Chapter 16
The Genetic Algorithm

The essence of machine learning is the *search* for the best solution to our problem: to find a classifier which classifies as correctly as possible not only the training examples, but also future examples. Chapter 1 explained the principle of one of the most popular AI-based search techniques, the so-called *hill-climbing*, and showed how it can be used in classifier induction.

There is another approach to search: the *Genetic Algorithm*, inspired by the principles of Darwinian evolution. The reader needs to be acquainted with it because the technique can be very useful in dealing with various machine-learning problems. This chapter presents the baseline version, and then illustrates its use using certain typical issues from the field of *k*-NN classifiers.

16.1 The Baseline Genetic Algorithm

Let us first briefly describe the general principle of the genetic algorithm, relegating the details of implementation to the next section.

The Basic Philosophy In this section, the classifier will encode in the form of a *chromosome*, which most of the time will be a string of bits that are sometimes referred to as "genes." The genetic algorithm operates with a population of chromosomes, each describing one individual (a classifier). Each such individual is assigned a value by a *fitness function*; this value will usually depend on the classifier's performance. The fitness function plays a role analogous to that of the evaluation function in heuristic search.[1]

[1]This chapter will use the terms "evaluation function," "survival function", and "fitness function" interchangeably.

M. Kubat, *An Introduction to Machine Learning*,
DOI 10.1007/978-3-319-63913-0_16

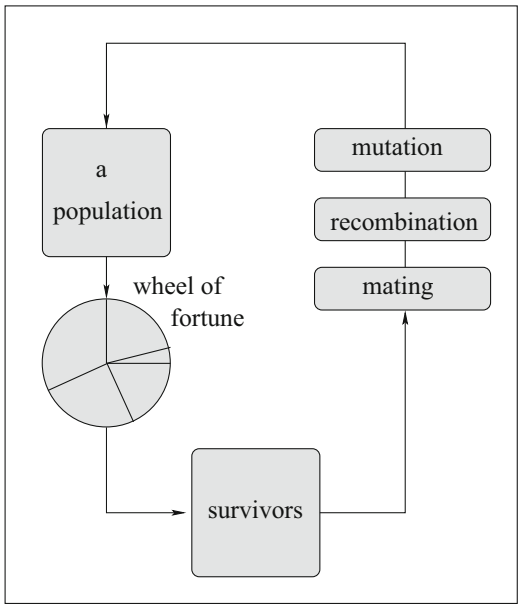

Fig. 16.1 The genetic algorithm's endless loop. Each individual in the population has its chance of survival. Recombination of the genetic information provided by mating partners creates new chromosomes that may be corrupted by mutation

The Genetic Algorithm's Loop The genetic algorithm operates in an endless loop depicted in Fig. 16.1. At each moment, there is a population of individuals, each with a certain value of the fitness function. This value then determines the size of the segment belonging to the individual in a "wheel of fortune" that determines the individual's chances of survival. It is important to understand the probabilistic nature of the process. While an individual with a larger segment enjoys a higher chance of survival, there is no guarantee of it because the survival game is non-deterministic. In the real world, too, a specimen with excellent genes may perish in a silly accident, while a weakling can make it by mere good luck. But in the long run, and in large populations, the laws of probability will favor genes that contribute to high fitness.

The surviving specimens will then choose "mating partners." In the process of mating, the chromosomes of the participating individuals are *recombined* (see below), which gives rise to a pair of new chromosomes. These new chromosomes may subsequently be subjected to *mutation*, which essentially adds noise to the strings of genes.

The whole principle is summarized by the pseudocode in Table 16.1.

How the Endless Loop Works Once a new population has been created, the process enters a new cycle in which the individuals are subjected to the same wheel of fortune, followed by mating, recombination, and mutation, and the story goes on and on until stopped by an appropriate termination criterion. Note how occasional wrong turns are eliminated by the probabilistic nature of process. A low-quality chromosome may survive the wheel of fortune by a fluke; but if its children's fitness values remain low, the genes will perish in subsequent generations

Table 16.1 The principle of the genetic algorithm

initial state: a population of individual chromosomes
1. The fitness of each individual is evaluated. Based on its value, individuals are randomly selected for *survival*.
2. Survivors select mating partners.
3. New individuals are created by chromosome *recombination* of the mating partners.
4. Individual chromosomes are corrupted by random *mutation*.
5. Unless a termination criterion is satisfied, the algorithm returns to step 1.

anyway. Alternatively, some of an unpromising individual's genes may prove to be useful when embedded in different chromosomes which they may enter through recombination. By giving them an occasional second chance, the process offers flexibility that would be impossible in a more deterministic setting.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the main principle of the genetic algorithm. How are the individuals described here? What is meant by their "survival chances"?
- Summarize the basic loop of the genetic algorithm.
- What is the advantage of the probabilistic implementation of the principle of survival as compared to a possible deterministic implementation?

16.2 Implementing the Individual Modules

Let us take a closer look at how to implement in a computer program the basic aspects of the genetic algorithm: the survival game, the mating process (partner selection), chromosome recombination, and mutation. To begin with, we will discuss only very simple solutions, relegating more advanced techniques to later sections.

For the sake of simplicity, we will assume that the chromosomes acquire the form of binary strings such as `[1 1 0 1 1 0 0 1]`, where each bit represents a certain property that is either present, in which case the bit has value `1`, or absent, in which case the bit is `0`. Thus in a simplified version of the "pies" problem, the first bit may indicate whether or not the `crust` is `thick`, the second bit may indicate whether or not the `filling` is `black`, and so on.

Initial Population The most common approach to creating the initial population will employ a random-number generator. Sometimes, the engineer can rely on some knowledge that may help her create initial chromosomes known to outperform randomly generated individuals. In the "pies" domain, this role can be played by the descriptions of the positive examples. However, one has to make sure that the initial population is sufficiently large and has sufficient diversity.

The Survival Game The genetic algorithm assumes that there is a way to calculate for each specimen its survival chances. In some applications, these chances can be established by a practical experiment that lets the individual specimens to fight it out. In other domains, the fitness is calculated by a user-specified evaluation function whose value depends on the chromosome's properties. And if the chromosome represents a classifier, the fitness function can rely on the percentage of the training examples correctly labeled by the classifier.

An individual's survival is determined probabilistically. Here is how to implement this "wheel of fortune" in a computer program. Let F_i denote the i-th specimen's fitness and let $F = \Sigma_i F_i$ be the sum of all individual's fitness values that are then arranged along the interval $(0, F]$. The survival is modeled by a random-number generator that returns some $r \in (0, F]$: the sequential number of the subinterval that has been "hit" by r then points to the survivor. The principle is illustrated in Fig. 16.2 for a small population of four specimens and a random number that lands in the third interval so that individual 3 is selected. If the fate wants 20 specimens to survive, it has to generate 20 random numbers whose locations in the interval $(0, F]$ identify the survivors.

Whereas specimens with small fitness are likely to get eliminated, those with higher values can appear in the pool of survivors more than once. A biologist will wince at this "cloning" idea, but in the pragmatic world of computer programmers, the same individual can "survive" twice, three times, or even many times.

The Mating Operator The survival game is followed by mating. In nature, an individual judges a partner's suitability by strength, speed, or sharp teeth. Something similar is accomplished in a computer implementation by means of the fitness

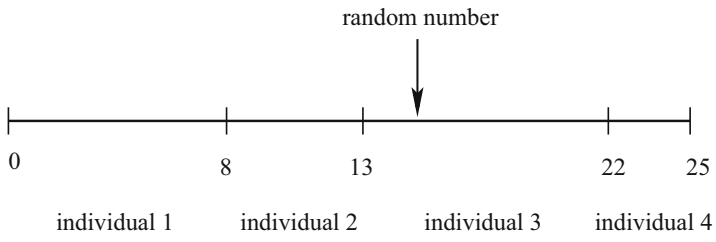

Fig. 16.2 The axis represents a population of four individuals whose fitness values are 8, 5, 9, and 3, respectively. Since the randomly generated number, 15, falls into the third subinterval, the third individual is selected

function. There is a difference, though: the notion of sex is usually ignored—any chromosome can mate with any other chromosome.

An almost trivial mating strategy will pair the individuals arbitrarily, perhaps generating random pairs of integers from the interval $[1, N_s]$, where N_s is the number of specimens in the population. However, this technique fails to do justice to the circumstance that specimens with high fitness are likely to be deemed more attractive than others. A simple way to reflect this in a computer program is to order the individuals in a descending order of their fitnesses, and then pair the neighbors.

Yet another strategy does it probabilistically. It takes the highest-ranking individual, then chooses its partner using the mechanism employed in the survival game—see Fig. 16.2. The same is done for the second highest-ranking individual, then for the third, and so on, until the new population has reached the required size. "Better" individuals are thus likely (though not guaranteed) to mate with other strong individuals. Sometimes, the partner will have low value (due to the probabilistic selection), but this gives rise to diversity that gives the system the opportunity to preserve valuable chromosome chunks that only have the bad luck of being currently incorporated in low-quality specimens.

Long-Living and Immortal Individuals One of the shortcomings of this algorithm is that a very good organism may be replaced by lower-valued children and useful genes may disappear. To prevent this from happening, some computer programs copy the best specimens into the new generation alongside their children. For instance, the program may directly insert in the new generation 20% best survivors, and then create the remaining 80% by applying the recombination and mutation operators to the best 95% individuals, totally ignoring the bottom 5%. In this way, not only will the best specimens live longer (even become "immortal"), but the program will also get rid of some very weak specimens that have survived by mere chance.

Chromosome Recombination: One-Point Crossover The simplest way to implement chromosome recombination is by the *one-point crossover*, an operator that swaps parts of the information in the parent chromosomes. The principle is simple. Suppose that each chromosome consists of a string of n bits and that a random-number generator has returned an integer $i \in [1, n]$. Then, the last i bits in the first chromosome (its i-bit tail) are replaced with the last i bits in the second chromosome and vice versa. A concrete implementation can permit the situation where $i = n$, in which case the two children are just replications of their parents. In the example below, the random integer is $i = 4$, which means that 4-bit tails are exchanged (the crossover point is indicated by a space).

$$\begin{array}{l} 1101\ 1001 \\ 0010\ 0111 \end{array} \Rightarrow \begin{array}{l} 1101\ 0111 \\ 0010\ 1001 \end{array}$$

The reader can see that the children tend to take after their parents, especially when the exchanged tails are short. The maximum distance between the children and the parents is achieved when $i = n - 1$.

In many applications, the recombination operator is applied only to a certain percentage of individuals. For instance, if 50 pairs have been selected for mating, and if the probability of recombination has been set by the user as 80%, then only 40 pairs will be subject to recombination, and the remaining 10 will just be copied into the next generation.

The Mutation Operator The task for mutation is to corrupt the inherited genetic information. Practically speaking, this is done by flip-flopping a small percentage of the bits in the sense that a bit's value `0` is changed to `1` or the other way round. The concrete percentage (the frequency of mutations) is a user-set parameter. Suppose that this parameter requires that $p = 0.001$ of the bits should on average be thus affected. The corresponding program module will then for each bit generate a random integer from the interval $[1, 1000]$. If the integer equals 1, then the bit's value is changed, otherwise it is left alone.

Let us give some thought to what frequency of mutations we need. At one extreme, very rare mutations will hardly have any effect at all. At the other extreme, very high mutation frequency would disrupt the genetic search by damaging too many chromosomes. If the frequency approaches 50%, then each new chromosome will behave as a randomly generated bit string; the genetic algorithm then degenerates to a random-number generator.

The mutation operator serves a different purpose than the crossover operator. In the one-point crossover, no new information is created, only existing substrings are swapped. Mutation introduces some new twist, previously absent in the population.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the main task of the survival game and how would you implement it in a computer program?
- Describe a simple mechanism to implement the selection of the mating partners. Describe the recombination operator, and the mutation operator.

16.3 Why It Works

Let us now offer an intuitive explanation of the genetic algorithm's performance.

Function Maximization The goal of the simple problem in Table 16.2 is to find the value of x for which the function $f(x) = x^2 - x$ is maximized. Each chromosome in the second column of the upper table is interpreted as a binary-encoded integer whose decadic value is given in the third column. The fourth column gives the

Table 16.2 Illustration of the genetic algorithm

Suppose we want the genetic algorithm to find the maximum of $f(x) = x^2 - x$. Let x be an integer represented by a binary string. The initial population consists of the four strings in the following table that for each of them gives the integer value, x, the corresponding $f(x)$, the survival chances (proportional to $f(x)$), and the number of times each exemplar was selected for the next generation.

No.	Initial population	x	$x^2 - x$	Survival chance	actual count
1	0 1 1 0 0	12	132	0.14	1
2	1 1 0 0 1	25	600	0.50	2
3	0 1 0 0 0	8	56	0.05	0
4	1 0 0 1 1	19	342	0.31	1
Average			282		
Maximum			600		

In the sample run reported here, the neighboring specimens mated, exchanging 1-bit tails and 3-bit tails, respectively, as dictated by the randomly generated tail lengths (the crossover sites indicated by spaces). No mutation is used here. The last two columns give the values of x and $f(x)$ for the new generation.

After reproduction	Mate with	Tail length	New population	x	$x^2 - x$
0 1 1 0 0	2	1	0 1 1 0 1	13	156
1 1 0 0 1	1	1	1 1 0 0 0	24	552
1 1 0 0 1	4	3	1 1 0 1 1	27	702
1 0 0 1 1	3	3	1 0 0 0 1	17	289
Average					425
Maximum					702

The reader can see that the value of the best specimen and the average value in the entire population have increased.

corresponding $f(x)$ whose relative value, shown in the fifth column, then determines for each individual its survival chances. For example, the first specimen has $f(x) = 12^2 - 12 = 132$ and the relative chances of survival (in this particular population) are 14% because $132/(132 + 600 + 56 + 342) = 0.14$. The rightmost column tells us how many times each individual has been selected for inclusion in the next generation.

In the next step, the survivors identify their mating partners. Let us assume that we have simply paired the neighboring specimens: the first with the second, and the third with the fourth. Then, the random selection of the crossover point dictates that 1-bit tails be exchanged in the first pair and 3-bit tails in the second. No mutation is applied. The result is shown in the bottom table where the last three columns

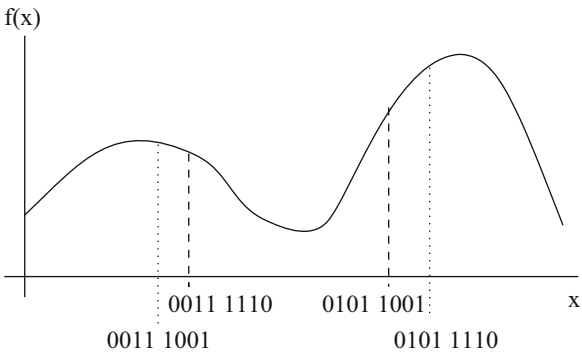

Fig. 16.3 After exchanging 4-bit tails, two parent chromosomes (upper strings) give rise to two children (lower strings). There is a chance that at least one child will "outperform" both parents

show, respectively, the new binary strings, their decadic values, and the values of $f(x)$. Note that both the average and the maximum value of the fitness function have increased.

Do Children Have to Outperform Their Parents? Let us ask what caused this improvement. An intuitive answer is illustrated in Fig. 16.3 that shows the location of two parents and the values of the survival function, $f(x)$, for each of them (the dashed vertical lines). When the two chromosomes swap their 4-bit tails, two children are created, each relatively close to one of the parents. The fact that each child finds itself in a region where the values of $f(x)$ are higher than those of the parents begs the question: are children always more fit than their parents? Far from that. All depends on the length of the exchanged tails and on the shape of the fitness function. Imagine that in the next generation the same two children get paired with each other and that the randomly generated crossover point is at the same location. Then, these children's children will be identical to the two original strings (their "grandparents"); this means that the survival chances decreased back to the original values. Sometimes, both children outperform their parents; in other cases, they are weaker than their parents; and quite often, we get a mixed bag. What matters is that in a sufficiently large population, most of the better specimens will survive because the selection process favors individuals with higher fitness, $f(x)$. Unfit specimens will occasionally make it, but they tend to lose in the long run.

If the exchanged string-tails are short, the children are close to their parent chromosomes. Long tails will give rise to children much less similar to their parents. As for mutation, its impact on the distance between the child and its parent depends on which bit is mutated. If it is the leftmost bit, the mutation will cause a big jump along the horizontal axis. If it is the rightmost bit, the jump is short. Either way, mutation complements recombination. Whereas the latter tends to explore the space in the vicinity of the parent chromosomes, the former may look elsewhere.

The Shape of the Fitness Function Some potential pitfalls inherent in the definition of the fitness functions are illustrated in Fig. 16.4. The function on the left is almost flat. The fact that different individuals have here virtually the same chances to survive defeats the purpose of the survival game. When the survivors are

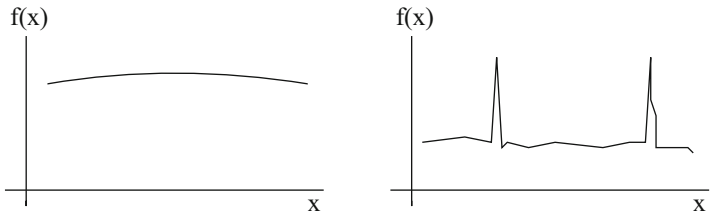

Fig. 16.4 Examples of two fitness functions that are poor guides for the genetic search. To be useful, the survival function should not be too flat and it should not contain isolated narrow peaks

chosen according to a near-uniform distribution, the qualities of the individuals will not give these individuals any perceptible competitive advantage. This drawback can be mitigated by making $f(x)$ less flat. There is an infinite number of ways this can be achieved, one possibility being to replace $f(x)$ with, say, $f(x) = f^2(x)$.

The right-hand part of Fig. 16.4 shows another pitfall: isolated narrow peaks. In comparison to the widths of the "peaks," children may find themselves too far from their parents. For instance, if the parent lies just at a hill's foot, the child may find itself on the opposite side, in which case the peak will go unnoticed. This problem is more difficult to prevent than the previous one.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain how the location of the crossover point determines how much the children will differ from their parents.
- Explain how the mutual interplay between recombination and mutation may affect the survival chances. Show how they also depend on the concrete shape of the survival function and on the location of the parents.

16.4 The Danger of Premature Degeneration

The fact that the genetic algorithm reached a value that does not seem to improve over a series of generation does not yet mean the search has been successful. The plateau may be explained by other circumstances.

Premature Degeneration A simple implementation of the genetic algorithm will stop after a predefined number of generations. A more sophisticated version will

keep track of the highest fitness value achieved so far, and then terminate the search when this value no longer improves.

There is a catch, though. The fact that the fitness value has reached a plateau may not guarantee that a solution has been found. Rather, the search might have reached the stage called *premature degeneration*. Suppose that the search from Table 16.2 has reached the following population:

```
0 1 0 0 0
0 1 0 0 1
0 1 0 0 0
0 1 0 0 0
```

What are the chances of improving this population? Recombination will not get us anywhere. If the (identical) last two chromosomes mate, the children will only be copies of the parents. If the first two are paired, then 1-point crossover will only swap the rightmost bit, an operation that does not create a new chromosome, either. The only way to cause a change is to use mutation. By changing the appropriate bits, mutation can reignite the search. For instance, this will happen after the mutation of the third bit in the first chromosome and the fourth bit (from the left) of the last chromosome. Unfortunately, mutations are rare, and to wait for this to happen may be impractical. For all practical purposes, premature degeneration means the search got stuck.

Preventing Premature Degeneration Premature degeneration has a lot to do with the population's *diversity*. The worst population is one in which all chromosomes have exactly the same bit string, something the engineer wants to avoid. Any computer implementation will therefore benefit from a module that monitors diversity and takes action whenever it drops below a certain level. A simple way to identify this situation is to calculate the average similarity between pairs of chromosomes, perhaps by counting the number of bits that have the same value in both strings. For instance, the similarity between `[0 0 1 0 0]` and `0 1 1 0 0]` will be 4 (four bits are equal) and the similarity between `[0 1 0 1 0]` and `[1 0 1 0 1]` will be 0.

Once a drop in average chromosome-to-chromosome similarity has been detected, the system has to react. This is not yet a cause for alarm. Thus in the function-maximization example, advanced generations will be marked by populations where most specimens are already close to the maximum. This kind of "degeneration" will certainly *not* be deemed "premature." However, the situation is different if the best chromosome can be shown to be very different from the solution. In this event, we have to increase diversity.

Increasing Diversity Several strategies can be used. The simplest will just insert in the current population one or more newly created random individuals. A more sophisticated approach will run the genetic algorithm on two or more populations in parallel, in isolation from each other. Then, either at random intervals, or whenever

premature degeneration is suspected, a specimen from one population will be permitted to choose its mating partner in a different population. When implementing this technique, the programmer has to decide in which population to place the children.

The Impact of Population Size Special attention has to be paid to the size of the population. Usually, though not always, the size is kept constant throughout the entire genetic search. The number of individuals in the population will be dictated by the concrete application. As a rule of thumb, smaller populations will need many generations to reach a good solution—unless they degenerated prematurely. Very large populations may be robust against degeneration, but they may incur impractical computational costs.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- In what way does the success of the genetic algorithm depend on the definition of the fitness function? What are the two main pitfalls? How would you handle them?
- What criteria to terminate the genetic search would you recommend? What are their advantages and disadvantages?
- What is *premature degeneration*? How can it be detected and how can the situation be rectified? Why do we need diversity in the population?
- Discuss the impact of the population size.

16.5 Other Genetic Operators

We have introduced only a very simple version of the genetic algorithm and its operators. Now that the reader understands the principle, we take a look at some alternatives.

Two-Point Crossover The one-point crossover introduced above is only a special case of the much more common *two-point crossover*. Here, the random-number generator is asked to return two integers that define two locations in the binary strings. The parents then swap the substrings between these two locations as illustrated below (the two crossover points are indicated by spaces).

$$\begin{array}{lcl} 110\ \ 110\ \ 01 & & 110\ \ 001\ \ 01 \\ 001\ \ 001\ \ 11 & \Rightarrow & 001\ \ 110\ \ 11 \end{array}$$

The two crossover points can be different for each chromosome. In this event, each parent will "trade" a different substring of its chromosome as indicated below.

$$\begin{array}{l} 1\ \ 101\ \ 1001 \\ 001\ \ 001\ \ 11 \end{array} \Rightarrow \begin{array}{l} 1\ \ 001\ \ 1001 \\ 001\ \ 101\ \ 11 \end{array}$$

Random Bit Exchange Yet another variation on the chromosome-recombination theme is the so-called *random bit exchange*. Here, the random-number generator selects a user-specified number of locations, and then swaps the bits at these locations as illustrated below.

$$\begin{array}{l} 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1 \\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 1 \end{array} \Rightarrow \begin{array}{l} 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1 \\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 1 \end{array}$$

Here, the second and the sixth bits (counting from the left) were swapped. Note that nothing will happen if the leftmost bit is exchanged because it has the same value in both chromosomes. The number of exchanged bits can vary but most applications prefer the number to be much smaller than the chromosome's length.

A common practice in realistic applications is to combine two or more recombination operators. For instance, the selected pair of parents will with 50% probability be subjected to a 2-point chromosome, with 30% probability to a random bit exchange, and with 20% probability there will be no recombination at all.

Inversion Whereas the recombination operators act on pairs of chromosomes, other operators act on single specimens. One such operator is mutation; another is *inversion*. In a typical implementation, the random-number generator returns two integers that define two locations in the binary string (similarly as in the 2-point crossover). Then, the substring between the two positions is inverted as shown below.

$$110\ 110\ 01 \Rightarrow 110\ 011\ 01$$

Note that the order of the zeros and ones in the substring between the third and the seventh bit (counting from the left) was reversed. The location of the two points determines how much inversion impacts the chromosome. If the two integers are close to each other, say, 4 and 7, then only a small part of the chromosome is affected.

In advanced implementations, inversion is used to supplement mutation. For instance, the probability that a given bit is mutated can be set to 0.2% whereas each chromosome may have a 0.7% chance to see its random substring inverted. Similarly as with mutation, care has to be taken to make sure the inversion operator is used rarely. Excessive use may destroy the positive contribution of recombination.

Inversion and Premature Degeneration Much more than mutation, inversion is very good at extricating the genetic search from premature degeneration. To see why, take a look at the following degenerated population.

```
0 1 0 0 0
0 1 0 0 1
0 1 0 0 0
0 1 0 0 0
```

Inverting the middle three bits of the first chromosome, and the last three bits of the second chromosome will result in the following population:

```
0 0 0 1 0
0 1 1 0 0
0 1 0 0 0
0 1 0 0 0
```

The reader can see that the diversity has indeed increased. This observation suggests a simple way to handle premature degeneration: just increase, for a while, the frequency of inversions, and perhaps also that of mutations.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the differences between one-point crossover, two-point crossover, and random bit exchange.
- What specific aspect makes the recombination operators different from the mutation and inversion operators?
- How does inversion affect the genetic search?

16.6 Some Advanced Versions

The genetic algorithm is a versatile general framework with almost infinite possibilities of variations. This section will introduce two interesting techniques.

A Note on the Lamarckian Alternative Computer programs are not constrained by the limitations of biology. Very often, the engineer discards some of these limitations, just as early aviators abandoned the idea of feathered wings. We have already encountered one such violation when making some specimens "immortal," copying them into the new generation to make sure they would not be destroyed by recombination and mutation. Let us now look at another deviation.

In the baseline genetic algorithm, new substrings come into being only as a result of random processes during such operators as recombination or mutation. After this,

the genetic information remains unchanged throughout the specimen's entire life. One pre-Darwinian biologist, Jean-Baptiste Lamarck, suggested something more flexible: in his view, evolution might be driven by the individuals' needs. A giraffe that keeps trying to reach the topmost leaves will stretch his neck that will thus become longer. This longer neck is then passed on to the offspring. While the lamarckian hypothesis is untenable in the realm of biology, it is not totally irrational in other fields. For instance, by publishing a scientific paper, a researcher leaves to posterity the knowledge acquired during his lifetime.

Lamarckian evolution is much faster than that of the classical darwinian process, which is why we sometimes implement it in the genetic algorithm. The simplest way to incorporate this concept in the general loop from Fig. 16.1 is to place the "lamarckian" operator between the "wheel of fortune" and recombination. The task for the operator is to improve the chromosome by adaptation. For instance, one can ask what happens if a certain bit gets flipped-flopped by mutation. Whereas mutation by itself is irreversible, we can add flexibility by explicitly testing what happens when the i bit is flipped-flopped, and then choose the better version.

Multi-Population Search One motivation for multi-population search has to do with the many parameters the genetic algorithm depends on. Most of the time, the engineer has to rely only on her experience. Alternatively, we may choose to subject the same initial population to several parallel runs of the genetic algorithm, each with its own mutation frequency, with or without inversion, with a different mixture of recombination operators, or with a modified fitness function. Among the many alternatives, some will reach the solution faster than the others.

The reader will recall having encountered multi-population search in the section that discussed the threat of premature degeneration. In that particular context, the suggestion was to let two or more populations evolve in relative isolation that is disrupted by occasional interbreeding. Note that this interbreeding may not be easy to implement if each population uses a different way of chromosome definition as suggested in the previous paragraphs. In that case, the programmer has to implement a special program module for the conversion from one encoding to another.

Strings of Numbers, Strings of Symbols Chromosomes do not have to be binary strings; they can consist of numbers, or characters. The same recombination operators as before can then be used, though mutation may call for creativity. Perhaps the most common kind of mutation in numeric strings is to use "noise" superimposed on some (or all) of the chromosome's "genes." For instance, if all locations contain numbers from the interval $[0, 100]$, then the noise can be modeled as a random number from $[-a, a]$ where a is a user-set parameter that plays here a role similar to that of mutation frequency in binary strings. Here is how it can work:

Before mutation	10	22	17	42	16
The "noise"		−3	1	−2	
After mutation	10	19	18	40	16

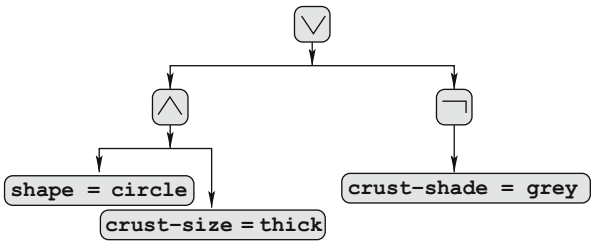

Fig. 16.5 A tree representation of a candidate expression from the "pies" domain

The situation is slightly different if the chromosomes have the form of strings of symbols. Here, mutation can replace a randomly selected symbol in the chromosome with another symbol chosen by the random-number generator. For instance, when applied to chromosome `[d s r d w k l]`, the mutation can change from `r` to `s` the third symbol from the left, the resulting chromosome being `[d s s d w k l]`.

Also possible are "mixed" chromosomes where some locations are binary, others numeric, and yet others symbolic. Here, mutation is usually implemented as a combination of the individual approaches. For instance, the program selects a random location in the chromosome, determines whether the location is binary, numeric, or symbolic, and then applies the appropriate type of mutation.

Chromosomes Implemented as Tree Structures In some applications, strings of bits, numbers, or symbols are inadequate; a tree-structure may then be more flexible. This, for instance, is the case of classifiers in the form of logical expressions—see the example in Fig. 16.5 where the following expression is represented by a tree-like chromosome.

```
(shape=circle ∧ crust-size=thick)  ∨  ¬ crust-shade=gray
```

The expression consists of attributes, the values of these attributes, and the logical operators of conjunction, disjunction, and negation. Note how naturally this is cast in the tree structure. The internal nodes represent the logical operations and the leaves contain the attribute-value pairs Recombination swaps random subtrees. Mutation can affect the leaves: either attribute names or attribute values or both. Another possibility for mutation is occasionally to replace $\wedge$ with $\vee$ or the other way round.

Special attention has to be paid to the way the initial population is generated. The programmer has to make sure that the population already contains some promising expressions. One possibility is to create a group of random expressions and to insert in it the descriptions of the positive examples. The survival function (to be maximized) can be defined as the classification accuracy on the training set.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the difference between the darwinian and the lamarckian evolution processes? Which of them is faster?
- What weakness is remedied by the multi-population genetic algorithm? In what way do multiple populations address this problem?
- How would you implement the mutation operator if the chromosome is a "mixed" string of bits, numeric values, and symbols?
- How would you implement the recombination and mutation operators in domains where chromosomes have the form of tree data structures?

16.7 Selections in *k*-NN Classifiers

Let us now illustrate a possible application of the genetic algorithm on a realistic problem from the field of machine learning.

Attribute Selection and Example Selection The reader knows that the success of the *k*-NN classifier depends on the quality of the stored examples and also on the choice of the attributes to describe these examples. The problem of choosing the right examples and attributes is easily cast in the search paradigm. For instance, the initial state can be defined as the complete set of examples, and the complete set of attributes; the search operators will remove examples and/or attributes; and the evaluation function (whose value is to be minimized) will be defined as the error rate reached by the 1-NN rule as measured on an independent set of testing examples.

Another possibility is to employ the genetic algorithm. In essence, we have to decide how to represent the problem in terms of chromosomes, how to define the fitness function, and the recombination and mutation operators. Then, we have to be clear about how to interpret (and utilize) the result of the search.

Chromosomes to Encode the Problem A very simple approach will divide the binary chromosome into two parts: each location in the first part corresponds to one training example, and each location in the second part corresponds to one attribute. If the value of a certain bit is `0`, the corresponding example or attribute is ignored, otherwise it is kept. The fitness function will be designed in a way that seeks to minimize the number of `1`s.

This solution may lead to impractically long chromosomes in domains where the training set contains many examples: if the training set has ten thousand examples, ten thousand bits would be needed. A better solution will then opt for the more flexible variable-length scheme where each element in the chromosome contains an integer that points to a training example or an attribute. The length of

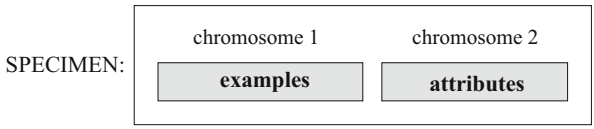

Fig. 16.6 Each specimen is described by two chromosomes, one representing examples and the other representing attributes. Recombination is applied to each of them separately

the chromosome would be the number of relevant attributes plus the number of representative examples. This mechanism is known as *value encoding*.

Interpreting the Chromosomes We must be sure to interpret the pairs of chromosomes properly. For instance, the specimen `[3,14,39],[2,4]` represents a training subset consisting of the third, the fourteenth, and the thirty-ninth training example, described by the second and the fourth attribute. When such specimen is used as a classifier, the system selects the examples determined by the first chromosome and describes them by the attributes determined by the second chromosome (Fig. 16.6). The distances between vectors $\mathbf{x} = (x_1, \ldots x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$ are calculated using the formula:

$$D(\mathbf{x}, \mathbf{y}) = \sqrt{\Sigma_{i=1}^{n} d(x_i, y_i)} \tag{16.1}$$

where $d(x_i, y_i)$ is the contribution of the ith dimension. For numeric attributes, this contribution can be calculated by the usual formula for Euclidean distance, $d(x_i, y_i) = (x_i - y_i)^2$; for boolean attributes and for discrete attributes, we may define $d(x_i, y_i) = 0$ if $x_i = y_i$ and $d(x_i, y_i) = 1$ if $x_i \neq y_i$.

The Fitness Function The next problem is how to quantify each individual's survival chances. Recall that we want to reduce the number of examples and the number of attributes without compromising classification accuracy. These requirements may contradict each other because, in noise-free domains, the entire training set tends to give higher classification performance than a reduced set. Likewise, removing attributes is hardly beneficial if each of them provides relevant information.

The involved trade-offs therefore should be reflected in fitness-function parameters that give the user the chance to specify the concrete preferences. The fitness function should make it possible to place emphasis either on maximizing the classification accuracy or on minimizing the number of the retained training examples and attributes. This requirement is expressed by the following formula where E_R is the number of training examples misclassified by the given specimen, N_E is the number of retained examples, and N_A is the number of retained attributes:

$$f = 1/(c_1 * E_R + c_2 * N_E + c_3 * N_A) \tag{16.2}$$

Note that the fitness of a specimen is high if its error rate is low, if the set of retained examples is small, and if many attributes have been eliminated. The function is controlled by three user-set parameters, c_1, c_2, and c_3, that weigh the user's preferences. For instance, if c_1 is high, emphasis is placed on classification accuracy. If c_2 or c_3 are high, emphasis is placed on minimizing the number of retained examples and on minimizing the number of retained attributes, respectively.

Genetic Operators for This Application Parents are selected probabilistically. In particular, the following formula is used to calculate the probability that the specimen S' will be chosen:

$$Prob(S') = \frac{f(S')}{\sum f(S)} \tag{16.3}$$

Here, $f(S)$ is the fitness of specimen S as calculated by Eq. (16.2). The denominator sums up the values of the fitness functions of all specimens in the population—this makes the probabilities sum up to 1.

Once the pair of parents have been chosen, their chromosomes are recombined by the two-point crossover. Since each specimen is defined by a pair of chromosomes, each with a different meaning, we apply the recombination operator to each of them separately. Let the length of one parent's chromosome be denoted by N_1 and let the length of the other parent's chromosome be denoted by N_2. Using the uniform distribution, the algorithm selects one pair of integers from the closed interval $[1, N_1]$ and another pair of integers from the closed interval $[1, N_2]$. Each of these pairs then defines a substring in the respective chromosome (the first and the last locations are included in the substring). The crossover operator then exchanges the substrings from one of the parent chromosomes with the substrings of the other parent. Note that, as each of these substrings can have a different size, the children's lengths are likely to be different from the parents' lengths.

Graphical Illustration The principle is illustrated in Fig. 16.7 where the middle parts of chromosomes A and B have been exchanged. Note how the lengths of A and B are affected. The engineer has to decide whether to permit the situation where the exchanged segments have size 0; in the other extreme, a segment can represent the entire parent.

The *mutation* operator should prevent premature degeneration of the population and make sure the population represents a representative part of the search space.

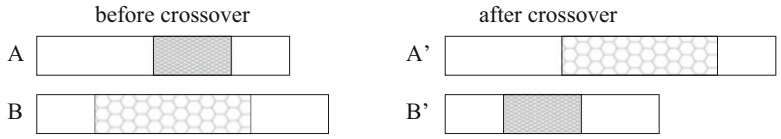

Fig. 16.7 The two-point crossover operator creates the children by exchanging randomly selected substrings in the parent chromosomes

One possibility is to select, randomly, a pre-specified percentage of the locations in the newly created population and to add to each of them a random integer generated separately for the location. The result is then taken modulo the number of examples/attributes. Let the original number of examples/attributes be 100 and let the location selected for mutation contains be 95. If the randomly generated integer is 22, then the value after mutation is $(95 + 22) \bmod 100 = 17$.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What can be accomplished by choosing the best attributes and the most representative examples?
- What are the advantages of using two chromosomes instead of just one?
- How does the chosen fitness function reflect the competing requirements of small sets of attributes and examples versus high classification accuracy?
- Why did we use a recombination operator that exchanges substrings of different lengths? How was mutation carried out?

16.8 Summary and Historical Remarks

- The genetic algorithm, inspired by the Darwinian evolution, is a popular alternative to classical artificial-intelligence search techniques. The simplest implementation works with binary strings.
- The algorithm subjects a population of individuals to three essential operations: fitness-function based survival, recombinations of pairs of chromosomes, and mutation. Also inversion of a substring is sometimes used.
- One of the frequently encountered problems in practical applications of the genetic algorithm is a population's premature degeneration. One way of detecting it is to consider the diversity of the chromosomes in the population. One solution will add artificially created chromosomes to the population. Also the inversion operator is useful, here.
- Alternative implementations of the genetic algorithm use strings of numbers, symbols, mixed strings, or even tree structures.
- The chapter illustrated the practical use of the genetic algorithm using a simple problem from the field of nearest-neighbor classifiers.

Historical Remarks The idea to cast the principle of biological evolution in the form of the genetic algorithm is due to Holland [37], although some other authors suggested something similar a little earlier. Among these, perhaps Rechenberg [80] deserves to be mentioned, while Fogel et al. [29] should be credited with

pioneering the idea of genetic programming. The concrete way of applying the genetic algorithm to selections in the k-classifier is from Rozsypal and Kubat [82].

16.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

Exercises

1. Hand-simulate the genetic algorithm with a pencil and paper in a similar way as in Table 16.2. Use a fitness function of your own choice, a different initial population, and the random points for a one-point crossover. Then repeat the exercise with the two-point crossover.

Give It Some Thought

1. Explain how different population sizes may affect the number of generations needed to reach a good solution. Elaborate on the relation of population size to the problem of premature degeneration. Discuss also the effect of the shape of the fitness function.
2. What types of search problems are likely to be more efficiently addressed by the genetic algorithm than by classical search algorithms?
3. Identify concrete engineering problems (other than those in the previous text) appropriate for the genetic algorithm. Suggest problems where the chromosomes are best represented by binary or numeric strings, and suggest problems where trees are more appropriate.
4. Name some differences between natural evolution and its computer model. Speculate on whether more inspiration can be taken from nature. Where do you think are the advantages of the computer programs as compared to biological evolution?

Computer Assignments

1. Implement the baseline genetic algorithm to operate on binary-string chromosomes. Make sure you have separate modules for the survival function, the wheel of fortune, recombination, and mutation, and that these modules are sufficiently general to enable easy modifications.
2. Create the initial populations for the "pies" and "circles" domains from Chap. 1 and use them as input to the program developed in the previous task. Note that, in the case of the "circles" domain, you might have to consider a slight modification of the original program so that it can handle numeric-string chromosomes.
3. For a domain of your choice, implement a few alternative mating strategies. Run systematic experiments to find out which strategy will most quickly find the solution. The speed can be measured by the number of chromosomes whose fitness values have to be evaluated before the solution is found.
4. For a domain of your choice, experiment with alternative "cocktails" of different recombination operators, and with different frequencies of recombinations, mutations, and inversions. Plot graphs that show how the speed of search (measured as in the previous task) depends on the concrete settings of these parameters.