Overfitting

Often there is a **bias–variance tradeoff**: a choice between more complex, low-bias hypotheses that fit the training data well and simpler, low-variance hypotheses that may generalize better. Albert Einstein said in 1933, "the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience." In other words, Einstein recommends choosing the simplest hypothesis that matches the data. This principle can be traced further back to the 14th-century English philosopher William of Ockham.[2] His principle that "plurality [of entities] should not be posited without necessity" is called **Ockham's razor** because it is used to "shave off" dubious explanations.

2 The name is often misspelled as "Occam."

Bias–variance tradeoff

Defining simplicity is not easy. It seems clear that a polynomial with only two parameters is simpler than one with thirteen parameters. We will make this intuition more precise in Section 19.3.4. However, in Chapter 21 we will see that deep neural network models can often generalize quite well, even though they are very complex—some of them have billions of parameters. So the number of parameters by itself is not a good measure of a model's fitness. Perhaps we should be aiming for "appropriateness," not "simplicity" in a model class. We will consider this issue in Section 19.4.1.

Which hypothesis is best in Figure 19.1? We can't be certain. If we knew the data represented, say, the number of hits to a Web site that grows from day to day, but also cycles depending on the time of day, then we might favor the sinusoidal function. If we knew the data was definitely not cyclic but had high noise, that would favor the linear function.

In some cases, an analyst is willing to say not just that a hypothesis is possible or impossible, but rather how probable it is. Supervised learning can be done by choosing the hypothesis h^* that is most probable given the data:

$$h^* = \underset{h \in H}{\operatorname{argmax}} P\left(h | data\right).$$

By Bayes' rule this is equivalent to

$$h^* = \underset{h \in H}{\operatorname{argmax}} P(data \middle| h)\, P(h).$$

Then we can say that the prior probability $P(h)$ is high for a smooth degree-1 or -2 polynomial and lower for a degree-12 polynomial with large, sharp spikes. We allow unusual-looking functions when the data say we really need them, but we discourage them by giving them a low prior probability.

Why not let H be the class of all computer programs, or all Turing machines? The problem is that *there is a tradeoff between the expressiveness of a hypothesis space and the computational complexity of finding a good hypothesis within that space.* For example, fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder; and fitting Turing machines is undecidable. A second reason to prefer simple hypothesis spaces is that presumably we will want to use h after we have learned it, and computing $h(x)$ when h is a linear function is guaranteed to be fast, while computing an arbitrary Turing machine program is not even guaranteed to terminate.

For these reasons, most work on learning has focused on simple representations. In recent years there has been great interest in deep learning (Chapter 21), where representations are not simple but where the $h(x)$ computation still takes only a *bounded number of steps* to compute with appropriate hardware.

We will see that the expressiveness–complexity tradeoff is not simple: it is often the case, as we saw with first-order logic in Chapter 8, that an expressive language makes it possible for a *simple* hypothesis to fit the data, whereas restricting the expressiveness of the language means that any consistent hypothesis must be complex.

19.2.1 Example problem: Restaurant waiting

We will describe a sample supervised learning problem in detail: the problem of deciding whether to wait for a table at a restaurant. This problem will be used throughout the chapter to demonstrate different model classes. For this problem the output, y, is a Boolean variable that we will call *WillWait*; it is true for examples where we do wait for a table. The input, x, is a vector of ten attribute values, each of which has discrete values:

1. **ALTERNATE:** whether there is a suitable alternative restaurant nearby.
2. **BAR:** whether the restaurant has a comfortable bar area to wait in.
3. **FRI/SAT:** true on Fridays and Saturdays.
4. **HUNGRY:** whether we are hungry right now.
5. **PATRONS:** how many people are in the restaurant (values are *None*, *Some*, and *Full*).
6. **PRICE:** the restaurant's price range ($, $$, $$$).
7. **RAINING:** whether it is raining outside.
8. **RESERVATION:** whether we made a reservation.
9. **TYPE:** the kind of restaurant (French, Italian, Thai, or burger).
10. **WAITESTIMATE:** host's wait estimate: $0-10$, $10-30$, $30-60$, or >60 minutes.

A set of 12 examples, taken from the experience of one of us (SR), is shown in Figure 19.2. Note how skimpy these data are: there are $2^6 \times 3^2 \times 4^2 = 9,216$ possible combinations of values for the input attributes, but we are given the correct output for only 12 of them; each of the other 9,204 could be either true or false; we don't know. This is the essence of induction: we need to make our best guess at these missing 9,204 output values, given only the evidence of the 12 examples.

Figure 19.2

Example	Input Attributes										Output
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

Examples for the restaurant domain.

19.3 Learning Decision Trees

A **decision tree** is a representation of a function that maps a vector of attribute values to a single output value—a "decision." A decision tree reaches its decision by performing a sequence of tests, starting at the root and following the appropriate branch until a leaf is reached. Each internal node in the tree corresponds to a test of the value of one of the input attributes, the branches from the node are labeled with the possible values of the attribute, and the leaf nodes specify what value is to be returned by the function.

Decision tree

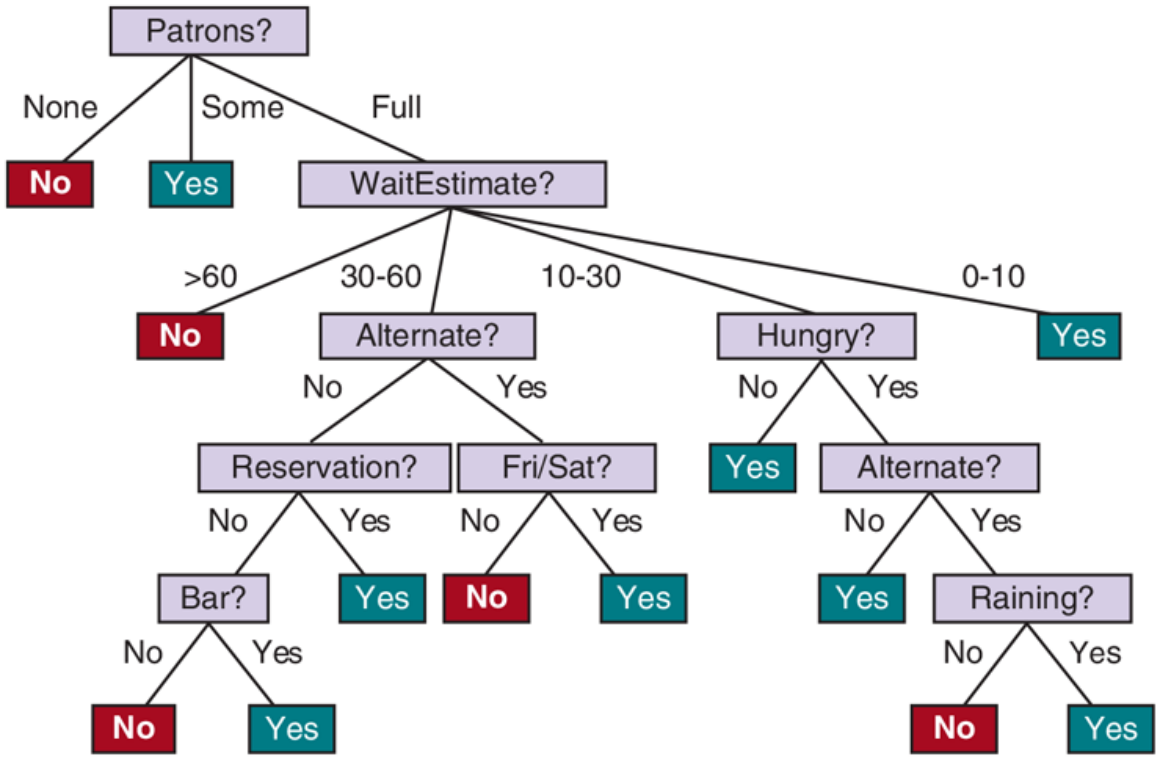
In general, the input and output values can be discrete or continuous, but for now we will consider only inputs consisting of discrete values and outputs that are either *true* (a **positive** example) or *false* (a **negative** example). We call this **Boolean classification**. We will use j to index the examples ($\mathbf{x}_j$ is the input vector for the jth example and y_j is the output), and $x_{j,i}$ for the ith attribute of the jth example.

Positive

Negative

The tree representing the decision function that SR uses for the restaurant problem is shown in Figure 19.3. Following the branches, we see that an example with $Patrons = Full$ and $WaitEstimate =$ 0–10 will be classified as positive (i.e., yes, we will wait for a table).

Figure 19.3



A decision tree for deciding whether to wait for a table.

19.3.1 Expressiveness of decision trees

A Boolean decision tree is equivalent to a logical statement of the form:

$$Output \quad \Leftrightarrow \quad (Path_1 \vee Path_2 \vee \cdots),$$

where each $Path_i$ is a conjunction of the form $(A_m = v_x \wedge A_n = v_y \wedge \cdots)$ of attribute-value tests corresponding to a path from the root to a *true* leaf. Thus, the whole expression is in disjunctive normal form, which means that any function in propositional logic can be expressed as a decision tree.

For many problems, the decision tree format yields a nice, concise, understandable result. Indeed, many "How To" manuals (e.g., for car repair) are written as decision trees. But some functions cannot be represented concisely. For example, the majority function, which returns true if and only if more than half of the inputs are true, requires an exponentially large decision tree, as does the parity function, which returns true if and only if an even number of input attributes are true. With real-valued attributes, the function $y > A_1 + A_2$ is

hard to represent with a decision tree because the decision boundary is a diagonal line, and all decision tree tests divide the space up into rectangular, axis-aligned boxes. We would have to stack a lot of boxes to closely approximate the diagonal line. In other words, decision trees are good for some kinds of functions and bad for others.

Is there *any* kind of representation that is efficient for *all* kinds of functions? Unfortunately, the answer is no—there are just too many functions to be able to represent them all with a small number of bits. Even just considering Boolean functions with n Boolean attributes, the truth table will have 2^n rows, and each row can output *true* or *false*, so there are 2^{2^n} different functions. With 20 attributes there are $2^{1,048,576} \approx 10^{300,000}$ functions, so if we limit ourselves to a million-bit representation, we can't represent all these functions.

19.3.2 Learning decision trees from examples

We want to find a tree that is consistent with the examples in Figure 19.2 and is as small as possible. Unfortunately, it is intractable to find a guaranteed smallest consistent tree. But with some simple heuristics, we can efficiently find one that is close to the smallest. The Learn-Decision-Tree algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first, then recursively solve the smaller subproblems that are defined by the possible results of the test. By "most important attribute," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be shallow.

Figure 19.4(a) shows that *Type* is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive as negative examples. On the other hand, in (b) we see that *Patrons* is a fairly important attribute, because if the value is *None* or *Some*, then we are left with example sets for which we can answer definitively (*No* and *Yes*, respectively). If the value is *Full*, we are left with a mixed set of examples. There are four cases to consider for these recursive subproblems:

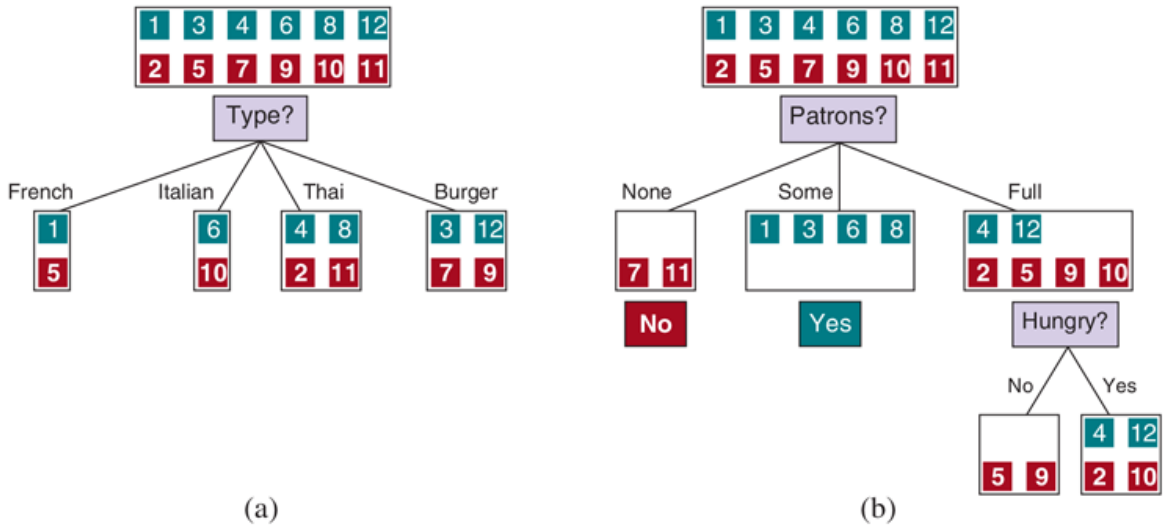
1. If the remaining examples are all positive (or all negative), then we are done: we can answer *Yes* or *No*. Figure 19.4(b) shows examples of this happening in the *None* and *Some* branches.
2. If there are some positive and some negative examples, then choose the best attribute to split them. Figure 19.4(b) shows *Hungry* being used to split the

remaining examples.

3. If there are no examples left, it means that no example has been observed for this combination of attribute values, and we return the most common output value from the set of examples that were used in constructing the node's parent.
4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can happen because there is an error or **noise** in the data; because the domain is nondeterministic; or because we can't observe an attribute that would distinguish the examples. The best we can do is return the most common output value of the remaining examples.

Noise

Figure 19.4



Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

The LEARN-DECISION-TREE algorithm is shown in Figure 19.5. Note that the set of examples is an input to the algorithm, but nowhere do the examples appear in the tree returned by the algorithm. A tree consists of tests on attributes in the interior nodes, values of attributes on

the branches, and output values on the leaf nodes. The details of the IMPORTANCE function are given in Section 19.3.3. The output of the learning algorithm on our sample training set is shown in Figure 19.6. The tree is clearly different from the original tree shown in Figure 19.3. One might conclude that the learning algorithm is not doing a very good job of learning the correct function. This would be the wrong conclusion to draw, however. The learning algorithm looks at the *examples*, not at the correct function, and in fact, its hypothesis (see Figure 19.6) not only is consistent with all the examples, but is considerably simpler than the original tree! With slightly different examples the tree might be very different, but the function it represents would be similar.

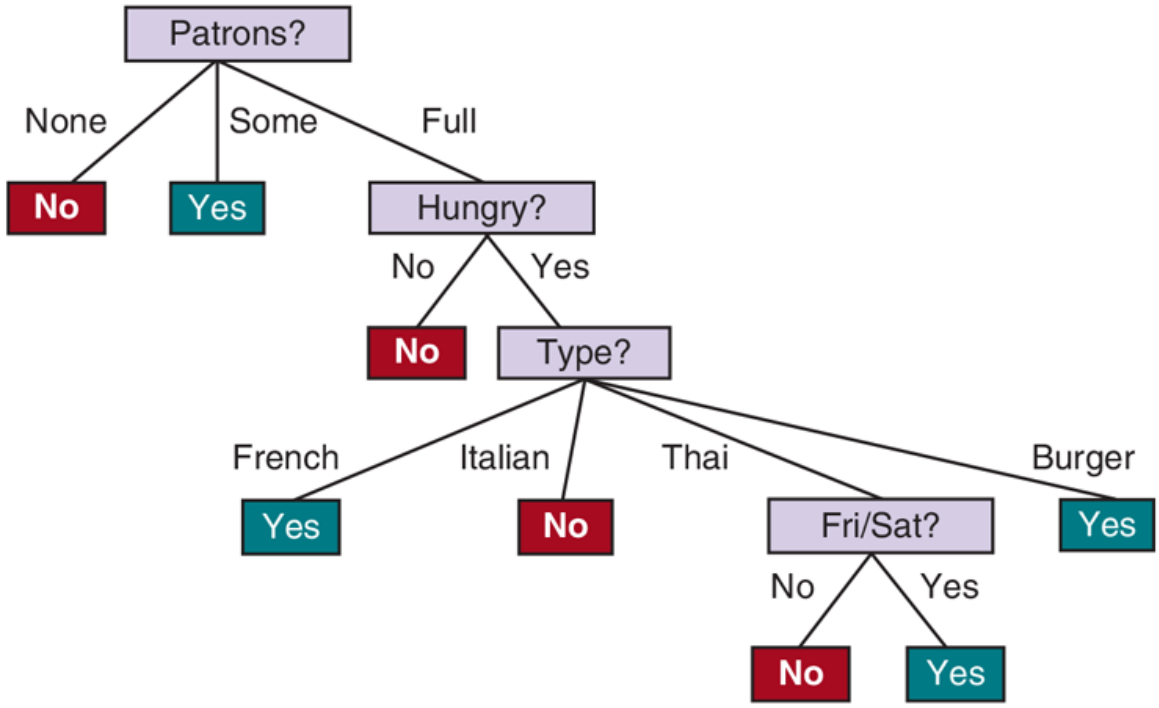
Figure 19.5

function LEARN-DECISION-TREE(*examples*, *attributes*, *parent_examples*) **returns** a tree

if *examples* is empty **then return** PLURALITY-VALUE(*parent_examples*)
else if all *examples* have the same classification **then return** the classification
else if *attributes* is empty **then return** PLURALITY-VALUE(*examples*)
else
  $A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE$(a, examples)$
  tree $\leftarrow$ a new decision tree with root test A
  for each value v of A **do**
    $exs \leftarrow \{e : e \in examples \text{ and } e.A = v\}$
    subtree $\leftarrow$ LEARN-DECISION-TREE$(exs, attributes - A, examples)$
    add a branch to *tree* with label $(A = v)$ and subtree *subtree*
  return *tree*

The decision tree learning algorithm. The function IMPORTANCE is described in Section 19.3.3. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

Figure 19.6

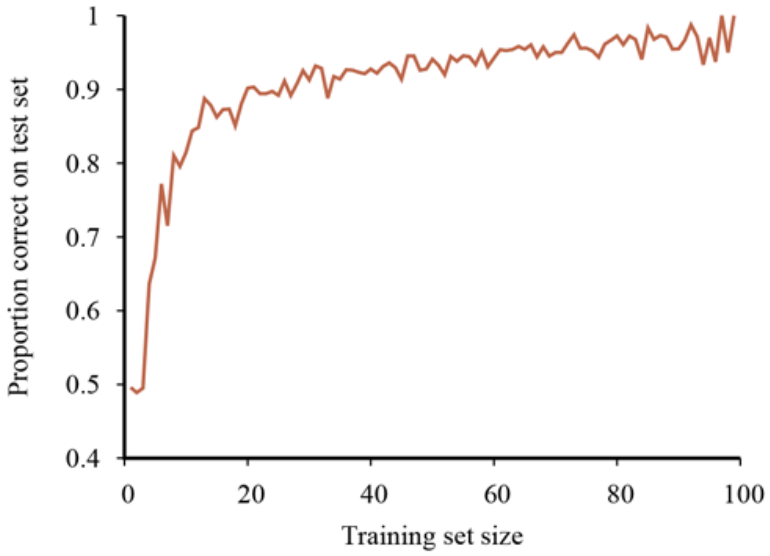


The decision tree induced from the 12-example training set.

The learning algorithm has no reason to include tests for *Raining* and *Reservation*, because it can classify all the examples without them. It has also detected an interesting and previously unsuspected pattern: SR will wait for Thai food on weekends. It is also bound to make some mistakes for cases where it has seen no examples. For example, it has never seen a case where the wait is 0–10 minutes but the restaurant is full. In that case it says not to wait when *Hungry* is false, but SR would certainly wait. With more training examples the learning program could correct this mistake.

We can evaluate the performance of a learning algorithm with a **learning curve**, as shown in Figure 19.7. For this figure we have 100 examples at our disposal, which we split randomly into a training set and a test set. We learn a hypothesis h with the training set and measure its accuracy with the test set. We can do this starting with a training set of size 1 and increasing one at a time up to size 99. For each size, we actually repeat the process of randomly splitting into training and test sets 20 times, and average the results of the 20 trials. The curve shows that as the training set size grows, the accuracy increases. (For this reason, learning curves are also called **happy graphs**.) In this graph we reach 95% accuracy, and it looks as if the curve might continue to increase if we had more data.

Figure 19.7



A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

Learning curve

Happy graphs

19.3.3 Choosing attribute tests

The decision tree learning algorithm chooses the attribute with the highest IMPORTANCE. We will now show how to measure importance, using the notion of information gain, which is defined in terms of **entropy**, which is the fundamental quantity in information theory (Shannon and Weaver, 1949).

Entropy

Entropy is a measure of the uncertainty of a random variable; the more information, the less entropy. A random variable with only one possible value—a coin that always comes up heads—has no uncertainty and thus its entropy is defined as zero. A fair coin is equally likely to come up heads or tails when flipped, and we will soon show that this counts as "1 bit" of entropy. The roll of a fair *four*-sided die has 2 bits of entropy, because there are 2^2 equally probable choices. Now consider an unfair coin that comes up heads 99% of the time. Intuitively, this coin has less uncertainty than the fair coin—if we guess heads we'll be wrong only 1% of the time—so we would like it to have an entropy measure that is close to zero, but positive. In general, the entropy of a random variable V with values v_k having probability $P(v_k)$ is defined as

$$\text{Entropy:} \quad H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k).$$

We can check that the entropy of a fair coin flip is indeed 1 bit:

$$H(Fair) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1 \ .$$

And of a four-sided die is 2 bits:

$$H(Die4) = -(0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25) = 2$$

For the loaded coin with 99% heads, we get

$$H(Loaded) = -(0.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits.}$$

It will help to define $B(q)$ as the entropy of a Boolean random variable that is true with probability q:

$$B(q) = -(q \log_2 q + (1-q) \log_2 (1-q)).$$

Thus, $H(Loaded) = B(0.99) \approx 0.08$. Now let's get back to decision tree learning. If a training set contains p positive examples and n negative examples, then the entropy of the output variable on the whole set is

$$H(Output) = B\left(\frac{p}{p+n}\right).$$

The restaurant training set in Figure 19.2 has $p = n = 6$, so the corresponding entropy is $B(0.5)$ or exactly 1 bit. The result of a test on an attribute A will give us some information, thus reducing the overall entropy by some amount. We can measure this reduction by looking at the entropy remaining after the attribute test.

An attribute A with d distinct values divides the training set E into subsets $E_1, \ldots, E_d$. Each subset E_k has p_k positive examples and n_k negative examples, so if we go along that branch, we will need an additional $B(p_k/(p_k + n_k))$ bits of information to answer the question. A randomly chosen example from the training set has the kth value for the attribute (i.e., is in E_k with probability $(p_k + n_k)/(p + n)$), so the expected entropy remaining after testing attribute A is

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right).$$

The **information gain** from the attribute test on A is the expected reduction in entropy:

$$Gain(A) = B\left(\frac{p}{p + n}\right) - Remainder(A).$$

Information gain

In fact $Gain(A)$ is just what we need to implement the IMPORTANCE function. Returning to the attributes considered in Figure 19.4, we have

$$Gain(Patrons) = 1 - \left[\tfrac{2}{12} B\left(\tfrac{0}{2}\right) + \tfrac{4}{12} B\left(\tfrac{4}{4}\right) + \tfrac{6}{12} B\left(\tfrac{2}{6}\right)\right] \approx 0.541 \text{ bits},$$
$$Gain(Type) = 1 - \left[\tfrac{2}{12} B\left(\tfrac{1}{2}\right) + \tfrac{2}{12} B\left(\tfrac{1}{2}\right) + \tfrac{4}{12} B\left(\tfrac{2}{4}\right) + \tfrac{4}{12} B\left(\tfrac{2}{4}\right)\right] = 0 \text{ bits},$$

confirming our intuition that *Patrons* is a better attribute to split on first. In fact, *Patrons* has the maximum information gain of any of the attributes and thus would be chosen by the decision tree learning algorithm as the root.

19.3.4 Generalization and overfitting

We want our learning algorithms to find a hypothesis that fits the training data, but more importantly, we want it to generalize well for previously unseen data. In Figure 19.1 we saw that a high-degree polynomial can fit all the data, but has wild swings that are not warranted by the data: it fits but can overfit. Overfitting becomes more likely as the number of attributes grows, and less likely as we increase the number of training examples. Larger hypothesis spaces (e.g., decision trees with more nodes or polynomials with high degree) have more capacity both to fit and to overfit; some model classes are more prone to overfitting than others.

For decision trees, a technique called **decision tree pruning** combats overfitting. Pruning works by eliminating nodes that are not clearly relevant. We start with a full tree, as generated by LEARN-DECISION-TREE. We then look at a test node that has only leaf nodes as descendants. If the test appears to be irrelevant—detecting only noise in the data—then we eliminate the test, replacing it with a leaf node. We repeat this process, considering each test with only leaf descendants, until each one has either been pruned or accepted as is.

Decision tree pruning

The question is, how do we detect that a node is testing an irrelevant attribute? Suppose we are at a node consisting of p positive and n negative examples. If the attribute is irrelevant, we would expect that it would split the examples into subsets such that each subset has roughly the same proportion of positive examples as the whole set, $p/(p+n)$, and so the information gain will be close to zero.[3] Thus, a low information gain is a good clue that the attribute is irrelevant. Now the question is, how large a gain should we require in order to split on a particular attribute?

3 The gain will be strictly positive except for the unlikely case where all the proportions are *exactly* the same. (See Exercise 19.NNGA.)

We can answer this question by using a statistical **significance test**. Such a test begins by assuming that there is no underlying pattern (the so-called **null hypothesis**). Then the actual data are analyzed to calculate the extent to which they deviate from a perfect absence of pattern. If the degree of deviation is statistically unlikely (usually taken to mean a 5% probability or less), then that is considered to be good evidence for the presence of a

significant pattern in the data. The probabilities are calculated from standard distributions of the amount of deviation one would expect to see in random sampling.

Significance test

Null hypothesis

In this case, the null hypothesis is that the attribute is irrelevant and, hence, that the information gain for an infinitely large sample would be zero. We need to calculate the probability that, under the null hypothesis, a sample of size $v = n + p$ would exhibit the observed deviation from the expected distribution of positive and negative examples. We can measure the deviation by comparing the actual numbers of positive and negative examples in each subset, p_k and n_k, with the expected numbers, $\hat{p}_k$ and $\hat{n}_k$, assuming true irrelevance:

$$\hat{p}_k = p \times \frac{p_k + n_k}{p + n} \qquad\qquad \hat{n}_k = n \times \frac{p_k + n_k}{p + n}.$$

A convenient measure of the total deviation is given by

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k}.$$

Under the null hypothesis, the value of Δ is distributed according to the χ^2 (chi-squared) distribution with $d - 1$ degrees of freedom. We can use a χ^2 statistics function to see if a particular Δ value confirms or rejects the null hypothesis. For example, consider the restaurant *Type* attribute, with four values and thus three degrees of freedom. A value of $\Delta = 7.82$ or more would reject the null hypothesis at the 5% level (and a value of $\Delta = 11.35$ or more would reject at the 1% level). Values below that lead to accepting the hypothesis that the attribute is irrelevant, and thus the associated branch of the tree should be pruned away. This is known as χ^2 **pruning**.

χ^2 pruning

With pruning, noise in the examples can be tolerated. Errors in the example's label (e.g., an example $(\mathbf{x}, Yes)$ that should be $(\mathbf{x}, No)$) give a linear increase in prediction error, whereas errors in the descriptions of examples (e.g., $Price = \$$ when it was actually $Price = \$\$$) have an asymptotic effect that gets worse as the tree shrinks down to smaller sets. Pruned trees perform significantly better than unpruned trees when the data contain a large amount of noise. Also, the pruned trees are often much smaller and hence easier to understand and more efficient to execute.

One final warning: You might think that χ^2 pruning and information gain look similar, so why not combine them using an approach called **early stopping**—have the decision tree algorithm stop generating nodes when there is no good attribute to split on, rather than going to all the trouble of generating nodes and then pruning them away. The problem with early stopping is that it stops us from recognizing situations where there is no one good attribute, but there are combinations of attributes that are informative. For example, consider the XOR function of two binary attributes. If there are roughly equal numbers of examples for all four combinations of input values, then neither attribute will be informative, yet the correct thing to do is to split on one of the attributes (it doesn't matter which one), and then at the second level we will get splits that are very informative. Early stopping would miss this, but generate-and-then-prune handles it correctly.

Early stopping

19.3.5 Broadening the applicability of decision trees

Decision trees can be made more widely useful by handling the following complications:

- **MISSING DATA:** In many domains, not all the attribute values will be known for every example. The values might have gone unrecorded, or they might be too expensive to obtain. This gives rise to two problems: First, given a complete decision tree, how

should one classify an example that is missing one of the test attributes? Second, how should one modify the information-gain formula when some examples have unknown values for the attribute? These questions are addressed in Exercise 19.MISS.

- **CONTINUOUS AND MULTIVALUED INPUT ATTRIBUTES:** For continuous attributes like *Height*, *Weight*, or *Time*, it may be that every example has a different attribute value. The information gain measure would give its highest score to such an attribute, giving us a shallow tree with this attribute at the root, and single-example subtrees for each possible value below it. But that doesn't help when we get a new example to classify with an attribute value that we haven't seen before.

Split point

A better way to deal with continuous values is a **split point** test—an inequality test on the value of an attribute. For example, at a given node in the tree, it might be the case that testing on $Weight > 160$ gives the most information. Efficient methods exist for finding good split points: start by sorting the values of the attribute, and then consider only split points that are between two examples in sorted order that have different classifications, while keeping track of the running totals of positive and negative examples on each side of the split point. Splitting is the most expensive part of real-world decision tree learning applications.

For attributes that are not continuous and do not have a meaningful ordering, but have a large number of possible values (e.g., *Zipcode* or *CreditCardNumber*), a measure called the **information gain ratio** (see Exercise 19.GAIN) can be used to avoid splitting into lots of single-example subtrees. Another useful approach is to allow an **equality test** of the form $A = v_k$. For example, the test $Zipcode = 10002$ could be used to pick out a large group of people in this zip code in New York City, and to lump everyone else into the "other" subtree.

- **CONTINUOUS-VALUED OUTPUT ATTRIBUTE:** If we are trying to predict a numerical output value, such as the price of an apartment, then we need a **regression tree** rather than a classification tree. A regression tree has at each leaf a linear function of some subset of numerical attributes, rather than a single output value. For example, the branch for two-bedroom apartments might end with a linear function of square

footage and number of bathrooms. The learning algorithm must decide when to stop splitting and begin applying linear regression (see Section 19.6) over the attributes. The name **CART**, standing for Classification And Regression Trees, is used to cover both classes.

Regression tree

CART

A decision tree learning system for real-world applications must be able to handle all of these problems. Handling continuous-valued variables is especially important, because both physical and financial processes provide numerical data. Several commercial packages have been built that meet these criteria, and they have been used to develop thousands of fielded systems. In many areas of industry and commerce, decision trees are the first method tried when a classification method is to be extracted from a data set.

Decision trees have a lot going for them: ease of understanding, scalability to large data sets, and versatility in handling discrete and continuous inputs as well as classification and regression. However, they can have suboptimal accuracy (largely due to the greedy search), and if trees are very deep, then getting a prediction for a new example can be expensive in run time. Decision trees are also **unstable** in that adding just one new example can change the test at the root, which changes the entire tree. In Section 19.8.2 we will see that the **random forest model** can fix some of these issues.

Unstable