worthwhile to develop a specialized operator that only creates feasible solutions from existing feasible solutions. For example, such an operator would process all the cars in a sequential manner, from 1 to 3,000, sending each car to an auction site that satisfies all constraints. Then, having a set of such feasible solutions, we can use a problem-specific operator to transform these feasible solutions into new feasible solutions. This way, again, there is no need to penalize or repair any infeasible solutions, as the operator only generates feasible solutions.

6.8 Additional Issues

In this chapter we explained the basic ideas behind several modern optimization techniques: The idea of traveling "up" in hill climbers, the idea of traveling almost randomly at first and gradually switching to uphill moves in simulated annealing, the idea of remembering our previous decisions in tabu search, and the idea of "breeding" the best possible solution in evolutionary algorithms. However, real-world problems are usually very complex, and many additional issues have to be taken into account when applying these techniques.

One potential issue in applying modern optimization techniques to real world problems is that of multiple objectives (as discussed in Sect. 2.4). For example, apart from maximizing the net profit from the sale of all cars, a business manager may want to minimize transportation cost as the second objective. On top of that, he may also want to balance the distribution of cars among all the auction sites, which would constitute the third objective.

As we discussed in Chap. 2, non-dominated solutions are of interest to us (i. e., solutions that cannot be improved according to one objective without being worse with respect to the remaining objectives). Ideally, a system that deals with multi-objective problems should return several diverse non-dominated solutions, as each of these solutions might be of interest. The concept of diversity is important, because many non-dominated solutions might be very similar to one another. For example, both solutions:

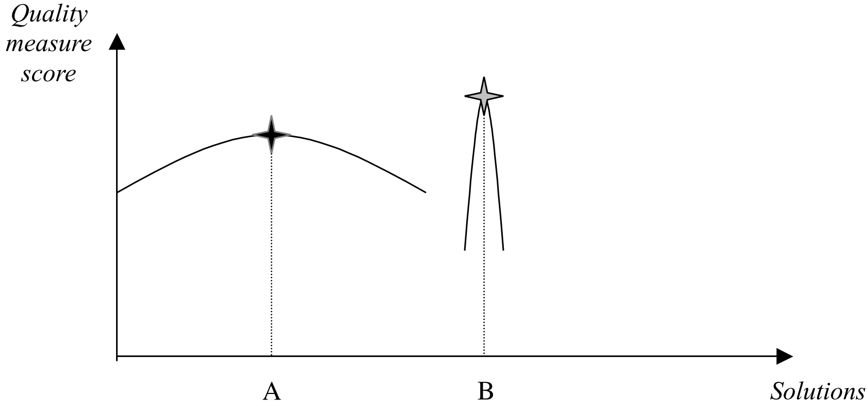
23	47	5	...	19	41

and:

23	47	6	...	19	41

might be non-dominant, but there is little point in returning both of them to a business manager, as they are almost identical. The simple fact that a business manager selects the final solution for implementation has two implications: (1) the number of returned solutions should not be too large, and (2) the returned solutions should be diverse. At this stage, human expertise is required to express different objectives on the same scale (e. g., net profit and balanced distribution) or to include additional higher-level information (e. g., strategic agreements made with certain auction sites). As indicated earlier, a business manager may assert the relative importance of each objective – perhaps by assigning numeric weights or imposing a ranking – and then select subsets of solutions that follow this ordering. Another option would be to select the most important objective and then convert the remaining objectives into constraints, using them as thresholds to be satisfied.

Also, referring back to the figure discussed in Sect. 2.2, it is prudent to point out that some solutions carry more risk than others:



For instance, we may consider two quality distribution solutions A and B, where solution B is slightly better than solution A. Although solution B might provide a $15 higher net profit per car than solution A, it might also carry more risk. For example, solution B might require transporting some cars for 2,000 miles (assuming there is no limit on the transportation distance), whereas the longest transportation distance imposed by solution A is only 550 miles. Now, is this additional risk (the longer distance, the higher probability that something can go wrong, e. g., accident, delay, truck failure) worth an additional $15 per car? By receiving a set of diverse non-dominated solutions, a business manager can make the final decision.

Lastly, most real-world problems also present another serious difficulty: *they are set in a dynamic environment.* This is the case in the car distribution example, as the problem contains several time components. Note that yesterday's distribution (and the distributions made during the past few days) will influence today's distribution. The reason is that auction sales are usually held twice a month, and all cars shipped during some time interval to a particular auction site would be sold together, on the same day. Because of this, it is necessary to include a memory buffer, where we record all recent decisions (like we did in tabu search). Recent decisions would influence the quality measure score of solutions that are processed today.

Because consumer preferences are also constantly changing (and makes, models, colors, etc. that are popular one year might be unpopular the next),[34] it is necessary to monitor these trends and update the prediction module on a regular basis. Other changes may include modifying the set of auctions sites by entering into new agreements and letting old ones expire, or by having some internal policy for using different auction sites at different times. Thus, the problem can easily change from one day to the next. Similarly, business managers want to insert or delete new rules (e. g., "red cars should not be sent to auction sites in Texas" or "cars with more than 100,000 miles should not be sent to auction sites in the South-East"), and such changes mean that today's distribution problem might be quite different from yesterday's distribution problem!

6.9 Recommended Reading

In this chapter we gave some background information on search spaces, different solution representations, basic hill-climbing algorithms, as well as a few modern optimization techniques that fall into the category of modern heuristics, such as simulated annealing, tabu search, and evolutionary algorithms. We also touched upon some additional issues related to searching for the optimum solution, which include constraint handling, solution robustness, multi-objective optimization, and dynamic environments. Lastly, despite the different terminology used for different techniques (e. g., *crossover* and *mutation operators* for evolutionary algorithms, or *temperature* for simulated annealing), it is important to remember that the techniques themselves are in fact very similar from a high-level perspective. They process one or more current solutions, generate one or more new solutions, evaluate the solutions found so far, and then make decisions on where to search next.

There are many texts available on modern optimization techniques. The book *Local Search in Combinatorial Optimization*, edited by Emile Aarts and Jan Karel Lenstra (Wiley, Chichester, 1997) provides many examples of local search based on the traveling salesman and vehicle routing problem, machine scheduling, VLSI layout synthesis, and code design. The book also includes additional chapters on simulated annealing, tabu search, evolutionary algorithms, and artificial neural networks.

A detailed treatment of simulated annealing is provided in *Simulated Annealing: Theory and Applications*, a book by P.J.M. van Laarhoven and Emile Aarts (D. Reidel, Dordrecht, 1987).

Tabu Search by Fred Glover and Manuel Laguna (Kluwer, Norwell, 1997) provides a good all-around introduction to tabu search, while *Introduction to Evolutionary Computing* by Gusz (A.E.) Eiben and Jim Smith (Springer, Berlin, 2003) provides an excellent introduction to evolutionary algorithms.

[34] For example, the rise in crude oil prices in 2004–2005 led to a sharp drop in consumer demand for sports utility vehicles (SUVs) and other heavy cars.

The book *How to Solve It: Modern Heuristics* by Zbigniew Michalewicz and David Fogel (Springer, 2nd edition, 2004) provides a general overview of evolutionary algorithms, simulated annealing, tabu search, and hill climbers, as well as a few additional issues like handling constraints, setting the parameters of a method, and dealing with time-changing environments.

Also available are many books that present various heuristic methods and their applications; see, for example, *Modern Heuristic Methods for Combinatorial Problems*, edited by Colin Reeves (Wiley, Chichester, 1993) or *Meta-heuristics: Theory and Applications*, edited by Ibrahim H. Osman and James P. Kelly (Kluwer, Norwell, 1996).

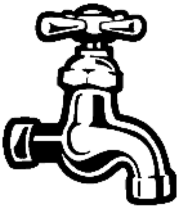
7 Fuzzy Logic

"Crime is common. Logic is rare."
The Adventure of the Copper Beeches

" 'Is there any point to which you would wish to draw my attention?'
'To the curious incident of the dog in the night-time.'
'The dog did nothing in the night-time.'
'That was the curious incident,' remarked Sherlock Holmes."
Silver Blaze

For many millennia, humans have tried to describe the world using models based on mathematics and logic, but only during the last few decades has it become possible to construct computer models of the real world. Computers are based on a binary language of zero's ("0"s) and one's ("1"s). This is an abstraction created by computer scientists to describe what is going on inside a computer chip, with "0" equating to "power off" and "1" equating to "power on." Every single transistor inside a computer chip is like a tap, and an electric current runs through the transistor like water through a tap. It looks very much like the following:

No water = no power = "0":



Water running = power = "1":

The water represents the electric current (i. e., electrons) running through the transistor inside a computer chip. All the “0”s and “1”s inside a chip are used to perform logical calculations according to *Boolean logic,* which treats 0 as “false” and 1 as “true.” So in other words, “0” = false = no power and “1” = true = power. The principles behind Boolean logic are not limited to computers, as this type of reasoning dates back to the ancient Greeks, especially the Greek philosopher Aristotle.

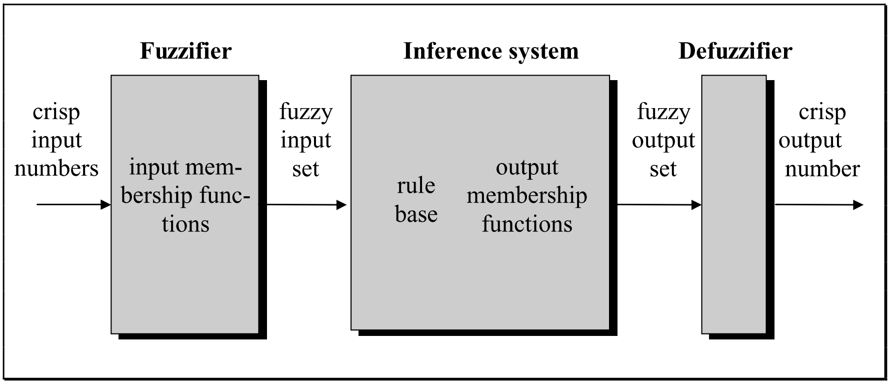
Nevertheless, the following tap can also be used to illustrate a fundamental problem with Boolean logic:

Is the water running in this tap? According to Boolean Logic the water must either be running or not, but in the above situation the water is only running to some *degree*. If we had to make a binary (true or false) decision then we might say that the water is running, and inside the transistor this would translate to a “1” = true. But now imagine the following situation:

Again, we can ask ourselves if the water is running or not, and try to make another binary decision. However, we can continue to make this situation more difficult by continuing to reduce the amount of water flowing out of the tap. These illustrations expose the fundamental problem with Boolean logic: Everything must be either true or false, all or nothing. Boolean logic cannot deal with something that is true to some degree. To deal with this problem, a relatively new type of logic has emerged, called *fuzzy logic.* Because it treats everything as a degree to which something is true, it can be used to create extremely powerful prediction and classification models. Let us take a closer look at how fuzzy logic works, and how it can be applied in an Adaptive Business Intelligence system.

7.1 Overview

Let us begin with one of the most common fuzzy logic systems, called the *Mamdani Fuzzy System.* It looks like the following:



Each component in this flowchart has a separate function:

- The *fuzzifier* takes the crisp input numbers and converts them into a *fuzzy input set* by using input membership functions (explained below) to calculate the degree of something being true.[35]
- The *inference system* takes the fuzzy input set from the fuzzifier, and applies a rule base and then the output membership functions to create a *fuzzy output set.*
- The *defuzzifier* takes the fuzzy output set from the inference system and converts it into a crisp output number (i. e., the prediction).

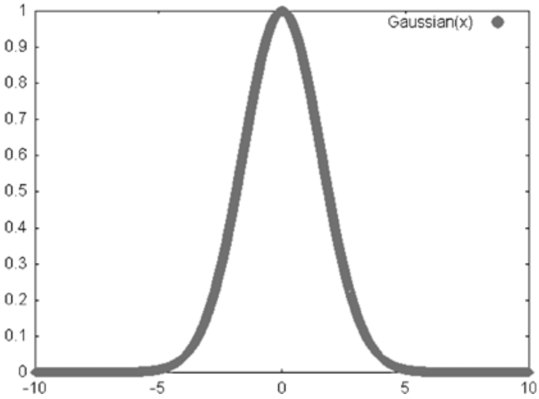
To gain a better understanding of how fuzzy logic extends Boolean logic, let us take a closer look at each component.

7.2 Fuzzifier

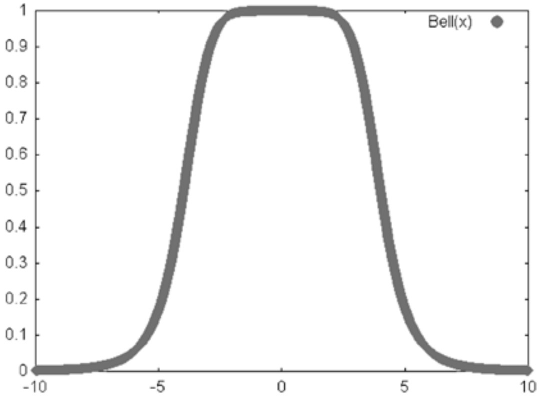
In order for fuzzy logic to work, *input membership functions* are used by the fuzzifier to turn the crisp input numbers into a fuzzy input set. There are many types of membership functions, and a few of them are shown below to provide an idea of what they may look like.

[35] Note that the *degree* of something being true is *not* the same as the *probability* of something being true.

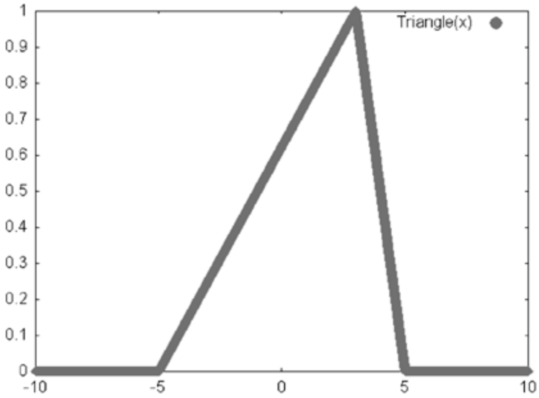
Below is an example of a *Gaussian membership function:*



Below is an example of a *Bell membership function:*



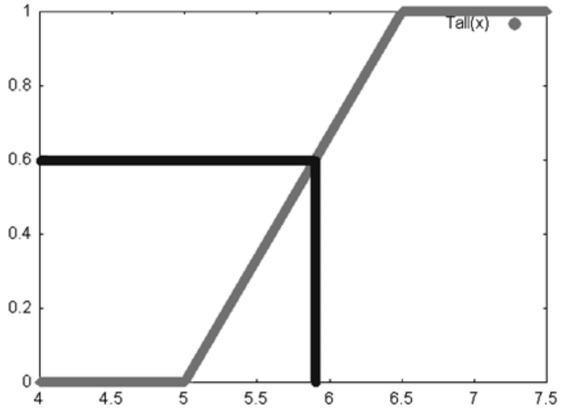
Below is an example of a *triangular membership function:*



The last figure shows something interesting: Membership functions do not have to be symmetrical and can have all kinds of forms. Nevertheless, simple membership functions are often preferred over complex ones because business managers and experts have an easier time understanding them.

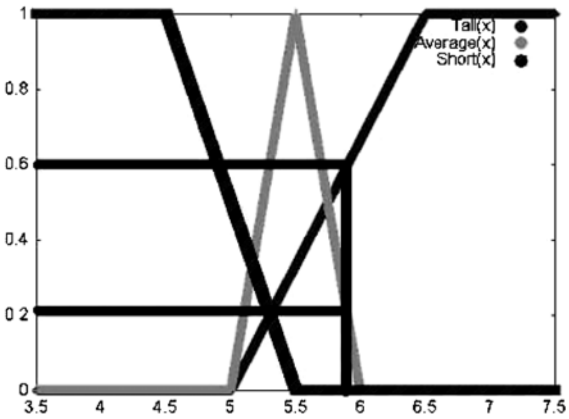
We can see from the above illustrations that membership functions always return a membership degree between 0 and 1, where 0 equates to "zero degree of membership" and 1 equates to "full degree of membership."

Let us look at an example using triangular membership functions. Imagine a man who is 5 feet 11 inches tall (about 1.8 meters). Somebody might say, "He is tall." Boolean logic would attach the value "true" or "false" to this statement, but fuzzy logic would try to evaluate to what degree the statement is true! Imagine the following triangular membership function that defines "tall":



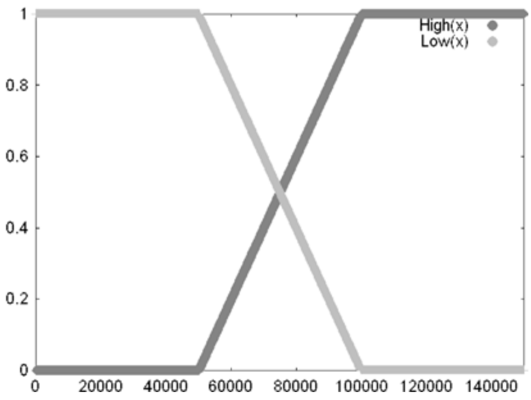
This membership function takes the height of a man and defines to what degree he is tall. If a man is shorter than 5 feet then his degree of being tall is 0; and if he is taller than 6 feet 6 inches, then his degree of being tall would be 1. If his height falls somewhere between 5 feet and 6 feet 6 inches feet, then his degree of being tall is somewhere in the range from 0 to 1. According to the membership function above, a man who is 5 feet 11 inches is tall to a degree of 0.6.

Imagine now that we also have membership functions defining "short" and "average":

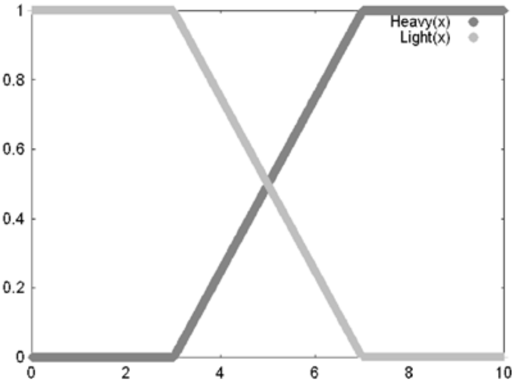


This figure illustrates an important point: Just because a man is "tall" to a certain degree, does not mean that he is not also "short" and "average" to some other degrees. In other words, the membership functions can overlap. A man who has a height of 5 feet 11 inches would be "short" to degree 0, average to a "degree" 0.2 and "tall" to a degree 0.6. The numbers do not have to add up to 1 because "short," "average," and "tall" are completely different linguistic values (some other examples of linguistic values would be "red," "heavy," "very fast," "close," "far," "warm," "cold," etc.).

In the car distribution example, we might have the following input membership function for "mileage," where the x-axis indicates the mileage:



And the following for "damage level," where the x-axis indicates the damage level:



These two input membership functions[36] refer to the variables "mileage" (which is found by reading the odometer) and "damage level" (which could be estimated on a scale from 0 to 10 by an auto mechanic). To create the fuzzy input set, the fuzzifier takes the crisp input number in the range of 0 to 10 for "damage level," along with the actual value for "mileage," and then applies the membership functions to calculate the membership degrees. For example, a car with "mileage" = 80,000 and "damage level" = 4 would be *low mileage* to a degree of 0.4, etc. Hence, the fuzzifier has transformed the crisp input numbers (80,000 miles and damage level 4) into a few linguistic values and degrees of membership:

- "High mileage" to a degree of 0.6.
- "Low mileage" to a degree of 0.4.
- "Heavy damage" to a degree of 0.25.
- "Light damage" to a degree of 0.75.

These membership degrees constitute the fuzzy input set, which is used by the inference system to generate a fuzzy output set.

7.3 Inference System

The *inference system* is the heart of fuzzy logic and it contains knowledge in the form of rules and output membership functions. If we require extremely precise predictions, then we will probably end up with many rules; if we do not require such extreme precision, then our rule base will probably contain fewer and more

[36] Although these two input membership functions look identical (because we have used straightforward membership functions to simplify the illustrations), this does not have to be the case, and some membership functions might be quite complex. Furthermore, there are usually more than two linguistic values defined for a variable (not just "high" and "low" for mileage or "heavy" and "light" for damage level).

general rules. Although more rules usually mean more precise predictions, the rule base can become too large (and, consequently, difficult for business managers and experts to comprehend). Hence, there needs to be a balance between generality and precision for any real-world business problem.

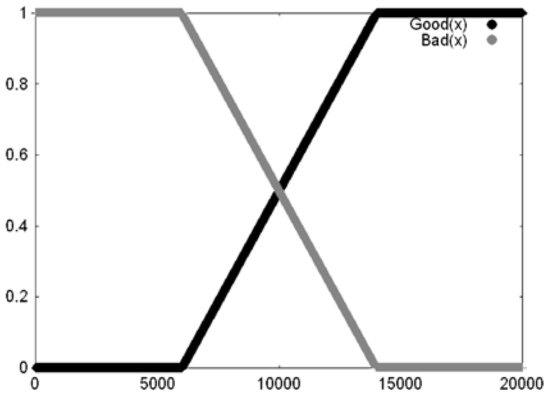
In the case of the car distribution example, we could build a rule base in one of two ways. The first way would be to ask a human expert to define all the rules that affect the sale price of a car. This manually built rule base might serve as a good starting point for further tuning (as the initial rules might be somewhat inaccurate), but a major drawback of this approach is the amount of time and effort it requires. For example, consider the countless membership functions that would need be to handcrafted and fine-tuned so that the predicted sale price is acceptably close to the actual sale price.

The second way is to carry out a data mining exercise on the available data. There might be a few reasons for using this approach rather than querying an expert: For instance, a human expert may not be available for questioning, or the available expert might be unable to define all the rules (as many decisions might be based on "intuition"). Another reason could be that we want the rule base to be "unguided" and free of human assumptions that might be flawed. For example, a thorough analysis of the data might lead to discovery of very effective rules based on "trim" and "color" – a combination that a human expert might not have considered. Sometimes human intuition is an effective guide, but at other times, it can lead us astray.

For the sake of simplicity, let us build a very simple rule base for estimating the sale price of a particular car at a particular auction site:

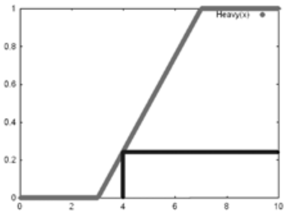
Rule 1: **If** damage level **is** heavy **and** mileage **is** high, **then** sale price **is** bad
Rule 2: **If** damage level **is** light **and** mileage **is** low, **then** sale price **is** good

These two rules refer to the variables "damage level," "mileage," and "sale price." Of course, the linguistic values for the variables "mileage" (i. e., "high," and "low") and "damage level" (i. e., "heavy" and "light") were already defined by some triangular input membership function (see Sect. 7.2). Now we also have to create a simple output membership function for "sale price," with two linguistic values "good" and "bad," where the x-axis indicates the sale price:

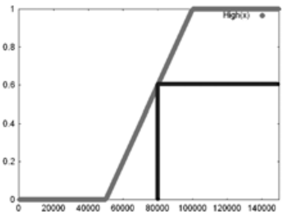


With a fuzzifier capable of generating fuzzy input sets, and an inference system consisting of a rule base (with two simple rules) and an output membership function, we can now create a fuzzy output set. By continuing our example of a car with “mileage” = 80,000 and “damage level” = 4, let us take a look at what the fuzzy output set would look like.

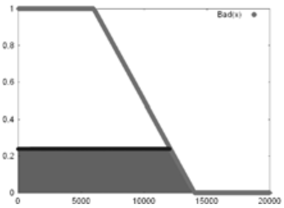
First of all, let us visualize the fuzzification process. As we have already discussed, the crisp input number “mileage” = 80,000 is transformed by the fuzzifier to calculate a membership degree of 0.6 for the mileage being “high” (illustrated in the top-middle graph below) and a membership degree of 0.4 for the mileage being “low” (illustrated in the middle-middle graph below). Similarly, the crisp input number “damage level” = 4 is transformed by the fuzzifier to calculate a membership degree of 0.25 for the damage level being “heavy (illustrated in the top-left graph below) and a membership degree of 0.75 for the damage level being “light” (illustrated in middle-left graph). These linguistic and membership values make up the fuzzy input set:



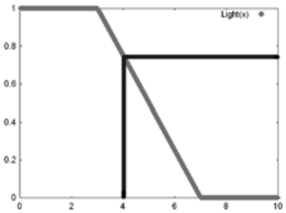
if *damage level* ***is*** *heavy*



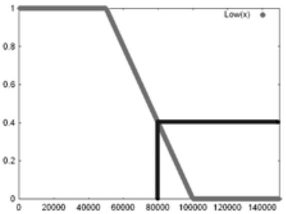
and *mileage* ***is*** *high*



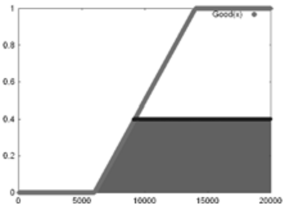
then *sale price* ***is*** *bad*



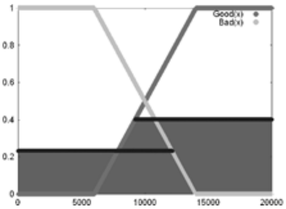
if *damage level* ***is*** *light*



and *mileage* ***is*** *low*



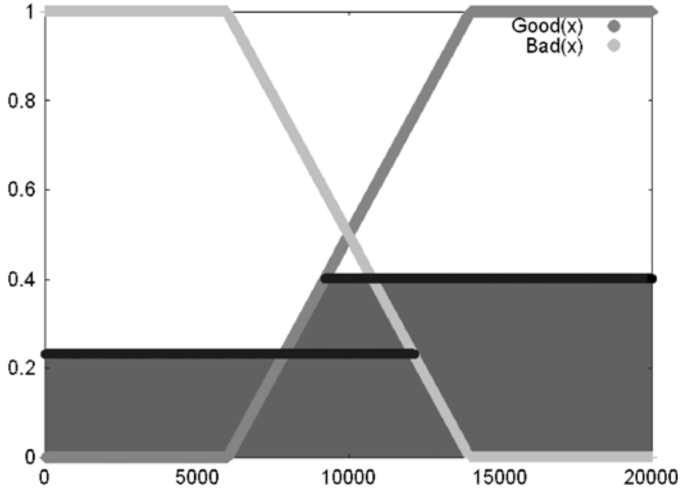
then *sale price* ***is*** *good*



fuzzy output set

The inference system then takes this fuzzy input set and applies a *fuzzy **and***, which can be performed in a number of different ways. Two frequently used methods are to take the minimum input membership degree, or to multiply the input membership degrees together.[37] In the above example, the inference system used the *minimum* input membership degree for *fuzzy **and***. Therefore, the degree of sale price being "bad" is set to 0.25 in the output membership function (illustrated in the top-right graph).[38] The inference system then repeats the above process for the second rule, but damage level is now "light" and mileage is "low." Since the input membership degree of mileage being "low" is lowest with 0.4 (while the degree of damage being "light" is 0.75), the inference system sets the degree of sale price being "good" to 0.4 in the output membership function (illustrated in the middle-right graph).

After all rules are processed sequentially, the next step is to combine the results of the output membership functions into one fuzzy output set for "sale price." This is accomplished by overlapping the gray areas illustrated in the top-right graph and middle-right graph above. The result of the inference system is a fuzzy output set illustrated by the gray area in the following illustration (this is an enlargement of the bottom-right graph above):



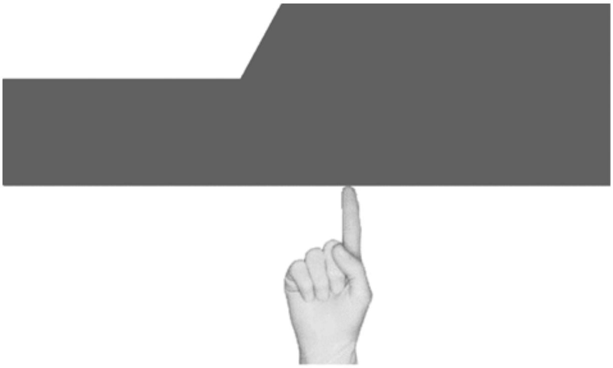
The question now is what to do with this gray area (i.e., fuzzy output set). Processing this fuzzy output set is the responsibility of the defuzzifier.

[37] *Fuzzy **and*** is an extension to the logical ***and*** in Boolean logic. If we denote *false* by 0 and *true* by 1, then using a *fuzzy minimum* for logical ***and*** would give the following results: *false* (=0) ***and*** *false* (=0) → *false* (=0), *true* (=1) ***and*** *false* (=0) → *false* (=0), *false* (=0) ***and*** *true* (=1) → *false* (=0), and *true* (=1) ***and*** *true* (=1) → *true* (=1). Hence, in the extreme (with "0"s and "1"s), fuzzy logic is reduced to Boolean logic.

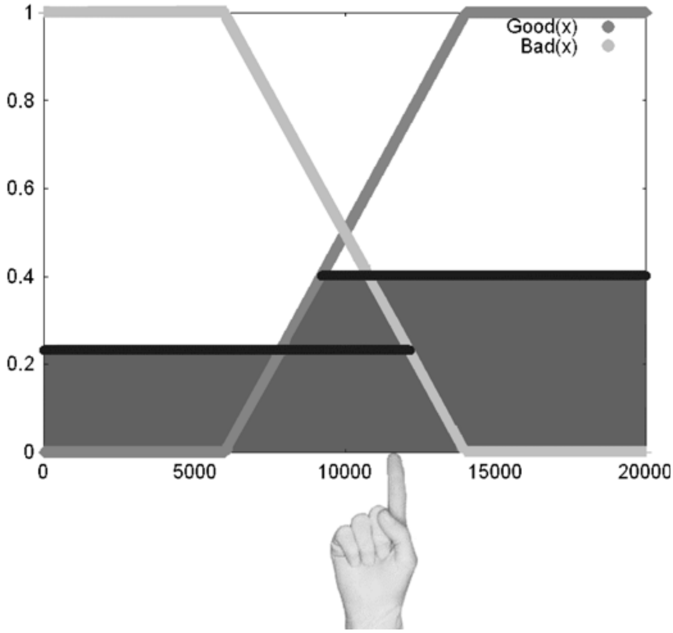
[38] 0.25 is equal to the degree of the damage level being "heavy," since that degree is the lowest of the two input membership degrees in the ***if*** part of the first fuzzy rule.

7.4 Defuzzifier

The *defuzzifier* takes the fuzzy output set from the inference system and converts it into a crisp output number. A defuzzifier can operate in a number of different ways, and one of the most common is the *center of mass defuzzifier*. It works in the following way: Imagine that the gray area (i. e., fuzzy output set) is a piece of wood that we have to balance on one finger:



The place where the finger touches the gray area is the *center of mass*. This defuzzifier calculates this exact spot, which results in the following defuzzification:



This illustration shows that the center of mass is 12,000 and so the defuzzifier would return a predicted sale price of $12,000 (i. e., crisp output number) for every car with "damage level" = 4 and "mileage" = 80,000.

7.5 Tuning the Membership Functions and Rule Base

Regardless of the methods used to construct the membership functions and rule base (e. g., through human expertise or data mining exercise), we need to tune them to get the best possible performance (e. g., in the car distribution example, this means minimizing the prediction error of the predicted sale price). Through the process of tuning, we can modify a few components of the fuzzy system. For instance:

- The output membership functions can be modified, while keeping the input membership functions static. For example, if we use triangular output membership functions, then we can adjust the triangles to get better predictions. This would make sense if we knew that the input membership functions were more or less perfect, or if the input membership functions are "given" (e. g., if they correlate to industry standards that we should follow).
- The input membership functions can be modified, while keeping the output membership functions static. This would make sense if we knew that the output is relatively static (e. g., when we have to make a binary classification).
- Both the input and output membership functions can be modified. This is the most general way to tune the rule base, and is typically the preferred choice. We would choose this form of tuning for the car distribution example, as both the input (e. g., car characteristics, such as new body styles or trim) and output (e. g., range of sale prices) can change over time.

If tuning the membership functions does not adequately reduce the prediction error, then we should consider making larger adjustments. For instance, we can add or delete some linguistic values in the existing rules, such as adding *"**and** color **is** dark"* to the ***if*** part of the rule ***if*** *damage level* ***is*** *heavy* ***and*** *mileage* ***is*** *high*, ***then*** *sale price* ***is*** *bad.* If we add a linguistic value to a rule, then the rule would become more specific; and if we remove such a value from an existing rule (e. g., dropping *"damage level **is** heavy"*), then the rule would become more general. If the prediction error is still unacceptable, then we should consider adding or deleting entire rules. After some rules are added or deleted, we can then perform the above steps to fine-tune the entire rule base.

The above steps for tuning the membership functions and rule base are presented in order of increasing "severity," beginning with relatively small tweaks of the membership functions to modifying entire rules. The level of tuning used depends on the size of the average prediction error, with larger errors typically requiring larger adjustments.

Note that the tuning process should be repeated in regular intervals (the frequency of which is always problem-dependent and can vary from a few hours to a few months), as the fuzzy logic implementation should adapt to changes in the environment (i. e., changes in the economy, makes/models, price ranges, etc.). This is the role of the adaptability module, which we will discuss in Sect. 10.3.

7.6 Recommended Reading

In this chapter, we discussed the various components of fuzzy logic: namely, the fuzzifier, inference system, and defuzzifier. Finally, we looked at how an adaptability module can be used to adapt an implementation of fuzzy logic.

Thinking in fuzzy logic terms is a very interesting experience: for example, when the traffic is slow, we should realize that "slow" is a linguistic value that has a degree of membership. Hence, the traffic is "slow" to a certain degree. When we are late for a meeting, "late" is not binary (even though our boss might tell us otherwise!); it is a linguistic value with a membership function. Hence, we are "late" to a certain degree. Finally, when we weigh ourselves and ask: "Am I too heavy?" We might realize that we are "heavy" to a certain degree between 0 and 1, as defined by some membership function in our head.

Many things in the real world are "true" to a certain degree. Someone can even be guilty of a crime to a certain degree and the judge might reduce the sentence. However, is *everything* a question of something being true to a certain degree? No, not everything is a question of degree. Some things in life are purely "true" or "false." For example, the statement "I won a million dollars in the lottery" is either true or false, because you cannot win a million dollars in the lottery to a certain degree (e. g., 0.0911). Nevertheless, the next time you ask your son if he broke the neighbor's window and he looks down at the floor and says "a little bit," just remember that he might actually be telling the truth. Then all you have to do is figure out what membership function to use for "a little bit."

For an easy introduction to fuzzy logic, we recommend *Fuzzy Thinking* by Bart Kosko (Hyperion, 1994). The book is a bestseller in the United States and it offers a philosophical perspective on fuzzy logic with hardly any functions. It introduces the concepts, origins, and reasoning behind fuzzy logic in an easy-to-grasp visual manner. This is a very easy book for beginners.

A more mathematical treatment of fuzzy logic can be found in the books *Neural Networks and Fuzzy Systems* by Bart Kosko (Prentice-Hall, Upper Saddle River, NJ, 1992) and *Adaptive Fuzzy Systems and Control* by Li-Xin Wang (Prentice-Hall, Upper Saddle River, NJ, 1994). Both books are written from an engineering point of view and do a great job of presenting all the aspects of fuzzy logic.

The book *Computational Intelligence PC Tools* by Russ Eberhart, Pat Simpson, and Roy Dobbins (Morgan Kaufmann, San Francisco, 1996) also provides an excellent introduction to fuzzy logic, neural networks and evolutionary algorithms. The concepts are presented through different examples and then generalized. The book also contains a very nice description of the different parts of fuzzy logic, along with an explanation of how they function.

8 Artificial Neural Networks

> "What one man can invent another can discover."
> *The Adventure of the Dancing Men*

> "Sherlock Holmes had pushed away his untasted breakfast and lit the unsavoury pipe which was the companion of his deepest meditations."
> *The Valley of Fear*

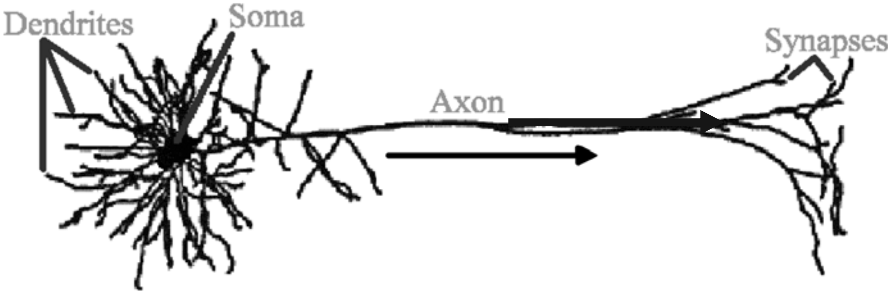
If you spend a significant amount of time trying to solve various business problems by "racking your brain," then you will inevitably come across the idea of "automating" your thinking process. The idea would be to simulate your brain functions on a computer so that it can solve problems for you. However, brains do not function in the same way as digital computers. First of all, biological processing is inherently and massively parallel in nature, while traditional computing is sequential (i. e., each step in an algorithm is processed "one at a time" until the termination condition is reached). Second, although conceptual similarities exist between the neurons in living brains and logic gates in computers, the firing rates of biological neurons are much slower than computer logic gates: milliseconds for neurons versus nanoseconds for computers. And, third, the response of a biological neuron is somewhat erratic and noisy (with misfirings or no firings at all), while a computer logic gate has very controlled "noise." Because of all these fundamental differences, we can conclude that different types of input-output devices can tackle different problems with different efficiency.

For example, because computers are excellent for quickly calculating arithmetic results, it is better to use a calculator rather then pen and paper for dividing 412.14823 by 519.442. By contrast, computers are not good at generalizing or handling conditions that fall outside the prescribed domain of possibilities. If one of your friends shaves his beard, you will probably still recognize him; but a computer would have considerably more trouble if it relies on a sequence of "if-then" rules that correspond to the identification of specific features of a person's face.

Does it have to be this way? Is this a fundamental restriction of computer processing? Or is it possible for computers to function more like biological neural networks? After all, a neural network is an input-output device. Hence, it should be possible to create models of how neural networks perform their input-output behavior and then capture this behavior in a computer. The resulting *artificial neural network* might yield some of the processing capabilities of living brains, while still providing the computational speed that can be attained in a computer chip. In this chapter, we will take a closer look at different artificial neural networks, and see how they fit into the context of Adaptive Business Intelligence.

8.1 Overview

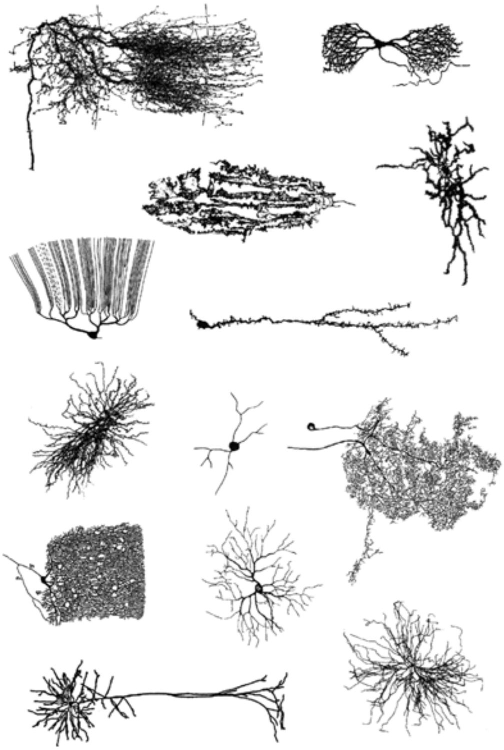
Living brains consist of a large number of different *neurons*,[39] and there are approximately 10^{14} neurons in a human brain. The following illustration is an example of a biological neuron:



A neuron's behavior is relatively simple: The incoming chemical activity feeds into the *soma* (the body) via the *dendrites*, and if the chemical activity overcomes a certain threshold, then the neuron sends an electrical spike down the *axon*. This spike triggers the "firing" of (chemical) neuro-transmitters at the *synapses*. Because the neuro-transmitters lie in close proximity to other neurons, the discharge creates a chemical reaction in the next set of neurons. The result of this relatively simple chemical-electrical behavior is responsible for the amazing achievements of the human race.

[39] Although the brain has thousands of different types of neurons, most of them behave in fundamentally the same way.

The above illustration is just one example of a neuron, and below are some other examples:

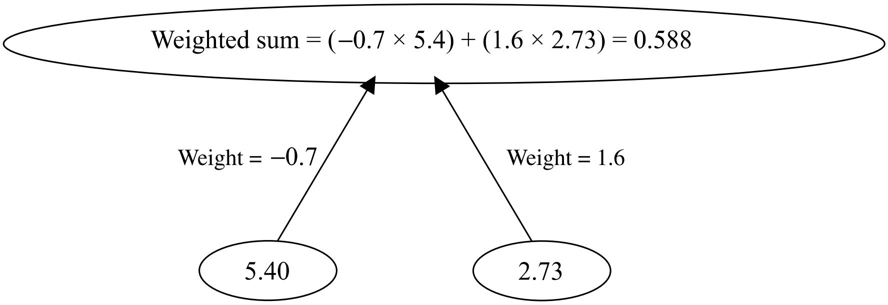


All these neurons are interconnected in a very complex manner. On average, each neuron in the human brain connects to 10,000 other neurons. Hence, the human brain is a complicated network of neurons with roughly 10^{18} connections,[40] and science has trouble explaining how the brain learns and performs its magnificent work. However, if you are reading this text, then your biological neural network is functioning properly and we can proceed with our discussion on artificial neural networks.

[40] Interestingly enough, recent research has shown that more neurons do not make us smarter. In other words, just because somebody has a big brain does not mean that he/she can process information more effectively. People with smaller brains can be smarter than people with bigger brains — size does not matter too much in this case.

8.2 Node Input and Output

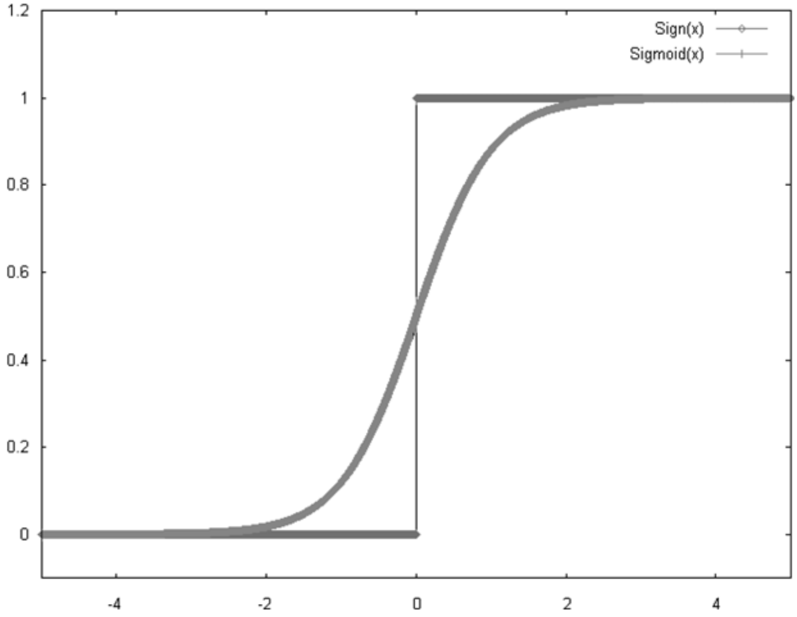
As mentioned earlier, neurons accumulate chemical input and create electrical output. In order to model a neuron in a computer the first step (most often) is to calculate a weighted sum of the input activity (i. e., the activity coming from other nodes):



This figure illustrates two nodes at the bottom (small ellipses) containing the values 5.40 and 2.73, respectively. The node at the top (big ellipse) calculates the weighted sum by multiplying the values of the input nodes with the associated *connection weights* (which are parameters that represent the strengths of the connections between nodes). Each arrow indicates the direction of the connection between two nodes, and the weights are floating-point numbers that assign a significance factor to each connection. Weights can be positive (exhibitory) or negative (inhibitory), which loosely resembles biological neurons and their connections.

Once the weighted sum is calculated, a node has to decide whether it should send an output signal. A *squashing function*[41] is often used to make this determination:

[41] A squashing function calculates the output of a node by taking the weighted sum as input.

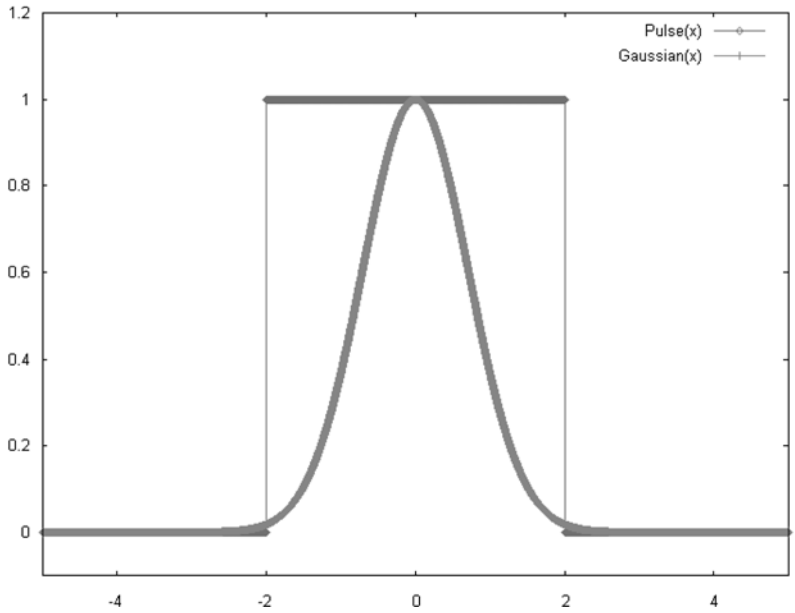


This figure shows the *sign* and *sigmoid* squashing functions. The horizontal axis illustrates the weighted sum of the input, while the vertical axis illustrates the output. As shown, the sign function generates an output when a specific threshold is crossed (0 in the above illustration). In other words, there is no output when the weighted sum is 0 or less, and there is an output (of strength 1) when the weighted sum is greater than 0. On the other hand, the sigmoid function is smoother than the discontinuous sign function, and the output value slowly rises toward the maximum output. In this example, the output values of the sigmoid function are:

- 0 for inputs less than −3,
- 0.5 for an input of 0, and
- 1 for inputs of 3 or more.

The sigmoid function is typically preferred over the sign function because it can be differentiated,[42] and some learning methods require differentiability (we will discuss learning methods in Sect. 8.4). Nevertheless, the sign and sigmoid functions are just two examples of how nodes can generate output; the following figure illustrates another type of output generation, where the nodes can create a spike instead of a plateau:

[42] In layman terms, a “differentiable” function is relatively smooth.



This figure illustrates the *pulse* and *Gaussian* squashing functions. Again, the horizontal axis illustrates the weighted sum of the input, and the vertical axis illustrates the generated output. The pulse function has a distinct area where it spikes (from −2 to +2 in the above illustration) to full strength of 1, and outside of this area the output is 0. The Gaussian also has a spike, but it is much smoother than the pulse function. In comparing these different squashing functions, the sign and pulse functions are conceptually very simple, returning either 0 (no output) or 1 (full strength), but are discontinuous and non-differentiable (which may exclude some learning methods). The sigmoid and Gaussian functions, on the other hand, are more complex, but they provide continuous output values.

8.3 Different Types of Networks

Fundamentally, there are two different types of artificial neural networks:

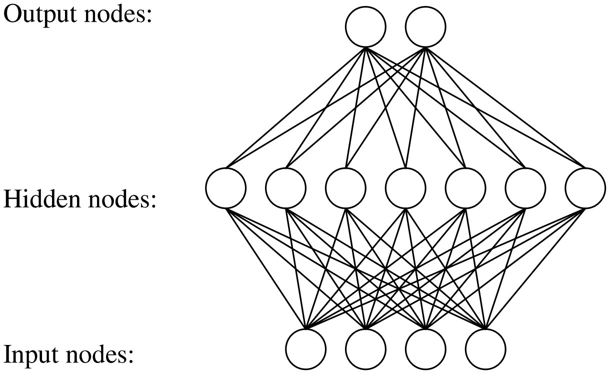
- *Feed-forward neural network.* This type of neural network has no recurrent connections between *nodes* (i. e., artificial neurons), and so the activity flows in one direction (i. e., the activity is fed forward step-by-step from the input nodes toward the output nodes). This type of neural network is most often used for function approximation and classification.

- *Recurrent neural network*. This type of neural network consists of a set of interconnected nodes, where the activity circles around in the neural network until it (maybe) settles down. This resembles a living brain in some ways (but is much simpler), and is typically used when data come in a stream (e. g., for spoken language processing, credit card fraud detection,[43] time series predictions).

Let us explore these two types in more detail.

8.3.1 Feed-Forward Neural Networks

The following figure illustrates a typical feed-forward neural network, which does not have any recurrent connections between nodes:



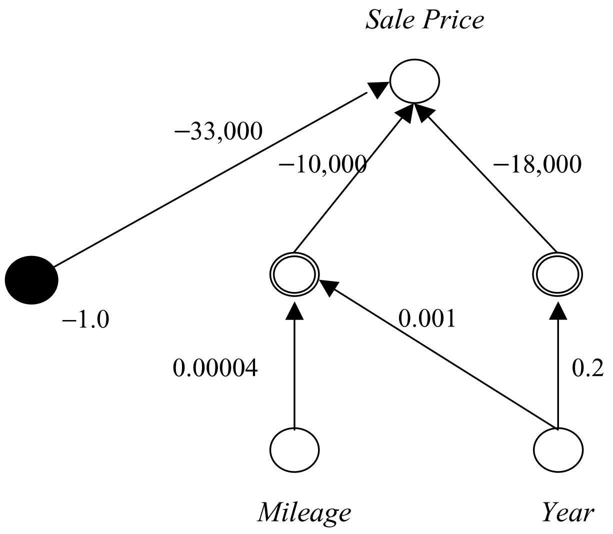
In this figure, there are three layers of nodes: The lowest layer is called the *input layer*, as these nodes accept the input values; the middle layer is called the *hidden layer*, as all the activity there remains hidden; and the top layer is called the *output layer*, as the activity that traversed the network finally arrives here.

Many different neural network architectures are possible, and the numbers of nodes in the input and output layers are usually determined by the problem at hand. However, the number of hidden layers and the connectivity between nodes and neighboring layers are design decisions that may vary from one implementation to the next. Thus, the number of possible architectures for any given problem is quite large. The illustration above depicts a "fully connected" neural network, as every node on a lower layer is connected with every node on the next level (but this need not be the case). Each of these connections also has its own weight. A feed-forward neural network that is fully connected and has assigned weights for all the connections "works" in the following manner:

[43] Most often, a considerable amount of authorized activity occurs before a card is lost or stolen and the fraudulent activity begins. The transition from authorized to unauthorized use needs to be detected in the stream of transactions.

1. The input values are fed directly into the nodes at the input layer.
2. The hidden nodes perform their calculations by summing up the weighted input received from the input layer nodes, and by applying their input-output squashing function to determine their output values.
3. Each output node determines its final output value by calculating the weighted sum using the values from the hidden nodes.[44]

Let us illustrate how a feed-forward neural network could be applied to the car distribution problem. If we use two variables as input (e. g. "mileage" and "year"), then our neural network could look like:



In the above example, the two hidden nodes (marked with double circles) use the sigmoid squashing function.[45] Rather than using a squashing function, the output node calculates its own weighted sum (using the black leftmost input unit with constant value −1.0 to calculate the *threshold* for the output node). The output of the above neural network can also be described mathematically as a weighted sum:

$$\begin{aligned}\text{Output} = {} & (-33{,}000 \times (-1.0)) \\ & + (-10{,}000 \times \text{Sigmoid}\ (0.00004 \times \text{Mileage} + 0.001 \times \text{Year})) \\ & + (-18{,}000 \times \text{Sigmoid}\ (0.2 \times \text{Year}))\end{aligned}$$

[44] Note that the "output" from the hidden nodes is the "input" to the output nodes.

[45] The exact function is: Sigmoid $(x) = 1/(1 + \exp(-x))$.

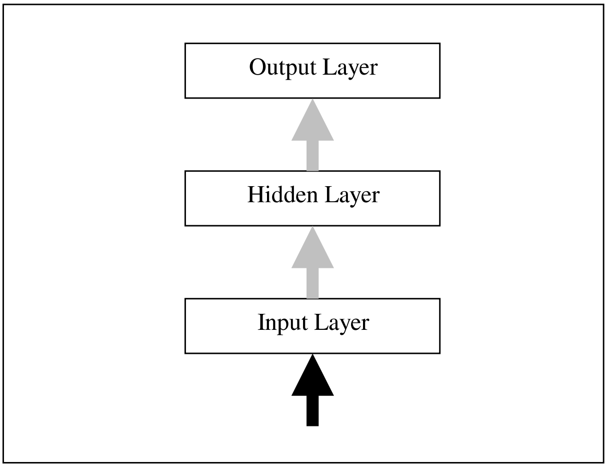
For this particular neural network, the following table shows the output values for different inputs of “mileage” and “year”:

Mileage	Year	Output
5000	1	17602.17
50000	1	14293.97
50000	5	11027.74

Not surprisingly, the predicted output (i. e., sale price) decreases as the mileage and year increase. Determining the various weights (such as 0.00004 between the “mileage” input node and the hidden node) occurs during the training process, which we will discuss in Sect. 8.4. Note also that no connection exists between the “mileage” input node and the rightmost hidden node. Here we can assume that there *is* a connection between these two nodes, but the assigned weight is 0.

Although this neural network has only a single hidden layer, more often than not additional hidden layers are used to improve the precision of the predicted output (in much the same way that more rules in fuzzy logic usually mean more precise predictions – as discussed in Sect. 7.3). The exact number of hidden layers, connections, and hidden nodes – as well as the input-output function of the nodes – is dependent on the problem at hand. This is why creating a successful neural network is still a “black art” and requires a considerable amount of experience …

When discussing various types of neural networks, it is convenient to illustrate the structure by showing how the different layers are connected. For example, the feed-forward neural network discussed earlier would have the following structure:



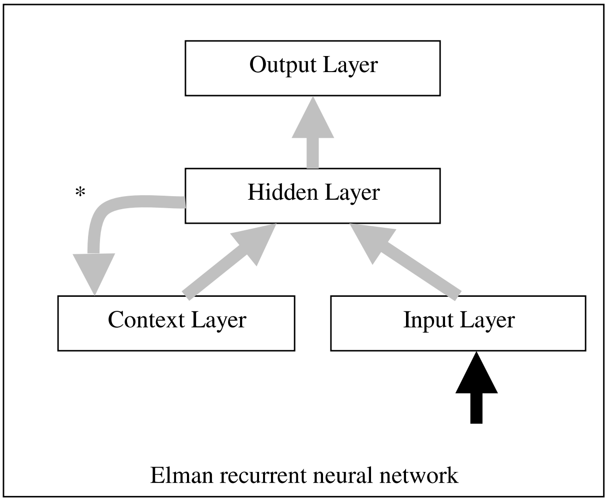
The bottom arrow indicates that the input values are fed into the nodes in the input layer. The middle arrow indicates that the activity is processed through the hidden layer toward the output layer. Once the nodes in the output layer have calculated their final output values, these values are (most often) used for prediction or classification.

While feed-forward neural networks are fast, precise, and able to generalize well, they are hard to understand because the learned features are "hidden" in the weights. For this reason neural networks are often referred to as "black boxes," because they do a great job of making predictions but nobody knows exactly how these predictions are made. Furthermore, because feed-forward neural networks can only process the entire input, one input at a time, they have no history/memory of earlier inputs, outputs, or processes. Recurrent neural networks, however, are able to overcome this limitation.

8.3.2 Recurrent Neural Networks

Recurrent neural networks have recurrent links between nodes, which means that the activity can "circle around." While this can create a situation where the activity never "settles down," it also opens the door for the concept of *memory*. This section illustrates a couple of recurrent neural network structures and explains how they operate.

The following figure illustrates the *Elman recurrent neural network:*



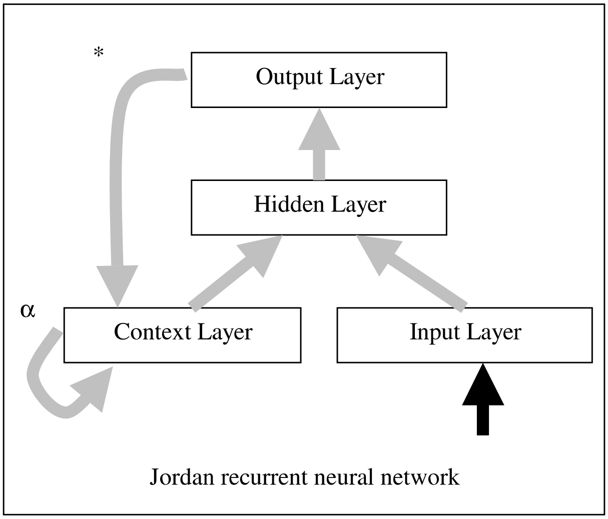
The Elman recurrent neural network operates in the following manner:

1. The input values are fed into the input layer.
2. The hidden layer calculates its activity using the input layer and the internal *context layer*, which serves as an internal memory that "remembers" the old hidden layer values.

3. The output values from the hidden layer are then copied into the context layer.[46] The asterisk ("*") next to the gray arrow indicates that the output values of the hidden layer are copied to the context layer, thereby overriding the old values.
4. The output layer calculates its final output values using the activity of the hidden layer.

The Elman neural network has *memory,* since the most recent output values of the hidden layer are available for further processing when the next set of input values are presented. Nevertheless, this only creates memory of the output values from the hidden layer, and not the final output values from the output layer.

The following figure illustrates another possible structure of a recurrent neural network, called the *Jordan recurrent neural network:*



The Jordan recurrent neural network operates as follows:

1. The input values are fed into the input layer.
2. The hidden layer calculates its activity using the input and context layers.
3. The output layer calculates its final output values using the activity of the hidden layer.
4. The context layer calculates its new values by using the new final output values[47] and the old final output values multiplied by the *decay rate* α.

Since the context layer is updated using the current final output values plus the "decayed" final output values, the hidden layer effectively contains all the

[46] Hence, the context layer is exactly the same size as the hidden layer (i.e., the number of nodes in the two layers is identical).

[47] The *decay rate* α is between 0 and 1.

former (decayed) final output values. Such an internal decaying memory is very useful for detecting patterns in streams of data.

Many other structures of recurrent neural networks can be created using one or more context layers. The memory in these neural networks is based on the decay of information using a decay rate. This decay of information allows recurrent neural networks to correlate two or more patterns that are separated in time (i. e., detect and predict output based on a stream of input data). Using the well-known *back-propagation* learning method, the training of such recurrent neural networks is quite fast (learning methods are discussed in the next section).

Lastly, rather than using context layers for memory, it is possible to use hidden nodes with recurrent connections that have their own adjustable weights. However, because the hidden nodes and recurrent connections can be arranged in almost an infinite number of ways, little is known about how to effectively structure or train such recurrent neural networks.

8.4 Learning Methods

A large number of different training methods have been developed to train, adjust, and update neural network models. Generally speaking, all of these methods fall into two major categories:

- *Supervised learning*. If we have data with both the input and output values (e. g., the characteristics of different cars and their actual sale prices), then we can apply supervised learning methods to train (i. e., adjust) the neural network.
- *Unsupervised learning*. If we have data without the output values (e. g., the characteristics of different cars that have not yet been sold), then we can apply unsupervised learning methods for clustering and data analysis.

In the remaining parts of this section, we will discuss both types of methods in more detail.

8.4.1 Supervised Learning

Supervised learning is typically used when the available data contain both the input *and* output values. As an example, recall that the input values for the car distribution example included VIN, make, model, body style, etc., and the output value was the sale price. Given all the data we possess about a particular car, plus the auction site location and the date of sale, we can build a neural network model for predicting the sale price. Let us assume for a moment that the actual sale price was $11,020 for a particular car at a particular auction site on a particular date, but our neural network model predicted that it would be $7,825. Obviously, our model made a significant prediction error, and so it needs to be adjusted/trained. This is most often done using the quadratic error function, i. e., the *least mean square error* (LMS error), such that:

$$\begin{aligned}\text{LMS error} &= 0.5 \times (\text{actual sale price} - \text{predicted sale price})^2\\ &= 0.5 \times (7{,}825 - 11{,}020)^2\\ &= 5{,}104{,}012\end{aligned}$$

The point of using the LMS error function is to *slightly* update the weights so that the predicted sale price would be closer to the actual sale price. Although the LMS error function is most commonly used to update the weights,[48] the most well-known method for training a feed-forward neural network is *back-propagation*. This learning method corrects the error at each layer by adjusting the weights, starting at the output layer and moving back toward the input layer.[49] The training process is typically repeated many times, until the LMS error is sufficiently low and the neural network has *learned* how to predict the sale price of all cars at all auction sites on all possible dates.

Let us illustrate this back-propagation method with a simple example. In Sect. 8.3, we illustrated the use of a neural network model for predicting the sale price of a particular make/model at a particular auction site. The feed-forward neural network accepted two variables as input, "mileage" and "year," and six weights were assigned to the different connections between nodes:

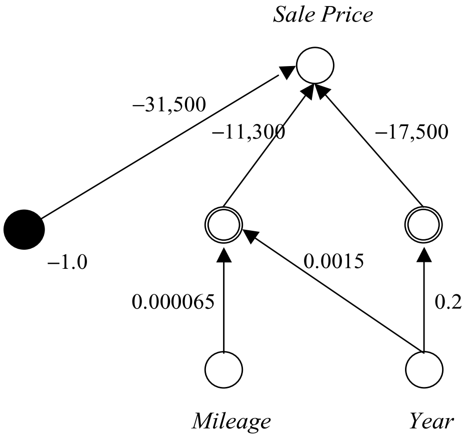
- The three weights between the input nodes and hidden nodes were 0.00004, 0.001, and 0.2.
- The three weights between the hidden nodes and the output node: −33,000, −10,000, and −18,000.

The question is: How did we determine these weights?

Assume that at some earlier stage of the training process (i. e., during the back-propagation process), the neural network had the following connections and weights:

[48] To apply the LMS error function, we have to calculate the "gradient" by differentiating the squashing function of the nodes. See suggested reading section for more details.

[49] The prediction of the sale price moves from input to output, while the error propagates in the reverse direction.



Assume further that for an input (say, "mileage" = 50,000 and "year" = 5) the model predicted a sale price of:

$$\begin{aligned}\$7{,}825 = {} & (-31{,}500 \times (-1.0)) \\ & + (-11{,}300 \times \text{Sigmoid}\ (0.000065 \times \text{Mileage} + 0.0015 \times \text{Year})) \\ & + (-17{,}500 \times \text{Sigmoid}\ (0.2 \times \text{Year}))\end{aligned}$$

and the actual sale price turns out to be \$11,020. Thus, the LMS error is:

$$\begin{aligned}5{,}104{,}012 &= 0.5 \times (\text{actual sale price} - \text{predicted sale price})^2 \\ &= 0.5 \times (7{,}825 - 11{,}020)^2\end{aligned}$$

We now have to update the weights between the output node and the hidden nodes. The output node does not use a squashing function, so the update rule is quite simple:

$$\text{weight}_{\text{new}} = \text{weight}_{\text{old}} + \alpha \times \text{error} \times \text{input}$$

where is the learning rate (larger values of α would result in larger adjustments of weights; in this example we assume $\alpha = 0.001$). So, the new weights for the connections between the output node and the hidden nodes are:

- $-31{,}500 + 0.001 \times \text{error} \times \text{input} = -31{,}500 + 0.001 \times 5{,}104{,}012 \times (-1) = \mathbf{-36{,}603}$
- $-11{,}300 + 0.001 \times \text{error} \times \text{input} = -11{,}300 + 0.001 \times 5{,}104{,}012 \times$ Sigmoid $(0.000065 \times \text{mileage} + 0.0015 \times \text{Year}) = \mathbf{-6{,}386}$
- $-17{,}500 + 0.001 \times \text{error} \times \text{input} = -17{,}500 + 0.001 \times 5{,}104{,}012 \times$ Sigmoid $(0.2 \times \text{Year}) = \mathbf{-13{,}769}$

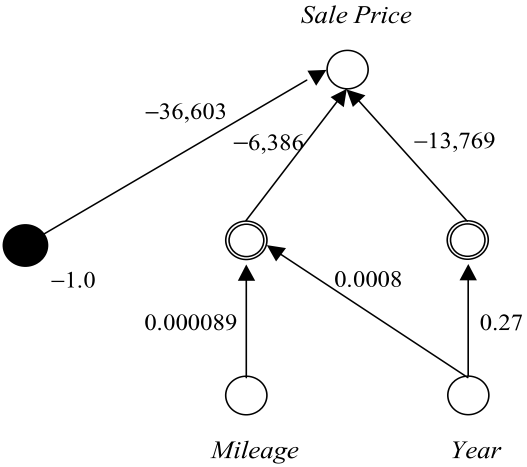
To update the weights of the hidden nodes, we use the same rule:

$$\text{weight}_{\text{new}} = \text{weight}_{\text{old}} + \alpha \times \text{error} \times \text{input}$$

However, the “error” is calculated differently, as the hidden nodes include squashing functions. Without going into the function details for error calculation, say the new values for these weights are:

- 0.000065 + 0.001 × error × input = **0.000089**
- 0.0015 + 0.001 × error × input = **0.0008**
- 0.2 + 0.001 × error × input = **0.27**

So, after a single iteration (e. g., after making adjustment for a single piece of data, where “mileage” = 50,000 and “year” = 5), the model was updated to:



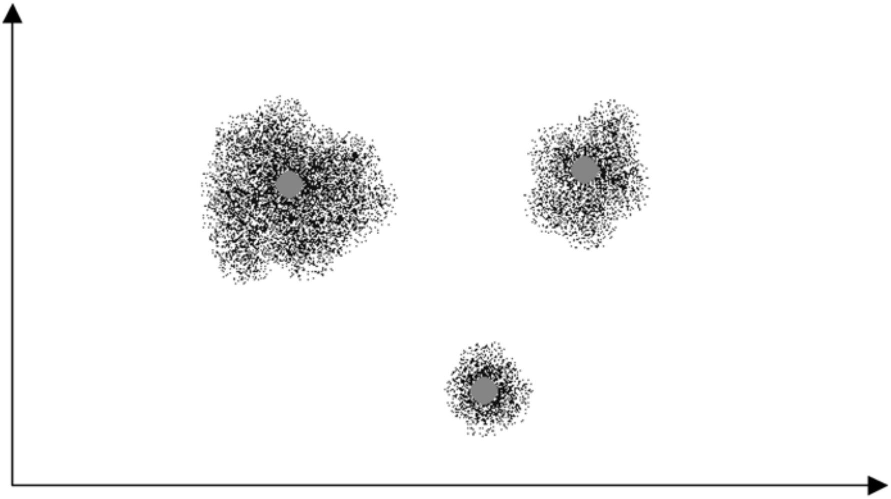
Several issues need to be addressed to make the training process work smoothly. First of all, we need to address the structure of the neural network. The fundamental question is: What kind of structure is optimal for a given problem? Although there are many methods and heuristics that try to construct the neural network layers and connections, no efficient way of selecting the optimal structure is known. Furthermore, because the training process is usually very slow (especially when multiple hidden layers are used), complicated learning methods have been invented to speed up the process.

Despite these difficulties, neural networks can make excellent predictions once they are properly trained, and can be easily retrained if additional data become available later. Neural networks also interpolate very well, especially if they do not contain too many nodes (which leads to *over-learning*, where the neural network simply *memorizes* the data). Lastly, if enough training data are available, neural networks can efficiently handle noise. In the car distribution example, it might happen that the actual sale price was incorrectly recorded and the resulting prediction error would constitute noise in the data.

8.4.2 Unsupervised Learning

Unsupervised learning is used in situations where only the input values are available (i. e., when there is no output value associated with the input values). As mentioned earlier, an example of this would be the characteristics of different cars that have not yet been sold. Assuming that no output values exist, we cannot predict the sale prices for these cars, but we can *cluster* the input values using unsupervised learning.

Imagine that the input values are distributed in the following manner:



Each data point can represent a car with certain variables (e. g. "color" and "mileage"). These data points form data clusters that display some degree of similarity, and the circles represent the cluster centers. Unsupervised learning can identify these cluster centers, which represent the "typical" values of the data clusters. Knowing the typical values makes it easier for us to spot atypical values (i. e., outliers) and make statements like "Most Toyota Camry cars are white with mileage in the range of 60,000 to 72,000, while most BMW 528i cars are silver with mileage in the range of 45,000 to 58,000." Such statements are very useful, as they can help us categorize new data points as either "typical" or "atypical."

The most well-known method for unsupervised learning is the *Kohonen* self-organized mapping method, where the data clusters are formed in such a way that they relate to each other in an organized manner. This method can be used to quickly identify the typical patterns within large data sets. Each "typical pattern" is (in simple terms) the average of the data points in a data cluster. Initially, the typical patterns are randomly distributed, but over several iterations of the learning process they end up in the center of the data clusters. "Typical patterns" are very useful for identifying typical car purchasing behavior, typical credit usage, typical health risks, etc. The Kohonen method resembles the well-known k-means clustering, but neural networks based on the Kohonen method learn differently. Moreover, they are inspired by biological evidence regarding how humans categorize data.

In general, unsupervised learning is used in many diverse areas, such as fraud detection, medical classifications, and categorization of consumer behavior. Since unsupervised learning identifies clusters of typical input value, it can also be used for signal compression of images and other electronic signals.

8.5 Data Representation

Neural networks are sometimes referred to as *universal approximators*, because of their efficiency in learning from data with an unknown underlying distribution. However, in order to construct an efficient neural network, some data representation issues need to be resolved first.

To start with, the typical input to a neural network is a vector of numeric values (such as –5.3425 or +7.935) and the output is another vector of numeric values. Note that the input does *not* include nominal values[50] like "Chevrolet," "yellow," "dog," "cat," etc. Simply assigning a numeric value to each nominal value would be disastrous, because neural networks cluster values that are numerically close together. If we randomly assigned numerical values to nominal values in the car distribution example, then we could have the following situation: "Chevrolet" = 1, "Accord" = 2, "Porsche" = 3, and so forth. Using only one numeric input, it would be very difficult for a neural network to separate these different car makes,[51] and, consequently, to accurately predict their sale prices. If we instead translated the cars into a Boolean-like vector using three inputs (where 1 is true and 0 is false) then we could express a "Chevrolet" as "1, 0, 0", "Accord" as "0, 1, 0," and "Porsche" as "0, 0, 1." Such a Boolean translation would make it significantly easier to train a neural network to predict sale prices, because different hidden nodes could *fire* more easily for different car makes.

Another issue is that of *high input dimensionality* (i. e., an extremely large number of inputs to the neural network). If we wanted to train a neural network to recognize faces, for example, we might assume that the task would be easy. After all, a high-quality camera has millions of pixels, and each pixel could be represented with three integers: one describing the intensity of red, another for blue, and the third for green. But millions of inputs require millions of connection weights, and each weight requires training data to correctly distinguish different examples. This results in the need for an *extremely* large number of training pictures, which is not feasible in practice since the training process would take too long. For this reason, a special form of "compression" is often applied to reduce the input dimensionality (the high-quality pictures are compressed by reducing the

[50] Nominal values are values without ordering or distance between them; see Chap. 5.

[51] Recall that the hidden nodes use a squashing function that makes it inherently difficult to separate nominal values.

resolution, applying retina preprocessing,[52] or using a Fourier transformation[53]). This compression significantly reduces the number of inputs to the neural network, which in turn reduces the number of connection weights, which in turn makes it more feasible to train the neural network within a reasonable time limit.

To summarize, we must keep the following things in mind about the input for a neural network:

- The input consists of numeric values.
- Numeric values that are "close" may be considered to "belong together."
- The input dimensionality must be reduced to a manageable size.

In addition to these input issues, the output also requires some special consideration – especially for classification tasks. In the facial recognition example, the output consists of a numeric value. Using a single node, we could assign specific output values to specific faces, but this would create immense difficulties for a neural network. Numerically close values would be regarded as "similar faces," making it difficult to correctly identify different faces. We can solve this problem by using a *Boolean output vector translation* (i. e., using a separate output node for each face). Once we let all the nodes complete their calculations, we could then select the node (and hence the face) that has the largest output value in a "winner-takes-all" fashion. Although this approach works well for a limited number of faces,[54] consideration has to be given to unknown faces.

8.6 Recommended Reading

In this chapter, we looked at the structure of artificial neural networks, how they are trained, and some possible applications. The relatively simple (but inherently parallel and distributed) computations that take place in neural networks are capable of impressive and robust performance. Tasks that are difficult for other techniques can sometimes be surprisingly easy for neural networks. Examples include the recognition of handwritten characters and numbers, faces, natural language, product defects, and fraudulent transactions.

Although artificial neural networks excel in many areas, they also inherit an undesirable characteristic of biological neural networks: they are *black boxes*. Artificial neural networks often perform very well, but we do not fully understand how or why. Although this is usually acceptable given the superior performance of a neural network, it can be a serious obstacle in some cases (where the reasoning behind a prediction or classification is required). New types of neural networks have been invented to partly or fully circumvent the black box issue, and can be

[52] *Retina preprocessing* reduces the effective resolution to the neural network by averaging the values of neighboring pixels.

[53] A *Fourier transformation* creates waveforms from a picture that are used as inputs into the neural network.

[54] This approach also works very well for the recognition of handwritten characters and numbers.

more readily understood by humans. Among these are *support vector machines, radial basis function neural networks,* and *neuro-fuzzy networks.*[55]

The book *Introduction to the Theory of Neural Computation* by John Hertz, Anders Krogh, and Richard G. Palmer (Addison-Wesley, Redwood City, CA, 1991) gives a good description of the different types of artificial neural networks. Although it can be challenging at times for those who are less technical, it leaves hardly any stones unturned and is highly recommended by the authors.

For a very inspiring book that deals with the relationship between biological and artificial neural networks we recommend *The Computational Brain* by Patricia S. Churchland and Terrence J. Sejnowski (MIT Press, Cambridge, MA, 1992). This great book describes many biological findings and explains them in algorithmic form. The book is easy to understand for most people who are new to the subject of biological-artificial neural networks. It explains much of the waste complexity of real biological brains and neurons, and the parallels with artificial neural networks.

The book *The Elements of Statistical Learning* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (Springer, New York, 2001) provides a solid mathematical explanation of the many different aspects of learning systems. The terminology and approach is statistical in nature, but with an emphasis on concepts rather than (too much) mathematics. It describes the many aspects of data, neural networks, support vector machines, supervised learning, unsupervised learning, and how to combine different prediction and classification systems. This is an excellent treatment of the entire subject of learning systems, but it is not recommended for readers who are not mathematically inclined.

[55] We have only presented the most common artificial neural networks, and many other structures (not mentioned in this chapter) are also possible.

9 Other Methods and Techniques

> "Mediocrity knows nothing higher than itself; but talent instantly recognizes genius."
> *The Valley of Fear*

> "… the quick inference, the subtle trap, the clever forecast of coming events, the triumphant vindication of bold theories – are these not the pride and the justification of our life's work?"
> *The Valley of Fear*

Thus far, we have discussed a variety of prediction methods and optimization techniques that have ranged from decision trees and hill climbers, to neural networks and evolutionary algorithms. We will now conclude Part II of this book with a review of some additional methods and techniques that can be applied to prediction and optimization problems. We moved these methods and techniques into a later chapter because they generate solutions that are considerably more complex than the static vectors discussed in Chap. 6. Furthermore, some of these methods (such as agent-based modeling) use an evaluation function that measures the "behavior" of a solution to determine its quality measure score. As a result, we can use some of these methods to observe an emergent behavior that would be difficult (if not impossible) to predict using other (more traditional) methods. This emergent behavior is one of the themes of this chapter …

9.1 Genetic Programming

Many real-world business problems require the discovery of some mathematical function. For example, we may have a data set with cases that contain both input and output values, and we would like to discover a function that can be used for predicting the output for future cases. Note that the formulation of this problem is quite different. We are not searching for a solution that is a vector of numbers, but rather for a solution that is a function.

Let us consider the following example: Say we have a collection of sales data for a particular product over some period of time, and we would like to discover a function that describes the relationship between sales volume, price, and marketing expenditure. In other words, we would like to discover a function *F:*

$$\text{Volume} = \text{F}\,(\text{Price, Marketing})$$

Of course, the above example is simplified (e. g., the marketing expenditure has a delayed effect on sales volume), but it will suffice for illustrating how genetic programming works. The typical approach to solving this problem is based on assuming the general structure of the function F and then directing our effort to tuning some parameters. For example, we can assume that the relationship between sales volume and the other two variables has the following structure:

$$\text{Volume} = (\text{a} \times (\text{Marketing} / \text{Price})) + (\text{b} \times (\text{Marketing})) + \text{c}$$

where *a, b,* and *c* are (unknown) parameters. This function states that the sales volume increases as the marketing expenditure increases and/or the product price decreases. The only issue then is to find parameters *a, b,* and *c* that minimize the prediction error for the historical data. And here we have many possibilities: we can use simulated annealing or evolutionary algorithms to find the optimal vector of these three numbers *a*, *b*, and *c*. However, the main weakness of this approach is that we *first* have to make an educated guess about the general form of the function. If our guess is correct (i. e., close to the real function), then tuning the parameters will be quite straightforward. On the other hand, if our guess is incorrect, then we will spend our time tuning the parameters of the “wrong” function … For example, it might be that the function we are searching for is:

$$\begin{aligned} \text{Volume} = {} & (12.3 \times (\text{Marketing}^{2} / \text{Price}^{2})) + (3.7 \times (\text{Marketing}^{2} / \text{Price})) \\ & + (2.9 \times (\text{Marketing} / \text{Price}^{2})) + (21.8 \times (\text{Marketing} / \text{Price})) \\ & + (8.7 \times (\text{Marketing}^{2})) + (3.3 \times \text{Marketing}) + 1346 \end{aligned}$$

Again, finding all the parameters (12.3, 3.7, etc.) is easy once we know the general form of the function. However, even in this very simple example, we can consider an almost countless number of possible functions! By using genetic programming, we do not have to make assumptions on what the correct function might look like. Genetic programming allows us to search the space of possible functions for one that fits the problem at hand – in particular, we may search for the best function that calculates the sales volume on the basis of the marketing expenditure and product price … How can we do that? Well, let us have a look.

First of all, genetic programming is a special type of evolutionary algorithm, and so many of the same concepts apply: There is a population of individual solutions (i. e., functions) that compete for a place in future generations and for the placement of their offspring solutions. The process of evolution is simulated and the best function emerges after some number of generations.

As we discussed in Chap. 6, we must follow some steps when applying the evolutionary approach to a specific problem. In particular, we have to design the “structure” of an individual solution, select an evaluation function that measures the quality of each solution, and decide upon the parameters (e. g., population size, probabilities of various crossover and mutation operators). Similarly, we must follow some steps when applying genetic programming, which are:

1. Selecting the set of *terminals.* These are all the variables, parameters, etc. that correspond to the inputs of the function. In our example, the set of terminals consists of the marketing expenditure and the product price, as well as a set of real numbers (e. g., 17.4).
2. Selecting the set of *primitive functions.* These are usually standard arithmetic operations like addition or subtraction, standard mathematical functions like log or square root, or domain-specific functions. In our example, the set of standard arithmetic operations (addition, subtraction, division, and multiplication), extended by the square root function, would be sufficient.
3. Selecting the evaluation function. This is the key decision that ties the genetic program to the problem at hand. The evaluation function evaluates how well a particular function solves the problem. In our case, the evaluation function has to estimate how well the developed function describes the relationship between sales volume, marketing expenditure, and product price (the smaller the error on historical data, the better the fit). Of course, the error is measured on many data points and is often expressed as a total of absolute errors on all data points.
4. Selecting the parameters of the genetic program. These would include the population size, number of generations, probabilities of various operators, and possibly some other parameters (e. g., parameters that influence the selective pressure of the algorithm: the higher the selective pressure, the smaller the chances that weaker individuals will be selected for reproduction).

Let us take a closer look at some individual solutions that can emerge during a simulated evolutionary run of a genetic program. Assume that at some generation (say, generation 215), one of the solutions in the population (of 500 individual solutions) is:

Volume = ((Marketing × 21.8 × Marketing) / (Price × Price))
– (3.7 × Marketing × Price) + (2.9 × (sqrt(Marketing)) / Price) + 1192

To facilitate the discussion of crossover and mutation operators, we can represent this individual solution as a tree. A particular solution can be represented by many different trees. For example, we can view the above solution as a sum of two parts:[56]

((Marketing × 21.8 × Marketing) / (Price × Price)) – (3.7 × Marketing × Price)

and:

(2.9 × (sqrt(Marketing)) / Price) + 1192

Thus, the root of the tree (the uppermost node) represents addition (+), and the first part of the solution is a subtraction between two subparts:

[56] We can also view the above solution as a subtraction of two parts: *(Marketing × 21.8 × Marketing) / (Price × Price))* and *(3.7 × Marketing × Price) + (2.9 × (sqrt(Marketing)) / Price) + 1192.* In this interpretation, the root of the tree (the uppermost node) would represent subtraction (–).

(Marketing × 21.8 × Marketing) / (Price × Price)

and:

(3.7 × Marketing × Price)

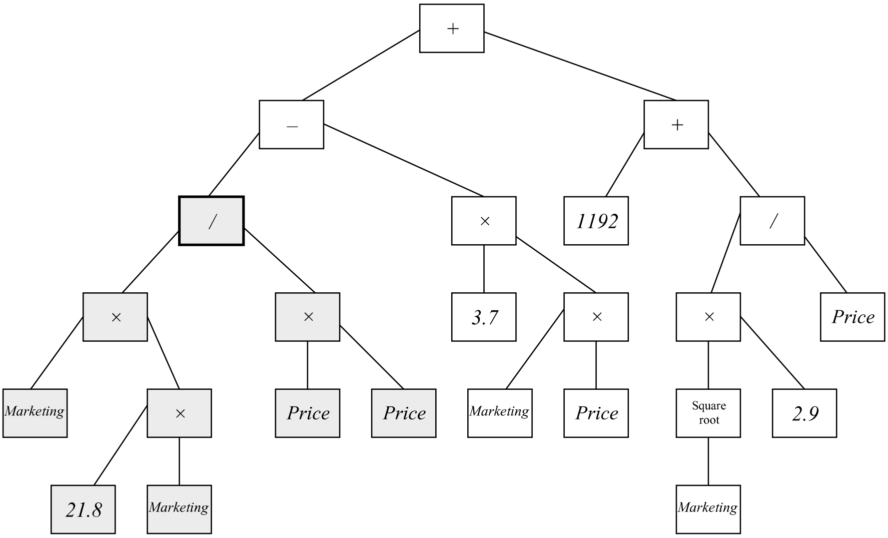
This is represented by the appropriate node (–) in the left sub-tree. This process then continues further: the first subpart is a division, where the enumerator is:

(Marketing × 21.8 × Marketing)

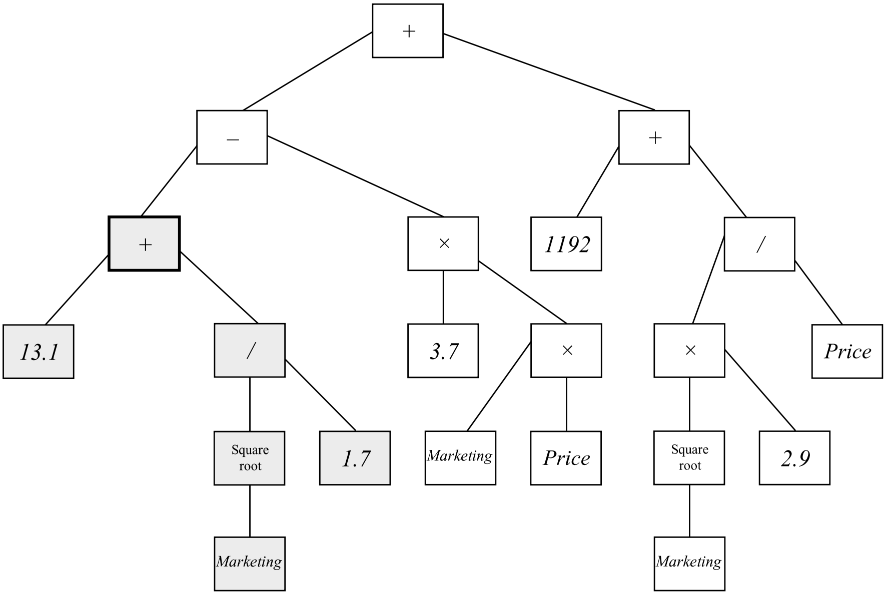
and the denominator is:

(Price × Price)

Hence, the next node down the tree represents division (/). The correspondence between the original solution and the tree below should now be straightforward:



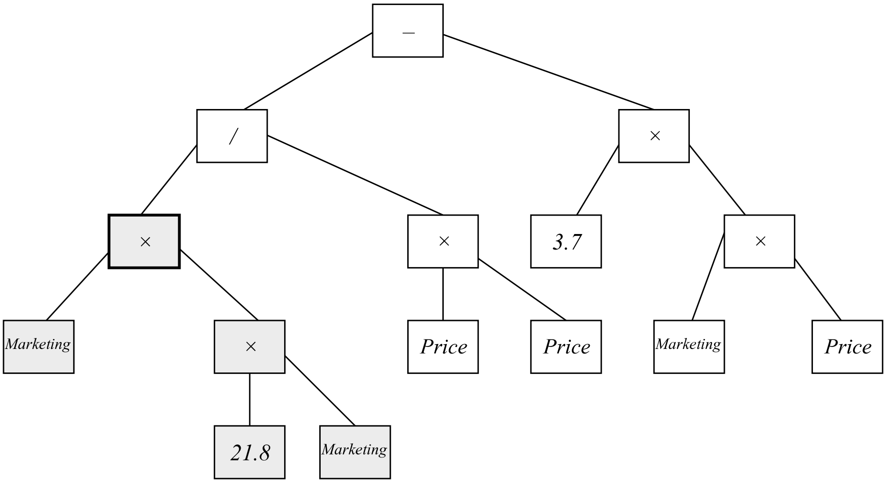
The above individual solution, which emerged in generation 215, will probably be modified further in subsequent generations. For example, assume a mutation operator was applied at the node with a bold outline (this node represents the division operator "/"). In further generations, a randomly generated sub-tree may replace this part of the tree with the final result being:



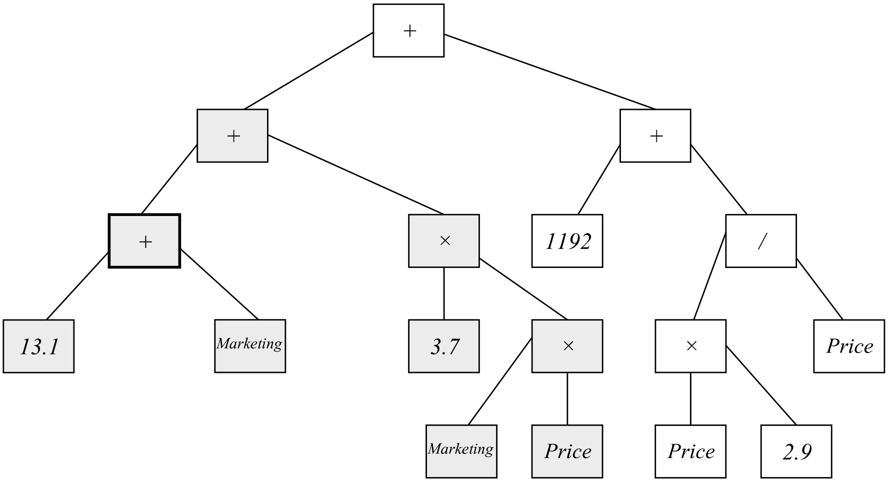
This individual solution corresponds to the following function:

Volume = 13.1 + (sqrt(Marketing)/ 1.7) – (3.7 × Marketing × Price)
+ (2.9 × (sqrt(Marketing))/ Price) + 1192

Similarly, individuals may undergo the crossover operator, where a sub-tree from one individual solution is swapped with a sub-tree from another solution. In other words, the crossover operator creates two offspring solutions by exchanging two sub-trees from two different parent solutions. For example, if the first parent is:

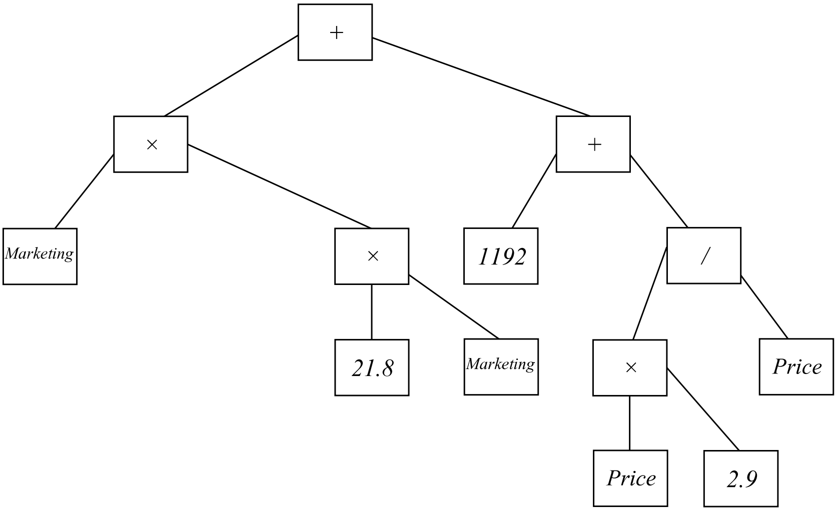


representing the function Volume = ((Marketing × 21.8 × Marketing) / (Price × Price)) – (3.7 × Marketing × Price), and the second parent is:



representing the function Volume = 13.1 + Marketing + (3.7 × Marketing × Price) + 1192 + ((2.9 × Price) / Price), then the first offspring is:[57]

[57] Note that the node with a bold outline marks the cutting points for the crossover in both parents. As for the mutation operator, these cutting points represent the "starting point" of the sub-tree that is cut off and replaced by a sub-tree from the other parent.



The other offspring is created by “the other” swap (i. e., where the “gray” sub-tree of the first parent is replaced by the “gray” sub-tree of the second parent). Anyhow, the first offspring represents the following function:

$$\text{Volume} = ((21.8 \times \text{Marketing} \times \text{Marketing}) + 1192 + (2.9 \times \text{Price}) / \text{Price})$$

There are many issues to deal with when experimenting with genetic programming. First of all, there is a tendency to get progressively more complex structures (trees), so the functions might be quite complex and difficult to interpret. The very simple function above, which emerged as an offspring solution after the crossover operator was applied, is an exception, rather than a rule. Second, there might be parts that are meaningless in the current function (like the division *Price / Price* above). Furthermore, most applications of genetic programming require relatively large population sizes.[58]

Recall that the evaluation function is responsible for measuring how well a particular solution solves the problem. In our case, the evaluation function has to estimate how well the developed function describes the relationship between sales volume, marketing expenditure, and product price (the smaller the error on the historical data, the better the fit). Thus, it should be relatively easy to determine the relative merit of all the individual solutions in the population. Again, as with other types of evolutionary algorithms, the quality measure score is used in the selection process of parents, and the mutation and crossover operators are with some probability. All the other mechanisms of evolutionary algorithms apply here as well. Thus, the genetic program generates subsequent iterations of individual solutions (in our case, a function for sales volume) by creating and evaluating an

[58] There are also some ways of introducing problem-specific knowledge into a genetic program, making the evolutionary process more efficient. However, this topic is beyond the scope of this book.

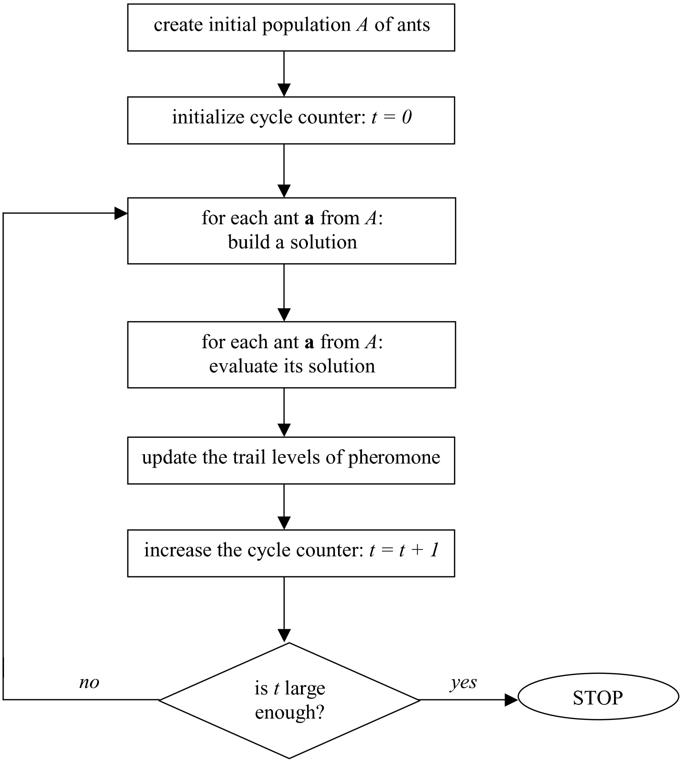
initial population of individual solutions, selecting parents, applying the mutation and crossover operators, selecting a new population from the existing parent and offspring solutions, evaluating all the solutions in the new population, and continuing this process for some pre-specified number of generations.

There are many possible applications of genetic programming. For the car distribution problem, one of the possibilities is to develop a genetic program for price prediction. After all, the predicted price of a car is a function of some variables ("make," "model," "body style," etc.), the auction site, and the estimated time of sale. Furthermore, it should be relatively easy to evaluate the precision of the functions that emerge during the simulated evolutionary process, as we have historic data at our disposal. Thus, genetic programming can be an important component of an Adaptive Business Intelligence system for some problem domains.

9.2 Ant Systems and Swarm Intelligence

Ant systems (also known as *ant colony optimization*) were inspired by colonies of real ants, which deposit a chemical substance (*pheromone*) on the ground. This substance influences the "behavior" of individual ants, as the greater the amount of pheromone deposited on a particular path, the larger the probability that an individual ant will select that path. Artificial ants behave in a similar way. In a nutshell, the ant colony optimization technique is a multi-agent system, where low-level interactions between artificial ants results in a complex behavior of the entire ant colony.

Ant systems are another population-based technique, much like evolutionary algorithms. In evolutionary algorithms, the parent solutions are modified through some operators (e. g., mutations, crossovers) to create offspring solutions; in ant systems, however, the pheromone levels influence the creation of new solutions. The general idea behind ant systems is provided in the following flowchart:


create initial population *A* of ants
initialize cycle counter: *t* = *0*
for each ant **a** from *A*: build a solution
for each ant **a** from *A*: evaluate its solution
update the trail levels of pheromone
increase the cycle counter: *t* = *t* + *1*
is *t* large enough?
no
yes
STOP

Most of the action boxes are self-explanatory: We create a population of ants, initialize each ant (i. e., set its various parameters), and then set a cycle counter to repeat the process several times. Now, in the loop above there is a central action box where each ant is responsible for *building* a solution. The process of building the solution is usually influenced by two factors: (a) problem-specific knowledge, and (b) decisions made during the previous cycles (these decisions are summarized by the current *pheromone* levels). Once the new solutions are ready, we can evaluate them, update the trail levels (thus summarizing the "popularity" of the various decisions made by the ants), increase the cycle counter, and then repeat the process. Through this loop, the updated trail levels would influence the future decisions of the ants, and better decisions made in previous cycles would be reinforced in future cycles!

So, what is an ant? In simple terms, an ant is a computer "agent" responsible for making decisions in the process of building new solutions, and each ant may have some parameters that influence its decisions. For example, it is typical for an ant to have parameters for controlling the relative importance of the pheromone trail versus problem-specific knowledge when a new solution is being built. In the car distribution example, these parameters may represent a trade-off between *visibility,* which states that closer auction sites should be chosen with a higher probability, and *trail intensity,* which states that if a particular auction site enjoys a lot of "traffic" for a particular car, then it must be a profitable path to follow. It is also common for each ant to have a memory structure that records earlier decisions. This data structure (sometimes called a *tabu list*) is useful in avoiding the construction of infeasible solutions.

As in Chap. 6, let us illustrate the concept of ant systems by using the car distribution example. As before, let us assume that the same representation is used (i. e., a vector of 3,000 values provides indices of auction sites 1 to 50). During the first stage of the algorithm, we initialize our ant colony (say there are 100 ants in our implementation). Each ant would be responsible for building a solution in each cycle of the loop. A solution, as discussed earlier, is a vector of 3,000 integer numbers, so an ant would generate a sequence of 3,000 integers. How can this be done?

As mentioned earlier, ants possess problem-specific knowledge. In our case, we can provide a list of available auction sites for each car (if there are no restrictions, then all 50 auction sites might be available), along with the distances between each car and auction site. Because no pheromone deposits exist at the beginning of the algorithm, problem-specific knowledge is the only factor ants take into account when building new solutions. To illustrate this process, let us assume that we have constructed an "attractiveness vector" for each car. This vector provides us with problem-specific knowledge (based on distances) of all the auction sites. For the first car in our solution vector, the attractiveness vector might look like:

0.000964	0.004739	0.001065	…	0.001197	0.003897

This attractiveness vector is only applicable to the first car, and the length of the vector is 50 (as there are 50 auction sites). If some auction sites are unavailable (for example, due to distance limitation), then the corresponding values would be zero. To arrive at these values, an intuitive heuristic to use might be "closer auction sites are more attractive." Let us assume that the distance between the first car and the first auction site is 1,037 miles, the distance between the first car and the second auction site is 211 miles, the distance between the first car and the third auction site is 409 miles, and so on. Now we have to convert these distances into measures of "attractiveness." Because attractiveness is inversely proportionate to distance, we can take the inverse of the distance to measure the attractiveness of an auction site. Hence, the attractiveness of the first auction site is 1/1,037 = 0.000964, and the attractiveness of the second and third auction sites are 1/211 = 0.004739 and 1/409 = 0.002445, respectively. Similar vectors must be constructed for the second car, third car, etc. Note that this preprocessing would define the attractiveness vectors (with each vector containing 50 values) for all 3,000 cars, making life much easier for our ants …

Another preprocessing activity is connected with the initial distribution of pheromone. Since the ants have not yet started, the pheromone levels would be zero (or close to zero). The data structures used for recording of pheromone levels are similar to those used for recording the attractiveness values. For each car, we maintain a vector of pheromone levels for each auction site. Initially, all these vectors might be identical and contain very small values (close to zero), as there is no ant activity at this stage. The pheromone vector for the first car may be:

0.000001	0.000001	0.000001	…	0.000001	0.000001

This pheromone vector can be interpreted in the following manner: The pheromone level for the connection between the first car and the first auction site is 0.000001; the pheromone level for the connection between the first car and the second auction site is 0.000001; and so forth. Later, we will see how these values change (as opposed to the values in the probability vectors), and how they influence the ants' decisions. They are of no importance at this stage, but that will change very soon.

Now, each ant has access to this preprocessed information, and at each cycle of the loop, an ant would create a new solution that consists of 3,000 decisions. All these decisions are based on the attractiveness values and the pheromone levels. Hence, the final issue to consider is the relative importance of these two measures. Several simple methods can be used to emphasize the importance of one measure against the other (e. g., by adding a small constant, raising a value to some power), but we will not discuss these in detail. It is sufficient to say that each pair of values (attractiveness and pheromone) is converted into a probability (this would require normalization, such that the total of all these probabilities – i. e., for all auction sites – would be 1). Therefore, after such normalization, the probability vector for the first car might be:

0.006035	0.029668	0.006667	…	0.011975	0.038979

Clearly, this probability vector is of great assistance: For the first car, the probability of selecting the first auction site is 0.006035; the probability of selecting the second auction site is 0.029668; and so on. It is relatively straightforward to select the auction sites according to this (or any other) probability distribution. Say, the first ant "selected" auction 23 as the destination for the first car, auction 12 for the second car, etc. and the solution vector built by the first ant is:

23	12	5	…	19	41

All auctions sites were selected in accordance with a distribution probability that favors closer auction sites with higher levels of pheromone. However, because this is the first cycle of the algorithm, the pheromone levels for influencing the ants' decisions are close to zero.

This process is repeated 100 times (because there are 100 ants in our implementation), with each ant making an independent decision on the auction site for each car. These decisions are based on selection probabilities, which are calculated on the basis of attractiveness (related to the distance between the car and auction site) and pheromone levels (related to the popularity of an auction site). At this stage, each ant has built its own solution, and we have a population of 100 solutions. The evaluation process is the same as for any other technique discussed in this chapter, with each solution receiving a quality measure score.

Now it is time to update the pheromone levels. Let us consider a particular value p, which represents the pheromone level for some car and some auction site. At the end of the cycle, when all the ants have built their solutions, there are two independent processes for updating the value p: *evaporation* (which decreases the value of p) and *accumulation* (which increases the value of p). The first process is responsible for reducing the pheromone values of less popular auctions, and the second process is responsible for increasing the pheromone values of more popular auctions.

We can easily implement the evaporation process by multiplying the original value p by a number smaller than 1 (e. g., assume that this number – known also as the *evaporation rate* – is equal to 0.9). This means that if, at the beginning of some cycle, $p = 0.087945$, then, after evaporation, $p = 0.079151$ (0.9 × 0.087945). The accumulation process, on the other hand, is a bit more complex. We have to measure the "popularity" of each auction site (for a given car) across all 100 ants. Moreover, better ants (i. e., ants that found better solutions) should be able to exert greater influence in this popularity contest. This can be done in the following way: Assume that each ant has some amount of pheromone, which is proportionate to the quality of the solution built. For example, we can award each ant 0.01 pheromone units per unit of the quality measure score. Thus, two ants that build two different solutions that evaluate to 176.23 and 191.07 would get 1.7623 and 1.9107 units of pheromone, respectively, for allocation to each auction site. If these two ants created the following solutions:

13	29	36
41	12	36

22	18
11	25

then the first ant would increase the pheromone level by 1.7623 for the first car and auction 13; by 1.7623 for the second car and auction 29; by 1.7623 for the third car and auction 36; etc. Similarly, the second ant would increase the pheromone level by 1.9107 for the first car and auction 41; by 1.9107 for the second car and auction 29; by 1.9107 for the third car and auction 36; and so on. Because both ants selected auction 36 for the third car, the increment of pheromone levels would be quite significant (1.7623 + 1.9107). Some auction sites would be selected several times (getting significant increments in pheromone levels), while others might not be selected at all. The latter ones would experience evaporation, so their chances of being selected in future cycles would be smaller. The updating of pheromone levels is an essential part of the ant system, as better solutions influence the probability of selecting the more promising parts of a solution in future cycles.[59] After many cycles, some ants should build a near-optimum solution to our problem.

This distribution and updating of pheromone levels is a "communication" process that is essential for the convergence of the ant system: the ants in the population become more and more similar in their behavior, as they *swarm* toward the optimum solution. Other related techniques also use this "swarming" phenomenon. For example, *particle swarm optimization* applies some variation operators (like mutation and crossover in evolutionary algorithms) to a population of agents, but without any selection process, as all agents are in constant motion and they "live" forever. In particle swarm optimization, the concept of "generation" is replaced with that of "iteration." Within a multi-dimensional matrix, each individual agent has a location and velocity that is updated according to the relationship between the agent's parameters and some other global parameters (e. g., the location of the best individual solution found so far). The search is biased toward better regions of the matrix, with the result being a sort of "flocking" (i. e., swarming) toward superior solutions. As in ant systems, the agents exchange information through some global medium that collects information on the locations and velocities of the particles, processes this information (e. g., selects the best particles in the current iteration, or selects the best locations found so far), and disseminates this information to other particles, influencing their subsequent direction and velocity.

In general, there are more and more applications of the "social insect metaphor" for solving problems. These approaches are often called *swarm intelligence systems*. As with ant systems and particle swarm optimization, swarm intelligence systems assume the presence of a number of simple agents (e. g., ants, bees, wasps, termites) with direct or indirect interactions/communications that influence their future behavior. There are many possible applications of swarm intelligence for problems in distribution, communication networks, robotics, etc. In all these cases, the underlying principle is the same: Each insect is an "independent individual" performing some "individual activities" (often specialized activities); however, all these activities *seem* very well organized, without any outside organizer (or supervisor). This "self-organized behavior" is the essence of systems based on swarm intelligence.

9.3 Agent-Based Modeling

During the past decade, there has been growing interest in *agent-based modeling.* Agent-based models are "behavior-based" computer programs that try to simulate complex phenomena through virtual "agents." The behavior of these agents is determined by programmable rules that reflect the constraints and conditions of a real-world system. Because agent-based modeling has its roots in the *Monte*

[59] This process is similar to the "selection pressure" in evolutionary algorithms.

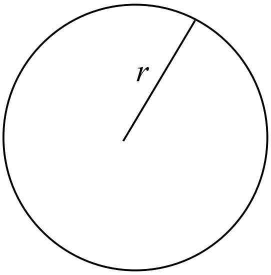
Carlo method, which now has a 60-year history, let us take a step back in time and discuss this classic method before moving on to agent-based modeling.

The idea behind the Monte Carlo method is to use statistical sampling to approximate a solution for some quantitative problem. The term “Monte Carlo method” is quite general and the method has universal applicability to a variety of problems in economics, environmental sciences, nuclear physics, chemistry, logistics, etc. Another popular term synonymous with the Monte Carlo method is *Monte Carlo simulation*. In general, the word *simulation* is defined as the imitative representation of the functioning of one system or process by means of the functioning of another (e. g., a computer simulation of an industrial process). Simulations are useful when other types of analysis are too difficult (e. g., they require solving thousands of differential equations). Many models include variables that have a known range of values, but an uncertain value for any particular time. This is true for the vast majority of economics problems (e. g., interest rates, currency exchange values, stock prices), logistics (e. g., inventory levels), etc. In such cases, the Monte Carlo simulation might be of assistance. The idea behind the Monte Carlo simulation is quite simple: By sampling the values of a model’s variables from their (predefined) probability distributions, many scenarios are generated and the outcome is calculated.

The best way to explain this concept is through a simple example that involves just one variable. Say that we would like to calculate (with some precision) the number π.[60] Because it is not straightforward to calculate the exact length of the circumference of a circle, we can approach this problem from a different angle. We know that the area *A* of a circle is expressed by:

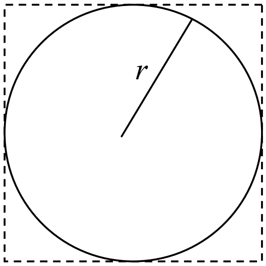
$$A = \pi r^2$$

where *r* represents half of the diameter of the circle:



[60] π is defined as the ratio between the circumference of a circle and its diameter, and its approximate value is 3.14159.

We can use a Monte Carlo simulation to approximate the area *A*. Let us circumscribe this circle with a square:



The area *S* of this square is:

$$(2 \times \mathrm{r}) \times (2 \times \mathrm{r}) = 4 \times \mathrm{r}^2$$

and the ratio between the area of the circle and the area of the square is:

$$\mathrm{A} / \mathrm{S} = (\pi \times \mathrm{r}^2) / (4 \times \mathrm{r}^2) = \pi / 4$$

Now we are ready for the simulation. Imagine throwing darts at a square target with a circle inside (as in the above figure). Each dart lands somewhere inside the square: the coordinates of a throw are *x* and *y* – the horizontal and vertical coordinates, respectively. If the center of the circle is positioned at point (0, 0), then the *x* and *y* coordinates can take any values from –*r* to *r*. We can simulate a "throw" by generating two random numbers from this range (one for *x* and the other for *y*), and then calculate how many throws landed inside the circle. A throw is inside the circle if the distance between the center of the circle (0, 0) and the position of the dart (*x, y*) is within the radius *r*:

$$\mathrm{x}^2 + \mathrm{y}^2 \leq \mathrm{r}^2$$

Say we simulated 10,000 throws, and the result of the simulation was that 7,854 darts landed inside the circle, while the remaining 2,146 darts landed inside the square, but outside the circle. This completes the simulation and we are ready to estimate the value of π. As the number 0.7854 (7,854/10,000) approximates the ratio *A / S*, and:

$$\mathrm{A} / \mathrm{S} = \pi / 4$$

then π is simply 4 × 0.7854 = 3.1416 (a pretty good approximation of 3.14159 after 10,000 throws). Clearly, the larger the number of throws, the better the approximation. Now that we know how the Monte Carlo simulation works, can we use it for something more useful?

Let us consider the casino environment for a moment (from which the method derives its name), and say we are holding an Ace and 6 in a game of Blackjack, with the dealer holding a Queen. To maximize our chances of winning the hand, should we "hit" or "stand"? Well, we can try to solve this problem by hand, but the number of possibilities might be too large (recall that the Ace can count as 1 or as 10, and if we hit and get a 2 then we will need to make another decision). This problem is an ideal candidate for the Monte Carlo simulation. We can generate millions of distributions of cards for a “shoe” (say we play six decks of cards, so we can consider random permutations of 6 × 52 = 312 cards), implement the dealer’s rules (e. g., hit on 16 or below, stay on 17 or higher), implement our own strategy, and then calculate the number of wins and losses. After running such a simulation, we will find that when holding an Ace and 6 against a dealer’s 10, we should hit! We may also discover some other “rules,” such as “always splitting two 8’s,” “doubling on 11 when the dealer’s hand is lower than 10,” and so forth.

In the car distribution problem, we can use the Monte Carlo method for estimating transportation risks. To do this, we can begin by estimating the transportation time between two locations as a function of weather (e. g., bad weather usually slows down a transport). We can then create a lookup table for the transportation time between two particular locations, which might look something like:

Weather Condition	**Time (in hours)**
Fair	36
Very hot	40
Windy	38
Rain	44
Snow	48
Ice	56

If we can estimate the weather conditions in a specific region (e. g., 10% chance of ice, 30% chance of snow, and 60% chance of having a fair day), then we can simulate many scenarios for how this will affect the transportation times. This is similar to generating various permutations of a shoe of cards, with the main difference being that the probability of each card arriving at a particular location in the shoe is the same – with weather, different weather patterns occur with higher or lower probabilities (at different seasons), so we should generate weather patterns that follow the same probability distributions. We can estimate the total transportation time for each of these scenarios, and then use the average in our decision-making process. This average transportation time is quite important, as many auction sites run their auctions once a fortnight, and if a car misses the auction it would sit on a lot (and depreciate) for two weeks waiting for the next auction.

Of course, the above example explains only the general idea of applying the Monte Carlo method for estimating transportation risks. In actuality, we would need to be more precise and have answers for many additional questions, such as: What does “very hot” or “windy” mean? Is it possible to have “rain” and “very hot” at the same time? If so, how would that affect the transportation time? Also, if the trip takes a few days, then there might be several “fair” days and several “rain” days, and so we would have to take an average of both categories.

Let us now transition from the Monte Carlo method to agent-based modeling with the following general remark: In Chap. 5 we discussed a variety of prediction methods that assume some amount of historic data are available. We can look at agent-based modeling as a special prediction method where no data are available! For example, the problem might be to minimize evacuation time from a conference room by finding the optimal arrangement of tables and chairs for 500 people. Clearly, we cannot dream of getting such data! We cannot load the room with 500 people, set the room on fire, measure the time needed to evacuate the room, change the arrangements of tables and chairs, load the room with 500 (preferably different) people, set the room on fire again, and continue this process for a few iterations to collect sufficient data! In situations like these, agent-based modeling is far more appropriate.

Now, the connection between the Monte Carlo method and agent-based modeling is that both methods generate many possible scenarios according to some probabilistic distributions of variables (e. g., windy weather conditions) or by using agents that follow some probabilistic rules. To continue our example from the previous paragraph of minimizing the evacuation time, psychological tests may indicate that in a fire emergency 17% of people run in random directions, 38% of people run in a straight line to the closest exit without paying attention to other people, 9% of people stand still and scream, etc. Additionally, the way people interact with one another during a fire emergency is also important (e. g., when two or more people collide, they might fall down and stay immobile for a few seconds). Now, by generating 500 agents whose behavior follows the above rules and interactions, and placing them together in a room that is set on fire, we can run many diverse simulations and observe the emergent behavior of the population.

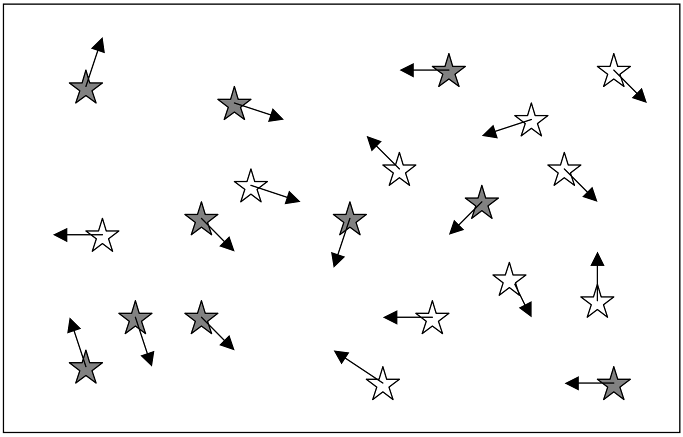
The general idea behind agent-based modeling is that the simulations are based on local interactions between the agents of a population. Furthermore, each agent may represent some object, whether it is a person in a crowd, a car in traffic, or an animal in an ecosystem. The entire model includes an environment in which the interactions occur. Agents may also have differing capabilities, and their behavior is based on probabilistic rules that determine their actions. The interaction of the agents (with the environment and one another) may result in an emergent behavior that may be impossible to predict (due to all the complexities of the interactions). Even a simple set of rules for each agent may result in a very complex system.

Some of the above points should sound familiar. Indeed, we have already looked at two particular instances of agent-based systems: ant colonies and particle swarms. In these systems, the local interactions of agents within a population may result in an emergent behavior of the entire population. The model – whether an ant colony or particle swarm – includes an environment in which interactions between the agents occur. Note, however, that these two examples constitute a simple case where all the agents are uniform. In general, this does not need to be the case: agents can represent a variety of complex (and different) objects, and the interactions among the agents can be quite complex (there may be rules for collecting and consuming food, trading resources, etc.).

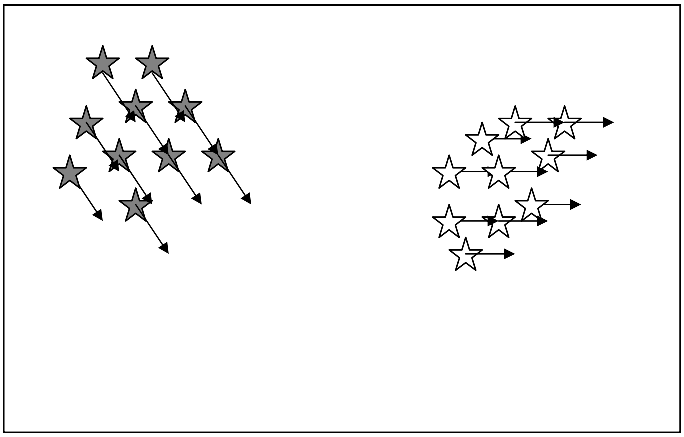
Let us illustrate the main characteristics of an agent-based modeling by using a classic example. Consider two species of birds (*A* and *B*) that fly around following three very simple rules:

Rule 1: If another bird of the same species is close by, then fly toward the other bird.
Rule 2: If another bird of the other species is close by, then fly away from the other bird.
Rule 3: Keep minimum distance between any two birds.

Note that such a model is extremely simple. There are only two species of birds (with 10 birds in each species) moving at the same speed, and their direction is set randomly at the beginning. The figure below illustrates the initial stage:



However, the interaction between these simple birds produces an emergent behavior that is complex, organized, and very life-like. After a while, we can ob serve a flocking phenomenon where birds of the same species fly together:



To define the behavior of these birds, we can add additional rules to the model. For example, we can modify their speed in some encounters, introduce obstacles in the environment, or create "sub-species" (which have slightly different rules). We can also make the environment more complex and then observe their behavior. One of the properties of this behavior is *unpredictability* over moderate time periods. For example, although the birds of one species might be flying primarily from left to right at one moment, it is impossible to predict which direction they might be moving at a later time.

In general, an agent can have many behavioral rules and internal states, some of which might be fixed, while others may change. An agent can process many sensory inputs, change its behavior according to these inputs, take into account the interactions with other agents, and make decisions based on the available information. Agents can also operate in an artificial environment, which might be a building, a city, a communication network, or a landscape that changes over time.[61] And, yet, as the above example shows, each agent is very simple in comparison with the complex behavior that eventually emerges.

So, how can we use agent-based modeling as a component of an Adaptive Business Intelligence system? Well, there are a few possibilities. In the car distribution example, we can develop an agent-based model to simulate the effect of a new make/model introduced by a competitor. Changes in new car demand would affect the resale value of off-lease cars, so the agent-based model could be used to enhance the effectiveness of the prediction module. Agent-based modeling can also be used to simulate a variety of other scenarios, such as the effect of higher gasoline prices on the sale price of used cars, or the impact of weather on the number of buyers that attend an auction sale.

Beyond the car distribution example, agent-based modeling is very appropriate for studying a variety of human social phenomena, including the propagation of diseases, trade habits, group formation (as illustrated by the above simple example of birds flocking), evacuation patterns, and migration.

9.4 Co-evolution

Throughout this book, we have emphasized the importance of the evaluation function, which serves as a link between the algorithm and the problem at hand. A thorough understanding of the evaluation function allows us to create the correct combination of representation, search operators, and selection criteria. There are problems, however, where we have no idea how to create a good evaluation function. Some of these involve discovering optimal strategies. For example: *What would be the best investment strategy? Best marketing strategy?* etc.

The problem with finding the "best strategy" in business resembles the problem of finding the best strategy in a game. Games are easier to model, as there are clear rules for what moves can and cannot be made, and the objective (e. g., the

[61] Because of this diversity, there is no general agreement on what an "agent" is, or what key features an agent should have.

criterion for winning and losing) is defined. Moreover, in many games there is only a single opponent and the moves are made "in turn." Real-world situations (e. g., the development of a marketing strategy) are much more complex. The rules are unclear (to say the least), there are many competitors (opponents), and the decisions (moves) are irregular. And yet, some powerful similarities exist between these two environments (a game versus the real world). In both cases, we have to devise a strategy for our possible moves, our opponents' countermoves, and the decision criteria for choosing one move over another. Furthermore, the process of learning is also based on trial and error in both cases.

To answer the question "how can we learn the best strategy?" let us refer to some co-evolutionary processes that exist in the natural environment. Most animals constantly face the crucial problem of survival. Many of their defensive and offensive survival strategies are genetically hardwired as instinctual behaviors. But how did these strategies emerge? Some species use coloration to blend into their background; their strategy is simply to be not noticed. Other species have developed a strategy based on "safety in numbers." Other species have learned to seek out high elevations and position themselves in a ring looking outwards, thus providing the earliest possible sighting of a potential predator. These complex strategies emerged over many generations of trial and error …

These examples illustrate the process of *co-evolution*, which is not the case of one individual against its environment, but rather individuals against other individuals, each competing for resources in an environment that poses its own hostile conditions but does not care which individuals win or lose in the struggle for existence. Competing individuals use random variation and selection to seek out superior survival strategies that will give them an edge over their opposition. Each innovation from one side may lead to an innovation from another, which is similar to an "arms race" of inventions.

For some problems, it is possible to develop a strategy by modeling co-evolutionary processes in the following way: There are several populations of solutions, where each population represents a "player" in the "game." For a two-person game, there are two competing populations. For some real-world cases, the number of populations would correspond to the number of competitors, with each of them seeking the best possible strategy. The key question is: how do we evaluate a strategy? The answer lies in running a strategy against the strategies of other "players," carefully evaluating the outcome of each "game." Better strategies are then selected, modified (with mutation and crossover operators – as described in Chap. 6), and, after some number of generations, the best strategy would emerge. This strategy would enjoy the property of robustness, as it would score well against the best opponent strategy!

Let us illustrate some co-evolutionary concepts by continuing the car distribution example. In the used car marketplace, many competing leasing companies face very similar problems. In particular, they face the problem of finding the best distribution of their off-lease cars. Of course, the distribution strategies of other leasing companies are not well known, and the decision process is complicated by the fact that used cars from many leasing companies are sold “together” at the same auction sites.[62] And because we do not know the distribution decisions made by other leasing companies, we cannot accurately measure the volume effect (see Sect. 3.5). For instance, say we decide to send 20 identical cars to a particular auction, and one of our competitors sends 40 similar cars to the same auction site (e. g., 20 white Toyota Camry cars and 40 white Mitsubishi Galant cars). The result may bias the volume effect and our sale price prediction would not be accurate.

As an example, say we are concerned with the distribution strategies of two very large competitors. We know the number of cars they lease each year (this is usually public information), and how these cars break down into different makes/models. Using this information, we can construct a “similarity matrix” that would provide us with information on which makes/models (“our” makes/models and “their” makes/models) should be clustered into one category for calculating the volume effect. If we grouped all the cars into 20 distinct categories (e. g., 2-door compact, 4-door compact, mid-sized), we can model a competitor’s strategy in the following way: Each auction site they send cars to can be modeled as a vector of 20 numbers (expressed as a percentage). This vector can represent a particular auction site X:

4.3%	0.0%	5.9%	…	3.9%	4.1%

The interpretation of this vector is that auction site X receives 4.3% of the cars from the first category, zero cars from the second category, 5.9% of the cars from the third category, etc. Since we know the total volume of cars in each category (e. g., a competitor would sell altogether 580 cars from the first category), we can easily convert these percentages into numbers (e. g., auction site X would receive 25 cars – 4.3% of 580 – from the first category). Thus, when we estimate the predicted sale price of a car, we can also count the number of additional cars in each category to calculate the volume effect.

Of course, the above vector of numbers represents just one hypothetical distribution of cars by this competitor, but we can use it as a starting point. Now we can model the overall situation using three separate populations of solutions. The first population represents a detailed distribution of the cars owned by our company. Note that we use the same representation of a solution as we did in Chap. 6: each individual solution in this population is a vector of auction site numbers, and the length of each vector corresponds to the number of processed cars. For example, the vector:

11	15	22	…	20	17

[62] This is not always the case. *Closed* auctions sell cars that belong to only one leasing company or automaker.

indicates that the first car should be sent to auction site 11, the second car to the auction site 15, etc., with the last car going to auction site 17. In other words, a single individual solution represents a complete, detailed distribution of all cars.

The two other populations, however, represent the distribution strategies of our two competitors. Individual solutions in these populations are very different to the individuals in the first population, as they have much shorter vectors of only 20 values each (corresponding to the 20 different categories). The total number of vectors corresponds to the total number of auctions sites. For example, an individual solution might be:

Auction 1	4.3%	0.0%	5.9%	…	3.9%	4.1%
Auction 2	2.5%	2.1%	1.9%	…	0.4%	0.3%
Auction 3	0.0%	2.9%	3.3%	…	0.9%	1.2%
	.				.	
	.				.	
Auction 50	1.1%	0.9%	1.3%	…	1.2%	1.4%

indicating the percentage of cars in each category (column) to be sent to a particular auction site (row). For example, 2.9% of cars from the second category are sent to auction site 3. The total of all percentages in one column (i. e., for one category) is 100%.

The co-evolutionary process of running these three evolutionary algorithms – each processing its own population – now becomes quite meaningful. Note that the connector between these three evolutionary processes is the evaluation function: To evaluate any individual in one population, we need to know the proposed distribution of cars in the other two other populations. Consider the following individual solution from "our" population:

11	15	22	…	20	17

To evaluate this individual solution, we take the best individuals from the second and third populations, as these individuals represent the best current strategies of our competitors. The combined distribution of cars from all three sources (i. e., our cars plus those of our two competitors) would allow us to measure the volume effect more precisely, thus we can better estimate the quality of our proposed distribution. The same can be done for each individual solution of each competitor: because we know our best distribution (at the moment) and the best distribution of the other competitor, we can precisely evaluate the quality of the first competitor's proposed distribution. In other words, we take into account the best distributions made by our competitors in evaluating our own distribution strategy.

Once we define the evaluation function (which serves as the main connector between three separate evolutionary processes), we are ready to start the co-evolutionary process. The initialization is simple, as the individual solutions in all three populations can be randomly generated. Once all the initial solutions for all three populations are evaluated, the other steps of the evolutionary algorithms are executed separately in each population: selection of parent solutions, application of crossover and mutation operators, replacement of some (inferior) individual solutions by newly generated offspring, and so on. The new populations are evaluated again at each generation, and the co-evolutionary process is responsible for finding superior distribution strategies for all three populations. This means that our distribution strategies become better over time, against the superior strategies of our competitors! Thus, the co-evolutionary process may give us the best solution for our distribution, as well as an insight into the likely behavior of our competitors!

Note also that we have some flexibility in the way we evaluate our solutions (i. e., distributions). It need not be the case that "our" distribution is combined with *the best* distributions of our competitors (at the particular generation of evolutionary process). To gain more robustness for our solution (after all, we cannot be certain that our competitors would select *the best* distributions), we can combine our solution with several – possibly randomly selected – solutions of our competitors and average the quality measure score. This way our distribution strategy that would emerge at the end of the co-evolutionary process would be scored against a variety of different strategies of our opponents!

9.5 Recommended Reading

In this chapter we explained a variety of additional prediction methods and optimization techniques, beginning with genetic programming, a special type of evolutionary algorithm where each individual solution is a function (i. e., a set of instructions) rather than a vector of numbers. We then looked at ant systems, where each artificial "ant" can be considered an intelligent agent that uses some communication medium (i. e., the pheromone trail) to leave messages for the other agents. This was followed by swarm intelligence systems, which assume the presence of a number of simple agents (e. g., ants, bees, wasps, termites) with direct or indirect interactions/communications. Next we provided an overview of the well-known Monte Carlo method and compared it with the relatively new agent-based modeling method, which can be used to model complex, dynamic, and open applications. And finally, we concluded the chapter with a section on co-evolutionary methods, explaining how they could be used to discover optimal "strategies." Some of these methods and techniques are inherently adaptive, and they allow the model or algorithm to change as the environment changes. As a result, these inherent characteristics can facilitate the construction of an Adaptive Business Intelligence system.

An excellent overview of genetic programming and related topics is given in the book *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications* by Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone (Morgan Kaufmann, San Francisco, 1997). The text begins with a basic introduction and then progresses into more advanced topics.

Ant Colony Optimization by Marco Dorigo and Thomas Stützle (MIT Press, Cambridge, MA, 2004) provides a great introduction to ant system heuristics (from real to artificial ants), and it also contains some chapters on theory, as well as a few applications.

An earlier book, *Swarm Intelligence: From Natural to Artificial Systems* by Eric Bonabeau, Marco Dorigo, and Guy Theraulaz (Oxford University Press, New York, 1999) discusses the social insect metaphor for solving many problems in optimization, communication networks, and robotics. Another book, *Swarm Intelligence*, by James Kennedy and Russell C. Eberhart (Morgan Kaufmann, San Francisco, 2001), talks about similar issues from the perspective of *particle swarm optimization*.

A recent book, *Introductory Econometrics: Using Monte Carlo Simulation with Microsoft Excel*, by Humberto Barreto and Frank Howland (Cambridge University Press, 2006) provides a clear undergraduate introduction to data analysis and Monte Carlo simulations.

The book *Growing Artificial Societies* by Joshua M. Epstein and Robert Axtell (MIT Press, Cambridge, MA, 1996) explains concepts behind the construction of artificial society models: agents, environment, and rules. The book introduces an artificial “sugarscape” model with more and more “advanced” agents, who live, trade, and die in their world, experience sex, culture, conflicts, and diseases …

The Evolution of Cooperation by Robert Axelrod (Basic Books, New York, 1984) was an early book describing co-evolutionary systems for game playing (iterated prisoner dilemma), and it examines the emergence of cooperation.

Part III:
Adaptive Business Intelligence

10 Hybrid Systems and Adaptability

"He was wrapped in some sort of dark ulster or blanket, which left only his face exposed, but that face was enough to give a man a sleepless night."
The Sign of Four

"Quite so! You have not observed! And yet you have seen. This is just my point."
A Scandal in Bohemia

In Part II of this book, we presented a variety of prediction methods and optimization techniques that can be viewed as being "hardware tools," in that each of them has different characteristics, strengths and weaknesses, and applications. Let us illustrate this point with the following analogy: Imagine that we have a hammer – which is a very handy tool for solving problems that involve nails – but we come across a problem that involves screws. What should we do? Well, if the hammer is our only tool then we might be tempted to "hammer" the screws. This, of course, is the wrong approach. A much better approach would be to find a better tool (e. g., a screwdriver) for solving the problem at hand.

The point of this analogy is that the right technique (i. e., tool) should be selected for the right problem. Unfortunately, many technology companies specialize in a single technique (e. g., evolutionary algorithms), and they apply this same technique to each and every problem they encounter. This is similar to having a hammer as the only tool and seeing all problems as nails. This approach is clearly wrong, as the technique should be selected for the problem at hand, not the other way around!

In general, there are several limitations of using a single prediction method or optimization technique. First, and most obvious, is the fact that it may perform superbly on some problem types and abysmally on others. Second, even if we limit ourselves to a single problem, remember that each problem type includes billions of possible "problem instances." In the car distribution example, we have to deal with a different instance of the same problem every day, as different cars have to be distributed. It is very important to understand the distinction between a problem (such as the car distribution example) and a particular instance of the problem (e. g., a particular input of cars to be distributed). It is often the case that different techniques are better or worse for different instances of the same problem. Although it would be great to claim that a particular technique is the best for a given problem, it is usually impossible to guarantee that this "best" technique would give the best results on each instance of the problem. Lastly, since the problem variables (e. g., weather, interest rates, consumer behavior) change over time, a single technique that performs well in the beginning may deteriorate over time.

In this chapter, we will discuss how the limitations of the single-technique approach can be overcome by packaging several techniques together into a *hybrid system* (i. e., a team of techniques). Specifically, we will look at hybrid systems for prediction and optimization, and then conclude the chapter with a discussion on the adaptability module.

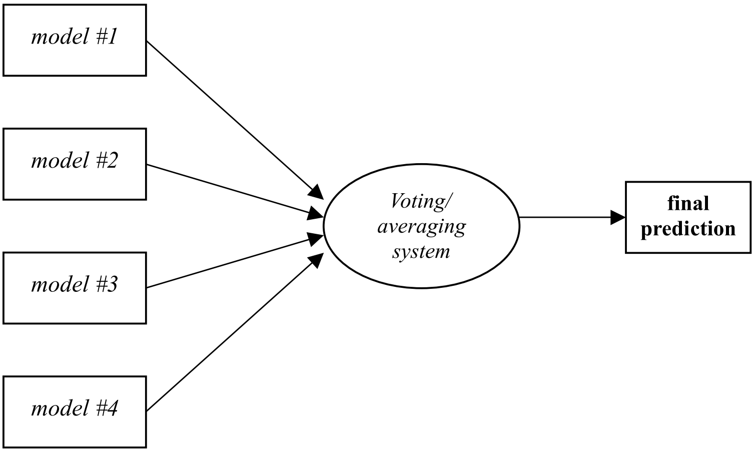
10.1 Hybrid Systems for Prediction

Most companies and government organizations make decisions that are based on predictions of the future. These predictions may involve future demand, effects of a marketing campaign, competitor actions, and so forth. As a simple example, imagine that we need a temperature prediction for an outdoor marketing event our company is planning. To predict the temperature, we could look at the last few days of temperature measurements and make a linear extrapolation, visit a weather website to obtain a forecast, watch the evening weather report, and so on. There are *many* ways to make a prediction, so the question is: Which method should we use to predict the temperature?

One possibility would be to use a forecast from our favorite weather channel. However, no prediction is perfect, and so we might get an inaccurate temperature prediction. Another possibility would be to calculate the average predicted temperature from many different sources, thereby minimizing the impact of any significant errors made by a single source. However, some sources may be biased (e. g., several channels may always predict toward the monthly average temperature). Hence, taking a simple average of all the predictions is sometimes not the best approach.

The safest way to make consistently good predictions is based on the concept of *hybrid systems.* Because response time, justification, model compactness, etc. are also important considerations (besides the prediction error, as discussed in Sect. 5.2.5), it may be difficult to select "the best" prediction method for the problem at hand (whether it be a prediction for temperature or something else). Different prediction methods have different properties, and so some of them may perform better or worse when trained on different data sets. Hence, instead of using a single prediction method to build a single model, it might be worthwhile to use a few methods to build a few models, and then use all the models to reach a consensus. This consensus might be achieved through voting or averaging, with the final prediction being the one with the largest number of votes (for classification problems) or some weighted average (for regression/time series problems). After all, this is commonly done within most organizations, where several people in a team express their opinions on "what may happen," and then the boss makes the final decision (i. e., the final prediction).

The general idea is presented in the following diagram:



This diagram illustrates four models that produce a "prediction" for each new case (either indicating a class for classification problems, or generating a number for regression/time series problems). These four predictions are then sent to a voting/averaging system that produces the final prediction. The arrows illustrate the weights for each prediction model, as the voting/averaging system learns (during the training session) that some models are more accurate than others and gives more weight to their output. Besides being simple and elegant, this approach has an intuitive advantage over a single prediction model: after all, "two heads are better than one"!

There are two primary ways of implementing this multi-model approach. The first way is to use models of the *same* type (e. g., linear regression, decision, tree, neural network). These models are then trained on different data sets, thereby ensuring that each prediction model is unique. The two most well-known techniques for this approach are called *bagging* and *boosting*:

- *Bagging*. Several prediction models of the same type are used together, and some voting (or averaging) is applied. The final prediction is either a weighted combination of predictions (for time series problems), or the prediction with the largest number of votes (for classification problems). Experimental evidence indicates that several prediction models built from one data set perform better than any single model built from the same data set.
- *Boosting*. This technique creates models through an iterative process that applies higher weights to the cases that are the most difficult to predict. Repetition of this process will produce a sequence of prediction models, where each new model focuses on the cases that were not accurately predicted in the previous model. This iterative process is the main difference between bagging (where all the models are developed separately) and boosting (where each new model is influenced by the performance of the previous model). This technique allows us to cover many "hard" cases, and develop many models. As with bagging, several prediction models of the same type are used together.

Recall from Sect. 5.3 that the bootstrap technique selects cases for the training data set *with repetition.* Moreover, the bootstrap technique uses the entire data set for training. This is also what happens in the early stages of the bagging technique: the size of the training data set is the same as the original size of the data set, and the sampling is performed with repetition.[1]

To illustrate the bagging technique, suppose we have selected several training data sets of the same size and decided to apply a neural network. For each training data set, we will get a *different* neural network model. Once we have created these different neural network models, we can use them together by taking an average of their predictions (if they are predicting numbers) or by counting the number of votes they produce (if they are voting for a class in a classification problem). Note that the number of neural network models is unlimited – as a matter of fact, the more "voters" the better! Because each "voter" has the same "power" – it is a pure democracy where every model influences the final outcome with the same weight.

Ideally, an effective prediction module consists of several *complementary* models that cover the vast majority of all possible cases. While *bagging* assumes that the "appropriate" type of prediction model has been selected, *boosting* addresses this issue directly by *seeking* models that complement each other. Unlike bagging, where each model is developed independently from one another, boosting takes advantage of pre-existing models. This makes it easier to find a model that "addresses" cases the earlier models did not handle well. This is usually accomplished by assigning weights to cases, so that the misclassification of a case with a high weight would have a more serious consequence than the misclassification of a case with low weight. Thus, each new prediction model would "concentrate" on the cases with higher weights (i. e., the cases that were the most difficult to predict).

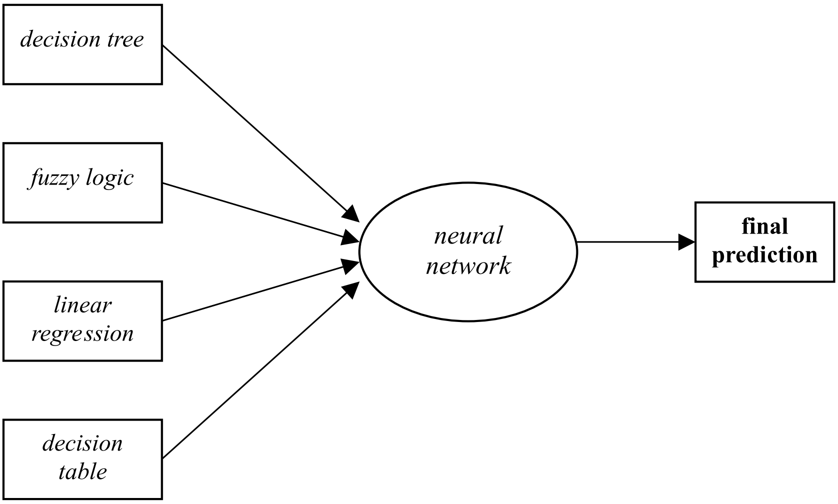
Keeping this in mind, the iterative process of boosting can be explained as follows: The first prediction model is built on the original data set, where all cases have the same weight. Then the weights are modified: the cases that were "easy" (i. e., those that the model classified correctly, or where the numerical prediction was accurate) would have their weights *decreased;* and the "hard" cases (i. e., those that the model misclassified or where the numerical prediction was inaccurate) would have their weights *increased.* Then the second prediction model is built using the original data set, but the cases now have modified weights so the new model would concentrate on the more difficult cases. Note that the weights are adjusted at the end of each iteration, so that each subsequent iteration produces a prediction model that "concentrates" on the harder cases.

Although there are many similarities between bagging and boosting – as both techniques require prediction models of the same type, and both techniques use voting or averaging – bagging creates a democracy of "equal" voters, while boosting uses weights to influence the model's performance. This is a major point of differentiation between the two techniques, and one that is particularly relevant to our discussion on adaptability in Sect. 10.3.

[1] This is not surprising, as *bagging* stands for *bootstrap aggregating.*

The second way of implementing the multi-model approach is to use *different* types of models. A well-known technique for this is *stacking,* where one prediction model uses the outputs (i. e., predictions) of several other models to come up with the final prediction. The best example of stacking is where the outputs from several prediction models constitute the input to a neural network model, which then produces the final prediction. Note that this final prediction is not the result of a vote or weighted average (as is the case in bagging and boosting), but rather of a "higher-level" prediction model that takes the outcome of the "lower-level" prediction models as inputs! Therefore, this higher-level prediction model makes the final prediction, on the basis of the preliminary, lower-level prediction models.

In the car distribution example, the different prediction models would try to predict the sale prices of the off-lease cars that need to be sold at auction. Each of the different models could be based on a different prediction method, such as linear regression, decision trees, fuzzy logic, etc. The "high-level" prediction model (e. g., a neural network) would take all of these predicted sale prices and produce the final prediction. Such a hybrid system based on stacking can be illustrated by the figure below, where four different prediction models independently predict the sale prices for cars sold at auction at some point in the future. These predictions, plus the original input data about the cars, are then fed into the neural network model for final processing:

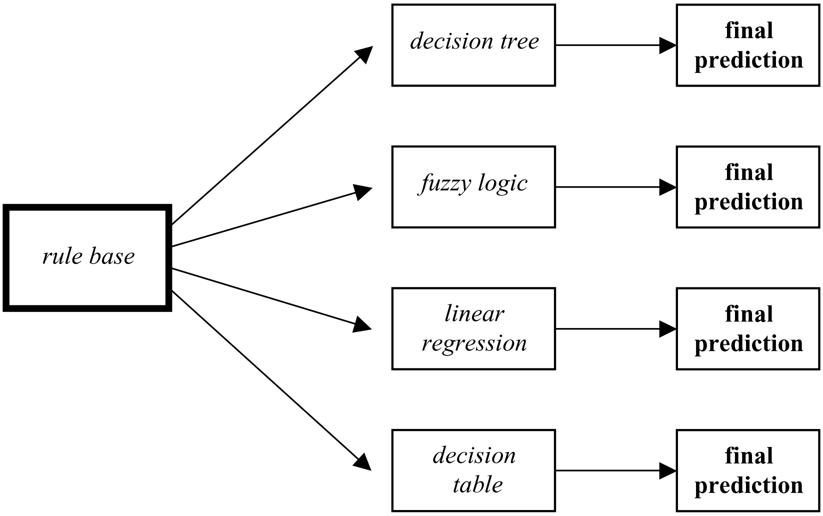


When implementing the stacking technique, there are several issues to consider. These include:

- Selecting the most appropriate prediction methods.
- Dividing the training data set into subsets for the lower-level prediction models and the main, higher-level prediction model.
- Performing cross-validation for the lower-level prediction models.

In addition to stacking, another popular technique for combining different prediction models is *gating*. In this method, a quick, upfront decision is made on which individual is most appropriate to process the input data and produce the final prediction. The major benefit of the gating technique is the speed of processing, as only the selected model is used to process a given input and the remaining models remain inactive.

The figure below shows a rule base that acts as a gate:



Based on the rules in the rule base, a decision tree, fuzzy logic system, linear regression model, or decision table is used to make the final prediction. Such selective processing speeds up the total classification/prediction time, which is important in time-critical applications. Consider a credit card fraud detection system that has to decide whether a transaction is fraudulent or not within a fraction of a second. Since large banks have tens of thousands of almost simultaneous transactions, the system has to produce a prediction almost instantly. In such situations, a gating method would be very appropriate.

In practice, the performance of a hybrid system is usually superior to any single technique that is used. Furthermore, the use of hybrid systems has become more attractive with the advent of affordable multi-processor servers, as the prediction models can be run in parallel using different processors. This would provide all the benefits of a hybrid system (including improved prediction accuracy) at virtually no extra computation time (as all the prediction models would perform their calculations in parallel).

10.2 Hybrid Systems for Optimization

For a moment, let us consider a problem that requires an optimal decision each day. This problem could involve the daily scheduling of tasks, distribution of assets, management of inventory, creation of transportation plans, etc. Note that different *instances* of this problem have to be solved every day, as a transportation plan for Monday might be very different from a transportation plan for Tuesday, due to different demands, availability of trucks and drivers, weather conditions, and so on.

The car distribution example is such a problem, as every day presents a new challenge. On Monday, for example, most of the 3,000 off-lease cars might be Ford Taurus models that are white, gray, and black because a large government organization returned these cars at the end of its fleet lease. Moreover, most of these cars were returned to two primary locations: Arlington, Virginia, and Atlanta, Georgia. The following day, however, the set of 3,000 off-lease cars may look very different, as many individual consumers may return different makes/models to various locations across the country. For this reason, each daily instance of the car distribution problem might different, because the distribution of makes/models and their location might be different from one day to the next. On top of this, the problem is set in a non-stationary environment, where things in the marketplace may change on a daily basis (e. g., a car manufacturer might offer regional incentives for certain makes/models, which will influence the wholesale price of these makes/models in some regions of the country).

The question is: Which optimization technique should be used to provide the best distribution of cars on a particular day? Is there any single technique that can provide us with the best recommendation across all possible instances? Or, rather, is one technique better for some instances, whereas another technique is better for other instances?

To answer this question, let us consider the following experiment: Say we have seven particular instances of the car distribution problem (i. e., we have seven sets of approximately 3,000 off-lease cars over seven consecutive days). Assume also that we have at our disposal seven different optimization techniques that were discussed in previous chapters. Each technique takes an input of 3,000 cars and produces the best possible distribution of these cars to the various auction sites. We can then measure the performance of each technique for each particular day by evaluating the "net profit gain" for each recommended distribution.[2]

[2] "Net profit gain" is calculated as difference in profit between the optimized solution and the standard solution that is based on expert rules developed by business managers over the years. Thus, if a technique provides a net profit gain of $100 on a particular day, it means that the recommended solution provides an average gain of $100 per car with respect to the standard solution (thus the total gain is of the order of 3,000 × $100 = $300,000).

Assume further that the following table summarizes the results of these seven different optimization techniques[3] over seven consecutive days:[4]

	AS	ES	EP	GA	SA	SI	TS
Day 1	$99	$129	$131	**$139**	$102	$110	$122
Day 2	$97	$89	**$103**	$91	$95	$91	$92
Day 3	**$119**	$104	$97	$101	$108	$93	$105
Day 4	$112	$94	$104	$109	$113	**$129**	$120
Day 5	$120	**$126**	$116	$119	$109	$110	$101
Day 6	$90	$101	$98	$102	$96	$95	**$108**
Day 7	$96	$102	$106	$105	**$113**	$92	$101
Average	**$105**	**$106**	**$108**	**$109**	**$105**	**$103**	**$107**

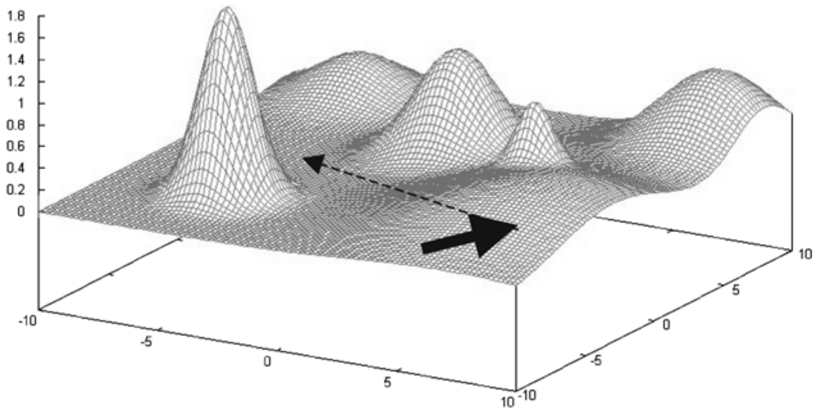
This table illustrates some very interesting points. First of all, it is obvious that no single technique had the best performance every day. As discussed earlier, each day is different and a different technique may provide the best solution for the given instance of the problem. For example, on Day 1, genetic algorithms produced the highest net profit gain of $139 per car, while on Day 6 it was tabu search with a net profit gain of $108 per car. Furthermore, if we had access to a fortune-teller and knew that on *average* the genetic algorithm would perform the best in the coming week, then we could use only that technique and achieve an *average* gain of $109 per car (the last row of the table shows the average net profit gain for each technique). However, by applying all the techniques in parallel, a hybrid system for optimization would generate an average gain of almost $120 per car (the average of the best results over seven days), which is approximately 10% better than using the single best-performing technique! And in the case of distributing 3,000 off-lease cars, this 10% extra improvement corresponds to an additional gain of $33,000 per day …

As with hybrid systems for prediction, we can run several optimization techniques in parallel and then select the best result. Thus, different optimization techniques can compete with one another and the best result is implemented. We can also "enhance" this competition by controlling the execution time allowed for each optimization technique, so that the technique with the best progress rate (i. e., improvement over the best solution found so far) gets more execution cycles.

[3] The following abbreviations are used: AS, ant systems; ES, evolution strategies; EP, evolutionary programming; GA, genetic algorithms; SA, simulated annealing; SI, swarm intelligence; TS, tabu search. Note that genetic algorithms, evolution strategies, and evolutionary programming are particular instances of evolutionary algorithms; see Footnote 38.

[4] The results presented in this table are based on a real experiment. However, we have changed the numbers slightly, as well as the order of presented algorithms. Note that each of the seven algorithms is a winner on exactly one of the seven consecutive days. This was done on purpose to: (1) point out the potential of hybrid systems, and (2) to avoid an unproductive discussion on "which technique is really the best?"

Hybrid systems for optimization also offer some additional possibilities. For example, by introducing mechanisms for exchanging information, the different optimization techniques can *cooperate* with one another to identify the best solution in the shortest possible time. Hence, the final gains are often much higher than by simply running all the techniques in parallel or sequentially.



The above figure illustrates how two optimization algorithms could cooperate with each other. The graph illustrates a quality measure score landscape (as discussed in Sect. 6.2) where the goal is to search for the highest peak (located in the left side of the landscape). The thick arrow illustrates how one technique (e. g., simulated annealing) searches up the hill to the right. Meanwhile another technique (e. g., genetic algorithms) also searches for the highest peak to the left. If the genetic algorithm finds a better solution in some other promising area of the search space then the appropriate "information" is passed to the competing technique (i. e., simulated annealing), which may "jump" to the new area to continue its search.

Of course, such information exchange can occur among any number of different optimization techniques, and herein lies the fundamental strength of cooperation: Due to this exchange of search information, more optimization techniques can explore the most promising areas of the search space. Consequently, such hybrid systems for optimization allow both competition and cooperation at the same time: competition, in that each technique tries to be "the best" by finding the best solution; and cooperation, in that all the techniques cooperate by exchanging their search information. Such hybrid systems for optimization often outperform any single optimization technique (especially when the set of problem instances is quite diverse) and, if properly implemented, this guarantees that the hybrid system never performs worse than any single technique.

Although the adaptability module (discussed in the next section) is responsible for adapting the prediction module in an Adaptive Business Intelligence system, the optimization module can also contain its own features of adaptability. Consider the fact that most optimization techniques have several parameters. For example:

- In genetic algorithms, the parameters include population size, probability of operators (crossover and mutation), and selection and replacement pressures.
- In simulated annealing, the parameters include the size of the neighborhood and the speed of the cooling scheme.
- In ant systems, the parameters include the number of ants, and the balance between attractiveness and pheromone levels.

Typically, an optimization technique is selected for the problem at hand, and then its parameters are tuned by experimenting with a variety of instances of the same problem. The question is: How can we tune the parameters of the algorithm to get the best performance across a wide gamut of various instances? One option might be to adapt the parameter values *during* the optimization run (and not *in between* runs). For instance, at some stages of the optimization process a small population of solutions may perform better, whereas at some other stages a larger population may do well. Thus, changing the population size parameter during the run might be beneficial, as a better solution could be found in a shorter time.

How can we use parameters that change values over time? Well, the methods for changing the parameter values can usually be classified into one of three categories:

- *Deterministic parameter control* uses deterministic rules to change the value of a strategy parameter (i. e., a parameter that controls how the search is made). This rule modifies the strategy parameter deterministically without using any feedback from the search. A time-varying schedule is often used, where a rule is applied when a set number of iterations have elapsed since the last time the rule was activated.
- *Adaptive parameter control* uses feedback from the search to determine the direction and/or magnitude of change in a strategy parameter value. In general, new values are assigned on the basis of the current state of the search.
- *Self-adaptive parameter control* uses "evolution" to determine the values for strategy parameters. The parameters are encoded into the data structure of the individual solutions and undergo variation (mutation and crossover). The "better" values of these encoded parameters lead to "better" solutions, which in turn are more likely to survive and produce offspring solutions, and hence propagate these "better" parameter values.

In summary, there are many ways to adapt the parameters of any single technique. In the case of using multiple techniques within the optimization module (i. e., the hybrid system approach), some additional parameters may undergo adaptation during the run. These parameters include:

- Parameters for controlling the type of information that is exchanged between the techniques. (For example: Should the optimization techniques exchange complete solutions or only partial solutions?)
- Parameters for controlling the frequency of the information exchange.

- Parameters for controlling the transmission of the information. (For example: Should the information be transmitted “across the board” to all techniques, or to only a few techniques? In the latter case, which ones?)
- Parameters for controlling how the information sent from one technique to another is used. (For example: Should the existing solution be replaced or recombined with an existing solution?)
- Parameters for controlling the execution time allocated to each technique during the run.

While all these issues can be parameterized and adapted during a run, most hybrid systems for optimization use both adaptive and static settings: Adaptive settings for the technique parameters (e. g., population size, probability of operators) and static settings for the hybrid system parameters (e. g., type and frequency of information exchange, the way a new solution is transferred and used by another optimization technique, run time, etc.).

10.3 Adaptability

New data are constantly entering the information systems of almost all organizations. In the car distribution example, these data would take the form of new cases containing the sale price, make, model, body style, etc. for each car sold at auction. Note that the prediction module (consisting of one or more models) has already predicted the sale price for these cars, and now we are getting back the *actual* sale prices! These data will tell us the error rate of our prediction module, and whether or not the underlying prediction model requires some adjustment. Because it is inevitable that some adjustments will be required over time (after all, the marketplace is constantly changing), we can do one of two things:

- Repeat the process of updating the prediction module at regular intervals. For example, we might update the parameters of the underlying prediction model (or models) every three to six months. However, the process of analyzing new data and updating the parameters accordingly might be expensive, so there is an incentive to repeat the process at longer intervals (e. g., once a year, instead of every quarter). This approach often causes a problematic tradeoff, as shorter intervals would be better from a prediction standpoint, but worse from a cost standpoint.
- Develop an adaptability module that is responsible for updating the parameters of the prediction model to fit new data. By automating this process, new data can be fed in quite frequently (i. e., at the end of each business day). If a new pattern emerges, an updated model can capture it almost immediately.

Clearly, there are several powerful advantages of the latter approach (i. e., developing an adaptability module):

- There is no time delay connected with building and implementing new models inside in the prediction module.
- The process is automatic.
- The frequency of updates can be much higher.
- There is a good chance of discovering a new, emerging pattern almost immediately.

Probably the most important consideration here is that the optimization module relies on accurate predictions. Consequently, if the predictions are inaccurate, then the optimization process may do more harm than good! For example, say the optimization module makes a recommendation of sending a car to an auction site thousands of miles away (e. g., from New York to California) because the predicted profit on this car (after deducting transportation costs, depreciation, etc.) is highest in California. However, if the actual sale price turns out to be considerably less than the predicted price, then the total loss might be hundreds of dollars (besides receiving a poor price for the car, additional costs were incurred for transportation and depreciation). Who is to blame? A poor prediction module …

Given the importance of keeping the prediction module current, developing an adaptability module that can automatically update the model's parameters is clearly the preferred approach. In the case of a prediction module that is based on multiple prediction models, it is usually feasible to update both the parameters of the individual models *and* the voting/averaging system. However, let us first discuss a prediction module that is based on a single model.

For example, in Sect. 5.2.1 we covered exponential smoothing methods (which generalize the *moving average method*, where the mean of past k observations is used as a prediction). Note that all exponential smoothing methods assign weights to past observations in such a way that recent observations are given more weight than older observations. They also require at least one parameter a, which plays an important role. A prediction for the time interval $t+1$ is calculated as:

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

which simply means that the prediction for the next (future) case is calculated as a total of two values: the last actual case (*Actual(t)*) with weight a, and the last prediction (*Prediction(t)*) with weight $1–a$. The performance of a prediction model based on this method depends on the selection of the parameter a, as the prediction would always be the last actual value if $a = 1$. Because different values of a may be required at different times, it might be reasonable to develop an adaptability module that would be responsible for adjusting this parameter. In other words, the adaptability module would systematically change the values of parameter a from interval to interval to allow for changes in the data. The fixed parameter a would be replaced by $a(t)$ and the adaptability module would assign a new value for $a(t)$ at every interval t (e. g., using a function that is based on the most recent prediction errors).

In this particular case, developing an adaptability module that systematically changes the values of parameter a has the following benefits: (1) it is automatic, so the administrative overhead connected with frequent adjustments is reduced, (2) even if the performance is slightly inferior to the "optimal" fixed value for parameter *a* it reduces the risk of serious errors, and (3) there is no need to specify the initial value of parameter *a* – even if it takes a few intervals for *a(t)* to catch up with the changes in the data, it will eventually do so.

Let us consider another example that is based on a single prediction model, but this time for a classification problem. The adaptability module could observe and record the performance of the prediction model over time. As cases of incorrect classification are determined, the adaptability module would store them (along with the correct response) for eventual processing. Once a critical mass of incorrect classifications has been saved, the adaptability module would "clone" and modify the prediction model so that it does a better job of handling these incorrectly classified cases. The adaptability module would then test the prediction model offline to ensure that it provides an advantage over the existing model. Once the adaptability module has adequately tested and verified the performance of the new model, it would be deployed (effectively replacing the existing model).

Regardless of whether we use exponential smoothing or some other prediction method (like fuzzy logic), we can develop an adaptability module that uses an optimization technique to search for best parameter values. The objective of the optimization technique would be to search for the parameter values that minimize the prediction error. After making an update, the adaptability module can measure the prediction error by comparing the predicted sale price of recently sold cars to the actual sale price. If we get better predictions, then our update was good; otherwise, the adaptability module should undo the changes.

For a prediction module that is based on a single prediction model, the development of an adaptability module that changes the parameters of the model usually works well in most cases. However, it is important to point out that very large changes in the marketplace might require an overhaul of the underlying prediction model, as any tuning of the model would probably be insufficient. The need for such massive adaptation often shows up in a sudden and significant increase in the prediction error, which cannot be reduced by simply updating the parameters. However, this is quite rare, as most changes are gradual. Furthermore, in practice the adaptability module would have a threshold that defines what prediction error is acceptable.[5]

In the case of a prediction module that is based on many models (i. e., a hybrid system), the adaptability module could:

- Update each individual prediction model.
- Update the voting/averaging system.
- Update both the individual models and the voting/averaging system.

Adapting each individual prediction model typically involves updating its parameters, while keeping the overall structure unchanged. As an example, imagine updating a neural network. As discussed in Chap. 8, a neural network consists of

[5] If the prediction error exceeds this threshold, then the adaptability module would indicate that an overhaul of the prediction model is required.

input nodes, hidden nodes, output nodes, and connection weights in between them. In order to update a neural network, an adaptability module could simply add new data to the training set and then update the weights, without changing the overall structure of the network (i. e., without adding or removing any nodes). If considerable data are available in short time intervals and the Adaptive Business Intelligence system operates in a dynamic environment, then the training data set is often like a sliding window (e. g., we would discard the old data, and only use the most recent data to update the neural network).

Another possibility is to update the voting/averaging system, while leaving the individual prediction models unchanged. This is usually straightforward, as it requires rewarding (i. e., increasing the weights) of the more accurate models, while punishing (i. e., decreasing the weights) of the less accurate models. In other words, we pay more attention to the models that produce the most accurate predictions for the given period of time. For example, consider a hybrid system based on the bagging technique, where each individual prediction model has a weight associated with it (i. e., each prediction model is weighted more or less heavily). These weights can be updated on the basis of new (recent) data, just like the weights of a neural network with no hidden layer.

The third possibility is to update both the individual models and the voting/averaging system, i. e., update the whole hybrid system. Usually this is done in two stages. Once the individual prediction models are updated, the weights that the hybrid system has assigned to the various prediction models are then modified on the basis of recent data. Of course, it is always possible to rebuild the entire hybrid system, but this is beyond the scope of the adaptability module.

Clearly, many possibilities exist for developing an adaptability module for an Adaptive Business Intelligence system. However, many decisions on the adaptability (e. g., what and when to adapt) are problem specific, and can be determined only after transactional data files are carefully examined.

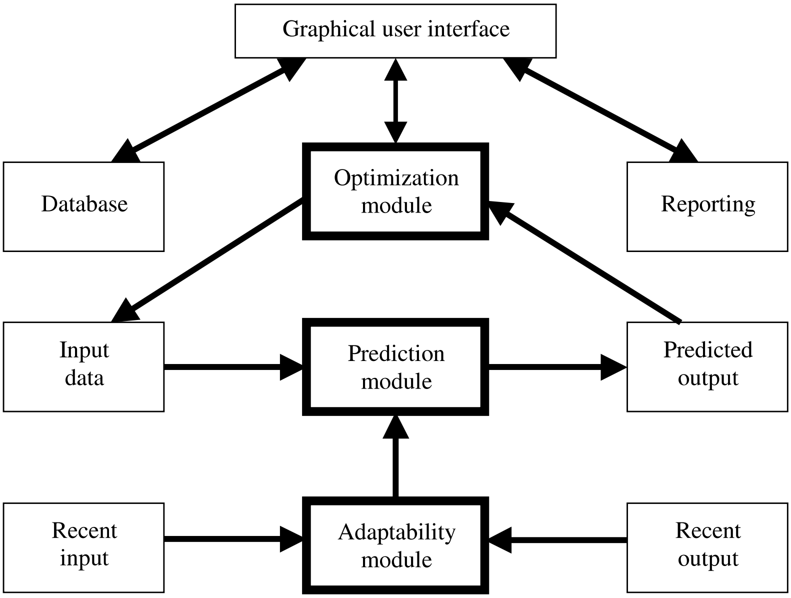
11 Car Distribution System

"We're not jealous of you at Scotland Yard. No, sir, we are very proud of you, and if you come down tomorrow, there's not a man, from the oldest inspector to the youngest constable, who wouldn't be glad to shake you by the hand."
The Adventure of the Six Napoleons

"When you follow two separate chains of thought, Watson, you will find some point of intersection which should approximate the truth."
The Disappearance of Lady Frances Carfax

The car distribution example, introduced in Chap. 3, is a real business problem that many leasing companies struggle with on a daily basis. To address this problem, the authors of this book designed and developed an Adaptive Business Intelligence system for optimizing the distribution of off-lease cars. In this chapter we discuss this system's core functionality, and take a closer look at how it was integrated into a particular company's computing environment.

To begin our discussion, recall from Sect. 4.5 that the overall structure of an Adaptive Business Intelligence system resembles the following diagram:



This diagram illustrates that the graphical user interface (which is what the end-user would see) is “in charge” of the remaining modules. The graphical user interface is also responsible for input/output data handling, which typically means either integrating or interfacing with existing data systems.[6] The input data are then processed via an internal database, and then passed on to the prediction and optimization modules.

In the case of the Adaptive Business Intelligence system for distributing cars:

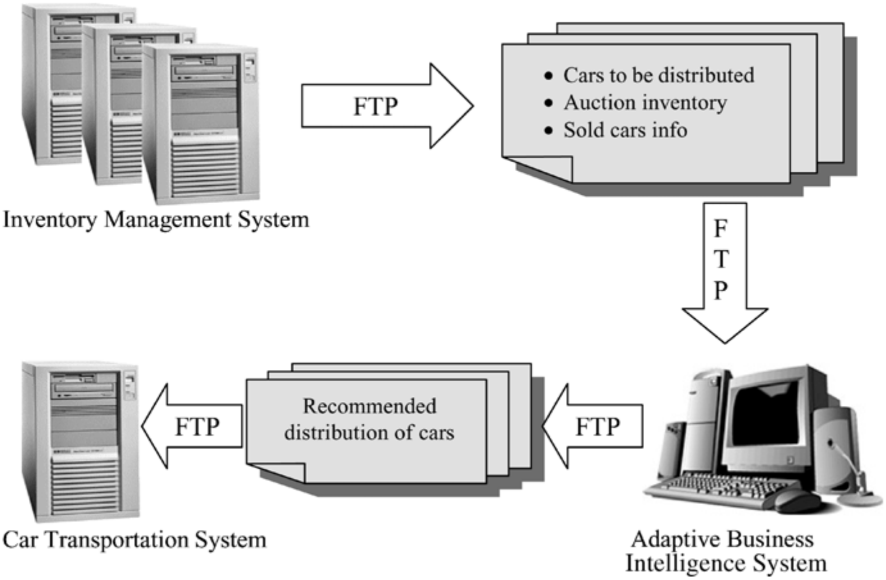
- The graphical user interface is the user-facing screen that controls the import and export of data, as well as the functionality of other modules.
- The prediction module consists of several different models to generate the most accurate price prediction for each car.
- The optimization module consists of several competing and cooperating techniques that adapt during the optimization process (and possibly between optimization runs) to find the best distribution of cars.
- The adaptability module uses the actual car sale prices to update the prediction module, thereby keeping the entire system “in tune” with the time-changing environment.

In the following sections of this chapter, we will take a closer look at each of these components.

11.1 Overview

The Adaptive Business Intelligence system described in this chapter was designed to run with minimal human supervision. Consequently, the system receives the inventory of cars to be distributed from the company’s *Inventory Management System*, generates a recommended distribution of cars, and then provides this solution to the *Car Transportation System.* The optimization process takes approximately 90 minutes, with additional time needed for loading all the required input files.

[6] *Integration* refers to a direct-programmed access from one system to another, while *interfacing* refers to a data exchange via an agreed data structure between two separate systems.



FTP
• Cars to be distributed
• Auction inventory
• Sold cars info
Inventory Management System
F T P
Recommended distribution of cars
FTP
FTP
Car Transportation System
Adaptive Business Intelligence System

Each morning, the Adaptive Business Intelligence system automatically downloads the input files from the server and processes them. The input files contain data about:

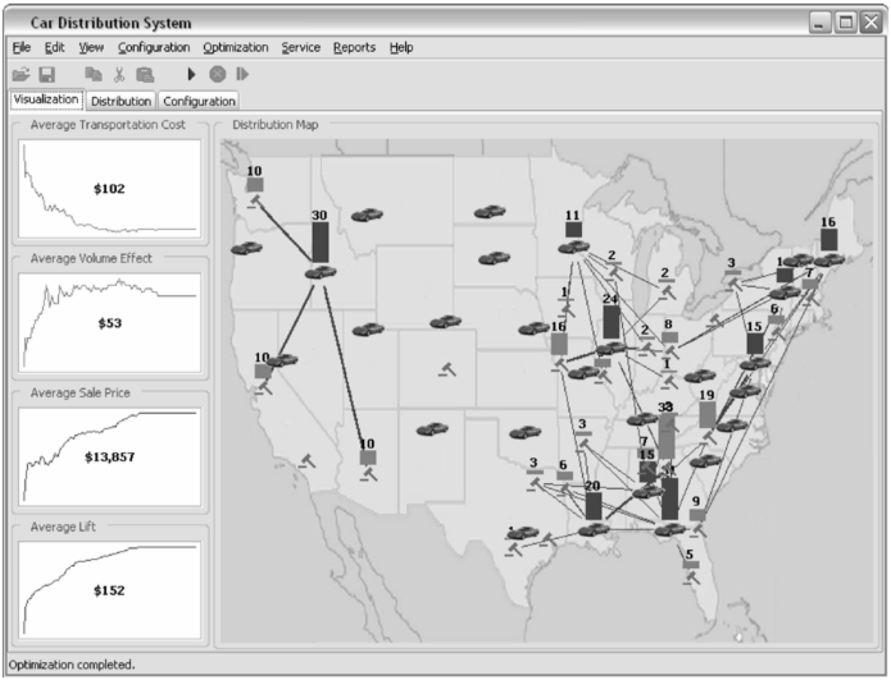
- The cars that need to be distributed that day.
- The current, up-to-date inventory levels of all cars already at auction.
- The cars that have been sold recently. This information is used by the adaptability module to tune the prediction module.

The optimization process starts automatically once the input files have been loaded. By the start of the business day, the process is complete and the recommended distribution is ready for review. However, the final solution is implemented only after a business manager checks the results and (possibly) makes some small adjustments. Less than 1% of the cars from the recommended distribution are changed manually, and these changes are last-minute decisions based on new information (e. g., an upcoming snowstorm has blocked some major highways). The final output file is then sent to the system that manages the transportation of the cars to the auction sites.

Because this Adaptive Business Intelligence system was designed to run with as little human intervention as possible, all of its functions can be executed automatically. The user specifies when the loading of input files should start, whether or not the optimization process should begin automatically, and if the generation of output files should be done with or without human intervention. These features allow the system to start processing files early in the morning, so that the recommended distribution is ready at the start of the business day (which is important for the efficiency of the overall business process).

11.2 Graphical User Interface

One of the functions of the graphical user interface is to allow business managers to "visualize" a particular distribution solution. In the sample screen below, there is a map of the United States with icons for each distribution center and each auction site, and four performance graphs on the left-hand side of the screen. The "car" icons represent *distribution centers*, where cars are collected, cleaned, and conditioned for eventual sale at an auction site.[7] The "hammer" icons represent *auction sites,* and the lines between the distribution centers and auction sites represent the volume of cars transported between these points (the thicker the line, the more cars are transported):



Optimization completed.

[7] Only the largest leasing companies have such distribution centers. For leasing companies that *do* have them, an off-lease car is dropped off at a dealership, then shipped to the closest distribution center for cleaning and conditioning, and then the Adaptive Business Intelligence system ships the car to the best auction site. For leasing companies that *do not* have them, the car is cleaned and conditioned at the dealership, and the Adaptive Business Intelligence system ships the car to the best auction site directly from the dealership.

The four graphs on the left-hand side of the screen display the optimization objectives during the run:

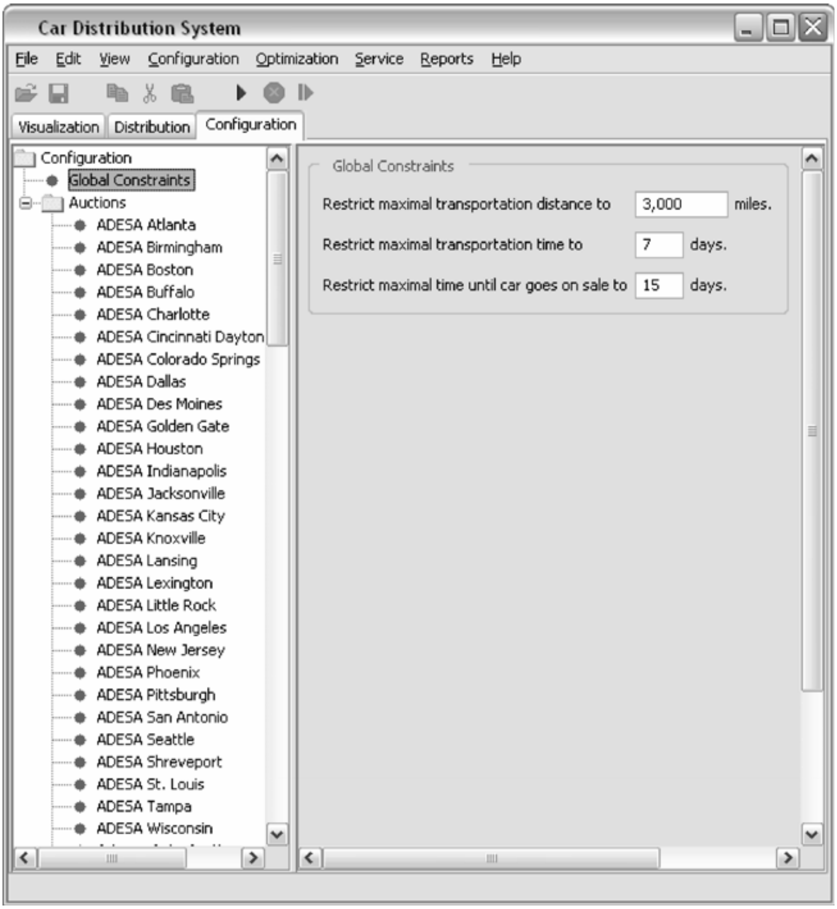
- *Average Transportation Cost.* The system calculates the total transportation cost, and the graph displays the average cost per car.
- *Average Volume Effect.* The system calculates the total amount of lost revenue due to sending too many similar cars to the same auction sites, and the graph displays the average value lost per car.
- *Average Sale Price.* The system calculates the expected sale price for all the cars, and the graph displays the average value per car.
- *Average (Net Sale Price) Lift.* This corresponds to the average "profit improvement" per car. The system calculates this as the difference between the predicted average net sale price (i. e., sale price after subtracting all auction fees, transportation costs, etc.) for the optimized solution, and the predicted net sale price for the standard solution (which is based on expert rules that were developed by business managers over the years).

In the sample screen above, the average transportation cost per car (first graph) has steadily decreased during the optimization run, while the average volume effect per car (second graph) has increased. The Adaptive Business Intelligence system has chosen a solution with a higher average volume effect, because it was more than offset by a lower average transportation cost and higher average sale price per car. This in turn has resulted in a higher average (net sale price) lift per car (fourth graph).

Once the optimization process is complete, the Adaptive Business Intelligence system generates an output file with the recommended distribution of cars. However, as mentioned earlier, sometimes a business manager modifies this distribution before the final output file is generated. To assist the business manager in this task, the graphical user interface provides an easy way to find specific cars or to select cars on the basis of distribution center location, make/model, year, color, etc. Furthermore, it is easy to "undo" any changes, as the system "remembers" the recommended distribution and can easily revert to it.

11.2.1 Constraint Handling

Another important function of the graphical user interface is to provide business managers with the ability to modify, add, or delete various constraints (i. e., business rules). Constraints that are applied to all the auction sites are regarded as *global constraints.* An example of this is the "maximal transportation distance constraint" (which limits the transportation distance of all cars), as shown in the following configuration sample screen:



Global Constraints

Restrict maximal transportation distance to 3,000 miles.

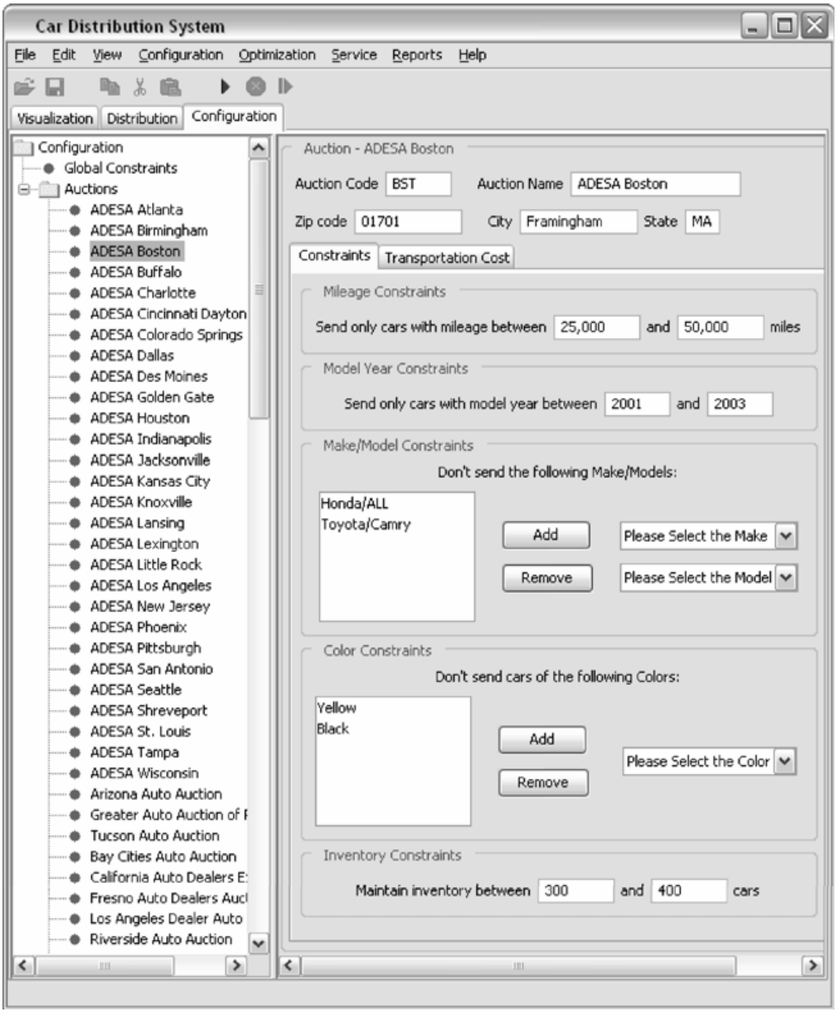
Restrict maximal transportation time to 7 days.

Restrict maximal time until car goes on sale to 15 days.

There is also a large set of local, auction-specific constraints, such as:

- *Mileage constraint*, which defines the upper and lower mileage of cars that can be shipped to specific auction sites. An example of this constraint would be: “Only ship cars that have between 30,000 and 70,000 miles to auction site ADESA Atlanta.”
- *Model year constraint,* which specifies a range of model years that can be sent to specific auction sites. For example, we could specify that a particular auction site only accepts cars built between 1997 and 2002.
- *Make/model exclusion constraint,* which specifies certain makes/models that are to be excluded from specific auction sites.
- *Color exclusion constraint*, which specifies certain colors that are to be excluded from specific auction sites.
- *Inventory constraint,* which specifies a desired inventory level at each auction site. For example, a business manager can specific an inventory level between 600 and 800 cars at a particular auction site at any particular time.

The sample screen below shows the local constraints for the “ADESA Boston” auction site:


Car Distribution System
File Edit View Configuration Optimization Service Reports Help
Visualization Distribution Configuration
Constraints Transportation Cost
Add
Remove
Please Select the Make
Please Select the Model
Add
Remove
Please Select the Color

Configuration
Global Constraints
Auctions
ADESA Atlanta
ADESA Birmingham
ADESA Boston
ADESA Buffalo
ADESA Charlotte
ADESA Cincinnati Dayton
ADESA Colorado Springs
ADESA Dallas
ADESA Des Moines
ADESA Golden Gate
ADESA Houston
ADESA Indianapolis
ADESA Jacksonville
ADESA Kansas City
ADESA Knoxville
ADESA Lansing
ADESA Lexington
ADESA Little Rock
ADESA Los Angeles
ADESA New Jersey
ADESA Phoenix
ADESA Pittsburgh
ADESA San Antonio
ADESA Seattle
ADESA Shreveport
ADESA St. Louis
ADESA Tampa
ADESA Wisconsin
Arizona Auto Auction
Greater Auto Auction of F
Tucson Auto Auction
Bay Cities Auto Auction
California Auto Dealers E:
Fresno Auto Dealers Aucl
Los Angeles Dealer Auto
Riverside Auto Auction

Auction - ADESA Boston

Auction Code BST Auction Name ADESA Boston

Zip code 01701 City Framingham State MA

Mileage Constraints

Send only cars with mileage between 25,000 and 50,000 miles

Model Year Constraints

Send only cars with model year between 2001 and 2003

Make/Model Constraints

Don't send the following Make/Models:

Honda/ALL
Toyota/Camry

Color Constraints

Don't send cars of the following Colors:

Yellow
Black

Inventory Constraints

Maintain inventory between 300 and 400 cars

Each auction site may have a different set of constraints, expressing its own local set of business rules. For the ADESA Boston auction, the constraints express the following business rules:

- “Send only cars with 25,000 to 50,000 miles.”
- “Send only 2001, 2002, or 2003 year models.”
- “Do not send any Honda or Toyota Camry cars.”
- “Do not send any yellow or black cars.”
- “Keep the inventory between 300 and 400 cars.”

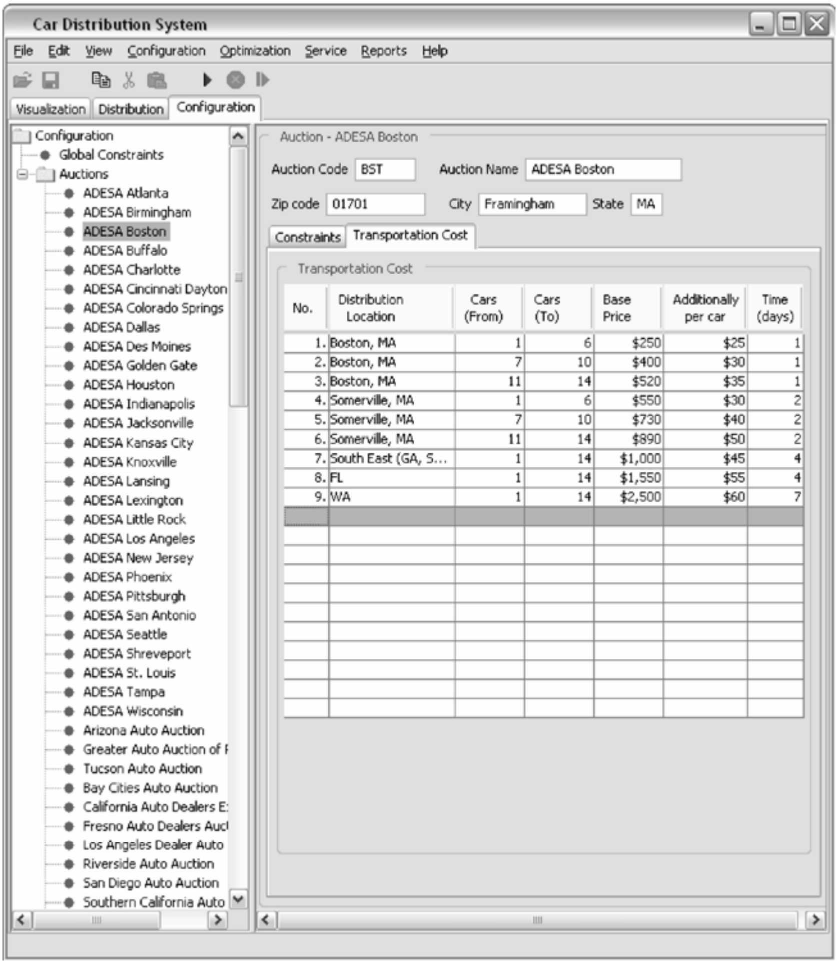
We should be careful in specifying these business rules, because it is possible to specify such a collection of rules (constraints) for which there is no feasible solution! As an example, imagine a yellow car arriving at a distribution center that has only two auction sites within a 300-mile radius. Now, if a global constraint limits the maximum transportation distance to 300 miles, and the local constraints for these two auction sites are such that neither accepts yellow cars, then the Adaptive Business Intelligence system has to violate at least one of these constraints to distribute this yellow car. This situation raises an interesting question, namely: How does the system deal with a situation where a car cannot be sent to any auction site without violating a constraint?

Except for the inventory constraint, all the constraints described above are "hard" constraints. If the Adaptive Business Intelligence system has to break a hard constraint, then it would send the car to the closest auction site and mark this recommendation as "violating a constraint." The inventory constraint, on the other hand, is a "soft" constraint. A penalty is assigned to solutions that violate a soft constraint, making them less competitive than other solutions. However, a solution that violates a soft constraint (and has a penalty) may be chosen over a solution that does not violate any soft constraints if its overall quality measure score is higher. For instance, let us assume that a specific auction site has a maximum inventory constraint of 500 cars. The system may generate a solution where this auction would accumulate an inventory of 510 cars. This solution would be penalized for having an extra inventory of 10 cars, but the price difference and transportation savings may overcome the cost of having 10 additional cars at this auction.

The penalty for violating a "soft" constraint grows exponentially for any auction site, and instances where the constraint is violated in a significant way are extremely rare. However, if the Adaptive Business Intelligence system has to process a very large number of cars on a single day, then the inventory constraint might be violated at almost every auction site. In such cases, the exponential penalty function would make these violations uniform. As an example, imagine a case where all the auction sites have a maximum inventory constraint of 300 cars, but the current number of cars to be distributed would inevitably increase the inventory to an average of 400 cars per auction. Under such circumstances, we could expect that the penalty for violating this "soft" constraint would be evenly distributed across all auctions (so that they have the same degree of violation).

As discussed earlier, constraints allow business managers to express various business rules (e. g., "do not send any red cars to Florida"), and the configuration screen serves as a communication link between a business manager and the Adaptive Business Intelligence system. Using the configuration screen, a business manager can investigate various "what-if" scenarios, such as "what would be the distribution of cars if I set the maximum transportation limit to 500 miles?" Note also that approximately 200 auction sites are present in the system, but only 50 of them are "active." A business manager can activate or deactivate any auction site, thus exploring various "what-if" scenarios such as "what would be the implication (e. g., in terms of the average lift) if the system uses 60 auction sites rather than 50?"

Different "what-if scenarios can also be investigated for different transportation cost options available from different suppliers. The Adaptive Business Intelligence system calculates the transportation cost from any distribution center to any auction site for any number of cars, and there are two factors that influence this cost: (1) the distance between a distribution center and an auction site, and (2) the number of cars being shipped. The sample screen below shows the transportation cost functions for the ADESA Boston auction:



Auction - ADESA Boston

Auction Code: BST   Auction Name: ADESA Boston

Zip code: 01701   City: Framingham   State: MA

Transportation Cost

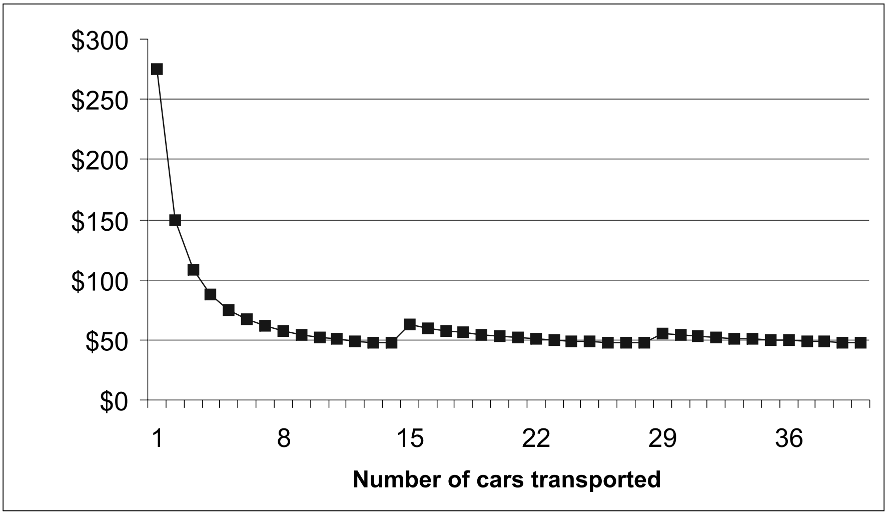
No.	Distribution Location	Cars (From)	Cars (To)	Base Price	Additionally per car	Time (days)
1.	Boston, MA	1	6	$250	$25	1
2.	Boston, MA	7	10	$400	$30	1
3.	Boston, MA	11	14	$520	$35	1
4.	Somerville, MA	1	6	$550	$30	2
5.	Somerville, MA	7	10	$730	$40	2
6.	Somerville, MA	11	14	$890	$50	2
7.	South East (GA, S...	1	14	$1,000	$45	4
8.	FL	1	14	$1,550	$55	4
9.	WA	1	14	$2,500	$60	7

In this sample screen, the transportation cost function is defined for shipping cars to the ADESA Boston auction from five different locations.[8] The first two

[8] The transportation cost may be defined in terms of how much it would cost to transport a car (or group of cars) from a particular ZIP code, city, state, or region to the auction site. Another way to define the transportation cost would be through the mileage (i.e., the distance from the distribution center to the auction site).

locations are defined by the cities: Boston, MA and Somerville, MA. The third location is defined by a region containing the states Georgia, South Carolina, and North Carolina. And the fourth and fifth locations are defined by the states Florida and Washington, respectively. According to the definitions given above, it would cost $250 to send a truck to Boston, MA, plus an additional $25 for each car that is picked up. If we wanted to pick up 6 cars, then the transportation cost would be $400 ($250 + $25 × 6 = $400).[9]

Note also that row no. 9 in the sample screen above defines the transportation cost between the ADESA Boston auction and the state of Washington. Because of the long distance (approximately 3000 miles), it would cost $2,500 to send a truck to Washington, plus an additional $60 for each car that is picked up. Although the cost of shipping one car would be $2,560, the cost of shipping 14 cars would be $3,340 ($2,500 + $60 × 14 = $3,340), or about $239 per car (which is 10 times less!). As the following graph illustrates, the more cars transported from the same location, the smaller the transportation cost per car (in this particular case, cars that are transported from Boston, MA to the ADESA Boston auction):



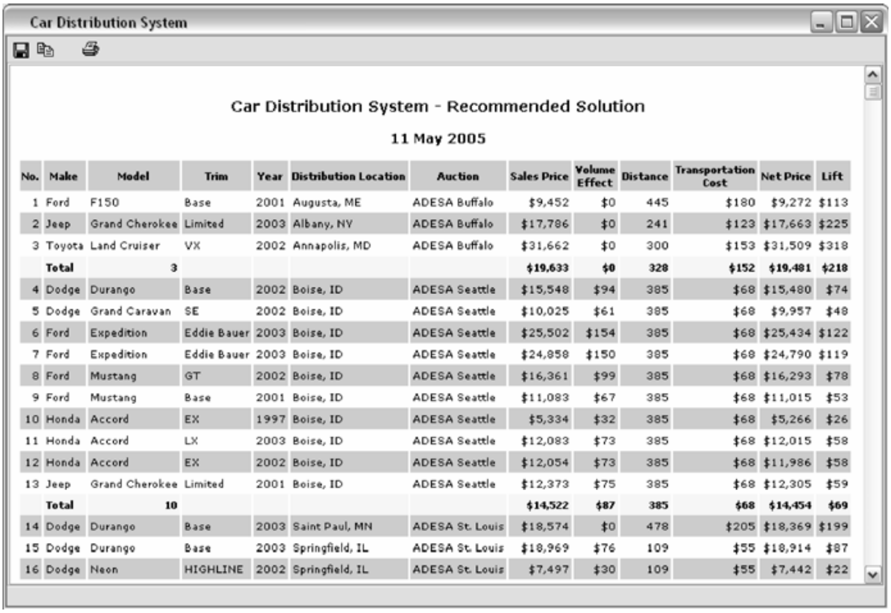
In this graph, the average transportation cost per car decreases from $275 for one car to just $47 for 14 cars. The graph also illustrates that the average transportation cost increases to about $62 when we need to transport 15 cars (because an additional truck is needed for the extra car). After the 15th car, the average transportation cost goes down again, with smaller spikes when additional trucks are needed.

[9] If we wanted to transport more than 6 cars, then the cost would be $400 for the first 6 cars ($250 plus $150 for 6 cars), plus $30 for each additional car. Hence, to transport 8 cars, the cost would be $400 for the first 6 cars, plus $60 for 2 additional cars, for a total of $460. Another price break occurs at the 11th car, and the incremental cost changes to $35 per car.

Besides the various transportation cost functions, the Adaptive Business Intelligence system also uses the inventory levels for each auction site to calculate several important parameters that are required for the optimization process. One of these parameters is the volume effect, which is based on how many similar makes/models (or cars of the same color) are present at a specific auction site. Another important parameter is the anticipated sale date. If we have 1,200 cars at a particular auction site (or in transit), and we know that about 500 are sold at each weekly auction session, then we can assume that a car shipped today will be sold in the third auction session. Therefore, we need to adjust the predicted sale price to take into account the depreciation and seasonality effect for these three weeks. Once the distribution solution has been approved, the auction inventory is updated with the new cars that have been assigned to each auction. Lastly, the cars that have been recently sold at these auction sites are removed from the inventory.[10]

11.2.2 Reporting

The graphical user interface also provides a set of reports that can be used to analyze a particular solution, or the efficiency of the Adaptive Business Intelligence system over some period of time. There are reports on the configuration settings, distribution of cars with different groupings, and auction inventory prognoses. The sample report below shows the distribution of cars grouped by auction site:



Car Distribution System - Recommended Solution

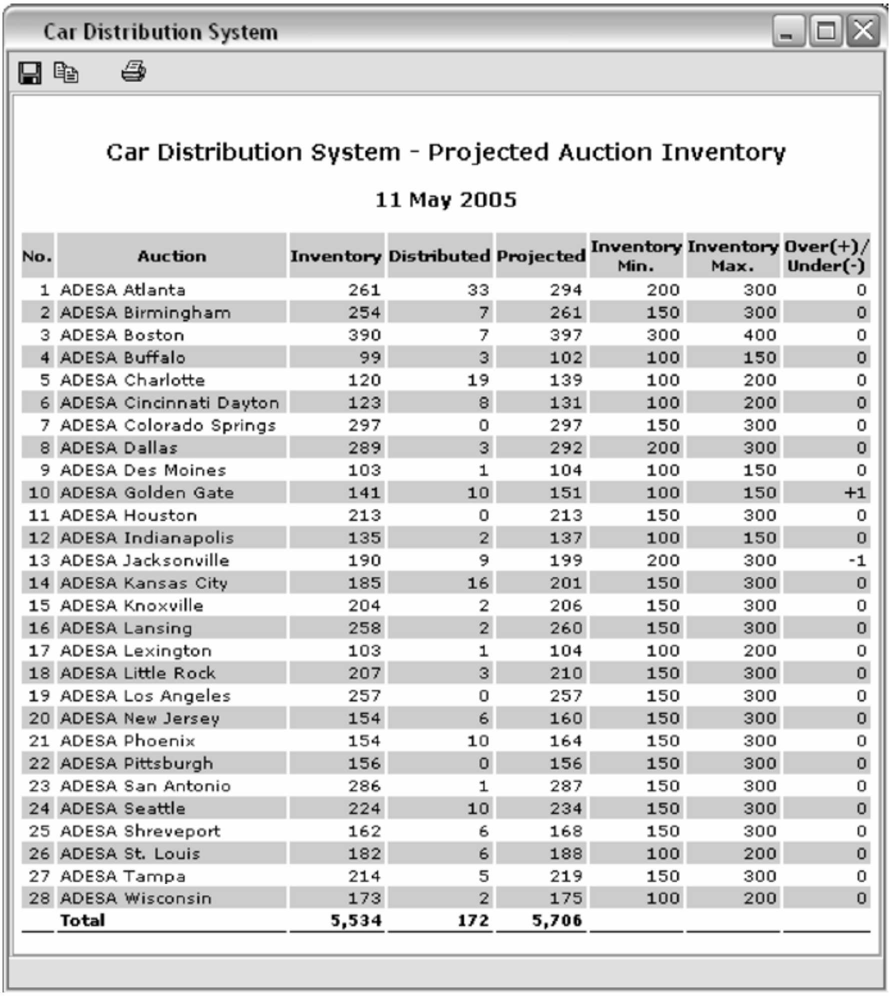
11 May 2005

No.	Make	Model	Trim	Year	Distribution Location	Auction	Sales Price	Volume Effect	Distance	Transportation Cost	Net Price	Lift
1	Ford	F150	Base	2001	Augusta, ME	ADESA Buffalo	$9,452	$0	445	$180	$9,272	$113
2	Jeep	Grand Cherokee	Limited	2003	Albany, NY	ADESA Buffalo	$17,786	$0	241	$123	$17,663	$225
3	Toyota	Land Cruiser	VX	2002	Annapolis, MD	ADESA Buffalo	$31,662	$0	300	$153	$31,509	$318
	Total	**3**					**$19,633**	**$0**	**328**	**$152**	**$19,481**	**$218**
4	Dodge	Durango	Base	2002	Boise, ID	ADESA Seattle	$15,548	$94	385	$68	$15,480	$74
5	Dodge	Grand Caravan	SE	2002	Boise, ID	ADESA Seattle	$10,025	$61	385	$68	$9,957	$48
6	Ford	Expedition	Eddie Bauer	2003	Boise, ID	ADESA Seattle	$25,502	$154	385	$68	$25,434	$122
7	Ford	Expedition	Eddie Bauer	2003	Boise, ID	ADESA Seattle	$24,858	$150	385	$68	$24,790	$119
8	Ford	Mustang	GT	2002	Boise, ID	ADESA Seattle	$16,361	$99	385	$68	$16,293	$78
9	Ford	Mustang	Base	2001	Boise, ID	ADESA Seattle	$11,083	$67	385	$68	$11,015	$53
10	Honda	Accord	EX	1997	Boise, ID	ADESA Seattle	$5,334	$32	385	$68	$5,266	$26
11	Honda	Accord	LX	2003	Boise, ID	ADESA Seattle	$12,083	$73	385	$68	$12,015	$58
12	Honda	Accord	EX	2002	Boise, ID	ADESA Seattle	$12,054	$73	385	$68	$11,986	$58
13	Jeep	Grand Cherokee	Limited	2001	Boise, ID	ADESA Seattle	$12,373	$75	385	$68	$12,305	$59
	Total	**10**					**$14,522**	**$87**	**385**	**$68**	**$14,454**	**$69**
14	Dodge	Durango	Base	2003	Saint Paul, MN	ADESA St. Louis	$18,574	$0	478	$205	$18,369	$199
15	Dodge	Durango	Base	2003	Springfield, IL	ADESA St. Louis	$18,969	$76	109	$55	$18,914	$87
16	Dodge	Neon	HIGHLINE	2002	Springfield, IL	ADESA St. Louis	$7,497	$30	109	$55	$7,442	$22

[10] The data about sold cars are also used to tune the prediction module (which is explained later in this chapter).

This sample screen shows all the cars that are to be distributed on the 11th of May, 2005, specifying the distribution center, recommended auction site, predicted sale price, transportation cost, and other data.

The projected auction inventory report below shows the inventory at each auction, the number of cars being sent to each auction, the projected number of cars at each auction, and whether or not the inventory constraints are violated:



Car Distribution System - Projected Auction Inventory

11 May 2005

No.	Auction	Inventory	Distributed	Projected	Inventory Min.	Inventory Max.	Over(+)/ Under(-)
1	ADESA Atlanta	261	33	294	200	300	0
2	ADESA Birmingham	254	7	261	150	300	0
3	ADESA Boston	390	7	397	300	400	0
4	ADESA Buffalo	99	3	102	100	150	0
5	ADESA Charlotte	120	19	139	100	200	0
6	ADESA Cincinnati Dayton	123	8	131	100	200	0
7	ADESA Colorado Springs	297	0	297	150	300	0
8	ADESA Dallas	289	3	292	200	300	0
9	ADESA Des Moines	103	1	104	100	150	0
10	ADESA Golden Gate	141	10	151	100	150	+1
11	ADESA Houston	213	0	213	150	300	0
12	ADESA Indianapolis	135	2	137	100	150	0
13	ADESA Jacksonville	190	9	199	200	300	-1
14	ADESA Kansas City	185	16	201	150	300	0
15	ADESA Knoxville	204	2	206	150	300	0
16	ADESA Lansing	258	2	260	150	300	0
17	ADESA Lexington	103	1	104	100	200	0
18	ADESA Little Rock	207	3	210	150	300	0
19	ADESA Los Angeles	257	0	257	150	300	0
20	ADESA New Jersey	154	6	160	150	300	0
21	ADESA Phoenix	154	10	164	150	300	0
22	ADESA Pittsburgh	156	0	156	150	300	0
23	ADESA San Antonio	286	1	287	150	300	0
24	ADESA Seattle	224	10	234	150	300	0
25	ADESA Shreveport	162	6	168	150	300	0
26	ADESA St. Louis	182	6	188	100	200	0
27	ADESA Tampa	214	5	219	150	300	0
28	ADESA Wisconsin	173	2	175	100	200	0
	Total	**5,534**	**172**	**5,706**			

The graphical user interface is capable of generating many other reports, and can be customized to provide business managers with the exact knowledge they need.

11.3 Prediction Module

Although many data sources report the auction sale prices of cars, each of these sources has some inherent advantages and disadvantages. For instance, the Black Book data provides "regional" sale prices, and each region usually contains several states and more than a dozen auction sites. Consequently, the actual sale price in certain states and auction sites is likely to differ from the average regional sale price. As an example of this, consider a car rental company that periodically dumps large numbers of rental cars at one specific auction site. Because of the volume effect, the sale price for some cars at this auction would be substantially different from the Black Book regional sale price.

Another important data source is the Manheim Market Report, as it tells us the sale prices of all cars sold at the auction sites owned by Manheim. Although these data are quite detailed (unlike the Black Book data), it would be quite difficult to make accurate predictions using only these data. For example, imagine that we want to sell a blue 2004 Toyota Camry with 23,000 miles at a specific auction site, but the closest matches in the most recent Manheim Market Report are:

- A blue 2004 Toyota Camry with 30,000 miles that sold for $9,500.
- A white 2004 Toyota Camry with 22,000 miles that sold for $9,000.

It looks like blue Toyota Camry cars sell for a premium at this auction site, and we should get more than $9,500 because our car has fewer miles. However, how much more can we expect to get? Also, what if the blue Toyota that sold for $9,500 had some fancy wheels that increased the price by $800, and the color really had very little to do with the sale price?

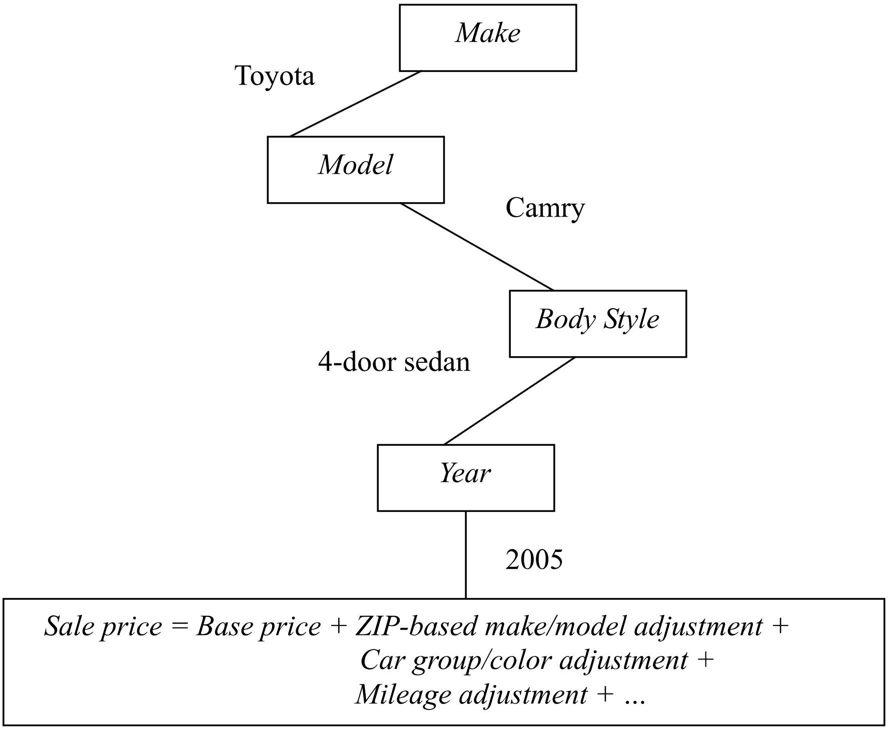
In addition to these external data sources, the sales data collected by the leasing company are also very important. Even though these data are usually quite sparse (e. g., the historical records for some makes/models might be limited to only a few cases), it can be analyzed to see if the sale prices of the company's off-lease cars deviate from the sale prices that are published by the external data sources. Such deviations may happen if a leasing company handles many ex-rental cars, which sell for substantially less than identical off-lease cars that were not used as rentals. Also, some deviations can be more prominent at certain auction sites than others.

On their own, each of these sources is somewhat limited, but together they provide an excellent data set for building and training various prediction models. During the data mining and model construction process, the goal is to identify the most important variables that influence the price of the car. If we start with the general data (such as the Black Book), we will discover that the most important factors influencing the car price are model year, make, and model. Consequently, the general data provide a convenient starting point, because it covers all the variables and regions. The auction-specific data (such as the Manheim Market Report) are then ideal for tuning the various prediction models, and the company's sales data are useful for detecting price deviations for certain cars at certain auction sites.

The resulting prediction module used in this Adaptive Business Intelligence system is based on decision trees (see Sect. 5.2.3), and generates sale price predictions in the following sequence of steps:

1. *Base price*. The prediction module generates a predicted “base price” for a car based on its make, model, body style, and year.
2. ZIP-based make/model adjustment. Some makes/models may sell for a premium or discount in certain regions, so the prediction module adjusts the base price for specific makes/models in specific regions (e. g., Chevrolet Corvettes might sell for a $300 premium in Florida and California, and a $600 discount in Montana and Idaho).
3. *Car group/color adjustment*. Some car groups/colors may sell for a premium or discount irrespective of the region, so the prediction module adjusts the base price for specific car groups and colors (e. g., yellow Chevrolet Corvettes might sell for a $500 premium, while a green one sells for a $1,000 discount).
4. *Mileage adjustment*. The prediction module adjusts the base price for mileage and *model-year-age*. The “model-year-age” is the age of a car according to its model year (i. e., when the 2005 Chevrolet Corvette became available in August 2004, the model-year-age of the 2004 Chevrolet Corvette became 1).
5. *Depreciation adjustment*. The prediction module adjusts the base price for daily depreciation, which is calculated from the day a car was returned to the predicted sale day. The daily depreciation rate is higher in the summer months preceding the introduction of new models. Consequently, the depreciation rate starts increasing from June, reaches its highest value in August, and then decreases to lower than average values for October, November, and December.
6. *Seasonality adjustment*. Some makes/models may sell for a premium or discount in certain regions at different times of the year, so the prediction module adjusts the base price for specific makes/models during certain seasons (e. g., convertible Chevrolet Corvettes may sell for a $1,800 discount in the northern states during the wintertime).
7. *UVC adjustment*. The Universal Car Code (UVC) component provides a more detailed car specification than the VIN. In the cases where the UVC is available, the prediction module adjusts the base price for additional options (e. g., the UVC might reveal that a specific Chevrolet Corvette is equipped with an upgraded suspension package).

As discussed in Sect. 5.2.3, there is no need to build a decision tree with every variable present at some level and a single, numeric value at each terminal node. Instead, the decision tree determines a car’s “base price” by its make, model, body style, and year, and then a linear model makes the abovementioned adjustments to the base price to come up with the final predicted sale price. Hence, to predict the price of a 2005 (“year”) Toyota (“make”) Camry (“model”) 4-door sedan (“body style”), the prediction module first finds the appropriate terminal node in the decision tree:


Make
Toyota
Model
Camry
Body Style
4-door sedan
Year
2005
Sale price = Base price + ZIP-based make/model adjustment +
Car group/color adjustment +
Mileage adjustment + ...

and then the linear model makes the necessary adjustments. It is quite possible that the base price is $18,500, but after making all adjustments the final predicted sale price is only $17,250. Note that the values for the different adjustments (e. g., mileage adjustment) might be different at each terminal node.

For an average load of cars, the prediction module follows the above steps to predict the final sale prices. However, if a company receives a large number of cars within one or more categories on a particular day, then the predicted sale prices are adjusted further to compensate for the volume effect. This additional adjustment is based on four factors:

- *Inventory.* A large inventory at any particular auction site would negatively affect the sale price of all cars sold at that particular auction.
- *Group.* Sending too many cars of the same group to a particular auction site would negatively affect the sale price of all cars within that particular group. For example, if we send an average number of 2002 Toyota Camry cars to an auction site but an excessive number of 2002 Honda Accord cars, then we can expect that the excess supply of Honda Accord cars would negatively affect the sale price of Toyota Camry cars. An excessive number of 2001 or 2003 Honda Accord cars would also negatively affect the sale price of 2002 Toyota Camry cars, but the effect would be less than that of having too many 2002 Honda Accord cars.

- *Make/model.* Sending too many cars of the same make/model to a particular auction site would negatively affect the sale price of these cars. For example, we would drive down the price of Chevrolet Corvettes by sending several truckloads to an auction site that already has plenty of Chevrolet Corvettes in inventory.
- *Color.* Sending too many cars of the same color (within a car group or make/model category) to a particular auction site would negatively affect the sale price of these cars. For example, sending an excessive amount of white mid-size cars to a particular auction site would drive down the sale price of all white mid-size cars, irrespective of make, model, or year.

The prediction module contains numerous parameters (different values for various adjustments at different terminal nodes), which are automatically updated by the adaptability module (discussed in Sect. 11.5) to capture changing trends in the used car marketplace.

11.4 Optimization Module

As we discussed in Sect. 4.3, the optimization module generates a possible solution that serves as input data for the prediction module. This input data provide a destination assignment (i. e., auction site) for each car, which the prediction module uses to generate the predicted sale prices. The optimization module then uses the sum of all the predicted sale prices (i. e., the output data) to gauge the quality of the input data: The higher the sum of the predicted sale prices, the better the distribution solution. Hence, there is a strong relationship between the prediction and optimization modules.

In this Adaptive Business Intelligence system for distributing cars, the optimization module employs several different techniques (i. e., the hybrid system approach discussed in Sect. 10.2) that use different solution representations. For instance, the evolutionary algorithms represent a solution based on indirect representation and some preprocessing (see Sect. 6.1), where all the available auction sites are sorted by *distance* from a particular car. In other words, auction 1 is the closest (distance-wise), auction 2 is the second closest, and so forth. Hence, each solution is represented by a vector of auction site indices (relative to a particular car), and the length of the vector is equal to the number of cars to be distributed:

3	4	4	…	1	1

This vector represents a solution where the first car is shipped to the third closest auction site (for this particular car), the second car is shipped to the fourth closest auction site (for this particular car), the third car is shipped also to the fourth closest auction site (note, however, that the second and third car are most likely shipped to different auction sites, as the fourth closest auction sites for the second and third car need not be the same), and so on, with the last two cars being shipped to the closest auction sites. In this particular implementation of evolutionary algorithms, the optimization module applies the elitist strategy, which forces the best solution from one generation to the next, as well as various mutation and crossover operators that were discovered after many experiments.

Furthermore, to make sure that only feasible solutions are generated (in terms of hard constraints), the mutation operator in the evolutionary algorithm is limited to using vector values that do not violate any constraints. For example, if the first car can only be sent to five auction sites, then the value for this position in the vector would be limited to those five auction sites. If the mutation operator decides to change this position in the vector to a value outside of these five auction sites, then the mutation will skip this position and leave its value unchanged. Such simple rules guarantee that infeasible solutions will not be created from feasible parents. As for the crossover operator, it cannot produce an infeasible solution from a feasible parent solution, because if the vector values in both parent solutions represent feasible auction sites for each car then the offspring solution will also have feasible vector values. Of course, both the mutation and crossover operators can create offspring solutions that violate an inventory constraint, but this constraint is considered to be soft and can be addressed by using penalties.

As we discussed in Sect. 10.2, each day brings a different "instance" of the same car distribution problem, as changes occur in the number and type of cars to be distributed. For this reason, the optimization module is based on the hybrid system approach, and the module itself contains a few features of adaptability. For instance, because the effectiveness of the mutation and crossover operators used within the evolutionary algorithm vary as the problem instance varies, the optimization module can adapt the execution frequency of these operators. To accomplish this, the optimization module records the outcome of each operator and counts its "effectiveness" rate. The optimization module then increases the execution frequency of the more effective operators, and decreases the execution frequency of the less effective ones. This approach provides the optimization module with some adaptability to better handle different problem instances.

In any real-world implementation of an Adaptive Business Intelligence system, while implementing the optimization module, we have also to consider two additional issues:

- "Run-time" performance.
- Finding a solution that is close to the global optimum, rather than an inferior local optimum.

From a software point of view, the issue of run-time performance is related to the fact that the various techniques inside the optimization module are exploring a vast number of possible solutions to avoid being guided to a local optimum. In light of this, a very common approach to increasing run-time performance is to pair up a global optimization technique with a hill climber that takes over when there is no longer any significant improvement in the quality measure score. Hence, the evolutionary algorithm might generate 3,000 initial solutions that correspond to the total number of cars being processed on a particular day. These 3,000 individual solutions are evaluated, the parents are selected, crossover and mutation operators are applied, the next generation of solutions is created, and so on. This process is repeated until the improvement in the quality measure score of the best solutions in subsequent generations becomes stagnant, at which point the hill climber is applied to find the best individual solution within the neighborhood of the best solutions found thus far.

From a hardware point of view, a distributed computing environment can decrease the run-time of the optimization module. By spreading the different techniques or evaluations over many processors, the optimization module can gain an almost linear speedup of execution. Note that many population-based optimization techniques (discussed in Chap. 6) are especially well suited for parallelization, as each solution in the population can be assigned to a dedicated processor. Also, to maximize the run-time performance of this type of implementation, the optimization module can control the amount of run time allocated to each technique.

The second issue – that of finding a solution that is close to the global optimum, rather than a local optimum – may arise if a large number of hard constraints are used. In such cases, the global optimum may be surrounded by a wall of infeasible solutions, which may prevent an optimization technique from exploring that area of the search space. The hybrid systems approach overcomes this problem, because even though the evolutionary algorithm (in this particular implementation) cannot process infeasible solutions, there are several other techniques running in parallel that can. By sharing this information during the search (i. e., cooperating), the evolutionary algorithm can "jump" over this wall of infeasible solutions if it gets information from another technique that higher-quality solutions exist on the other side.

In summary, the optimization module used in this Adaptive Business Intelligence system can produce a solution that is close to the global optimum in a reasonable running time. By using certain features of adaptability, the optimization module can also handle a wide variety of different instances of the car distribution problem. Lastly, the usage of several optimization techniques together generates a solution that is better than the result of any single technique.

11.5 Adaptability Module

Besides making regular (daily) recommendations on where to ship cars, the Adaptive Business Intelligence system also provides a detailed sale price prediction for each car. These sale price predictions are recorded in the system's database, along with the actual sale prices when the cars are eventually sold at auction. These actual sale prices constitute the *recent output*, and the auction sites, dates, and detailed information about each car constitute the *recent input*. Thus, for each car, the Adaptive Business Intelligence system can compare the predicted sale price with the actual sale price. Such comparisons are very useful for detecting pricing trends in the used car marketplace.

For example, a new report that shows gray-colored cars to be involved in more accidents than other colors in the Midwest (because gray cars "blur" with the road during hot weather) may cause the price of used gray colors to fall in this region of the United States. If this happened, the system would need to identify the trend, and then adapt the parameters of the prediction module so that fewer gray cars are shipped to the auction sites located in Midwest states. To accomplish this, the adaptability module would take the recent output and input, and adapt the parameters of the prediction module to decrease the prediction error. In other words, the adaptability module would "update" the prediction module so it makes better predictions in light of the changes that have occurred in the environment.

Recall from Sect. 11.3 that the prediction module generates a predicted base price for each car, which is based on its make, model, body style, and year. The prediction module then "adjusts" this base price to come up with the final predicted sale price for each car. Let us illustrate this process with an example: Say the prediction module predicts that the base price for a 2005 Toyota Camry 4-door sedan is $18,500. Because the system considers shipping this car to an auction site in Florida, the following adjustments are made: ZIP-based make/model adjustment ($320 premium on this type of car), car group/color adjustment ($120 premium on silver cars), mileage adjustment ($750 penalty for exceeding the average mileage), depreciation adjustment ($280 penalty for depreciation, as the car will probably be sold in 3 weeks' time), and so forth.

The adaptation is done on the basis of actual car prices obtained at the auction sites and is done in two stages. First, the value of the base price is adapted. For each make, model, body style, and year (i. e., for each branch leading to the terminal node of the decision tree), the adaptability module determines a new value for the base price. The applied technique is similar to exponential smoothing (Sect. 5.2.1), as it assigns weights to past cases to distinguish between recent and older cases. Hence, a prediction for the time $t{+}1$ is calculated as:[11]

$$\text{Base price}\,(\text{t+1}) = (\alpha \times \text{Recent}\,(\text{t})) + ((1-\alpha) \times \text{Base price}\,(\text{t}))$$

so the prediction for the next (future) case is calculated as a total of two values: the recent cases (*Recent(t)*) with parameter α, and the last prediction of the base price (*Base price(t)* with the weight $1-\alpha$). Note that parameter α assigns the significance of the recent cases in making the prediction. If α is close to *1*, then the older cases would be ignored to a large extent. If α is close to *0*, then the most recent cases would be ignored to a large extent. This is important, as the value of the parameter α depends on the number of cars sold in the appropriate category (e. g., the number of 2004 Toyota Camry 4-door cars): the smaller the number of sold cars, the smaller the value of α. The adaptability module updates the base price every three months.

[11] Note that recent cases provide us with sale prices, so additional transformations (based on the current values of appropriate adjustments) are necessary to estimate their base prices.

The adaptability module updates the other adjustments in a similar way. For example, to tune the value of the ZIP-based make/model adjustment (for a particular category of a car) in the linear model:

$$\begin{aligned}\text{Sale price} &= \text{Base price} + \text{ZIP-based make/model adjustment}\\ &\quad + \text{Car group/color adjustment} + \text{Mileage adjustment} + \ldots\end{aligned}$$

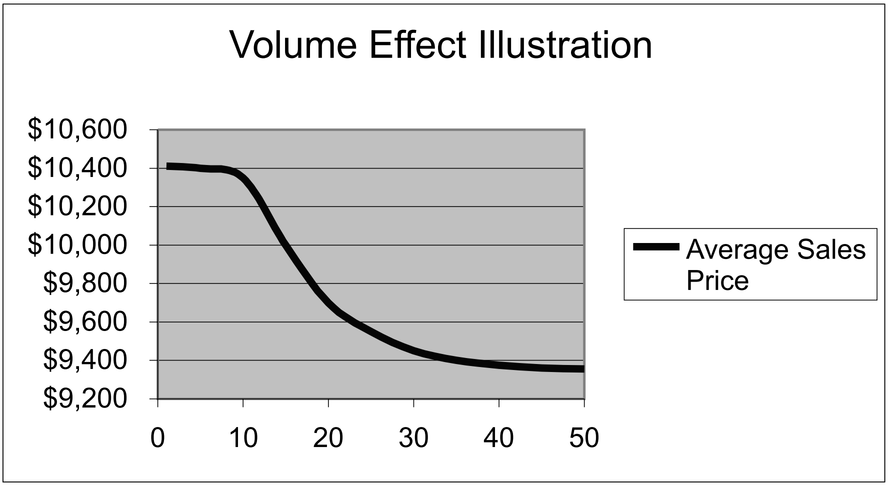
the adaptability module compares the national sales records with ZIP-specific sales records. To tune the car group/color adjustment, the adaptability module compares national sales records of a particular car with the sales records of a specific color of the same car, and so on.

As with the updates to the base price, a new value of an adjustment is based on its current value and the recent cases. For example

$$\begin{aligned}\text{ZIP-based make/model adjustment}\,(t{+}1) &= (\beta \times \text{Recent}\,(t))\\ &\quad + ((1-\beta) \times \text{ZIP-based make/model adjustment}\,(t))\end{aligned}$$

so the prediction for the next (future) case is calculated as a total of two values: the recent cases (*Recent(t)*) with parameter β, and the last prediction of the ZIP-based make/model adjustment (*ZIP-based make/model adjustment(t)* with the weight $1-\beta$). As for parameter α, parameter β assigns the significance of the recent cases in making the prediction. Again, the value of parameter β depends on the number of cars sold in a category: the smaller the number of cars sold, the smaller the value of β. The adaptability module updates each adjustment at different intervals, which usually range from one to six months. Although the adaptability module determines the values of parameters α, β, etc. on the basis of a lookup table in this particular implementation, it is also possible for the adaptability module to adapt these values.

As we indicated in Sect. 3.5, we might inadvertently trigger the "volume effect" by sending too many similar cars to the same auction site. In such cases, the sale price determined by the prediction module is adjusted further to compensate for the volume effect. This additional adjustment is very different from the previous ones, and is based on four factors: inventory, group, make/model, and color. Here the prediction module has to take into account the distribution of all cars before making any adjustments. Thus, the sale price of each car is adjusted further to reflect the number of similar cars sold at each auction site. Recall the volume effect illustration from Sect. 3.5:



The shape of the curve in this figure is defined by the parameters p and q, which define the decay rate of the price as a function of the number of cars sold. These parameters are adjusted on a quarterly basis, which means that the shape of the volume effect curve changes every three months. As with the other adjustments, the updates for parameters p and q are based on their current values and the recent cases.

In summary, this particular implementation of the adaptability module required a significant effort in identifying and adapting all the relevant parameters. This is not surprising, given that the functionality of the adaptability module depends on the availability and the amount of new data, the frequency of their arrival, and many other problem-specific details (e. g., the importance of mileage, color, and volume effect in making adjustments).

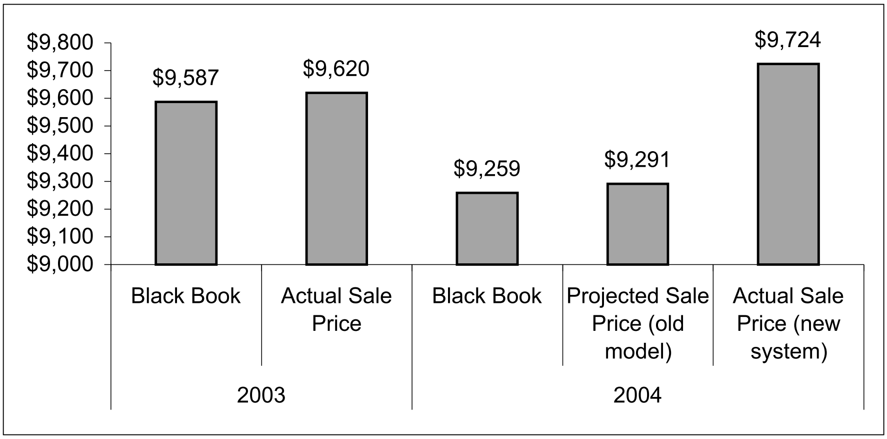
11.6 Validation

The Adaptive Business Intelligence system discussed in this chapter was designed and developed by the authors of this book to help leasing companies make near-optimal decisions on the distribution of off-lease cars. As discussed earlier, the car distribution problem is extremely complex and the system addresses the issues of transportation, volume effect, price depreciation, inventory levels, risk factors, and dynamic market changes.

When used in a high-volume production setting – where thousands of cars are returned each day – the Adaptive Business Intelligence system can generate a net profit lift of hundreds of millions of dollars per year. There are a few ways of validating this lift. One way is to divide the daily load of cars into two equal sets with an almost identical division of makes/models. One set would be distributed using the “old” method, whereas the other set would be distributed using the

Adaptive Business Intelligence system, and then the results can be compared when all the cars are sold. Another way would be to use the old method on selected days of the week (e. g., Mondays, Wednesdays, and Fridays) and the Adaptive Business Intelligence system on the remaining days (e. g., Tuesdays and Thursdays). Again, the results can be compared later, when all the cars are sold.

Another way to estimate the lift (in cases where a company may not want to use two methods simultaneously) is to use the Adaptive Business Intelligence system for one year and then compare the average sale prices with those of the previous year (which were distributed using the old method). The comparison should use a trusted pricing source as a benchmark. For example, if we have data about the cars sold by a company in 2003 (before the implementation of an Adaptive Business Intelligence system), we can select a set of cars that have the same mix of makes/models, year, trim, etc. and compare the average sale price of these cars with the average Black Book sale price. A chart depicting this comparison is presented below:



In this example, the average Black Book sale price for a particular mix of makes/models, year, trim, etc. in 2003 was $9,587 per car, and the company sold these cars for an average of $9,620 per car, or 0.344% higher than the Black Book sale price. The next step would be to compare the sale prices in 2004 (when the Adaptive Business Intelligence system replaced the old method of distributing cars) against the Black Book sale prices for that year. In this example, the average Black Book sale price was $9,259 per car in 2004, and the average actual sale price obtained by the system was $9,724. If the cars had been distributed using the old method in 2004, then the company would have attained similar results to those attained in the previous year (i. e., a 0.344% improvement over the Black Book benchmark, or an average of $9,291 per car). We can attribute an increase in average sale price to the Adaptive Business Intelligence system of $9,724 minus $9,291, or $433 per car.

In Sect. 5.3, we covered the difficulties in validating various prediction models on *historic* data. In particular, we discussed splitting the available data set into a training set (for building a prediction model), validation set (for tuning the parameters of the model), and test set (for evaluating the performance of the model). In this section, we discussed how the *whole* system could be validated using *real-time* data. There is an enormous difference between these two validations, the most important being that real-time data can only be processed once (e. g., if a car is sold at auction X, we will never know what price the car would have fetched at auction Y). Consequently, different validation techniques are necessary.

12 Applying Adaptive Business Intelligence

> "I knew not what wild beast we were about to hunt down in the dark jungle of criminal London, but I was well assured, from the bearing of this master huntsman, that the adventure was a most grave one."
> *The Adventure of the Empty House*

> "There's money in this case, Watson, if there is nothing else."
> *A Scandal in Bohemia*

The car distribution example that we have used throughout this book has served its purpose in underscoring the importance of Adaptive Business Intelligence. Without a doubt, software systems that can make decisions and adapt to changes in the marketplace are the future of the business intelligence industry! As discussed in Chap. 2, many real-world business problems have similar characteristics to the car distribution problem: an enormous number of possible solutions, many complex constraints, and a time-changing environment. Therefore, we will discuss several other business problems in this chapter – ranging from production-line optimization to fraud detection – and look at possible solutions from the viewpoint of Adaptive Business Intelligence.

12.1 Marketing Campaigns

Many large corporations spend millions of dollars each year to advertise their products and services on different television channels. However, rather than buying airtime directly from the media stations, most of these companies use specialized advertising firms to handle their marketing campaigns. These campaigns have to achieve several objectives, the most important of which is to reach the target market while complying with predefined business rules and limitations.[12] Because advertising firms first purchase airtime from media stations and then later allocate this time to different brands (clients), they face a major optimization problem: How to allocate the purchased airtime among the various brands? This optimization task (of allocating purchased airtime among different brands) has two objectives: The first is to maximize the number of *target rating points* for the primary target, and the second is to maximize the *audience reach*.

[12] Examples of such rules and limitations may include budget constraints, or restrictions on the use of advertisements within certain time intervals.

Before delving into this problem, let us first define what *target rating points* and *audience reach* mean in the context of "reaching the target audience":

- The *target rating points* is the percentage of the target audience that is watching a specific program at a specific point in time. Let us say that the primary target audience is "22- to 29-year-old males," which consists of 607,500 people.[13] If the advertisement broadcast during Program A is viewed by 12,000 of these people, then Program A receives 2 Target Rating Points (12,000 / 607,500 = 2%).
- *Audience reach* is the percentage of the target audience that will see the advertisement at least once. For example, if the advertisement is aired during Program A twice, then the total number of target rating points will be 4. However, it does not mean that the advertisement will be seen by 4% of the target audience, as it is very likely that some people will see the advertisement twice. If the target audience is 607,500 people and Program A is watched by 12,000 of these people, then 5,000 people might see the advertisement twice, 7,000 might see the first broadcast but not the second, and the remaining 7,000 might miss the first broadcast but catch the second. In this case, the percentage of the target audience that will see the advertisement at least once is 3.13% (5,000 + 7,000 + 7,000 = 19,000 / 607,500), so the audience reach[14] is 3.13.

The target rating points for each program are obtained by researching the popularity of different programs among various consumer groups**.** The target rating points are readily available for programs that aired in the past. However, because the task of allocating airtime is performed before the programs are aired, it is necessary to predict what their target rating points will be. This is the responsibility of the prediction module of the Adaptive Business Intelligence system.

The prediction module needs to take into consideration all the factors that influence the target rating points for different programs. Seasonality is a significant factor (among others), because the number of target rating points generated by a program during Christmas week might be very different from the number of points generated during a regular week in September. To identify the impact that factors such as seasonality have on different programs, it is necessary to analyze the available data (i. e., conduct some data mining activity). During this process, the programs are classified into groups that have similar characteristics, and the appropriate adjustments are calculated for each day of the year. Additional things to consider during the data mining process include program trends (e. g., certain programs steadily gain or lose popularity over time) and the calendar of events

[13] The audience is the number of people that can see the advertisement. For example, if an advertisement is aired on five paid channels, and those channels have 4,500,000 subscribed households with an average of 2.7 people per household, then the audience is 12,150,000 people (4,500,000 × 2.7). If the target audience is "22- to 29-year-old males" and they constitute 5% of the total audience, then the size of the target audience is = 607,500 people (12,150,000 × 5%).

[14] Many marketing and advertising theories state that an advertisement needs to be viewed by the target audience several times to be effective.

(e. g., Super Bowl Sunday, election day). In such cases, the predicted target rating points will require further adjustment.

Note that predicting the target rating points is a typical time series problem (see Sect. 5.2.1). The problem might be expressed as follows:

Given $v[1], v[2], \ldots, v[t]$, predict the value of $v[t+1]$

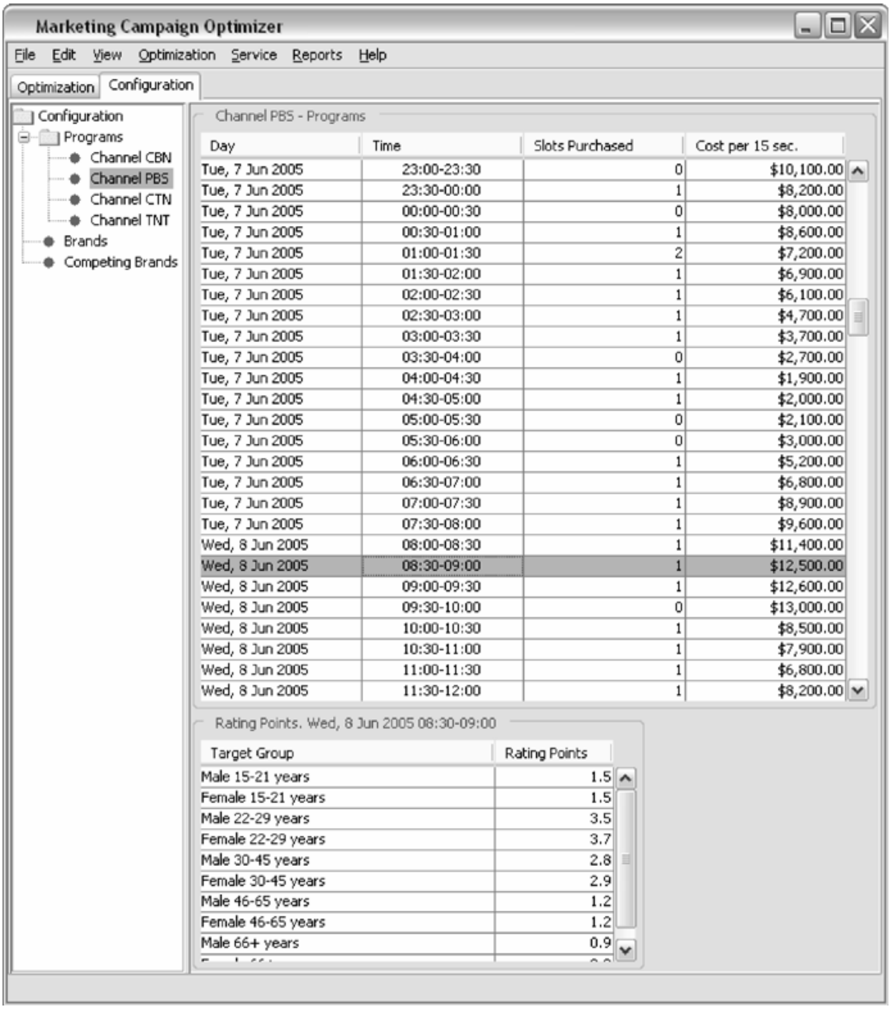
where v is the target rating points and t is the time of the most recent market research data. Note also that because other variables are available (e. g., calendar of events) beside the target rating points from earlier time intervals, we are talking about a composite forecasting model, which consists of past time series values, past variables, and past errors.

As with the other applications of Adaptive Business Intelligence, an adaptability module is required to make the system capable of learning and recognizing new trends in the marketplace. This module is responsible for adapting the parameters of the prediction module. Each day brings a new set of program ratings for recently aired shows, and these ratings need to be taken into account when making future predictions. Thus, the adaptability module adjusts the appropriate coefficients in the model to accommodate the new data, so that future predictions are more accurate.

Before we turn our attention to the optimization process, note that each day many programs are broadcast on a multitude of different channels. If there are a few dozen brands to be advertised during different programs on different channels, then the number of possible advertising schedules can be extremely large. A typical scenario might involve 5,000 time slots to be filled by advertisements of 50 brands (so each brand would require 100 advertisement slots on average). In this case, there are more than 10^{200} ways to allocate the 100 slots of *just the first brand!* This number is already much larger than the estimated number of atoms in the Universe, so for 50 brands the number of possible allocations is truly overwhelming!

To find the best allocation of airtime for the different brands, the optimization module takes into account the predicted target rating points for each program, as well as various brand-specific constraints. The most important of these is the *budget constraint*, which is the advertising campaign budget that should not be exceeded for each brand. Another important constraint is that some (competing) brands should not be aired too closely (e. g., broadcasting an advertisement from Coca Cola followed by one from Pepsi Cola on the same channel). Some companies might impose additional requirements that would become constraints for their particular brands. For example, they might require that a portion of their advertisement campaign be aired on certain weekdays and weekends, or during prime time.

Before the optimization process can begin, the optimization module loads several input data files, which are summarized in the following configuration screens. The first configuration screen (presented in the sample screen below) provides information on purchased slots:


Marketing Campaign Optimizer
File Edit View Optimization Service Reports Help
Optimization Configuration
Configuration
Programs
Channel CBN
Channel PBS
Channel CTN
Channel TNT
Brands
Competing Brands

Channel PBS - Programs

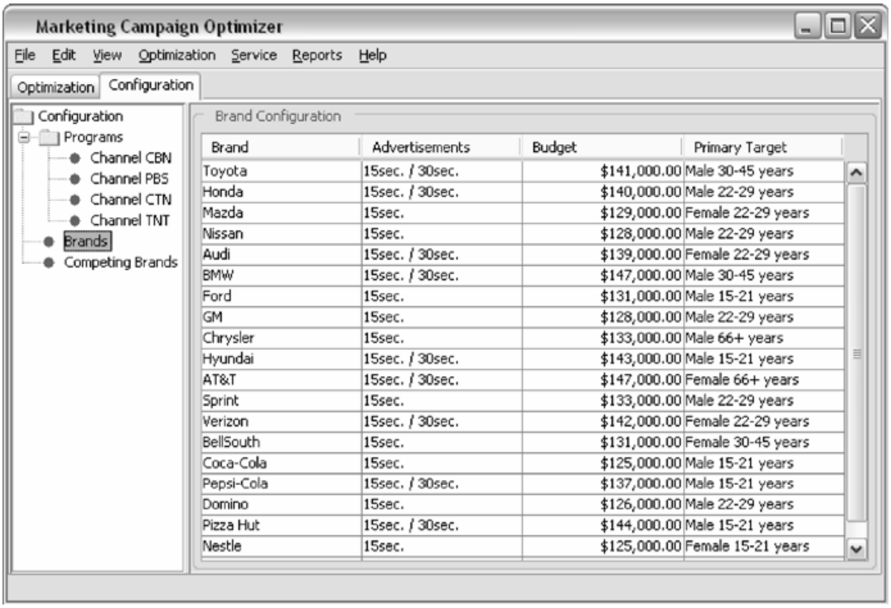
Day	Time	Slots Purchased	Cost per 15 sec.
Tue, 7 Jun 2005	23:00-23:30	0	$10,100.00
Tue, 7 Jun 2005	23:30-00:00	1	$8,200.00
Tue, 7 Jun 2005	00:00-00:30	0	$8,000.00
Tue, 7 Jun 2005	00:30-01:00	1	$8,600.00
Tue, 7 Jun 2005	01:00-01:30	2	$7,200.00
Tue, 7 Jun 2005	01:30-02:00	1	$6,900.00
Tue, 7 Jun 2005	02:00-02:30	1	$6,100.00
Tue, 7 Jun 2005	02:30-03:00	1	$4,700.00
Tue, 7 Jun 2005	03:00-03:30	1	$3,700.00
Tue, 7 Jun 2005	03:30-04:00	0	$2,700.00
Tue, 7 Jun 2005	04:00-04:30	1	$1,900.00
Tue, 7 Jun 2005	04:30-05:00	1	$2,000.00
Tue, 7 Jun 2005	05:00-05:30	0	$2,100.00
Tue, 7 Jun 2005	05:30-06:00	0	$3,000.00
Tue, 7 Jun 2005	06:00-06:30	1	$5,200.00
Tue, 7 Jun 2005	06:30-07:00	1	$6,800.00
Tue, 7 Jun 2005	07:00-07:30	1	$8,900.00
Tue, 7 Jun 2005	07:30-08:00	1	$9,600.00
Wed, 8 Jun 2005	08:00-08:30	1	$11,400.00
Wed, 8 Jun 2005	08:30-09:00	1	$12,500.00
Wed, 8 Jun 2005	09:00-09:30	1	$12,600.00
Wed, 8 Jun 2005	09:30-10:00	0	$13,000.00
Wed, 8 Jun 2005	10:00-10:30	1	$8,500.00
Wed, 8 Jun 2005	10:30-11:00	1	$7,900.00
Wed, 8 Jun 2005	11:00-11:30	1	$6,800.00
Wed, 8 Jun 2005	11:30-12:00	1	$8,200.00

Rating Points. Wed, 8 Jun 2005 08:30-09:00

Target Group	Rating Points
Male 15-21 years	1.5
Female 15-21 years	1.5
Male 22-29 years	3.5
Female 22-29 years	3.7
Male 30-45 years	2.8
Female 30-45 years	2.9
Male 46-65 years	1.2
Female 46-65 years	1.2
Male 66+ years	0.9

For each channel on the left-hand side of this screen, it is possible to define the available programs and their corresponding target rating points. The screen displays the number of purchased slots and their cost for the PBS channel. The target rating points are defined for each program and each target group. For example, the program running on 8 June from 8:30 am to 9:00 am (highlighted) costs $12,500 per 15-second time slot, and fetches 1.5 target rating points for 15- to 21-year-old males, 3.5 target rating points for 22- to 29-year-old males, and so on. These target rating points for each target group are generated by the prediction module.

The second configuration screen (presented below) summarizes the information related to different brands:

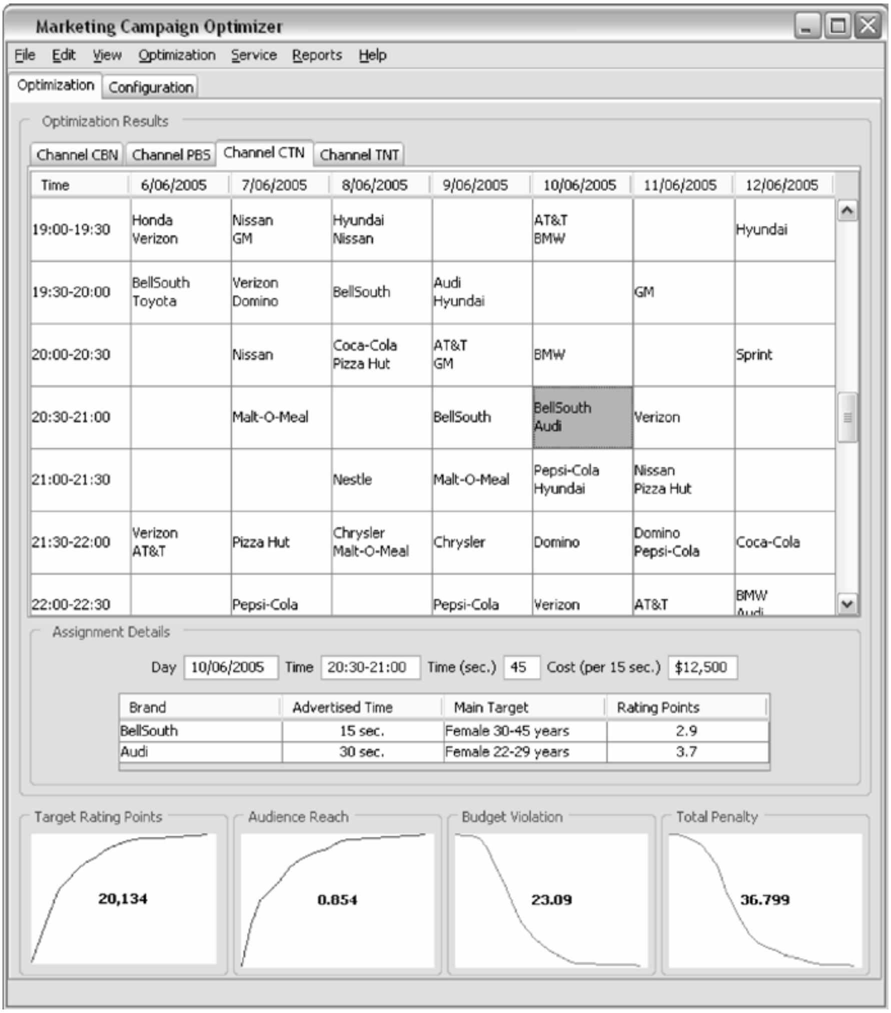


Brand	Advertisements	Budget	Primary Target
Toyota	15sec. / 30sec.	$141,000.00	Male 30-45 years
Honda	15sec. / 30sec.	$140,000.00	Male 22-29 years
Mazda	15sec.	$129,000.00	Female 22-29 years
Nissan	15sec.	$128,000.00	Male 22-29 years
Audi	15sec. / 30sec.	$139,000.00	Female 22-29 years
BMW	15sec. / 30sec.	$147,000.00	Male 30-45 years
Ford	15sec.	$131,000.00	Male 15-21 years
GM	15sec.	$128,000.00	Male 22-29 years
Chrysler	15sec.	$133,000.00	Male 66+ years
Hyundai	15sec. / 30sec.	$143,000.00	Male 15-21 years
AT&T	15sec. / 30sec.	$147,000.00	Female 66+ years
Sprint	15sec.	$133,000.00	Male 22-29 years
Verizon	15sec. / 30sec.	$142,000.00	Female 22-29 years
BellSouth	15sec.	$131,000.00	Female 30-45 years
Coca-Cola	15sec.	$125,000.00	Male 15-21 years
Pepsi-Cola	15sec. / 30sec.	$137,000.00	Male 15-21 years
Domino	15sec.	$126,000.00	Male 22-29 years
Pizza Hut	15sec. / 30sec.	$144,000.00	Male 15-21 years
Nestle	15sec.	$125,000.00	Female 15-21 years

The total budget, primary target group, and the length of airtime (15 seconds only, 30 seconds only, or both) are defined for each brand. In this particular example, Honda has a total budget of $140,000, uses both 15- and 30-second time slots, and the primary target group is 22- to 29-year-old males.

Finally, we are ready to optimize. A solution to this marketing problem might be represented as a multi-dimensional table where the first dimension represents possible channels, the second dimension represents the available programs, and the third dimension represents all the days of the month. Many different optimization techniques can be applied to this problem (e. g., evolutionary algorithms, simulated annealing, tabu search), and some of the problem-specific constraints might be best dealt with by using a decoder (a sort-of repair algorithm, which would “repair” the current solution to make it feasible). Furthermore, a hill climber can be used to improve the current solution by searching its neighborhood.

The sample screen below displays a possible solution:


Marketing Campaign Optimizer
File Edit View Optimization Service Reports Help
Optimization Configuration
Optimization Results
Channel CBN Channel PBS Channel CTN Channel TNT
Target Rating Points
20,134
Audience Reach
0.854
Budget Violation
23.09
Total Penalty
36.799

Time	6/06/2005	7/06/2005	8/06/2005	9/06/2005	10/06/2005	11/06/2005	12/06/2005
19:00-19:30	Honda Verizon	Nissan GM	Hyundai Nissan		AT&T BMW		Hyundai
19:30-20:00	BellSouth Toyota	Verizon Domino	BellSouth	Audi Hyundai		GM	
20:00-20:30		Nissan	Coca-Cola Pizza Hut	AT&T GM	BMW		Sprint
20:30-21:00		Malt-O-Meal		BellSouth	BellSouth Audi	Verizon	
21:00-21:30			Nestle	Malt-O-Meal	Pepsi-Cola Hyundai	Nissan Pizza Hut	
21:30-22:00	Verizon AT&T	Pizza Hut	Chrysler Malt-O-Meal	Chrysler	Domino	Domino Pepsi-Cola	Coca-Cola
22:00-22:30		Pepsi-Cola		Pepsi-Cola	Verizon	AT&T	BMW Audi

Assignment Details

Day 10/06/2005 Time 20:30-21:00 Time (sec.) 45 Cost (per 15 sec.) $12,500

Brand	Advertised Time	Main Target	Rating Points
BellSouth	15 sec.	Female 30-45 years	2.9
Audi	30 sec.	Female 22-29 years	3.7

In this sample screen, there are four tabs corresponding to the four different channels (which represent the first dimension of the solution). The table in the middle of the screen shows the allocation of airtime to the different brands, and the rows and columns in this table represent the second and third dimensions of the solution. In this particular example, it is recommended to broadcast BellSouth and Audi advertisements on 10 June from 20:30 to 21:30. In total, 45 seconds of advertisement time should be allocated for that time period at a cost of $12,500 per 15 seconds. The BellSouth advertisement has 2.9 target rating points for 30- to 45-year-old females, while the Audi advertisement has 3.7 target rating points for 22- to 29-year-old females.

The four graphs in the lower part of the screen show the progress of the optimization objectives during the optimization run. The current value of each graph is displayed in the corresponding window:

- The *Target Rating Points* window shows the total number of target rating points accumulated for all brands. Maximizing this number is an important objective of the optimization module.
- The *Audience Reach* window shows what percentage of the target audience is reached by the marketing campaign. As discussed earlier, this is another important objective of the optimization process, and a value of 85.4% is achieved in this particular example.
- The *Budget Violation* window shows if the current allocation exceeds the allowed budget (and by how much, in thousands of dollars). Another objective of the optimization module is to minimize this number, which the system can view as a soft constraint.
- The *Total Penalty* window shows the "general level" of constraint violation. This number includes the budget violation penalties plus penalties for violating other brand-specific constraints.

These four graphs show the optimization module in action, allowing the business manager to observe the progress made during the run (increases in target rating points, increases in audience reach, etc.).

Using an Adaptive Business Intelligence system to optimize marketing campaigns can result in better predictions and better allocations of television airtime. Such benefits translate into a reduction in marketing spend, as the same objectives can be achieved with a smaller budget. Companies that do not outsource their marketing campaigns could also use Adaptive Business Intelligence to decide what airtime to buy from what media stations, which is another (related) prediction and optimization problem.

12.2 Manufacturing

The optimization of production lines is a very challenging manufacturing problem, which involves elements of demand forecasting, scheduling of labor, and sequencing of production orders. Although the production process at many manufacturing companies might be conceptually straightforward, it contains many details that make scheduling quite complex. For example, an iron foundry that operates several furnaces and production lines has to schedule many independent processes, such as the preparation of cores and molds, pouring the molds, finishing the castings, and so forth. Furthermore, there are relationships between the "base-grades" and metal grades, as well as transition times for changing the melting process (on a furnace) from one iron base-grade to another. Because of these complexities, the efficient coordination of the overall operation can be quite difficult.

Using the iron foundry as an example, many production constraints have to be taken into account during the scheduling process. Some of these constraints represent physical limitations (e. g., melting time, the capacity of each furnace), while others may represent operational/business rules. The set of constraints and business rules may include:

1. Production lines that only operate on particular days of the week (e. g., from Monday morning to Thursday evening).
2. Products that may only be produced during the day or night shift.
3. Production deadlines that have to be satisfied.[15]
4. Earliest "start date" for each order (e. g., cores and molds need to be prepared before the order can be produced).
5. Products that cannot be produced in parallel (e. g., some products are very similar in appearance, which makes it difficult to separate them at the end of a production run).
6. Production limits, such as furnace tons per hour, furnace min/max tons per hour, molds per hour per production line, etc.
7. Some additional business rules can also include the avoidance of overproduction (for some orders it might be necessary to overproduce), the avoidance of deviating from production-line recommendations for processing "heavy" jobs, and so on.

In the application of Adaptive Business Intelligence to this problem, the prediction module is responsible for accurately forecasting the production orders. A prediction module based on several models of the same type (e. g., neural networks) can be used,[16] with the final prediction being a weighted average of the individual models. As with the other business problems we discussed, each day brings a new set of data that needs to be taken into account for making future predictions. Again, the adaptability module is responsible for adapting the parameters (weights) of the prediction module.

While the prediction module works on forecasting new orders, the optimization module is responsible for optimizing the interplay between the furnaces and production lines. After all, this interplay represents the core scheduling issue. The primary objective is to optimize – in terms of the utilization of furnaces and throughput of production lines – the distribution of production orders over some period of time. Because the furnaces and production lines work together in the production process, the maximization of furnace utilization and production-line throughput have to be considered jointly. Secondary objectives might include the maximization of yield, or the minimization of labor costs, electricity costs, storage costs, the fettling process, etc.

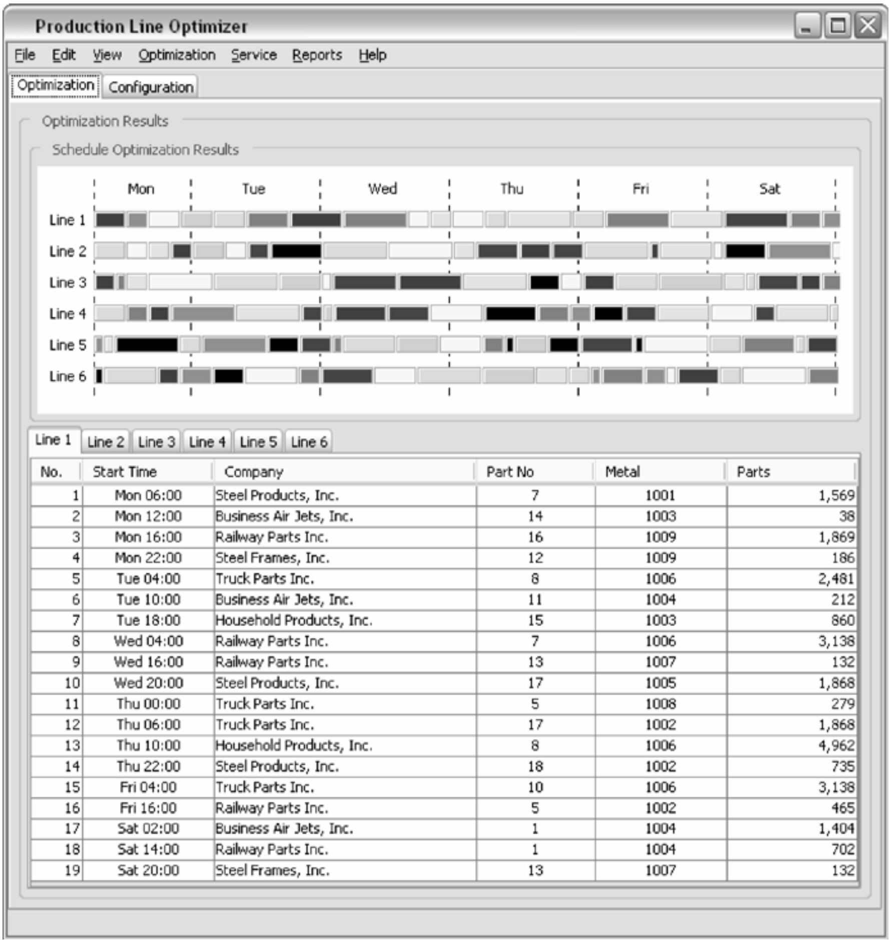
The approach for handling constraints can be based on decoders, which separate between objectives and constraints. Using this approach, the optimization module can use the constraints to "guide" the optimization process toward feasible solutions of high quality. This constraint-handling approach also allows for easy and "at will" modification of the constraints to fit the current situation. Furthermore, to

[15] Although the production deadline is often viewed as the objective, this is in fact a constraint that should not be violated.

[16] In other words, a bagging technique is used (as discussed in Sect. 10.1).

generate valid scheduling solutions right from the start of the run, the optimization module can use a combination of evolutionary algorithms and simulated annealing (plus a decoder responsible for generating near-feasible solutions). Although the scheduling solution will increase in quality as the optimization run progresses, a business manager can stop the run at any given moment and use the best available solution rather than wait to the end. This allows for very flexible usage and provides a business manager with ultimate control over the optimization process.

The sample screen below shows a scheduling solution for a particular week:


Production Line Optimizer
File Edit View Optimization Service Reports Help
Optimization Configuration
Optimization Results
Schedule Optimization Results
Mon
Tue
Wed
Thu
Fri
Sat
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 1 Line 2 Line 3 Line 4 Line 5 Line 6

No.	Start Time	Company	Part No	Metal	Parts
1	Mon 06:00	Steel Products, Inc.	7	1001	1,569
2	Mon 12:00	Business Air Jets, Inc.	14	1003	38
3	Mon 16:00	Railway Parts Inc.	16	1009	1,869
4	Mon 22:00	Steel Frames, Inc.	12	1009	186
5	Tue 04:00	Truck Parts Inc.	8	1006	2,481
6	Tue 10:00	Business Air Jets, Inc.	11	1004	212
7	Tue 18:00	Household Products, Inc.	15	1003	860
8	Wed 04:00	Railway Parts Inc.	7	1006	3,138
9	Wed 16:00	Railway Parts Inc.	13	1007	132
10	Wed 20:00	Steel Products, Inc.	17	1005	1,868
11	Thu 00:00	Truck Parts Inc.	5	1008	279
12	Thu 06:00	Truck Parts Inc.	17	1002	1,868
13	Thu 10:00	Household Products, Inc.	8	1006	4,962
14	Thu 22:00	Steel Products, Inc.	18	1002	735
15	Fri 04:00	Truck Parts Inc.	10	1006	3,138
16	Fri 16:00	Railway Parts Inc.	5	1002	465
17	Sat 02:00	Business Air Jets, Inc.	1	1004	1,404
18	Sat 14:00	Railway Parts Inc.	1	1004	702
19	Sat 20:00	Steel Frames, Inc.	13	1007	132

Each bar represents a production order, and in the lower part of the screen we can view the details of each order. Separate screens illustrate the different furnaces and iron base-grades. Obviously, each base-grade is aligned with the production orders that require that base-grade (to produce a particular product).

Although there are many advantages of this graphical user interface, the most important is that it allows a business manager to carry out numerous “what-if” scenarios, including:

- Examining the effect of moving a production order forward or backward in time, or from one production line to another.
- Splitting a production order into smaller pieces and examining the effect of such a modification.
- Examining the effect of constraining certain orders so they cannot run in parallel.
- Visually examining what effect changes to the production calendar, furnaces, and production lines would have.

The final output of the Adaptive Business Intelligence system is an ordered list of production orders to be processed each day on each production line. The schedule may contain both confirmed and forecasted orders, as determined by the prediction module, scheduled in such a way that they maximize the utilization and throughput of the production lines. Some additional reports can be used to display various utilization ratios, throughputs, and other key performance indicators in graphical and/or numerical form. Such reporting would allow a business manager to evaluate the performance of the production schedules.

12.3 Investment Strategies

Most fund managers who oversee large pools of money (such as pension funds, mutual funds, hedge funds, etc.) use a stock ranking system based on fundamental analysis to decide what to buy. They execute their orders over a two- to three-day period, depending on the size of the order and the liquidity of the stock. Although the overall objective is to maximize profit, there are usually many business rules that must be taken into account when making buy/sell decisions. In addition to various risk factors that have to be considered, there are also maximum and minimum limits to the amount that can be invested in any one sector or country, the turnover for any particular period, etc.

In this setting, an Adaptive Business Intelligence system could provide a complementary technical trading rule component to the fundamental analysis, thereby helping to determine whether there are technical “trade signals” for buying or selling specific stocks. As with the other Adaptive Business Intelligence systems discussed in this chapter, the general goal is to develop a dynamic and adaptive system that can consider changing market conditions (rather than just a static environment). Thus, in this particular problem, the system must continuously adapt the short- to mid-term trading rules to changes in the market.

Using past stock market data, a prediction module based on fuzzy trading rules can be used. For each linguistic variable (e. g., price change, volume change, single moving average buy signal, single moving average sell signal, plus many additional trend, volume, and momentum indicators) we can define seven fuzzy logic membership functions: *Extremely Low*, *Very Low*, *Low*, *Medium*, *High*, *Very High*, and *Extremely High*. The prediction module automatically calculates these membership functions, which are adaptive in the sense that each membership function covers the same number of input data (thus the shape of each membership function changes slightly at the arrival of new data).

The fuzzy rule base consists of a variable number of trading rules, which are optimized by the optimization module. For example, one of the initial solutions (i. e., a rule base, which is a member of the initial population of solutions) might be:

- **If** Single Moving Average Buy Signal **is** Extremely Low, **then** rating = 0.7.
- **If** Volume Change **is** Medium **and** Single Moving Average Buy Signal **is** Extremely Low, **then** rating = 0.9.
- **If** Price Change **is** High **and** Volume Change **is** Very High, **then** rating = 0.4.
- **If** Price Change **is** High **and** Single Moving Average Buy Signal **is** Very Low, **then** rating = 0.6.
- **If** Price Change **is** Extremely High **and** Volume Change **is** Very High, **then** rating = 0.3.
- **If** *Price Change* **is** *Medium* **and** *Volume Change* **is** *Extremely High* **and** *Single Moving Average Buy Signal* **is** *Medium* **and** *Single Moving Average Buy Signal* **is** *Medium,* **then** rating = 0.5.

A population of such rule bases[17] undergoes an evolutionary process (as discussed in Chap. 6). During each generation, all the solutions (rule bases) are evaluated on the basis of some historical time window (e. g., the last 3 years). In the search for optimal trading rules, various evaluation functions are used (such as excess return or risk-adjusted return), whereas transaction costs and market impact serve as constraints. Once each rule base is evaluated, the standard steps of the evolutionary algorithm are executed (e. g., crossover, mutation). After some number of generations, the best rule base in the population might be:

- **If** Volume Change **is** Medium **and** Single Moving Average Buy Signal **is** Extremely Low, **then** rating = 0.9.
- **If** Single Moving Average Buy Signal **is** Extremely Low, **then** rating = 0.6.
- **If** Price Change **is** High, **then** rating = 0.2.
- **If** Price Change **is** High **and** Volume Change **is** High, **then** rating = 0.4.
- **If** Single Moving Average Buy Signal **is** Low **and** Single Moving Average Buy Signal **is** Medium, **then** rating = 0.3.
- **If** *Volume Change* **is** *High,* **then** rating = 0.3.

[17] Note that each rule base may have a different number of rules, and each individual rule may have a different number of conditions.

The resulting rule base is then used by the prediction module to create a *technical* ranking, which is the final output of the system. Each end of the technical ranking list represents the stocks with the strongest buy and sell signals, and this list can be used in conjunction with fundamental analysis to make buy/sell decisions. To generate a ranking from the rule base, the stocks are rated according to the rules of the best rule base. Note that each individual rule assigns a rating for all the stocks that satisfy the "if" part of the rule. For example, the rule

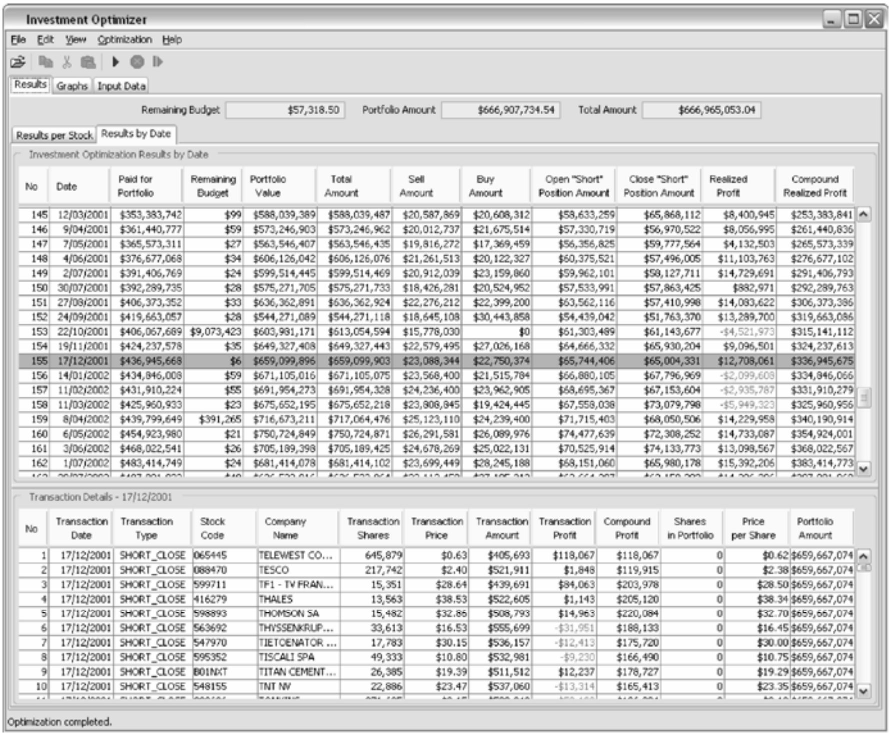
If *Price Change* **is** *High*, **then** rating = 0.2.

would assign a rating of 0.2 to all stocks for which *Price Change* is *High*. The defuzzifier (see Sect. 7.4) then assigns the final rating to each stock by creating a weighted average of all the recommended ratings for each stock (the weights correspond to the "degree" to which the "if" part of the rule is satisfied), divided by the total of all weights. Finally, these rating numbers are used to create a technical ranking of all the stocks.

For this particular problem, the optimization module is responsible for evolving the "optimal" set of technical trading rules (i. e., rule base). One way of speeding up the optimization process is to initialize the population with some classic trading rules, which might be based on time series forecasting, price/volume momentum, etc. Although the actual trading rules that are evolved would be *very different*, these initial rules allow the optimization module to begin with "something" (i. e., a set of previously tested rules) rather than "nothing" (i. e., completely in the dark).

Lastly, the adaptability module is responsible for taking into account new information and continuously re-evaluating the rules. The time-window of information is actively managed so that outdated information is discarded and critical information is retained (new regimes are registered, yet repeated past regimes are also recognized). The adaptability module might also be extended with the ability to generate and test new market indicators to see what kind of impact they would have on the existing trading rules. These indicators can be introduced into the rule bases during the optimization process.

The outcome of the combined fundamental analysis and the technical buy/sell recommendations made by the Adaptive Business Intelligence system are reported in a series of profit/loss screens. For example, the sample screen below shows the profit/loss result for a particular time period:



Remaining Budget $57,318.50 Portfolio Amount $666,907,734.54 Total Amount $666,965,053.04

Investment Optimization Results by Date

No	Date	Paid for Portfolio	Remaining Budget	Portfolio Value	Total Amount	Sell Amount	Buy Amount	Open "Short" Position Amount	Close "Short" Position Amount	Realized Profit	Compound Realized Profit
145	12/03/2001	$353,383,742	$99	$588,039,389	$588,039,487	$20,587,869	$20,608,312	$58,633,259	$65,868,112	$8,400,945	$253,383,841
146	9/04/2001	$361,440,777	$59	$573,246,903	$573,246,962	$20,012,737	$21,675,514	$57,330,719	$56,970,522	$8,056,995	$261,440,836
147	7/05/2001	$365,573,311	$27	$563,546,407	$563,546,435	$19,816,272	$17,369,459	$56,356,825	$59,777,564	$4,132,503	$265,573,339
148	4/06/2001	$376,677,068	$34	$606,126,042	$606,126,076	$21,261,513	$20,122,327	$60,375,521	$57,496,005	$11,103,763	$276,677,102
149	2/07/2001	$391,406,769	$24	$599,514,445	$599,514,469	$20,912,039	$23,159,860	$59,962,101	$58,127,711	$14,729,691	$291,406,793
150	30/07/2001	$392,289,735	$28	$575,271,705	$575,271,733	$18,426,281	$20,524,952	$57,533,991	$57,863,425	$882,971	$292,289,763
151	27/08/2001	$406,373,352	$33	$636,362,891	$636,362,924	$22,276,212	$22,399,200	$63,562,116	$57,410,998	$14,083,622	$306,373,386
152	24/09/2001	$419,663,057	$28	$544,271,089	$544,271,118	$18,645,108	$30,443,858	$54,439,042	$51,763,370	$13,289,700	$319,663,086
153	22/10/2001	$406,067,689	$9,073,423	$603,981,171	$613,054,594	$15,778,030	$0	$61,303,489	$61,143,677	-$4,521,973	$315,141,112
154	19/11/2001	$424,237,578	$35	$649,327,408	$649,327,443	$22,579,495	$27,026,168	$64,666,332	$65,930,204	$9,096,501	$324,237,613
155	17/12/2001	$436,945,668	$6	$659,099,896	$659,099,903	$23,088,344	$22,750,374	$65,744,406	$65,004,331	$12,708,061	$336,945,675
156	14/01/2002	$434,846,008	$59	$671,105,016	$671,105,075	$23,568,400	$21,515,784	$66,880,105	$67,796,969	-$2,099,608	$334,846,066
157	11/02/2002	$431,910,224	$55	$691,954,273	$691,954,328	$24,236,400	$23,962,905	$68,695,367	$67,153,604	-$2,935,787	$331,910,279
158	11/03/2002	$425,960,933	$23	$675,652,195	$675,652,218	$23,808,845	$19,424,445	$67,558,038	$73,079,798	-$5,949,323	$325,960,956
159	8/04/2002	$439,799,649	$391,265	$716,673,211	$717,064,476	$25,123,110	$24,239,400	$71,715,403	$68,050,506	$14,229,958	$340,190,914
160	6/05/2002	$454,923,980	$21	$750,724,849	$750,724,871	$26,291,581	$26,089,976	$74,477,639	$72,308,252	$14,733,087	$354,924,001
161	3/06/2002	$468,022,541	$26	$705,189,398	$705,189,425	$24,678,269	$25,022,131	$70,525,914	$74,133,773	$13,098,567	$368,022,567
162	1/07/2002	$483,414,749	$24	$681,414,078	$681,414,102	$23,699,449	$28,245,188	$68,151,060	$65,980,178	$15,392,206	$383,414,773

Transaction Details - 17/12/2001

No	Transaction Date	Transaction Type	Stock Code	Company Name	Transaction Shares	Transaction Price	Transaction Amount	Transaction Profit	Compound Profit	Shares in Portfolio	Price per Share	Portfolio Amount
1	17/12/2001	SHORT_CLOSE	065445	TELEWEST CO...	645,879	$0.63	$405,693	$118,067	$118,067	0	$0.62	$659,667,074
2	17/12/2001	SHORT_CLOSE	088470	TESCO	217,742	$2.40	$521,911	$1,848	$119,915	0	$2.38	$659,667,074
3	17/12/2001	SHORT_CLOSE	599711	TF1 - TV FRAN...	15,351	$28.64	$439,691	$84,063	$203,978	0	$28.50	$659,667,074
4	17/12/2001	SHORT_CLOSE	416279	THALES	13,563	$38.53	$522,605	$1,143	$205,120	0	$38.34	$659,667,074
5	17/12/2001	SHORT_CLOSE	598893	THOMSON SA	15,482	$32.86	$508,793	$14,963	$220,084	0	$32.70	$659,667,074
6	17/12/2001	SHORT_CLOSE	563692	THYSSENKRUP...	33,613	$16.53	$555,699	-$31,951	$188,133	0	$16.45	$659,667,074
7	17/12/2001	SHORT_CLOSE	547970	TIETOENATOR ...	17,783	$30.15	$536,157	-$12,413	$175,720	0	$30.00	$659,667,074
8	17/12/2001	SHORT_CLOSE	595352	TISCALI SPA	49,333	$10.80	$532,981	-$9,230	$166,490	0	$10.75	$659,667,074
9	17/12/2001	SHORT_CLOSE	B01NXT	TITAN CEMENT...	26,385	$19.39	$511,512	$12,237	$178,727	0	$19.29	$659,667,074
10	17/12/2001	SHORT_CLOSE	548155	TNT NV	22,886	$23.47	$537,060	-$13,314	$165,413	0	$23.35	$659,667,074

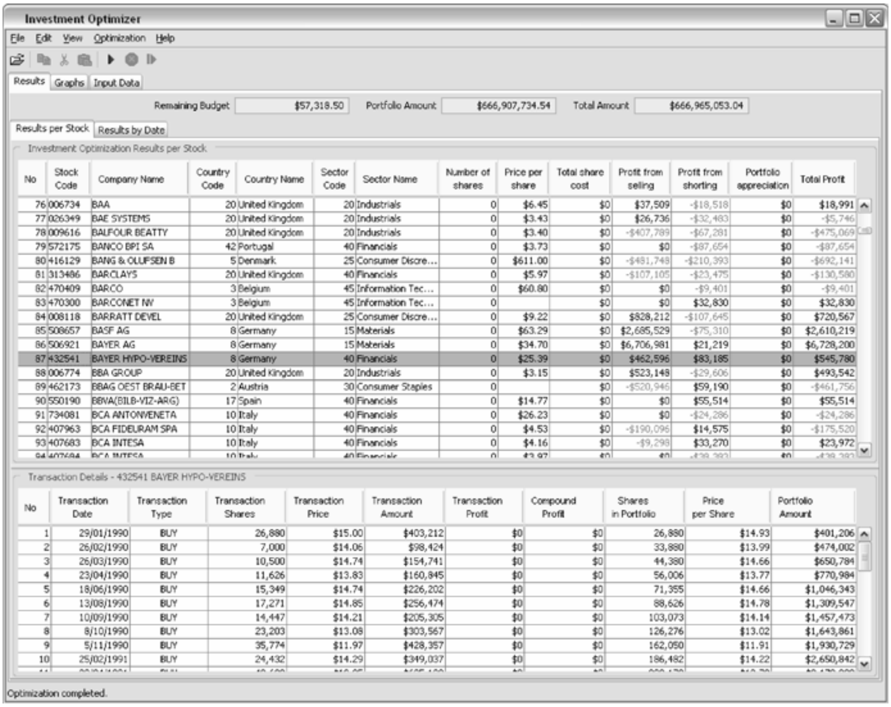
Optimization completed.

In this report, the columns of the table display the:

- *Remaining budget:* The cash not used for buying stocks.
- *Total amount:* The total of the Portfolio Value and Remaining Budget.
- *Sell/buy:* The amount sold or bought on a particular date.
- *Open "short" position:* The amount used to open a short position.
- *Close "short" position:* The profit or loss realized on the closing of a short position.
- *Realized profit:* The total net profit realized on a particular day (including transaction costs).
- *Compounded realized profit:* The cumulative net profit since the inception of the fund.

The *Transaction Details* table on the lower part of the screen shows what was sold/bought/shorted on the selected date (highlighted on the upper part of the screen).

Several additional reports, such as the following sample screen, are used to summarize the profit and loss result for particular stocks:



Investment Optimizer
File Edit View Optimization Help
Results Graphs Input Data
Results per Stock Results by Date

Remaining Budget $57,318.50 Portfolio Amount $666,907,734.54 Total Amount $666,965,053.04

Investment Optimization Results per Stock

No	Stock Code	Company Name	Country Code	Country Name	Sector Code	Sector Name	Number of shares	Price per share	Total share cost	Profit from selling	Profit from shorting	Portfolio appreciation	Total Profit
76	006734	BAA	20	United Kingdom	20	Industrials	0	$6.45	$0	$37,509	-$18,518	$0	$18,991
77	026349	BAE SYSTEMS	20	United Kingdom	20	Industrials	0	$3.43	$0	$26,736	-$32,483	$0	-$5,746
78	009616	BALFOUR BEATTY	20	United Kingdom	20	Industrials	0	$3.40	$0	-$407,789	-$67,281	$0	-$475,069
79	572175	BANCO BPI SA	42	Portugal	40	Financials	0	$3.73	$0	$0	-$87,654	$0	-$87,654
80	416129	BANG & OLUFSEN B	5	Denmark	25	Consumer Discre…	0	$611.00	$0	-$481,748	-$210,393	$0	-$692,141
81	313486	BARCLAYS	20	United Kingdom	40	Financials	0	$5.97	$0	-$107,105	-$23,475	$0	-$130,580
82	470409	BARCO	3	Belgium	45	Information Tec…	0	$60.80	$0	$0	-$9,401	$0	-$9,401
83	470300	BARCONET NV	3	Belgium	45	Information Tec…	0		$0	$0	$32,830	$0	$32,830
84	008118	BARRATT DEVEL	20	United Kingdom	25	Consumer Discre…	0	$9.22	$0	$828,212	-$107,645	$0	$720,567
85	508657	BASF AG	8	Germany	15	Materials	0	$63.29	$0	$2,685,529	-$75,310	$0	$2,610,219
86	506921	BAYER AG	8	Germany	15	Materials	0	$34.70	$0	$6,706,981	$21,219	$0	$6,728,200
87	432541	BAYER HYPO-VEREINS	8	Germany	40	Financials	0	$25.39	$0	$462,596	$83,185	$0	$545,780
88	006774	BBA GROUP	20	United Kingdom	20	Industrials	0	$3.15	$0	$523,148	-$29,606	$0	$493,542
89	462173	BBAG OEST BRAU-BET	2	Austria	30	Consumer Staples	0		$0	-$520,946	$59,190	$0	-$461,756
90	550190	BBVA(BILB-VIZ-ARG)	17	Spain	40	Financials	0	$14.77	$0	$0	$55,514	$0	$55,514
91	734081	BCA ANTONVENETA	10	Italy	40	Financials	0	$26.23	$0	$0	-$24,286	$0	-$24,286
92	407963	BCA FIDEURAM SPA	10	Italy	40	Financials	0	$4.53	$0	-$190,096	$14,575	$0	-$175,520
93	407683	BCA INTESA	10	Italy	40	Financials	0	$4.16	$0	-$9,298	$33,270	$0	$23,972

Transaction Details - 432541 BAYER HYPO-VEREINS

No	Transaction Date	Transaction Type	Transaction Shares	Transaction Price	Transaction Amount	Transaction Profit	Compound Profit	Shares in Portfolio	Price per Share	Portfolio Amount
1	29/01/1990	BUY	26,880	$15.00	$403,212	$0	$0	26,880	$14.93	$401,206
2	26/02/1990	BUY	7,000	$14.06	$98,424	$0	$0	33,880	$13.99	$474,002
3	26/03/1990	BUY	10,500	$14.74	$154,741	$0	$0	44,380	$14.66	$650,784
4	23/04/1990	BUY	11,626	$13.83	$160,845	$0	$0	56,006	$13.77	$770,984
5	18/06/1990	BUY	15,349	$14.74	$226,202	$0	$0	71,355	$14.66	$1,046,343
6	13/08/1990	BUY	17,271	$14.85	$256,474	$0	$0	88,626	$14.78	$1,309,547
7	10/09/1990	BUY	14,447	$14.21	$205,305	$0	$0	103,073	$14.14	$1,457,473
8	8/10/1990	BUY	23,203	$13.08	$303,567	$0	$0	126,276	$13.02	$1,643,861
9	5/11/1990	BUY	35,774	$11.97	$428,357	$0	$0	162,050	$11.91	$1,930,729
10	25/02/1991	BUY	24,432	$14.29	$349,037	$0	$0	186,482	$14.22	$2,650,842

Optimization completed.

In this screen, the upper part of the table contains information about each stock, such as the country, sector, number of shares owned, price paid per share, amount bought or sold, and total profit. The lower part of the table shows the detailed transaction history for the highlighted stock in the upper part of the table.

In summary, it is important to highlight that such an Adaptive Business Intelligence system is not based on a *fixed* set of technical trading rules. Rather, the adaptability module updates these rules as new information becomes available. As mentioned earlier, the output of the Adaptive Business Intelligence system is a technical ranking of stocks, with each end of the list representing the stocks with the strongest buy and sell signals. When used in conjunction with fundamental analysis, such an investment optimization system could provide fund managers with an edge in performance.

12.4 Emergency Response Services

The emergency response service in any major city has a number of ambulance and police cars that respond to calls. The problem faced by these organizations is “where should these cars be located so that the response time is minimized?” The ambulances usually stay and wait for a call, while police cars patrol some predefined zone.

An Adaptive Business Intelligence system can be applied to this problem, as prediction, optimization, and adaptability are all required components.

The prediction module for this problem is based on three sets of historical data:

1. The frequencies and types of emergencies and their locations.
2. The calendar (weekdays, weekends, national and local holidays, etc.).
3. Weather conditions.

By analyzing these data sets for a specific city, we might discover that more emergencies occur on weekdays than weekends, or that more cases occur in residential areas during the weekends. Also, by taking into account weather conditions, we might also discover that more traffic accidents occur during periods of rain, snow, and ice. This knowledge is used to develop a prediction module capable of identifying the places (and frequencies) where emergencies are most likely to occur.

To make the prediction task easier, we can divide the city into a number of zones, with the size and the shape of each zone depending on the geographical topology and some constraints. One constraint, for instance, might be that the travel time between any two internal points of a zone cannot exceed 15 minutes. Thus, the zones would be smaller in urban areas and larger in rural areas (especially in the presence of highways intersecting the zones). It is also possible to have different zones for different days of the week, time of the day (e. g., during peak hours), and weather conditions.

For this type of problem, the prediction module can be based on two models:

1. The first model is responsible for predicting the total number of emergencies during any block of time.
2. The second model is responsible for predicting the distribution of these emergencies across the different zones.

For example, a prediction model based on artificial neural networks could estimate the total number of emergencies (together with their type) for Friday morning between 6 am and noon on a rainy day, and a second model based on expert rules could predict where these emergencies will occur (i. e., in which zone). By combining the weighted "recommendations" of the individual rules in the second model (which are normalized to match the average number of emergencies predicted by the neural network model), the rule base would act as a voting system. The final output of the prediction module is the predicted number of emergencies in each zone during a six-hour block.

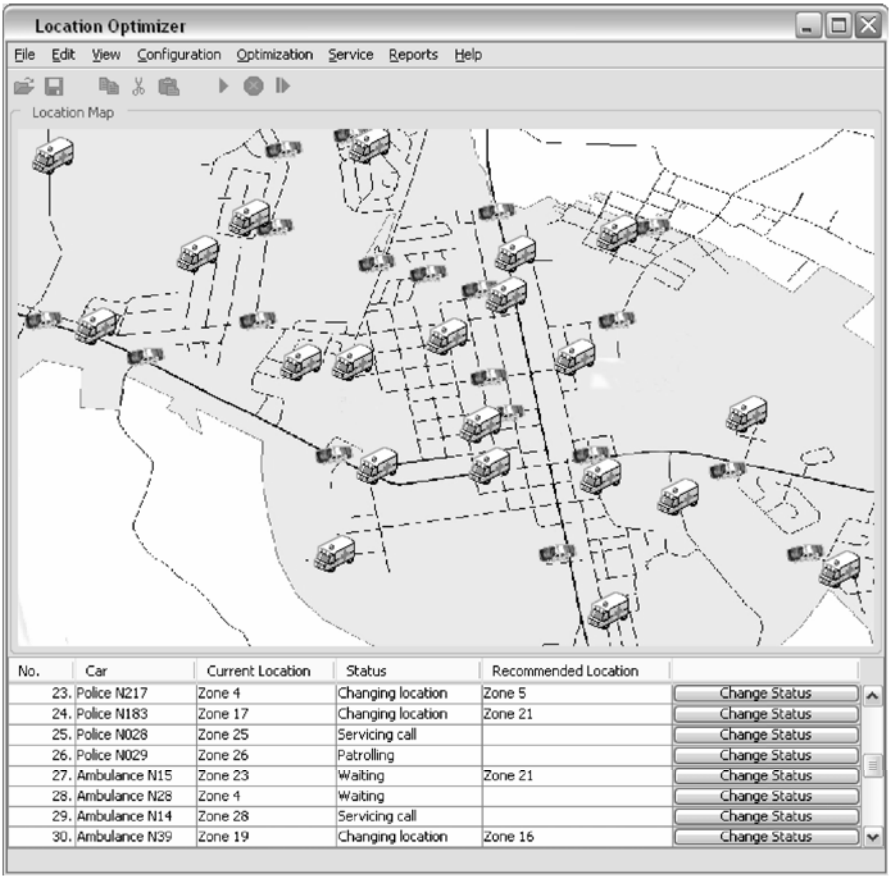
Note, however, that whatever the output of the prediction module, we must usually satisfy a "maximum response time" constraint: For example, the distribution of emergency response cars should be such that any accident can be reached within 15 minutes. Also note that each day brings a new set of cases, and some rules may gain or lose accuracy over time. Consequently, the adaptability module must accommodate the new data by adjusting the weights of the individual rules.

The optimization module for this problem is responsible for recommending the best locations for the emergency response cars. This recommendation is made on the basis of the number of available cars, their current positions, and the output of the prediction module (i. e., the number and distribution of the predicted emergencies). We could use a vector of integers for the solution representation, with each integer representing a zone for a specific car (i. e., the first number represents the zone for the first car, the second number represents the zone for the second car, and so forth). As an example, the vector:

32	14	9	…	91	57

represents a solution where the first ambulance is assigned to zone 32, the second ambulance is assigned to zone 14, and so on, with the last two ambulances assigned to zones 91 and 57, respectively. A similar vector may represent the assignment of police cars. By using evolutionary algorithms (discussed in Chap. 6), the optimization module can evolve solutions that find better and better locations for the emergency cars. To evaluate a solution vector, the evaluation function must take into account the average response time, maximum response time, etc.

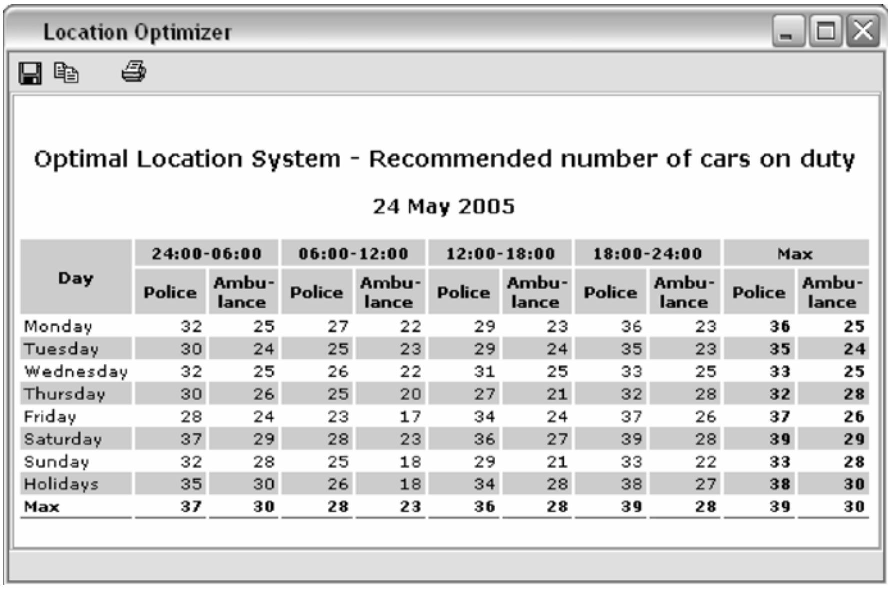
The sample screen below shows the current location of several emergency cars. The table in the lower part of the screen contains the status and recommended zone of each car:



No.	Car	Current Location	Status	Recommended Location	
23.	Police N217	Zone 4	Changing location	Zone 5	Change Status
24.	Police N183	Zone 17	Changing location	Zone 21	Change Status
25.	Police N028	Zone 25	Servicing call		Change Status
26.	Police N029	Zone 26	Patrolling		Change Status
27.	Ambulance N15	Zone 23	Waiting	Zone 21	Change Status
28.	Ambulance N28	Zone 4	Waiting		Change Status
29.	Ambulance N14	Zone 28	Servicing call		Change Status
30.	Ambulance N39	Zone 19	Changing location	Zone 16	Change Status

In this screen, police car N217 is changing locations from zone 4 to zone 5. Once this car arrives at zone 5, its status will change to “Patrolling.” Police car N028 is servicing a call. Once the call is serviced, the system will recommend the best zone for patrolling.

Another important task for the system is to recommend the minimum number of ambulances and police cars needed to meet some service level criteria. The user specifies the desired response time for different types of emergencies, and the system finds the minimum number of cars that should be kept on duty to achieve this criteria. In the sample screen below, the system recommends the minimum number of cars for a specific day:



Optimal Location System - Recommended number of cars on duty

24 May 2005

Day	24:00-06:00		06:00-12:00		12:00-18:00		18:00-24:00		Max	
	Police	Ambu-lance	Police	Ambu-lance	Police	Ambu-lance	Police	Ambu-lance	Police	Ambu-lance
Monday	32	25	27	22	29	23	36	23	**36**	**25**
Tuesday	30	24	25	23	29	24	35	23	**35**	**24**
Wednesday	32	25	26	22	31	25	33	25	**33**	**25**
Thursday	30	26	25	20	27	21	32	28	**32**	**28**
Friday	28	24	23	17	34	24	37	26	**37**	**26**
Saturday	37	29	28	23	36	27	39	28	**39**	**29**
Sunday	32	28	25	18	29	21	33	22	**33**	**28**
Holidays	35	30	26	18	34	28	38	27	**38**	**30**
Max	**37**	**30**	**28**	**23**	**36**	**28**	**39**	**28**	**39**	**30**

In this report, the maximum number of police cars required for any six-hour period is 39, while the maximum number of ambulances is 30 (the 6 pm–midnight time block requires 39 police cars, and the midnight–6 am time block requires 30 ambulances).

Note that this location optimization problem is set in a dynamic environment, where different types of changes occur at different rates. Some of the more predictable and "regular" changes might occur on an hourly or weekly basis (e. g., changes in traffic patterns due to rush hour or holidays), while the less predictable changes might include adverse weather conditions or roadwork. Some changes might also be the result of the current operation: For example, a few cars servicing a particular incident may leave some other zones of the city exposed. Furthermore, as a city grows – with some areas growing faster than others – the distribution and frequency of emergencies will inevitably change.

By recommending the best locations for emergency response cars and keeping their number to a minimum, an Adaptive Business Intelligence system may provide significant savings over the current operation. Dynamically reassigning emergency cars in time-changing environments is an additional benefit.

12.5 Credit Card Fraud

The purpose of fraud detection systems is to evaluate and classify financial transactions. In the case of credit and debit cards, a system has to check thousands of transactions per second and flag those that are potentially fraudulent. Note that a set of tested transactions can be divided into four groups:

1. *True-positive*s. Transactions that were fraudulent and which the system correctly classified as fraudulent.
2. *False-positive*s. Transactions that were legitimate, but which the system incorrectly classified as fraudulent.
3. *True-negative*s. Transactions that were legitimate and which the system correctly classified as legitimate.
4. *False-negative*s. Transactions that were fraudulent, but which the system incorrectly classified as legitimate.

In other words, a transaction that is classified as fraudulent is called *positive,* while a transaction that is classified as legitimate is *negative.* If the classification is correct then the transaction will be *true* (i. e., correctly classified), and if the classification is incorrect then the transaction will be *false* (i. e., incorrectly classified). The table below summarizes these classification results:

	Tested Transactions	
	Fraudulent (Positive)	**Legitimate (Negative)**
Fraudulent	True-positive	False-negative
Legitimate	False-positive	True-negative

The efficiency of a fraud detection system can be measured by the number of correct classifications. Clearly, a bank would like to minimize the number of false-positives and false-negatives (incorrect classifications); however, these two measures very often work against each other. For instance, we can reduce the number of false-negatives by making a system more "suspicious" (i. e., thereby flagging more transactions as fraudulent); but flagging a larger number of transactions would increase the number of false-positives (as many of these transactions would be legitimate). What is more, any change in these two measures might have significant consequences for a bank.

To illustrate this point, let us consider the following example: A bank uses a fraud detection system that flags "suspicious" transactions, and, later, a subset of these transactions is classified as fraudulent. Now the bank can drastically reduce fraud by flagging *all* suspicious transactions (i. e., reducing the number of false-negatives), but this is not feasible for two reasons:

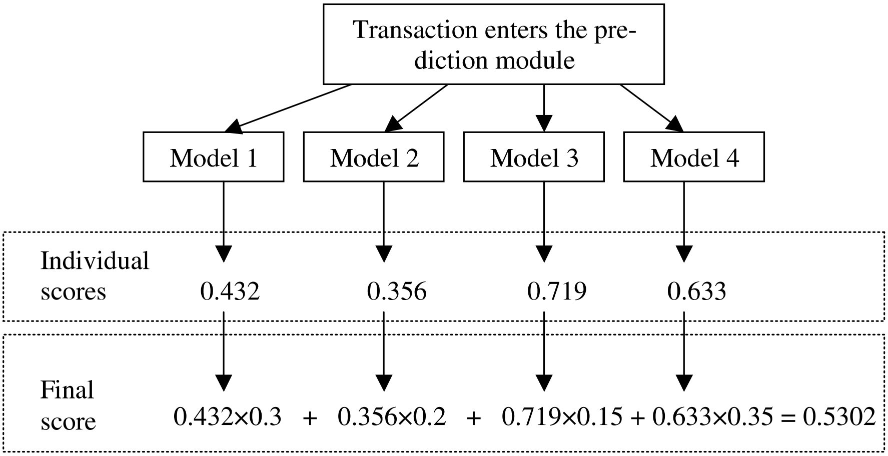
1. A battalion of fraud prevention officers would be needed to review all the flagged transactions.
2. The bank would annoy its customers by blocking many legitimate transactions (i. e., increasing the number of false-positives, as not all suspicious transactions turn out to be fraudulent).

Hence, the "suspicion threshold" cannot be set too high. However, although there will be fewer transactions to review and less upset customers when the threshold is lowered, this might result in an increased number of false-negatives (which will increase the financial losses of the bank). This problem is compounded by the fact that many new fraud patterns emerge each year, and a bank might not recognize them until the fraud has already occurred. This problem of minimizing the number of credit and debit card transactions to review, while maximizing the number of fraudulent transactions detected (in an ever-changing environment), is ideally suited for an Adaptive Business Intelligence system.

Clearly, the heart of the system is a prediction module that assigns a "suspicion score" to each transaction. For this complex classification task, it might be prudent to build and train several prediction models and then combine them together into one module (as discussed in Sect. 10.1). Let us take a closer look at the possible design of such a prediction module.

In order to make a decision, the prediction module can use several models that are based on different prediction methods. Each model assigns a suspicion score in the range of 0 to 1: the higher the score, the greater the probability that the transaction is fraudulent. Once each of the models produces a score for the transaction, the prediction module then combines the scores. Each model has a weight that represents the importance of its prediction, and the sum of all the weights equals 1; therefore, the combined suspicion score is also in the range of 0 to 1. The final decision of "fraudulent" versus "legitimate" is based on whether the weighted score is higher than the "suspicion threshold" parameter in the system.

The diagram below provides an example of the prediction process, which is based on four models and a voting system (averaging of scores):

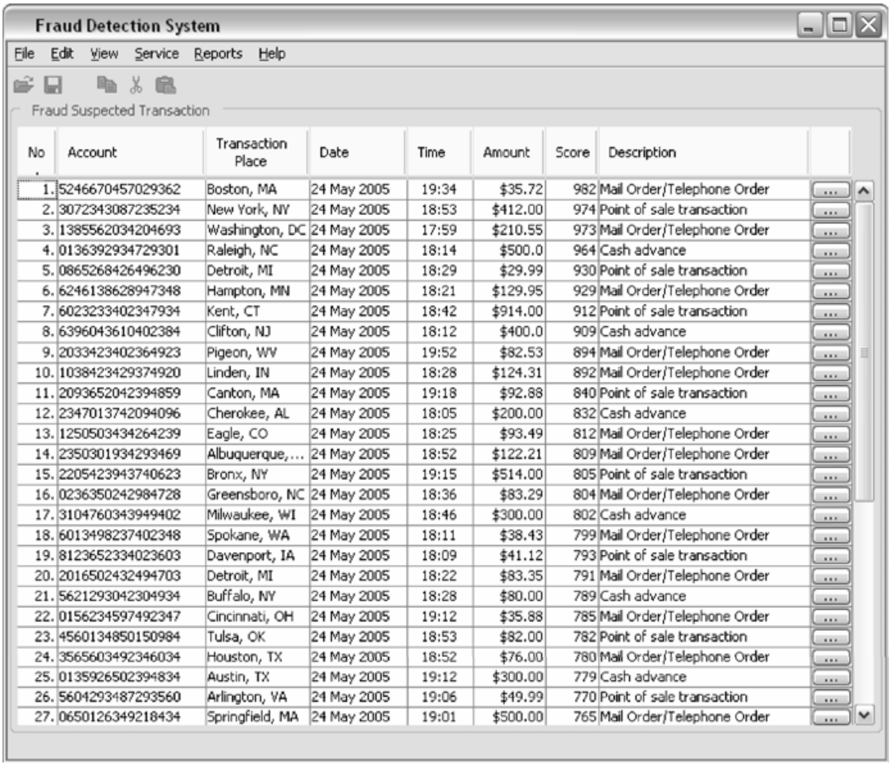


In this example, the four models have weights equal to 0.3, 0.2, 0.15, and 0.35, respectively. The individual scores are combined to come up with the final suspicion score, which is a weighted average.

Now, the prediction module may have different suspicion thresholds that define how a transaction should be processed. For example, the thresholds may be:

- Not greater than 0.6 – grant authorization.
- Greater than 0.6, but not greater than 0.9 – grant authorization, but flag the transaction for later review by a fraud prevention officer.
- Greater than 0.9 – deny authorization and call customer.

The sample screen below shows a list of *all* suspicious transactions (i. e., transactions with the final score greater than 0.6)[18] that have been assigned to a fraud prevention officer for review. The suspicious transactions are sorted by decreasing final score:


Fraud Detection System
File Edit View Service Reports Help
Fraud Suspected Transaction

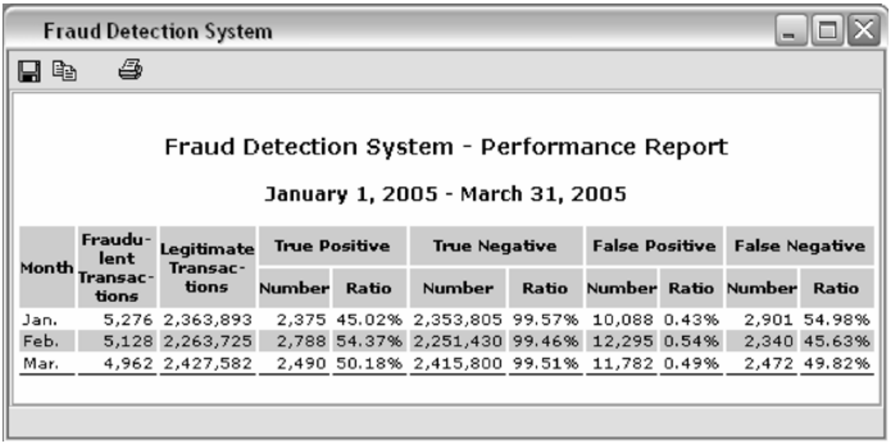
No.	Account	Transaction Place	Date	Time	Amount	Score	Description	
1.	5246670457029362	Boston, MA	24 May 2005	19:34	$35.72	982	Mail Order/Telephone Order	...
2.	3072343087235234	New York, NY	24 May 2005	18:53	$412.00	974	Point of sale transaction	...
3.	1385562034204693	Washington, DC	24 May 2005	17:59	$210.55	973	Mail Order/Telephone Order	...
4.	0136392934729301	Raleigh, NC	24 May 2005	18:14	$500.0	964	Cash advance	...
5.	0865268426496230	Detroit, MI	24 May 2005	18:29	$29.99	930	Point of sale transaction	...
6.	6246138628947348	Hampton, MN	24 May 2005	18:21	$129.95	929	Mail Order/Telephone Order	...
7.	6023233402347934	Kent, CT	24 May 2005	18:42	$914.00	912	Point of sale transaction	...
8.	6396043610402384	Clifton, NJ	24 May 2005	18:12	$400.0	909	Cash advance	...
9.	2033423402364923	Pigeon, WV	24 May 2005	19:52	$82.53	894	Mail Order/Telephone Order	...
10.	1038423429374920	Linden, IN	24 May 2005	18:28	$124.31	892	Mail Order/Telephone Order	...
11.	2093652042394859	Canton, MA	24 May 2005	19:18	$92.88	840	Point of sale transaction	...
12.	2347013742094096	Cherokee, AL	24 May 2005	18:05	$200.00	832	Cash advance	...
13.	1250503434264239	Eagle, CO	24 May 2005	18:25	$93.49	812	Mail Order/Telephone Order	...
14.	2350301934293469	Albuquerque,…	24 May 2005	18:52	$122.21	809	Mail Order/Telephone Order	...
15.	2205423943740623	Bronx, NY	24 May 2005	19:15	$514.00	805	Point of sale transaction	...
16.	0236350242984728	Greensboro, NC	24 May 2005	18:36	$83.29	804	Mail Order/Telephone Order	...
17.	3104760343949402	Milwaukee, WI	24 May 2005	18:46	$300.00	802	Cash advance	...
18.	6013498237402348	Spokane, WA	24 May 2005	18:11	$38.43	799	Mail Order/Telephone Order	...
19.	8123652334023603	Davenport, IA	24 May 2005	18:09	$41.12	793	Point of sale transaction	...
20.	2016502432494703	Detroit, MI	24 May 2005	18:22	$83.35	791	Mail Order/Telephone Order	...
21.	5621293042304934	Buffalo, NY	24 May 2005	18:28	$80.00	789	Cash advance	...
22.	0156234597492347	Cincinnati, OH	24 May 2005	19:12	$35.88	785	Mail Order/Telephone Order	...
23.	4560134850150984	Tulsa, OK	24 May 2005	18:53	$82.00	782	Point of sale transaction	...
24.	3565603492346034	Houston, TX	24 May 2005	18:52	$76.00	780	Mail Order/Telephone Order	...
25.	0135926502394834	Austin, TX	24 May 2005	19:12	$300.00	779	Cash advance	...
26.	5604293487293560	Arlington, VA	24 May 2005	19:06	$49.99	770	Point of sale transaction	...
27.	0650126349218434	Springfield, MA	24 May 2005	19:01	$500.00	765	Mail Order/Telephone Order	...

To measure the effectiveness of the prediction module over some period of time, the system can compute several performance ratios based on the measures discussed earlier in this section:

[18] The screen shows only decimal digits, so the score 0.982 is displayed as 982.

- *True-positive ratio:* Correctly classified fraudulent transactions / total number of fraudulent transactions.
- *True-negative ratio:* Correctly classified legitimate transactions / total number of legitimate transactions.
- *False-positive ratio:* Incorrectly classified legitimate transactions / total number of legitimate transactions.
- *False-negative ratio:* Incorrectly classified fraudulent transactions / total number of fraudulent transactions

The sample report below displays these ratios (together with absolute numbers) for the first quarter of the year:



Fraud Detection System - Performance Report

January 1, 2005 - March 31, 2005

Month	Fraudulent Transactions	Legitimate Transactions	True Positive		True Negative		False Positive		False Negative	
			Number	Ratio	Number	Ratio	Number	Ratio	Number	Ratio
Jan.	5,276	2,363,893	2,375	45.02%	2,353,805	99.57%	10,088	0.43%	2,901	54.98%
Feb.	5,128	2,263,725	2,788	54.37%	2,251,430	99.46%	12,295	0.54%	2,340	45.63%
Mar.	4,962	2,427,582	2,490	50.18%	2,415,800	99.51%	11,782	0.49%	2,472	49.82%

This report indicates that the true-positive ratio for January was 45.02%, which means that during the month of January the system correctly classified 45.02% of the fraudulent transactions (2,375 fraudulent transactions were correctly classified from a total number of 5,276). During the same time period, the false-positive ratio was 0.43%, which means that the system incorrectly classified 0.43% of the legitimate transactions during the month of January (10,088 legitimate transactions were incorrectly classified from a total number of 2,363,893).

As mentioned earlier, it might be desirable to use several prediction methods for the problem of detecting fraudulent transactions. One of the more popular prediction methods used for this type of problem is based on rules, which are defined and maintained for classifying new transactions. Some other widely used methods for fraud detection include artificial neural networks, fuzzy logic, and decision trees – all of which have their own advantages and disadvantages. As a case in point, a considerable deficiency of artificial neural networks is their inability to provide a logical explanation for why a transaction was classified as fraudulent. However, models based on artificial neural networks are computationally very fast, and so the prediction is made in a fraction of a second.

Because the models used for detecting fraudulent transactions are trained and tested on historical data, the performance of these prediction models depends on the quality and quantity of available data. As an example, a prediction model that is based on credit card transaction data will detect fraud by looking at the amount, frequency, and location of the transactions. If we also had additional cardholder data, such as recent changes of address, age, etc., then we could improve the model's performance.

To allow the prediction module to learn and recognize new fraud patterns as they occur, an adaptability module is required. For example, the rule-base might be modified at regular intervals (i. e., adjustment of weights for individual rules, dropping existing rules, adding new rules), so that the number of false-positives and false-negatives is reduced. The adaptability module is also responsible for continuously adjusting the weights of the individual prediction models, thereby ensuring that the more useful (i. e., precise) models exert more influence on the final prediction. In our earlier example, these weights were equal to 0.3, 0.2, 0.15, and 0.35, respectively.

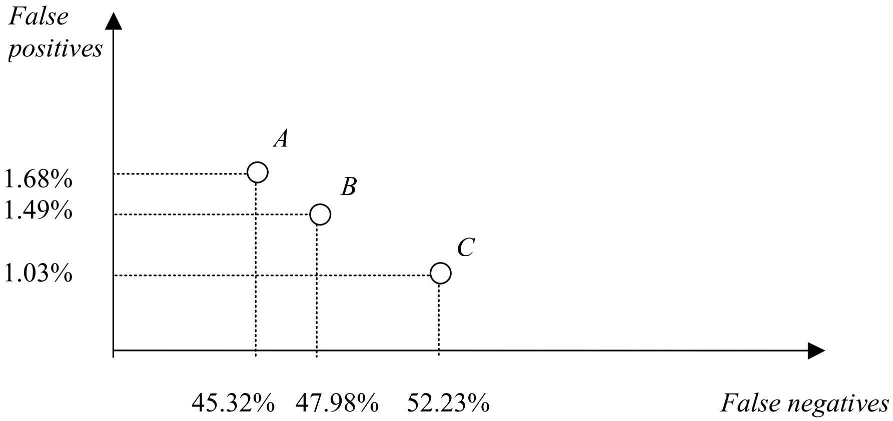
For the problem of detecting fraudulent transactions, the optimization module is responsible for recommending the best suspicion thresholds. Recall that the prediction module has different thresholds for the final suspicion score, which define various courses of action:

- If the score is not greater than α – grant authorization.
- If the score is greater than α, but not greater than β – grant authorization, but flag the transaction for later review by a fraud prevention officer.
- If the score is greater than β – deny authorization and call customer.

Now, what are the "best" suspicion thresholds? Well, this is not so straightforward, as many issues should be taken into account. It is important to minimize losses (thus minimizing the number of false-negatives), but it is also very important to take into account customer satisfaction (thus minimizing the number of false-positives). As we indicated in Sect. 2.4, it is quite unusual for any real-world business problem to have only one objective. This is precisely the case here, and so fraud detection can be viewed as a multi-objective optimization problem. Hence, the optimization module should find the optimal values of α and β (assuming two thresholds in the decision making process) to minimize false-negatives and false-positives.

The task of the optimization module is to identify a set of non-dominated solutions,[19] like *A*, *B*, and *C* below:

[19] For a discussion on non-dominated solutions, see Sect. 2.4.



Each of these solutions produces some percentage of false-positives and false-negatives for threshold values α and β. Solution *A* produces the lowest number of false-negatives (45.32% versus 47.98% and 52.23% for solutions *B* and *C*, respectively), whereas solution *C* produces the lowest number of false-positives (1.03% versus 1.68% and 1.49% for solutions *A* and *B*, respectively). Based upon management's priorities for customer satisfaction and loss prevention, the implemented solution will determine the thresholds for α and β.

Using an Adaptive Business Intelligence system for fraud detection may produce significant savings over a static rule-based system. These savings can be attributed to the system's ability to recommend the best suspicion thresholds for flagging transactions, as well as its capacity to adapt to the ever-changing environment.

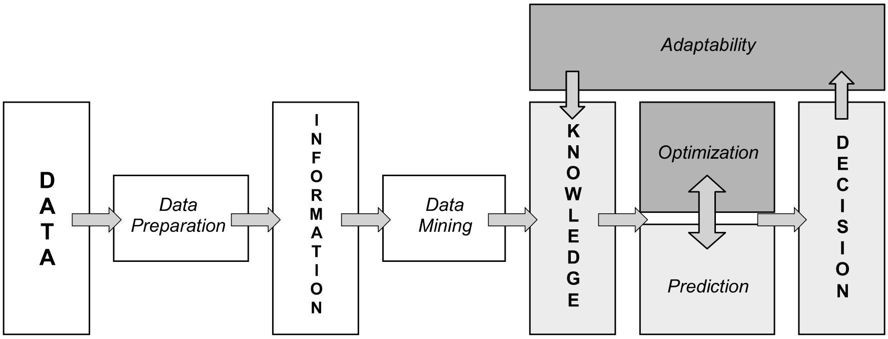
13 Conclusions

> “ ‘Come, Watson, come!’, he cried. ‘The game is afoot. Not a word! Into your clothes and come!’ ”
> *The Adventure of the Abbey Grange*
>
> “ ‘You wish to employ me as a consulting detective?’
> ‘Not that only. I want your opinion as a judicious man – as a man of the world. I want to know what I ought to do next.’ ”
> *The Yellow Face*

During the past few decades, organizations have strived to improve their decision-making capability by collecting and storing more data. The need to convert this raw data into useable knowledge has fueled the growth of the business intelligence industry, which provides software tools for retrieving, summarizing, and interpreting data for end users. Most of these business intelligence systems can: (a) access data from a variety of sources, (b) facilitate the transformation of data into knowledge, and (c) display this knowledge through a variety of graphical reports. *But is this enough?*

Although the major goal of business intelligence is to help managers make faster and smarter decisions, organizations now realize that a vast gulf exists between having the “right knowledge” and making the “right decision.” To answer questions such as “*What should be done to increase profits? reduce costs? and increase market share*?” business managers need more than graphs, charts, and numerical reports. They need systems that can predict the future and recommend the best course of action. For this reason, the future of the business intelligence industry lies in systems that can predict, optimize, and adapt – put another way, the future of the industry lies in *Adaptive Business Intelligence.*

The following diagram (copied from Chap. 1) illustrates this concept:



The power and appeal of Adaptive Business Intelligence systems reside in their ability to answer the two fundamental questions behind all business decisions: *What is likely to happen in the future?* and *What is the best decision right now?* Without a doubt, organizations that can accurately answer these questions on a consistent basis will have a competitive advantage over those that cannot. By combining prediction (*What is likely to happen in the future?)* and optimization (*What is the decision right now?*) into one system, business managers can reach new heights in their decision-making efficiency.

In addition to prediction and optimization, the concept of *adaptability* is also central to Adaptive Business Intelligence. As we discussed in Chap. 1, adaptability has already been introduced into many consumer products, such as car transmissions, television sets, and running shoes. Because these mass-produced products can adapt to the preferences of each unique owner, they are rapidly gaining popularity with consumers around the world. The logical extension of this trend is *adaptive software*: imagine a mass-produced software system that can adapt to the daily operation of any organization!

Note that the whole concept of Adaptive Business Intelligence also mimics the human brain, which constantly makes a variety of predictions (e. g., "the price of technology-related stocks will go up" or "there will be less traffic on the roads today"), decisions (e. g., "I will buy 10,000 shares of XYZ technology company" or "I will do my shopping today"), and later modifications (e. g., "I bought too many shares of XYZ technology" or "I should have done my shopping in the morning, rather than the afternoon"). Thus, Adaptive Business Intelligence addresses the core issue of Artificial Intelligence, which is how to make computers more useful and intelligent.

In this book, we discussed the prediction, optimization, and adaptability modules in detail, and illustrated how they can be integrated to create an Adaptive Business Intelligence system. Besides explaining the working principles behind Adaptive Business Intelligence, this book can also serve as a "high-level" guide for solving specific business problems. To that end, Part II covered a variety of prediction methods and modern optimization techniques (e. g., evolutionary algorithms, simulated annealing, tabu search, ant systems), and Part III explained how to combine these methods and techniques into an Adaptive Business Intelligence system. In addition to the real-world distribution example used throughout the text, Chap. 12 also provided some additional problem domains to which Adaptive Business Intelligence can be applied.

Although the functionality of Adaptive Business Intelligence systems will evolve over time, the goal of these systems will remain the same: solving real-world business problems that have complex constraints, multiple (possibly conflicting) objectives, an enormous number of possible solutions, and which are set in a time-changing environment. Consequently, these systems will always depend on modules for prediction, optimization, and adaptability.

So, what can we expect to see during the next decade?

First of all, the knowledge underpinning Adaptive Business Intelligence will become "standard." Business managers, IT practitioners, and students will need to understand the methods and techniques used to create Adaptive Business Intelligence systems. Many universities will institute courses on Adaptive Business Intelligence, covering topics such as data mining, prediction methods, optimization techniques, and modern heuristics. Until now, these topics were usually taught in separate courses, often across different departments and schools (e. g., operations research, computer science, engineering, applied mathematics, statistics).

Furthermore, we will see a continuation of topic-specific research related to Adaptive Business Intelligence. For example, most of the research in machine learning has been focused on using historical data to build prediction models. Once the model is built and evaluated, the goal is accomplished. However, because new data arrive at regular intervals, building and evaluating a model is just the first step in Adaptive Business Intelligence. Because these models need to be updated regularly (something that the adaptability module is responsible for), we expect to see more emphasis on this updating process in machine learning research. The frequency of updating the prediction module is also a very important research subject in Adaptive Business Intelligence. These frequencies can vary from seconds (e. g., in currency trading systems), to hours (e. g., pollution control system), to days (e. g., car distribution system), to weeks and months (e. g., fraud detection systems). Different update frequencies require different techniques and methodologies; thus, systems based on Adaptive Business Intelligence would include the research results from control theory, statistics, operations research, machine learning, and modern heuristic methods, to name a few. Recall also that different prediction models (e. g., decision trees or neural networks) require different methodologies for implementing adaptability.

We also expect that major advances will continue to be made in modern optimization techniques. Some of these advances already include new techniques based on the paradigm of adaptive memory and the concept of variable neighborhoods. In the years to come, more and more research papers will be published on constrained and multi-objective optimization problems, and on optimization problems set in dynamic environments. This is essential, as most (if not all) real-world business problems are constrained, multi-objective, and set in a time-changing environment.

Without any doubt, the next decade will bring many advances in Adaptive Business Intelligence and its related disciplines. These advances will be reported (in detail) in future proceedings of international conferences on Adaptive Business Intelligence. And in the words of Sherlock Holmes:

"*If you care to smoke a cigar in our rooms, Colonel, I shall be happy to give you any other details which might interest you.*"

We could not have said it better ourselves.

Index