Chapter 1
A Simple Machine-Learning Task

You will find it difficult to describe your mother's face accurately enough for your friend to recognize her in a supermarket. But if you show him a few of her photos, he will immediately spot the tell-tale traits he needs. As they say, a picture—an example—is worth a thousand words.

This is what we want our technology to emulate. Unable to define certain objects or concepts with adequate accuracy, we want to convey them to the machine by way of examples. For this to work, however, the computer has to be able to convert the examples into knowledge. Hence our interest in algorithms and techniques for *machine learning*, the topic of this textbook.

The first chapter formulates the task as a search problem, introducing hill-climbing search not only as our preliminary attempt to address the machine-learning task, but also as a tool that will come handy in a few auxiliary problems to be encountered in later chapters. Having thus established the foundation, we will proceed to such issues as performance criteria, experimental methodology, and certain aspects that make the learning process difficult—and interesting.

1.1 Training Sets and Classifiers

Let us introduce the problem, and certain fundamental concepts that will accompany us throughout the rest of the book.

The Set of Pre-Classified Training Examples Figure 1.1 shows six pies that Johnny likes, and six that he does not. These *positive* and *negative examples* of the underlying concept constitute a *training set* from which the machine is to induce a *classifier*—an algorithm capable of categorizing any future pie into one of the two *classes*: positive and negative.

M. Kubat, *An Introduction to Machine Learning*,
DOI 10.1007/978-3-319-63913-0_1

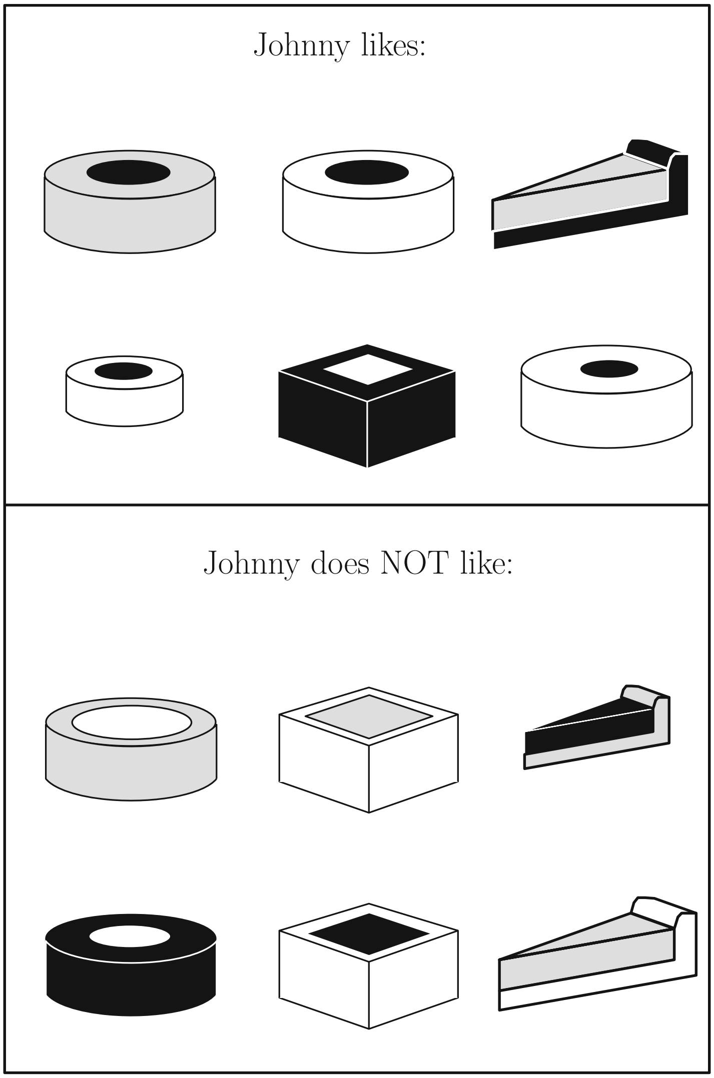

Fig. 1.1 A simple machine-learning task: induce a classifier capable of labeling future pies as positive and negative instances of “a pie that Johnny likes”

The number of classes can of course be greater. Thus a classifier that decides whether a landscape snapshot was taken in `spring`, `summer`, `fall`, or `winter` distinguishes four. Software that identifies characters scribbled on an *iPad* needs at least 36 classes: 26 for letters and 10 for digits. And document-categorization systems are capable of identifying hundreds, even thousands of different topics. Our only motivation for choosing a two-class domain is its simplicity.

Table 1.1 The twelve training examples expressed in a matrix form

Example	Shape	Crust		Filling		Class
		Size	Shade	Size	Shade	
ex1	Circle	Thick	Gray	Thick	Dark	pos
ex2	Circle	Thick	White	Thick	Dark	pos
ex3	Triangle	Thick	Dark	Thick	Gray	pos
ex4	Circle	Thin	White	Thin	Dark	pos
ex5	Square	Thick	Dark	Thin	White	pos
ex6	Circle	Thick	White	Thin	Dark	pos
ex7	Circle	Thick	Gray	Thick	White	neg
ex8	Square	Thick	White	Thick	Gray	neg
ex9	Triangle	Thin	Gray	Thin	Dark	neg
ex10	Circle	Thick	Dark	Thick	White	neg
ex11	Square	Thick	White	Thick	Dark	neg
ex12	Triangle	Thick	White	Thick	Gray	neg

Attribute Vectors To be able to communicate the training examples to the machine, we have to describe them in an appropriate way. The most common mechanism relies on so-called *attributes*. In the "pies" domain, five may be suggested: `shape` (circle, triangle, and square), `crust-size` (thin or thick), `crust-shade` (white, gray, or dark), `filling-size` (thin or thick), and `filling-shade` (white, gray, or dark). Table 1.1 specifies the values of these attributes for the twelve examples in Fig. 1.1. For instance, the pie in the upper-left corner of the picture (the table calls it `ex1`) is described by the following conjunction:

```
(shape=circle) AND (crust-size=thick) AND (crust-shade=gray)
AND (filling-size=thick) AND (filling-shade=dark)
```

A Classifier to Be Induced The training set constitutes the input from which we are to induce the classifier. But *what* classifier?

Suppose we want it in the form of a boolean function that is *true* for positive examples and *false* for negative ones. Checking the expression `[(shape=circle) AND (filling-shade=dark)]` against the training set, we can see that its value is *false* for all negative examples: while it *is* possible to find negative examples that are circular, none of these has a dark filling. As for the positive examples, however, the expression is *true* for four of them and *false* for the remaining two. This means that the classifier makes two errors, a transgression we might refuse to tolerate, suspecting there is a better solution. Indeed, the reader will easily verify that the following expression never goes wrong on the entire training set:

```
[ (shape=circle) AND (filling-shade=dark) ] OR
[ NOT(shape=circle) AND (crust-shade=dark) ]
```

Problems with a Brute-Force Approach How does a machine find a classifier of this kind? Brute force (something that computers are so good at) will not do here. Just consider how many different examples can be distinguished by the given set of attributes in the "pies" domain. For each of the three different `shapes`, there are two alternative `crust-sizes`, the number of combinations being $3 \times 2 = 6$. For each of these, the next attribute, `crust-shade`, can acquire three different values, which brings the number of combinations to $3 \times 2 \times 3 = 18$. Extending this line of reasoning to *all* attributes, we realize that the size of the *instance space* is $3 \times 2 \times 3 \times 2 \times 3 = 108$ different examples.

Each subset of these examples—and there are 2^{108} subsets!—may constitute the list of positive examples of someone's notion of a "good pie." And each such subset can be characterized by at least one boolean expression. Running each of these classifiers through the training set is clearly out of the question.

Manual Approach and Search Uncertain about how to invent a classifier-inducing algorithm, we may try to glean some inspiration from an attempt to create a classifier "manually," by the good old-fashioned pencil-and-paper method. When doing so, we begin with some tentative initial version, say, `shape=circular`. Having checked it against the training set, we find it to be *true* for four positive examples, but also for two negative ones. Apparently, the classifier needs to be "narrowed" (specialized) so as to exclude the two negative examples. One way to go about the specialization is to add a conjunction, such as when turning `shape=circular` into `[(shape=circular) AND (filling-shade=dark)]`. This new expression, while *false* for all negative examples, is still imperfect because it covers only four (`ex1`, `ex2`, `ex4`, and `ex6`) of the six positive examples. The next step should therefore attempt some generalization, perhaps by adding a disjunction: {`[(shape=circular) AND (filling-shade=dark)] OR (crust-size=thick)`}. We continue in this way until we find a 100% accurate classifier (if it exists).

The lesson from this little introspection is that the classifier can be created by means of a sequence of specialization and generalization steps which gradually modify a given version of the classifier until it satisfies certain predefined requirements. This is encouraging. Readers with background in Artificial Intelligence will recognize this procedure as a *search* through the space of boolean expressions. And Artificial Intelligence is known to have developed and explored quite a few of search algorithms. It may be an idea to take a look at least at one of them.

What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- What is the input and output of the learning problem we have just described?
- How do we describe the training examples? What is *instance space*? Can we calculate its size?

- In the “pies” domain, find a boolean expression that correctly classifies all the training examples from Table 1.1.

1.2 Minor Digression: Hill-Climbing Search

Let us now formalize what we mean by *search*, and then introduce one popular algorithm, the so-called *hill climbing*. Artificial Intelligence defines *search* something like this: starting from an *initial state*, find a sequence of steps which, proceeding through a set of interim *search states*, lead to a predefined *final state*. The individual steps—transitions from one search state to another—are carried out by *search operators* which, too, have been pre-specified by the programmer. The order in which the search operators are applied follows a specific *search strategy* (Fig. 1.2).

Hill Climbing: An Illustration One popular search strategy is *hill climbing*. Let us illustrate its essence on a well-known brain-teaser, the sliding-tiles puzzle. The board of a trivial version of this game consists of nine squares arranged in three rows, eight covered by numbered tiles (integers from 1 to 8), the last left empty. We convert one search state into another by sliding to the empty square a tile from one of its neighbors. The goal is to achieve a pre-specified arrangement of the tiles.

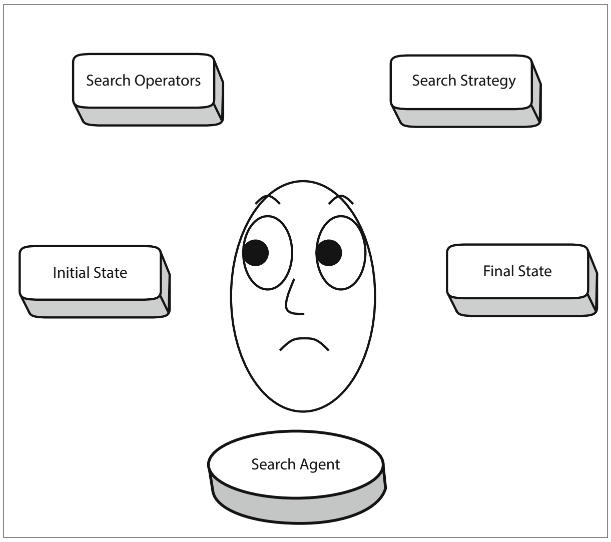

Fig. 1.2 A search problem is characterized by an initial state, final state, search operators, and a search strategy

The flowchart in Fig. 1.3 starts with a concrete initial state, in which we can choose between two operators: "move `tile-6` up" and "move `tile-2` to the left." The choice is guided by an *evaluation function* that estimates for each state its distance from the goal. A simple possibility is to count the squares that the tiles have to traverse before reaching their final destinations. In the initial state, tiles 2, 4, and 5 are already in the right locations; tile 3 has to be moved by four squares; and each of the tiles 1, 6, 7, and 8 have to be moved by two squares. This sums up to distance $d = 4 + 4 \times 2 = 12$.

In Fig. 1.3, each of the two operators applicable to the initial state leads to a state whose distance from the final state is $d = 13$. In the absence of any other guidance, we choose randomly and go to the left, reaching the situation where the empty square is in the middle of the top row. Here, three moves are possible. One of them would only get us back to the initial state, and can thus be ignored; as for the remaining two, one results in a state with $d = 14$, the other in a state with $d = 12$. The latter being the lower value, this is where we go. The next step is trivial because only one move gets us to a state that has not been visited before. After this, we again face the choice between two alternatives …and this how the search continues until it reaches the final state.

Alternative Termination Criteria and Evaluation Functions Other *termination criteria* can be considered, too. The search can be instructed to stop when the maximum allotted time has elapsed (we do not want the computer to run forever), when the number of visited states has exceeded a certain limit, when something sufficiently close to the final state has been found, when we have realized that all states have already been visited, and so on, the concrete formulation reflecting critical aspects of the given application, sometimes combining two or more criteria in one.

By the way, the evaluation function employed in the sliding-tiles example was fairly simple, barely accomplishing its mission: to let the user convey some notion of his or her understanding of the problem, to provide a hint as to which move a human solver might prefer. To succeed in a realistic application, we would have to come up with a more sophisticated function. Quite often, *many* different alternatives can be devised, each engendering a different sequence of steps. Some will be quick in reaching the solution, others will follow a more circuitous path. The program's performance will then depend on the programmer's ability to pick the right one.

The Algorithm of Hill Combing The algorithm is summarized by the pseudocode in Table 1.2. Details will of course depend on each individual's programming style, but the code will almost always contain a few typical functions. One of them compares two states and returns *true* if they are identical; this is how the program ascertains that the final state has been reached. Another function takes a given search state and applies to it all search operators, thus creating a complete set of "child states." To avoid infinite loops, a third function checks whether a state has already been investigated. A fourth calculates for a given state its distance from the final

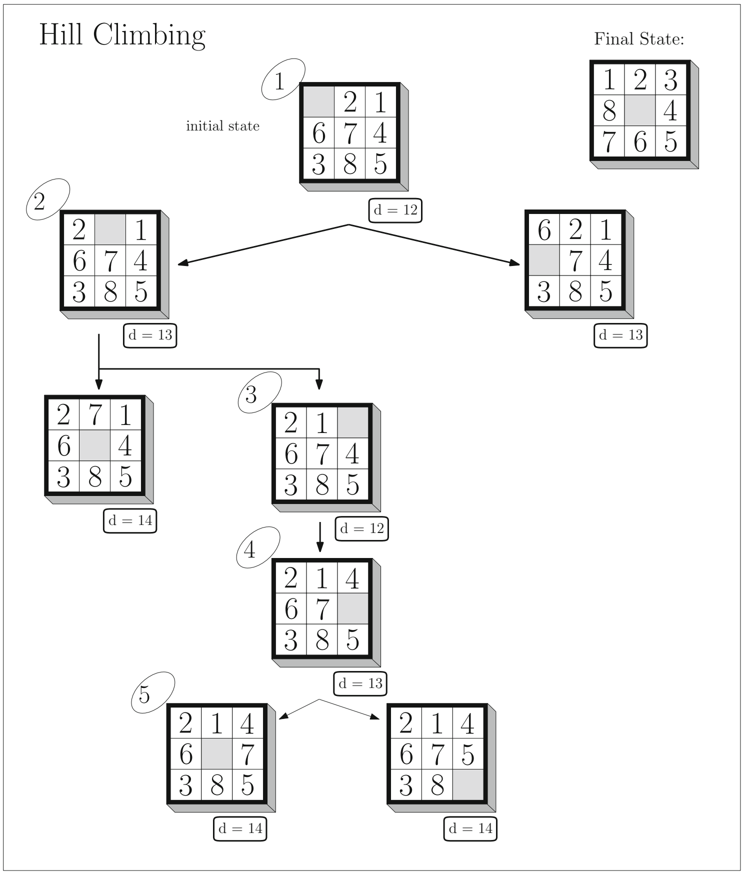

Fig. 1.3 Hill climbing. Circled integers indicate the order in which the search states are visited. *d* is a state's distance from the final state as calculated by the given evaluation function. Ties are broken randomly

state, and a fifth sorts the "child" states according to the distances thus calculated and places them at the front of the list *L*. And the last function checks if a termination criterion has been satisfied.[1]

One last observation: at some of the states in Fig. 1.3, no "child" offers any improvement over its "parent," a lower *d*-value being achieved only after temporary compromises. This is what a mountain climber may experience, too: sometimes, he has to traverse a valley before being able to resume the ascent. The mountain-climbing metaphor, by the way, is what gave this technique its name.

[1]For simplicity, the pseudocode ignores termination criteria other than reaching, or failing to reach, the final state.