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

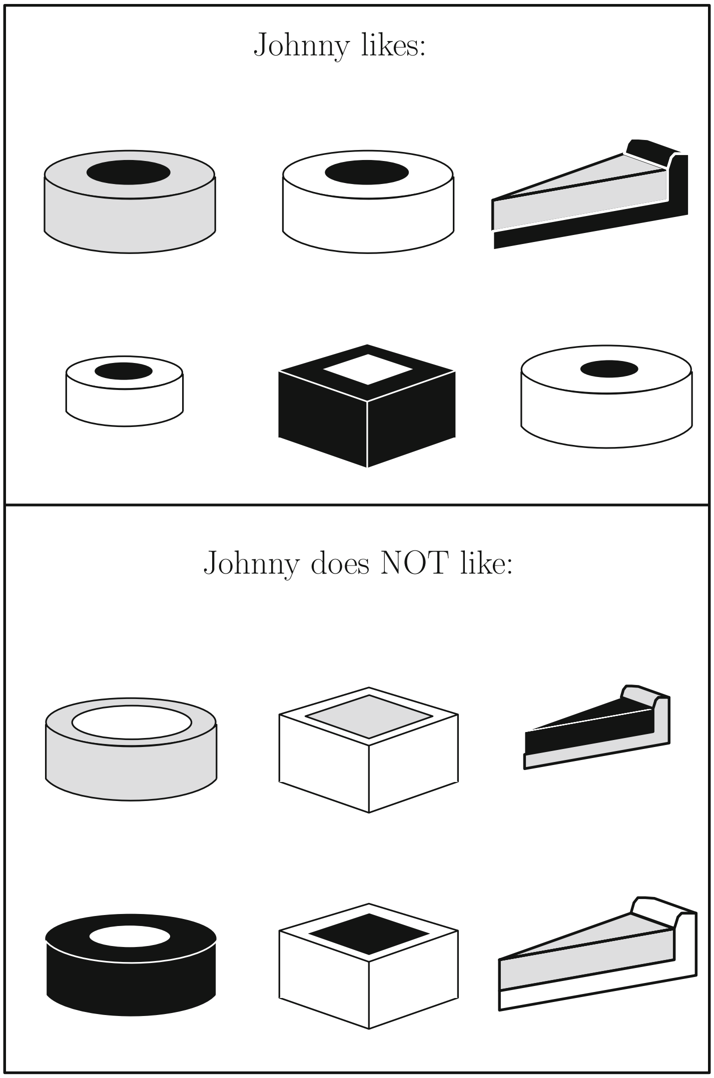

Fig. 1.1 A simple machine-learning task: induce a classifier capable of labeling future pies as positive and negative instances of "a pie that Johnny likes"

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.

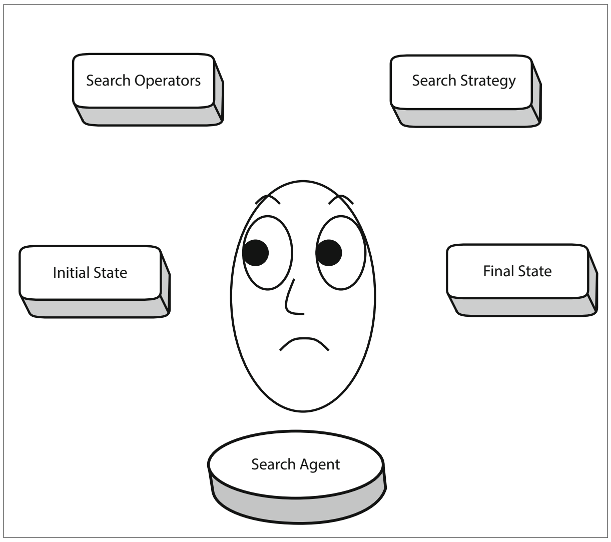

Fig. 1.2 A search problem is characterized by an initial state, final state, search operators, and a search strategy

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

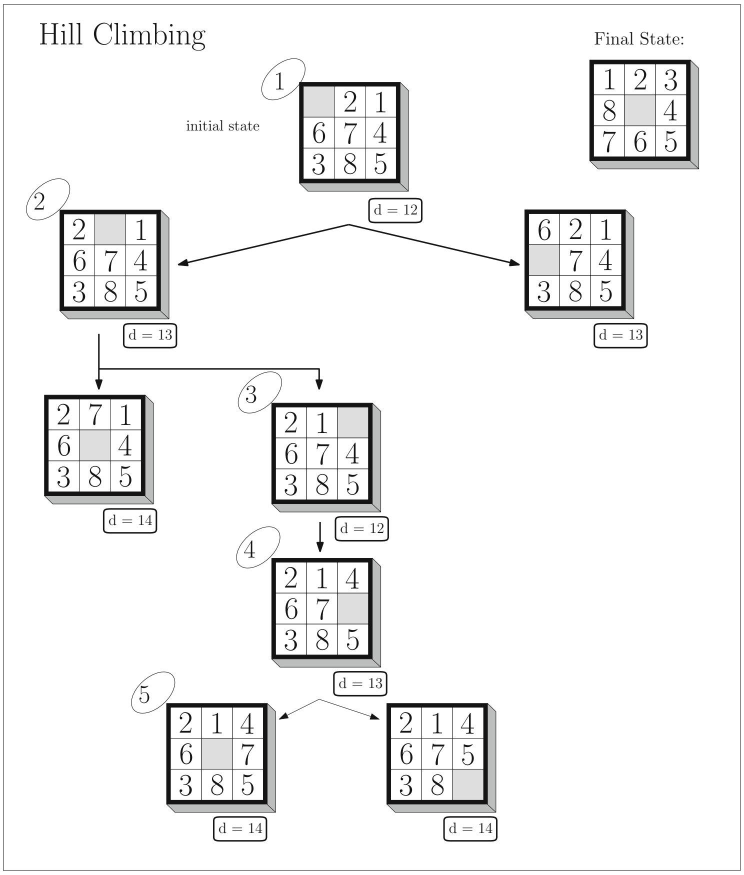

Fig. 1.3 Hill climbing. Circled integers indicate the order in which the search states are visited. *d* is a state's distance from the final state as calculated by the given evaluation function. Ties are broken randomly

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.

Table 1.2 Hill-climbing search algorithm

1.	Create two lists, L and L_{seen}. At the beginning, L contains only the initial state, and L_{seen} is empty.
2.	Let n be the first element of L. Compare this state with the final state. If they are identical, stop with success.
3.	Apply to n all available search operators, thus obtaining a set of new states. Discard those states that already exist in L_{seen}. As for the rest, sort them by the evaluation function and place them at the front of L.
4.	Transfer n from L into the list, L_{seen}, of the states that have been investigated.
5.	If $L = \emptyset$, stop and report failure. Otherwise, go to 2.

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.

- How does Artificial Intelligence define the search problem? What do we understand under the terms, “search space” and “search operators”?
- What is the role of the evaluation function? How does it affect the hill-climbing behavior?

1.3 Hill Climbing in Machine Learning

We are ready to explore the concrete ways of applying hill climbing to the needs of machine learning.

Hill Climbing and Johnny’s Pies Let us begin with the problem of how to decide which pies Johnny likes. The input consists of a set of training examples, each described by the available attributes. The output—the *final state*—is a boolean expression that is *true* for each positive example in the training set, and *false* for each negative example. The expression involves attribute-value pairs, logical operators (conjunction, disjunction, and negation), and such combination of parentheses as may be needed. The evaluation function measures the given expression’s error rate on the training set. For the *initial state*, any randomly generated expression can be used. In Fig. 1.4, we chose `(shape=circle)`, on the grounds that more than a half of the training examples are circular.

As for the *search operator*, one possibility is to add a conjunction as illustrated in the upper part of Fig. 1.4: for instance, the root’s leftmost child is obtained by replacing `(shape=circle)` with `[(shape=circle) AND (filling-shade=dark)]` (in the picture, logical AND is represented by the symbol “$\wedge$.”). Note how many different expressions this operator generates even in our toy domain. To `shape=circle`, any other attribute-value pair can be

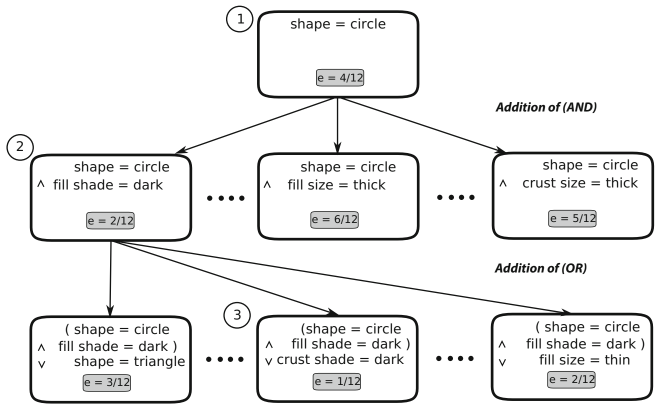
1
shape = circle
e = 4/12
Addition of (AND)
2
shape = circle ∧ fill shade = dark
e = 2/12
shape = circle ∧ fill size = thick
e = 6/12
shape = circle ∧ crust size = thick
e = 5/12
Addition of (OR)
(shape = circle ∧ fill shade = dark) ∨ shape = triangle
e = 3/12
3
(shape = circle ∧ fill shade = dark) ∨ crust shade = dark
e = 1/12
(shape = circle ∧ fill shade = dark) ∨ fill size = thin
e = 2/12

Fig. 1.4 Hill-climbing search in the “pies” domain

“ANDed.” Since the remaining four attributes (apart from `shape`) acquire 2, 3, 2, and 3 different values, respectively, the total number of terms that can be added to `(shape=circle)` is $2 \times 2 \times 3 = 36$.[2]

Alternatively, we may choose to add a disjunction, as illustrated (in the picture) by the three expansions of the leftmost child. Other operators may “remove a conjunct,” “remove a disjunct,” “add a negation,” “negate a term,” various ways of manipulating parentheses, and so on. All in all, hundreds of search operators can be applied to each state, and then again to the resulting states. This can be hard to manage even in this very simple domain.

Numeric Attributes In the “pies” domain, each attribute acquires one out of a few discrete values, but in realistic applications, some attributes will probably be numeric. For instance, each pie has a `price`, an attribute whose values come from a continuous domain. What will the search look like then?

To keep things simple, suppose there are only two attributes: `weight` and `price`. This limitation makes it possible, in Fig. 1.5, to represent each training example by a point in a plane. The reader can see that examples belonging to the same class tend to occupy a specific region, and curves separating individual regions can be defined—expressed mathematically as lines, circles, polynomials. For instance, the right part of Fig. 1.5 shows three different circles, each of which can act as a classifier: examples inside the circle are deemed positive; those outside, negative. Again, some of these classifiers are better than others. How will hill climbing go about finding the best ones? Here is one possibility.

[2]Of the 36 new states thus created, Fig. 1.4 shows only three.

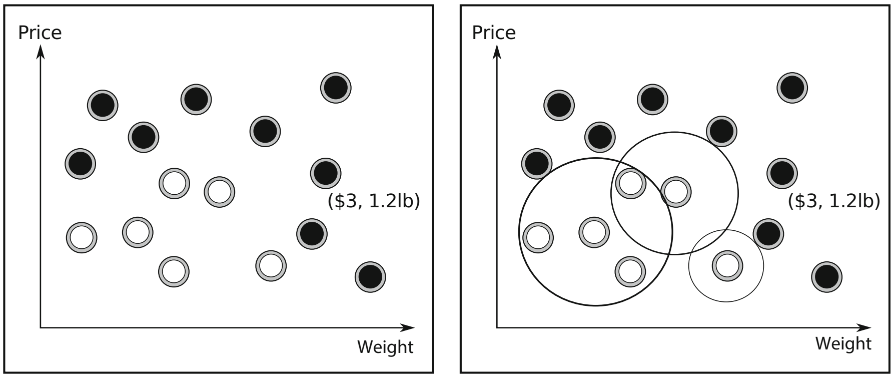

Fig. 1.5 *On the left*: a domain with continuous attributes; *on the right*: some "circular" classifiers

Hill Climbing in a Domain with Numeric Attributes

Initial State A circle is defined by its center and radius. We can identify the initial center with a randomly selected positive example, making the initial radius so small that the circle contains only this single example.

Search Operators Two search operators can be used: one increases the circle's radius, and the other shifts the center from one training example to another. In the former, we also have to determine *how much* the radius should change. One idea is to increase it only so much as to make the circle encompass one additional training example. At the beginning, only one training example is inside. After the first step, there will be two, then three, four, and so on.

Final State The circle may not be an ideal figure to represent the positive region. In this event, a 100% accuracy may not be achievable, and we may prefer to define the final state as, say, a "classifier that correctly classifies 95% of the training examples."

Evaluation Function As before, we choose to minimize the error rate.

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 aspects of search must be specified before we can employ hill climbing in machine learning?
- What search operators can be used in the "pies" domain and what in the "circles" domain? How can we define the evaluation function, the initial state, and the final state?

1.4 The Induced Classifier's Performance

So far, we have measured the error rate by comparing the training examples' known classes with those recommended by the classifier. Practically speaking, though, our goal is *not* to re-classify objects whose classes we already know; what we really want is to label *future examples*, those of whose classes we are as yet ignorant. The classifier's anticipated performance on these is estimated experimentally. It is important to know how.

Independent Testing Examples The simplest scenario will divide the available pre-classified examples into two parts: the training set, from which the classifier is induced, and the *testing set*, on which it is evaluated (Fig. 1.6). Thus in the "pies" domain, with its 12 pre-classified examples, the induction may be carried out on randomly selected eight, and the testing on the remaining four. If the classifier then "guesses" correctly the class of three testing examples (while going wrong on one), its performance is estimated as 75%.

Reasonable though this approach may appear, it suffers from a major drawback: a random choice of eight training examples may not be sufficiently representative of the underlying concept—and the same applies to the (even smaller) testing set. If we induce the meaning of a `mammal` from a training set consisting of a whale, a dolphin, and a platypus, the learner may be led to believe that mammals live in the sea (whale, dolphin), and sometimes lay eggs (platypus), hardly an opinion a biologist will embrace. And yet, another choice of trainingexamples may result in a

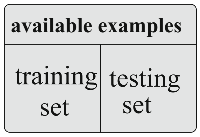

Fig. 1.6 Pre-classified examples are divided into the training and testing sets

classifier satisfying the highest standards. The point is, a different training/testing set division gives rise to a different classifier—and also to a different estimate of future performance. This is particularly serious if the number of pre-classified examples is small.

Suppose we want to compare two machine learning algorithms in terms of the quality of the products they induce. The problem of non-representative training sets can be mitigated by so-called *random subsampling*.[3] The idea is to repeat the random division into the training and testing sets several times, always inducing a classifier from the i-th training set, and then measuring the error rate, E_i, on the i-th testing set. The algorithm that delivers classifiers with the lower average value of E_i's is deemed better—as far as classification performance is concerned.

[3]Later, we will describe some other methodologies.

The Need for Explanations In some applications, establishing the class of each example is not enough. Just as desirable is to know the reasons behind the classification. Thus a patient is unlikely to give consent to amputation if the only argument in support of surgery is, “this is what our computer says.” But how to find a better explanation?

In the “pies” domain, a lot can be gleaned from the boolean expression itself. For instance, we may notice that a pie was labeled as negative whenever its shape was square, and its filling white. Combining this observation with alternative sources of knowledge may offer useful insights: the dark shade of the filling may indicate poppy, an ingredient Johnny is known to love; or the crust of circular pies turns out to be more crispy than that of square ones; and so on. The knowledge obtained in this manner can be more desirable than the classification itself.

By contrast, the classifier in the “circles” domain is a mathematical expression that acts as a “black box” which accepts an example’s description and returns the class label without telling us anything else. This is not necessarily a shortcoming. In some applications, an explanation is nothing more than a welcome bonus; in others, it is superfluous. Consider a classifier that accepts a digital image of a hand-written character and returns the letter it represents. The user who expects several pages of text to be converted into a Word document will hardly insist on a detailed explanation for each single character.

Existence of Alternative Solutions By the way, we should notice that *many* apparently perfect classifiers can be induced from the given data. In the “pies” domain, the training set contained 12 examples, and the classes of the remaining 96 examples were unknown. Using some simple combinatorics, we realize that there are 2^{96} classifiers that label correctly all training examples but differ in the way they label the unknown 96. One induced classifier may label correctly every single future example—and another will misclassify them all.

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.

- How can we estimate the error rate on examples that have not been seen during learning?
- Why is error rate usually higher on the testing set than on the training set?
- Give an example of a domain where the classifier also has to explain its action, and an example of a domain where this is unnecessary.
- What do we mean by saying that, “there is a combinatorial number of classifiers that correctly classify all training examples”?

1.5 Some Difficulties with Available Data

In some applications, the training set is created manually: an expert prepares the examples, tags them with class labels, chooses the attributes, and specifies the value of each attribute in each example. In other domains, the process is computerized. For instance, a company may want to be able to anticipate an employee's intention to leave. Their database contains, for each person, the address, gender, marital status, function, salary raises, promotions—as well as the information about whether the person is still with the company or, if not, the day they left. From this, a program can obtain the attribute vectors, labeled as positive if the given person left within a year since the last update of the database record.

Sometimes, the attribute vectors are automatically extracted from a database, and labeled by an expert. Alternatively, some examples can be obtained from a database, and others added manually. Often, two or more databases are combined. The number of such variations is virtually unlimited.

But whatever the source of the examples, they are likely to suffer from imperfections whose essence and consequences the engineer has to understand.

Irrelevant Attributes To begin with, some attributes are important, while others are not. While Johnny may be truly fond of poppy filling, his preference for a pie will hardly be driven by the cook's shoe size. This is something to be concerned about: *irrelevant* attributes add to computational costs; they can even mislead the learner. Can they be avoided?

Usually not. True, in manually created domains, the expert is supposed to know which attributes really matter, but even here, things are not so simple. Thus the author of the "pies" domain might have done her best to choose those attributes she believed to matter. But unsure about the real reasons behind Johnny's tastes, she may have included attributes whose necessity she suspected—but could not guarantee. Even more often the problems with relevance occur when the examples are extracted from a database. Databases are developed primarily with the intention to provide access to lots of information—of which usually only a tiny part pertains to the learning task. As to which part this is, we usually have no idea.

Missing Attributes Conversely, some critical attributes can be missing. Mindful of his parents' finances, Johnny may be prejudiced against expensive pies. The absence of attribute `price` will then make it impossible to induce a good classifier: two examples, identical in terms of the available attributes, can differ in the values of the vital "missing" attribute. No wonder that, though identically described, one example is positive, and the other is negative. When this happens, we say that the training set is *inconsistent*. The situation is sometimes difficult to avoid: not only may the expert be ignorant of the relevance of attribute `price`; it may be impossible to provide this attribute's values, and the attribute thus cannot be used anyway.

Redundant Attributes Somewhat less damaging are attributes that are *redundant* in the sense that their values can be obtained from other attributes. If the database contains a patient's `date-of-birth` as well as `age`, the latter is unnecessary because it can be calculated by subtracting `date-of-birth` from today's date. Fortunately, redundant attributes are less dangerous than irrelevant or missing ones.

Missing Attribute Values In some applications, the user has no problems identifying the right choice of attributes. The problem is, however, that the value of some attributes are not known. For instance, the company analyzing the database of its employees may not know, for each person, the number of children.

Attribute: Value Noise Attribute values and class labels often cannot be trusted on account of unreliable sources of information, poor measurement devices, typos, the user's confusion, and many other reasons. We say that the data suffer from various kinds of *noise*.

Stochastic noise is random. For instance, since our body-weight varies during the day, the reading we get in the morning is different from the one in the evening. A human error can also play a part: lacking the time to take a patient's blood pressure, a negligent nurse simply scribbles down a modification of the previous reading. By contrast, *systematic noise* drags all values in the same direction. For instance, a poorly calibrated thermometer always gives a lower reading than it should. And something different occurs in the case of *arbitrary artifacts*; here, the given value bears no relation to reality such as when an EEG electrode gets loose and, from that moment on, all subsequent readings will be zero.

Class-Label Noise Class labels suffer from similar problems as attributes. The labels recommended by an expert may not have been properly recorded; alternatively, some examples find themselves in a "gray area" between two classes, in which event the correct labels are not certain. Both cases represent stochastic noise, of which the latter may affect negatively only examples from the borderline region between the two classes. However, class-label noise can also be systematic: a physician may be reluctant to diagnose a rare disease unless the evidence is overwhelming—his class labels are then more likely to be negative than positive. Finally, arbitrary artifacts in class labels are encountered in domains where the classes are supplied by an automated process that has gone wrong.

Class-label noise can be more dangerous than attribute-value noise. Thus in the "circles" domain, an example located deep inside the positive region will stay there even if an attribute's value is slightly modified; only the borderline example will suffer from being "sent across the border." By contrast, class-label noise will invalidate *any* example.

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.

- Explain the following types of attributes: irrelevant, redundant, and missing. Illustrate each of them using the "pies" domain.

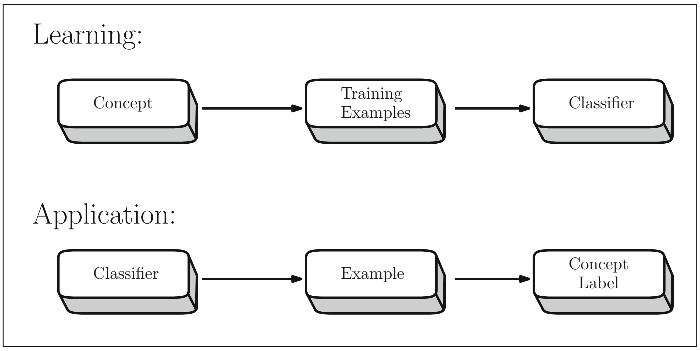

Fig. 1.7 The training examples are used to induce a classifier. The classifier is then employed to classify future examples

- What is meant by "inconsistent training set"? What can be the cause? How can it affect the learning process?
- What kinds of noise do we know? What are their possible sources?

1.6 Summary and Historical Remarks

- Induction from a training set of pre-classified examples is the most deeply studied machine-learning task.
- Historically, the task is cast as search. One can propose a mechanism that exploits the well-established search technique of hill climbing defined by an initial state, final state, interim states, search operators, and evaluation functions.
- Mechanical use of search is not the ultimate solution, though. The rest of the book will explore more useful techniques.
- Classifier performance is estimated with the help of pre-classified testing data. The simplest performance criterion is error rate, the percentage of examples misclassified by the classifier. The baseline scenario is shown in Fig. 1.7.
- Two classifiers that both correctly classify all training examples may differ significantly in their handling of the testing set.
- Apart from low error rate, some applications require that the classifier provides the reasons behind the classification.
- The quality of the induced classifier depends on training examples. The quality of the training examples depends not only on their choice, but also on the attributes used to describe them. Some attributes are relevant, others irrelevant or redundant. Quite often, critical attributes are missing.
- The attribute values and class labels may suffer from stochastic noise, systematic noise, and random artefacts. The value of an attribute in a concrete example may not be known.

Historical Remarks The idea of casting the machine-learning task as search was popular in the 1980s and 1990s. While several "founding fathers" came to see things this way independently of each other, Mitchell [67] is often credited with being the first to promote the search-based approach; just as influential, however, was the family of AQ-algorithms proposed by Michalski [59]. The discipline got a major boost by the collection of papers edited by Michalski et al. [61]. They framed the mindset of a whole generation.

There is much more to search algorithms. The interested reader is referred to textbooks of Artificial Intelligence, of which perhaps the most comprehensive is Russell and Norvig [84] or Coppin [17].

The reader may find it interesting that the question of proper representation of concepts or classes intrigued philosophers for centuries. Thus John Stuart Mill [65] explored concepts that are related towhat the next chapter calls *probabilistic*

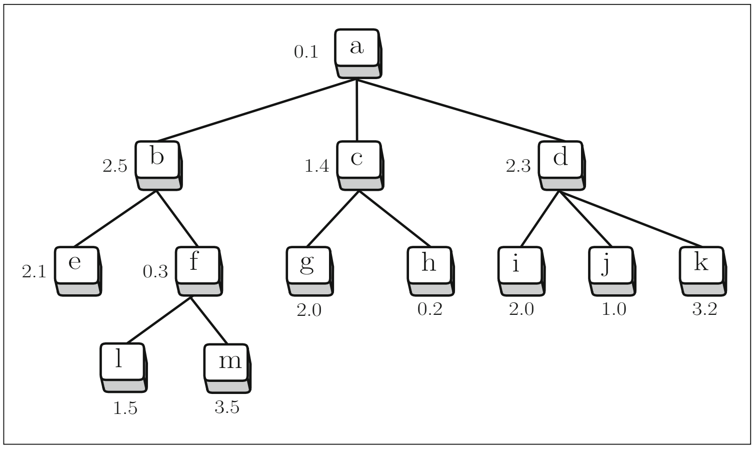

Fig. 1.8 Determine the order in which these search states are visited by heuristic search algorithms. The numbers next to the "boxes" give the values of the evaluation function for the individual search states

representation; and William Whewel [96] advocated *prototypical* representations that are close to the subject of our Chap. 3.

1.7 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. In the sliding-tiles puzzle, suggest a better evaluation function than the one used in the text.
2. Figure 1.8 shows a search tree where each node represents one search state and is tagged with the value of the evaluation function. In what order will these states be visited by hill-climbing search?
3. Suppose the evaluation function in the "pies" domain calculates the percentage of correctly classified training examples. Let the initial state be the expression describing the second positive example in Table 1.1. Hand-simulate the hill-climbing search that uses generalization and specialization operators.
4. What is the size of the instance space in a domain where examples are described by ten boolean attributes? How large is then the space of classifiers?

Give It Some Thought

1. In the "pies" domain, the size of the space of all classifiers is 2^{108}, provided that each subset of the instance space can be represented by a distinct classifier. How much will the search space shrink if we permit only classifiers in the form of conjunctions of attribute-value pairs?
2. What kind of noise can you think of in the "pies" domain? What can be the source of this noise? What other issues may render training sets of this kind less than perfect?
3. Some classifiers behave as black boxes that do not offer much in the way of explanations. This, for instance, was the case of the "circles" domain. Suggest examples of domains where black-box classifiers are impractical, and suggest domains where this limitation does not matter.
4. Consider the data-related difficulties summarized in Sect. 1.5. Which of them are really serious, and which can perhaps be tolerated?
5. What is the difference between redundant attributes and irrelevant attributes?
6. Take a class that you think is difficult to describe—for instance, the recognition of a complex biological object (oak tree, ostrich, etc.) or the recognition of a music genre (rock, folk, jazz, etc.). Suggest the list of attributes to describe the training examples. Are the values of these attributes easy to obtain? Which of the problems discussed in this chapter do you expect will complicate the learning process?

Computer Assignments

1. Write a program implementing hill climbing and apply it to the sliding-tiles puzzle. Choose appropriate representation for the search states, write a module that decides whether a state is a final state, and implement the search operators. Define two or three alternative evaluation functions and observe how each of them leads to a different sequence of search steps.
2. Write a program that will implement the "growing circles" algorithm from Sect. 1.3. Create a training set of two-dimensional examples such as those in Fig. 1.5. The learning program will use the hill-climbing search. The evaluation function will calculate the percentage of training examples correctly classified by the classifier. Consider the following search operators: (1) increase/decrease the radius of the circle, (2) use a different training example as the circle's center.
3. Write a program that will implement the search for the description of the "pies that Johnny likes." Define your own generalization and specialization operators. The evaluation function will rely on the error rate observed on the training examples.