CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

Schedule

- 11/30, 12/5, 12/7: Machine learning (classification, regression, clustering, deep learning(neural networks))
- 12/7: Project presentations and class project due
  - Project code due Monday 12/4 at 2PM on Moodle.
- Final exam on 12/14

Announcements

- HW4 out week of 11/14 (final homework assignment) due Friday 12/1 at 2pm on Moodle
  - https://www.cs.cmu.edu/~sganzfri/HW4_AI.pdf
- HW3 back today.
  - Average 71.5, standard deviation 17.2.

Machine learning

- An agent is **learning** if it improves its performance on future tasks after making observations about the world. Learning can range from the trivial, as exhibited by jotting down a phone number, to the profound, as exhibited by Albert Einstein, who inferred a new theory of the universe.
- We will start by concentrating on one class of learning problem, which seems restricted but actually has vast applicability: from a collection of input-output pairs, learn a function that predicts the output for new inputs.

Machine learning

- Why would we want an agent to learn? If the design of the agent can be improved, why wouldn’t the designers just program in that improvement to begin with? There are three main reasons.

- First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters.

- Second, the designers cannot anticipate all changes over time; a program designed to predict tomorrow’s stock market prices must learn to adapt when conditions change from boom to bust.

- Third, sometimes human programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of family members, but even the best programmers are unable to program a computer to accomplish that task, except by using learning algorithms.

Supervised learning

- The task of supervised learning is this: Given a **training set** of N example input-output pairs $(x_1, y_1),(x_2, y_2),\ldots,(x_N, y_N)$,
- Where each y_j was generated by an unknown function $y = f(x)$, discover a function h that approximates the true function f.
- Example: x_i, can be True/False for whether email says "Prize" in it, and y_i can be True/False for whether or not it is Spam.
- x and y can be any value, they need not be numbers.
  - E.g., x can be {red, green, blue} for jacket color, and y can be price.
- The function h is a **hypothesis**. Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set.

Supervised learning

- To measure the accuracy of a hypothesis we give it a **test set** of examples that are distinct from the training set.
  - What would happen if we tested on the examples that were trained on?
- We say a hypothesis **generalizes** well if it correctly predicts the value of y for novel examples. Sometimes the function f is stochastic—it is not strictly a function of x, and what we have to learn is a conditional probability distribution, P(Y|x).

Supervised learning

- When the output y is one of a finite set of values (such as *sunny*, *cloudy*, or *rainy*), the learning problem is called **classification**, and is called Boolean or binary classification if there are only two values. When y is a number (such as tomorrow's temperature), the learning problem is called **regression**. (Technically, solving a regression problem is finding a conditional expectation or average value of y, because the probability that we have found *exactly* the right real-valued number for y is 0).

Supervised learning

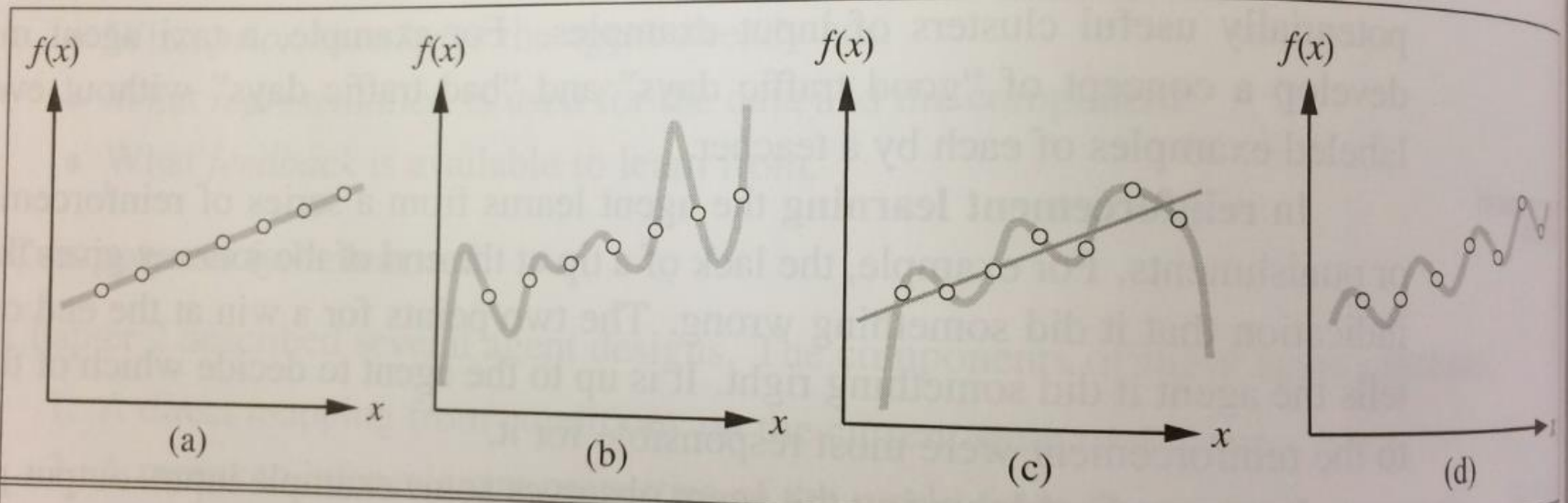

Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

Supervised learning

- The figure shows a familiar example: fitting a function of a single variable to some data points. The examples are points in the (x,y) plane, where y = f(x). We don't know what f is, but we will approximate it with a function h selected from a **hypothesis space**, H, which for this example we will take to be the set of polynomials such as $x^5 + 3x^2 + 2$. Figure a shows some data with an exact fit by a straight line (the polynomial 0.4x + 3). The line is called a **consistent** hypothesis because it agrees with all the data. Figure b shows a high-degree polynomial that is also consistent with all the data. This illustrates a fundamental problem in inductive learning: *how do we choose from among multiple consistent hypotheses*? The answer is to prefer the *simplest* hypothesis consistent with the data. This principle is called **Ockham's razor**, after the 14th-century English philosopher William of Ockham, who used it to argue sharply against all sorts of complications. Defining simplicity is not easy, but it seems clear that a degree-1 polynomial is simpler than a degree-7 polynomial, and thus (a) should be preferred to (b). We will make this intuition more precise later.

Supervised learning

- Figure c shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial for an exact fit. There are just 7 data points, so a polynomial with 7 parameters does not seem to be finding any pattern in the data and we do not expect it to generalize well. A straight line that is not consistent with any of the data points, but might generalize fairly well for unseen values of x, is also shown in c. *In general, there is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better*. In figure d we expand the hypothesis space H to allow polynomials over both x and sin(x), and find that the data in c can be fitted exactly by a simple function of the form ax + b + csin(x). This shows the importance of the hypothesis space.

Supervised learning

- In some cases, an analyst looking at a problem is willing to make more fine-grained distinctions about the hypothesis space, to say—even before seeing any data—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:
  - $h^* = \text{argmax}_{h \text{ in } H} P(h|data)$
  - By Bayes' rule, this is equivalent to $h^* = \text{argmax}_{h \text{ in } H} P(data|h)\ P(h)$
- Then we can say that the prior probability P(h) is high for a degree-1 or -2 polynomial, lower for a degree-7 polynomial, and especially low for degree-7 polynomials with large, sharp spikes as in Figure 18.1(b). We allow unusual-looking functions when the data say we really need them, but we discourage them by giving them a low prior probability.

Supervised learning

- Why not let H be the class of all Java programs, or Turing machines? After all, every computable function can be represented y some Turing machine, and that is the best we can do. One problem with this idea is that it does not take into account the computational complexity of learning. *There is a tradeoff between the expressiveness of a hypothesis space and the 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 in general 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.

Learning decision trees

- A **decision tree** represents a function that takes as input a vector of attribute values and returns a "decision"—a single output value. The input and output values can be discrete or continuous. For now we will concentrate on problems where the inputs have discrete values and the output has exactly two possible values; this is Boolean classification, where each example input will be classified as true (a **positive** example) or false (a **negative** example).

Decision trees

- A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the input attributes, A_i, and the branches from the node are labeled with the possible values of the attribute, $A_i = v_{ik}$. Each leaf node in the tree specifies a value to be returned by the function. The decision tree representation is natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

Decision tree

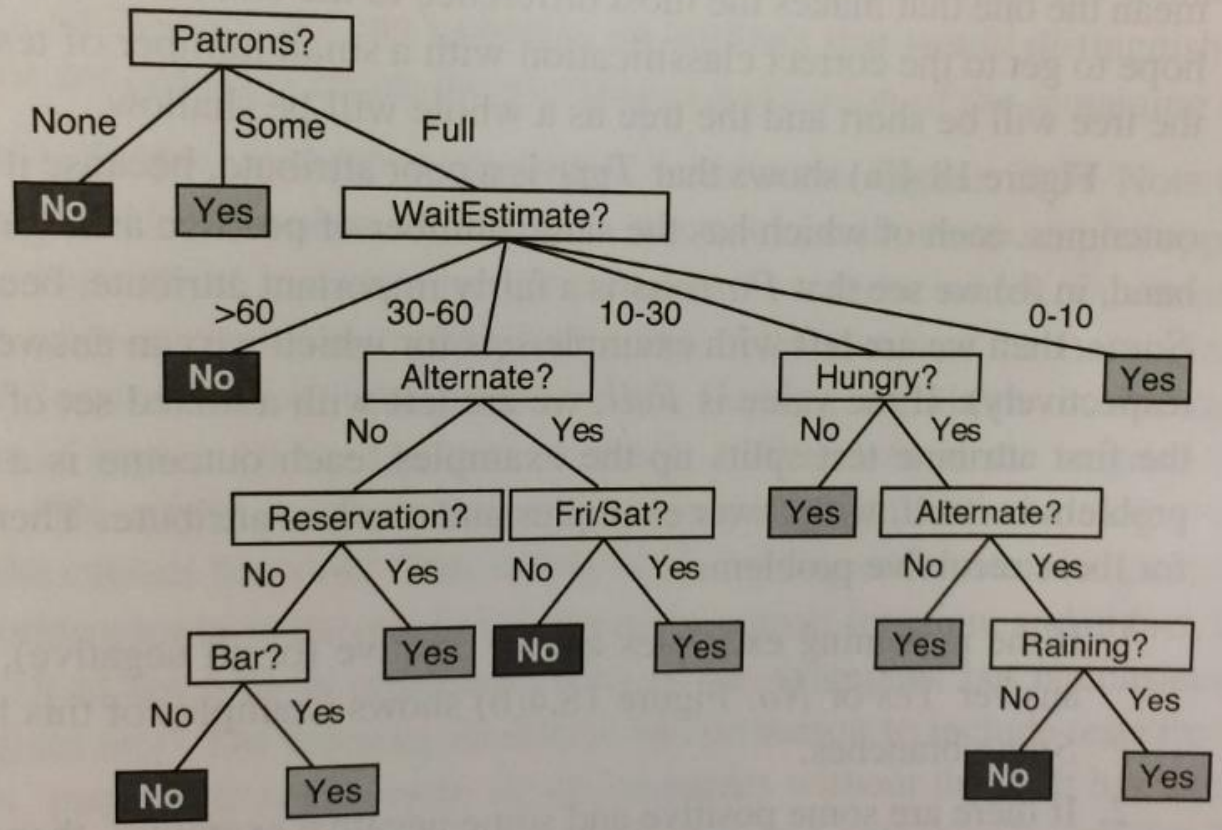

Figure 18.2 A decision tree for deciding whether to wait for a table.

Decision trees

- As an example, we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** *WillWait*. First we list the **attributes** that we will consider as part of the input:
  - *Alternate*: whether there is a suitable alternative restaurant nearby.
  - *Bar*: whether the restaurant has a comfortable bar area to wait in.
  - *Fri/Sat*: true on Fridays and Saturdays.
  - *Hungry*: whether we are hungry.
  - *Patrons*: how many people are in the restaurant (values are None, Some, and Full).
  - *Price*: the restaurant's price range ($, $$, $$$).
  - *Raining*: whether it is raining outside.
  - *Reservation*: whether we made a reservation.
  - *Type*: the kind of restaurant (French, Italian, Thai, or burger).
  - *WaitEstimate*: the wait estimated by the host (0-10 minutes, 10-30, 30-60, or >60).

Decision trees

- Note that every variable has a small set of possible values; the value of *WaitEstimate*, for example, is not an integer, rather it is one of the four discrete values 0-10, 10-30, 30-60, or >60. The decision tree usually used by one of us for this domain is shown in Figure 18.2. Notice that the tree ignores the Price and Type attributes. Examples are processed by the tree starting at the root and following the appropriate branch until a leaf is reached. For instance, an example with *Patrons* = Full and *WaitEstimate* = 0-10 will be classified as positive (i.e., yes, we will wait for a table).

Decision trees

- An example for a Boolean decision tree consists of an (x,y) pair, where x is a vector of values for the input attributes, and y is a single Boolean output value. A training set of 12 examples is shown in Figure 18.3. The positive examples are the ones in which the goal WillWait is true (x_1, x_3,…); the negative examples are the ones in which it is false (x_2, x_5,…).

Decision tree

Example	Input Attributes										Goal
	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*	$y_{12} = Yes$

Figure 18.3 Examples for the restaurant domain.

is shown in Figure

Decision trees

- Consider the set of all Boolean functions on n attributes. How many different functions are in this set? This is just the number of different truth tables that we can write down, because the function is defined by its truth table. A truth table over n attributes has 2^n rows, one for each combination of values of the attributes. We can consider the "answer" column of the table as a 2^n-bit number that defines the function. That means that there are $2^{2\wedge n}$ different functions (and there will be more than that number of trees, since more than one tree can compute the same function). This is a scary number. For example, with just the ten Boolean attributes of our restaurant problem there are 2^{1024}-bit, or about 10^{308}, different functions to choose from, and for 20 attributes there are over $10^{300,000}$. We will need some ingenious algorithms to find good hypotheses in such a large space.

Decision trees

- We want a tree that is consistent with the examples and is as small as possible. Unfortunately, no matter how we measure size, it is an intractable problem to find the smallest consistent tree; there is no way to efficiently search through the $2^{2^{\wedge n}}$ trees. With some simple heuristics, however, we can find a good approximate solution: a small (but not smallest) consistent tree. The DECISION-TREE-LEARNING ALGORITHM adopts a greedy divide-and-conquer strategy; always test the most important attribute first. This test divides the problem up into smaller subproblems that can then be solved recursively. 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.

Decision trees

- Figure 18.4(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. In general, after the first attribute test splits up the examples, each outcome is a new decision tree problem in itself, with fewer examples and one less attribute. There are four cases to consider for these recursive problems:

Decision trees

1. If the remaining examples are all positive (or all negative), then we are done: we can answer Yes or No. Figure 18.4(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 18.4(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 a default value calculated from the plurality classification of all the examples that were used in constructing the node's parent. These are passed along in the variable parent_examples.
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 plurality classification of the remaining examples.

Decision tree learning algorithm

- The DECISION-TREE-LEARNING algorithm is shown in Figure 18.5. Note that the set of examples is crucial for *constructing* the tree, but nowhere do the examples appear in the tree itself. A tree consists of just tests on attributes in the interior nodes, values of attributes on the branches, and output values on the leaf nodes. The output of the learning algorithm on our sample training set is shown in Figure 18.6.

Decision tree

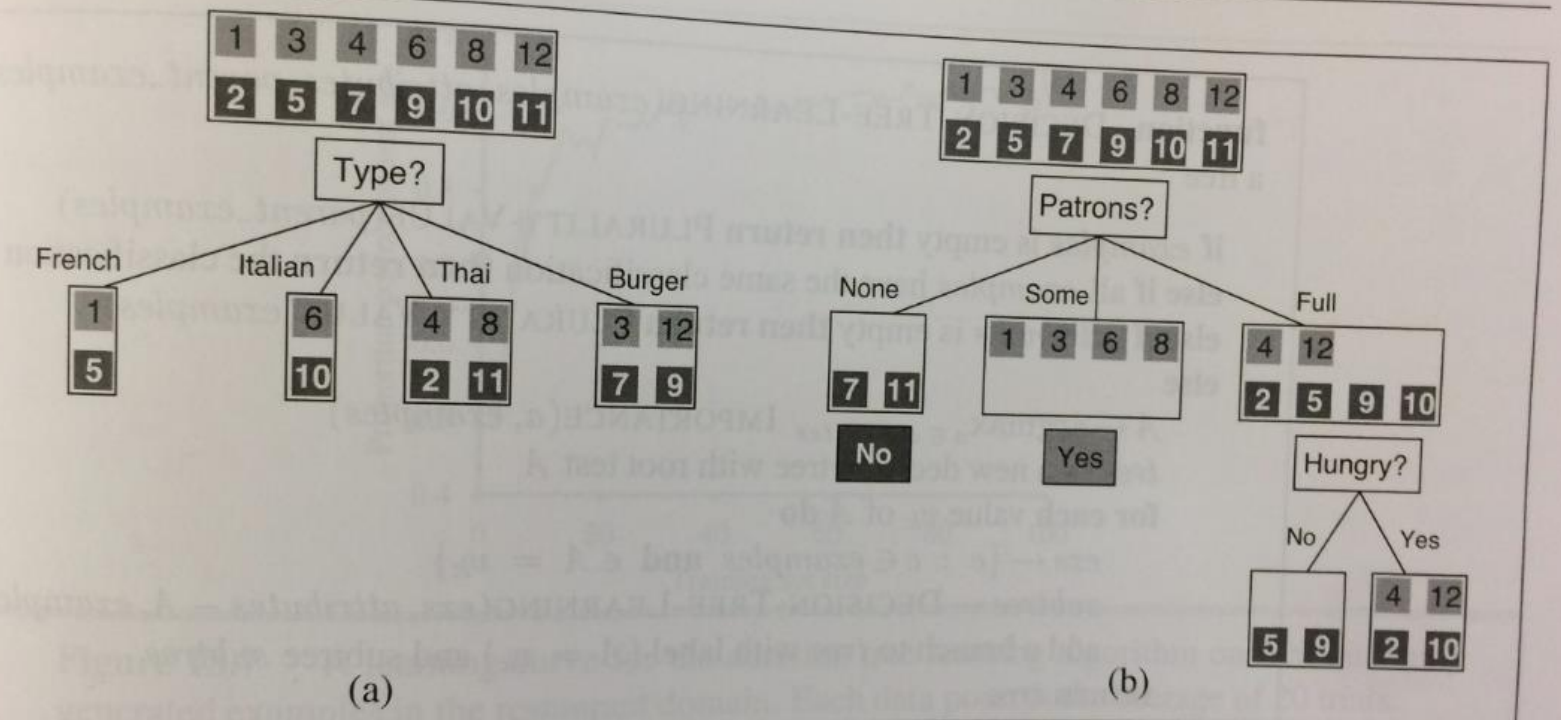

Figure 18.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

Decision trees

- The tree is clearly different from the original tree shown in Figure 18.2. 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 not only is consistent with all the examples, but is considerably simpler than the original tree! 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: the first author 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 I would certainly wait. With more training examples the learning program could correct this mistake.

Decision tree algorithm

```
function DECISION-TREE-LEARNING(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 ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← a new decision tree with root test A
      for each value v_k of A do
          exs ← {e : e ∈ examples and e.A = v_k}
          subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
          add a branch to tree with label (A = v_k) and subtree subtree
      return tree
```

Figure 18.5 The decision-tree learning algorithm. The function IMPORTANCE is described in Section 18.3.4. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

Decision tree from 12-example training set

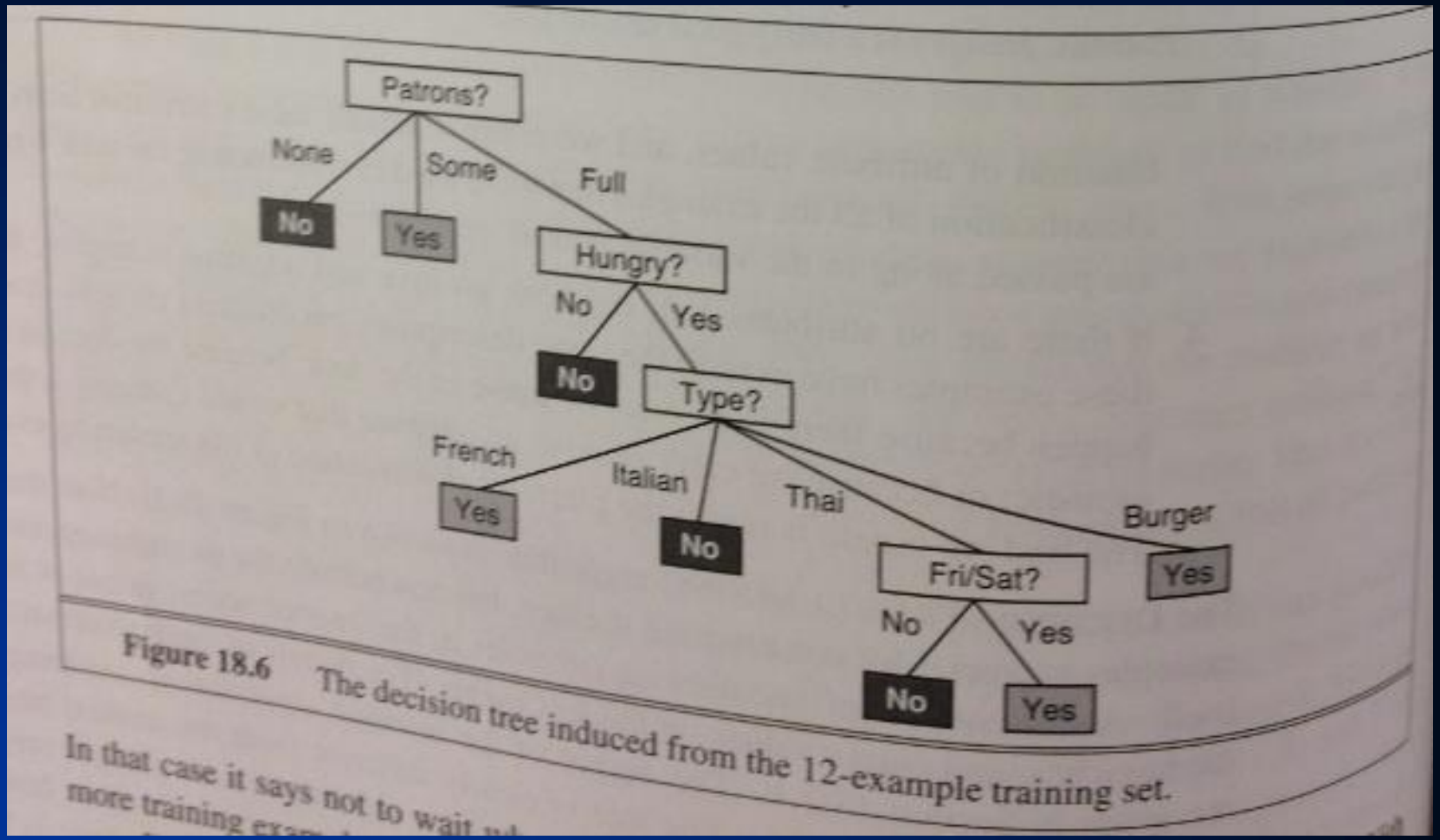

Figure 18.6 The decision tree induced from the 12-example training set.

In that case it says not to wait

Decision tree

- We note there is a danger of over-interpreting the tree that the algorithm selects. When there are several variables of similar importance, the choice between them is somewhat arbitrary: with slightly different input examples, a different variable would be chosen to split on first, and the whole tree would look completely different. The function computed by the tree would still be similar, but the structure of the tree can vary widely.

Learning curves

- We can evaluate the accuracy of a learning algorithm with a **learning curve**, as shown in Figure 18.7. We have 100 examples at our disposal, which we split into a training and a test set. We learn a hypothesis h with the training set and measure its accuracy with the test set. We 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 20 times, and average the results of the 20 trials. The curve shows that as the training 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 like the curve might continue to increase with more data.

Learning curve

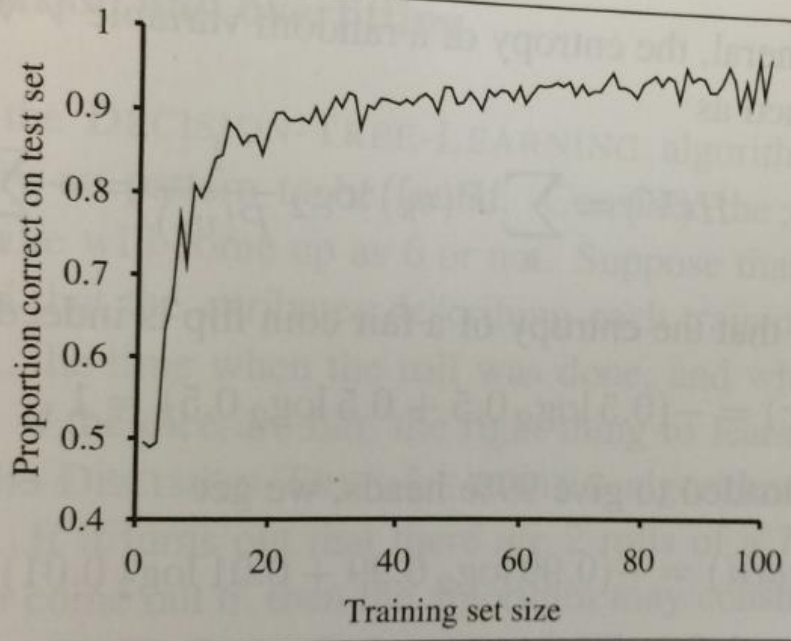

Figure 18.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

Choosing attribute tests

- The greedy search used in decision tree learning is designed to approximately minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. A perfect attribute divides the examples into sets, each of which are all positive or all negative and thus will be leaves of the tree. The *Patrons* attribute is not perfect, but it is fairly good. A really useless attribute, such as *Type*, leaves the example sets with roughly the same proportion of positive and negative examples as the original set.

Choosing attribute tests

- All we need, then, is a formal measure of "fairly good" and "really useless" and we can implement the IMPORTANCE function of Figure 18.5. We will use the notion of **information gain**, which is defined in terms of **entropy**, the fundamental quantity in information theory.

Choosing attribute tests

- Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy. A random variable with only one value—a coin that always comes up heads—has no uncertainty and thus its entropy is defined as zero; this, we gain no information by observing its value. A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, 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 it takes two bits to describe one of four 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.

Choosing attribute tests

- In general, the entropy of a random variable V with values v_k, each with probability $P(v_k)$, is defined as

  Entropy: $H(V) = \Sigma_k P(v_k) \log_2 (1/ P(v_k))$

  $= -\Sigma_k P(v_k) \log_2 (P(v_k))$

- We can check that the entropy of a fair coin flip is indeed 1 bit: H(Fair) = -(0.5 log(0.5) + 0.5 log(0.5)) = 1.
- If the coin is loaded to give 99% heads, we get

  H(Loaded) = -(0.99 log 0.99 + 0.01 log 0.01 ~= 0.08 bits.

Choosing attribute tests

- 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 q + (1-q) log(1-q))
- Thus, H(*Loaded*) = B(0.99) ~= 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 goal attribute on the whole set is H(Goal) = B(p/(p+n)).
- The restaurant training set in Figure 18.3 has p = n = 6, so the corresponding entropy is B(0.5) or exactly 1 bit. A test on a single attribute A might give us only part of this 1 bit. We can measure exactly how much by looking at the entropy remaining *after* the attribute test.

Choosing attribute tests

- 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 with probability $(p_k + n_k)/(p+n)$, so the expected entropy remaining after testing attribute A is
  - $\text{Remainder}(A) = \Sigma^d_{k=1} (p_k + n_k)/(p+n)\, B(p_k)/(p_k + n_k)$
- The **information gain** from the attribute test on A is the expected reduction in entropy: $\text{Gain}(A) = B(p/(p+n)) - \text{Remainder}(A)$.

Choosing attribute tests

- In fact Gain(A) is just what we need to implement the IMPORTANCE function. Returning to the attributes considered in Figure 18.4, we have
  - Gains(*Patrons*) = 1 – [2/12 B(0/2) + 4/12 B(4/4) + 6/12 B(2/6)] ~= 0.541 bits.
  - Gain(*Type*) = 1 – [2/12 B(1/2) + 2/12 B(1/2) + 4/12 B(2/4) + 4/12 B(2/4)] = 0 bits.
- This confirms our intuition that *Patrons* is a better attribute to split on. In fact, *Patrons* has the maximum gain of any of the attributes and would be chosen by the decision-tree learning algorithm as the root.

Bias-variance tradeoff

- We can create a graphical visualization of bias and variance using a bulls-eye diagram. Imagine that the center of the target is a model that perfectly predicts the correct values. As we move away from the bulls-eye, our predictions get worse and worse. Imagine we can repeat our entire model building process to get a number of separate hits on the target. Each hit represents an individual realization of our model, given the chance variability in the training data we gather. Sometimes we will get a good distribution of training data so we predict very well and we are close to the bulls-eye, while sometimes our training data might be full of outliers or non-standard values resulting in poorer predictions. These different realizations result in a scatter of hits on the target.
- We can plot four different cases representing combinations of both high and low bias and variance.

Bias-variance tradeoff

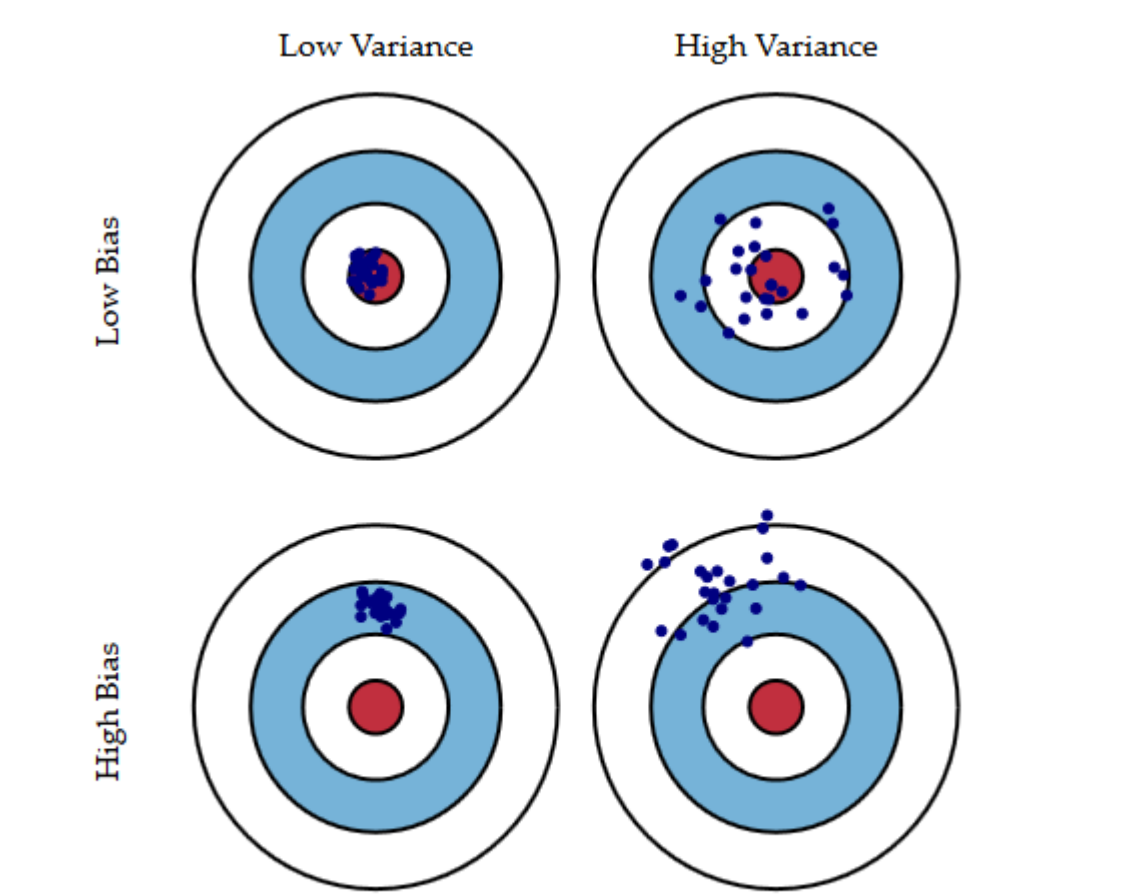

Fig. 1 Graphical illustration of bias and variance.

Bias-variance tradeoff

- The bias–variance tradeoff is a central problem in supervised learning. Ideally, one wants to choose a model that both accurately captures the regularities in its training data, but also generalizes well to unseen data. Unfortunately, it is typically impossible to do both simultaneously. High-variance learning methods may be able to represent their training set well, but are at risk of overfitting to noisy or unrepresentative training data. In contrast, algorithms with high bias typically produce simpler models that don't tend to overfit, but may underfit their training data, failing to capture important regularities.
- Models with low bias are usually more complex (e.g. higher-order regression polynomials), enabling them to represent the training set more accurately. In the process, however, they may also represent a large noise component in the training set, making their predictions less accurate - despite their added complexity. In contrast, models with higher bias tend to be relatively simple (low-order or even linear regression polynomials), but may produce lower variance predictions when applied beyond the training set.

Bias-variance tradeoff

lowermost image the approximated values for x=0 varies wildly depending on where the data points were located.

Bias–variance decomposition of squared error [edit]

Suppose that we have a training set consisting of a set of points $x_1, \ldots, x_n$ and real values y_i associated with each point x_i. We assume that there is a function with noise $y = f(x) + \epsilon$, where the noise, ϵ, has zero mean and variance σ^2.

We want to find a function $\hat{f}(x)$, that approximates the true function $f(x)$ as well as possible, by means of some learning algorithm. We make "as well as possible" precise by measuring the mean squared error between y and $\hat{f}(x)$: we want $(y - \hat{f}(x))^2$ to be minimal, both for $x_1, \ldots, x_n$ *and for points outside of our sample*. Of course, we cannot hope to do so perfectly, since the y_i contain noise ϵ; this means we must be prepared to accept an *irreducible error* in any function we come up with.

Finding an $\hat{f}$ that generalizes to points outside of the training set can be done with any of the countless algorithms used for supervised learning. It turns out that whichever function $\hat{f}$ we select, we can decompose its expected error on an unseen sample x as follows:[4]:34[5]:223

$$\mathrm{E}\left[\left(y - \hat{f}(x)\right)^2\right] = \mathrm{Bias}\left[\hat{f}(x)\right]^2 + \mathrm{Var}\left[\hat{f}(x)\right] + \sigma^2$$

Where:

$$\mathrm{Bias}\left[\hat{f}(x)\right] = \mathrm{E}\left[\hat{f}(x) - f(x)\right]$$

and

$$\mathrm{Var}\left[\hat{f}(x)\right] = \mathrm{E}[\hat{f}(x)^2] - \mathrm{E}[\hat{f}(x)]^2$$

The expectation ranges over different choices of the training set $x_1, \ldots, x_n, y_1, \ldots, y_n$, all sampled from the same joint distribution $P(x, y)$. The three terms represent:

- the square of the *bias* of the learning method, which can be thought of as the error caused by the simplifying assumptions built into the method. E.g., when approximating a non-linear function $f(x)$ using a learning method for linear models, there will be error in the estimates $\hat{f}(x)$ due to this assumption;
- the *variance* of the learning method, or, intuitively, how much the learning method $\hat{f}(x)$ will move around its mean;
- the irreducible error σ^2. Since all three terms are non-negative, this forms a lower bound on the expected error on unseen samples.[4]:34

The more complex the model $\hat{f}(x)$ is, the more data points it will capture, and the lower the bias will be. However, complexity will make the model "move" more to capture the data points, and hence its variance will be larger.

Bias-variance tradeoff

- Application to regression:

- The bias–variance decomposition forms the conceptual basis for regression regularization methods such as Lasso and ridge regression. Regularization methods introduce bias into the regression solution that can reduce variance considerably relative to the ordinary least-squares (OLS) solution. Although the OLS solution provides non-biased regression estimates, the lower variance solutions produced by regularization techniques provide superior MSE performance.

Bias-variance tradeoff

- Application to classification:

- The bias–variance decomposition was originally formulated for least-squares regression. For the case of classification under the 0-1 loss (misclassification rate), it's possible to find a similar decomposition. Alternatively, if the classification problem can be phrased as probabilistic classification, then the expected squared error of the predicted probabilities with respect to the true probabilities can be decomposed as before.

Bias-variance tradeoff

- Dimensionality reduction and feature selection can decrease variance by simplifying models. Similarly, a larger training set tends to decrease variance. Adding features (predictors) tends to decrease bias, at the expense of introducing additional variance. Learning algorithms typically have some tunable parameters that control bias and variance, e.g.:
  - (Generalized) linear models can be regularized to decrease their variance at the cost of increasing their bias.
  - In artificial neural networks, the variance increases and the bias decreases with the number of hidden units. Like in GLMs, regularization is typically applied.
  - In k-nearest neighbor models, a high value of k leads to high bias and low variance.
  - In decision trees, the depth of the tree determines the variance. Decision trees are commonly pruned to control variance.

Cross validation

- Cross-validation, sometimes called rotation estimation, is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. In a prediction problem, a model is usually given a dataset of known data on which training is run (training dataset), and a dataset of unknown data (or first seen data) against which the model is tested (called the validation dataset or testing set). The goal of cross validation is to define a dataset to "test" the model in the training phase (i.e., the validation set), in order to limit problems like overfitting, give an insight on how the model will generalize to an independent dataset (i.e., an unknown dataset, for instance from a real problem), etc.

Cross validation

- One round of cross-validation involves partitioning a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set). To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are combined (e.g. averaged) over the rounds to estimate a final predictive model.
- One of the main reasons for using cross-validation instead of using the conventional validation (e.g. partitioning the data set into two sets of 70% for training and 30% for test) is that there is not enough data available to partition it into separate training and test sets without losing significant modelling or testing capability. In these cases, a fair way to properly estimate model prediction performance is to use cross-validation as a powerful general technique.
- In summary, cross-validation combines (averages) measures of fit (prediction error) to derive a more accurate estimate of model prediction performance.

Cross validation

Common types of cross-validation [edit]

Two types of cross-validation can be distinguished, exhaustive and non-exhaustive cross-validation.

Exhaustive cross-validation [edit]

Exhaustive cross-validation methods are cross-validation methods which learn and test on all possible ways to divide the original sample into a training and a validation set.

Leave-p-out cross-validation [edit]

Leave-*p*-out cross-validation (**LpO CV**) involves using *p* observations as the validation set and the remaining observations as the training set. This is repeated on all ways to cut the original sample on a validation set of *p* observations and a training set.

LpO cross-validation requires training and validating the model C_p^n times, where *n* is the number of observations in the original sample, and where C_p^n is the binomial coefficient. For $p > 1$ and for even moderately large *n*, LpO CV can become computationally infeasible. For example, with $n = 100$ and $p = 30 = 30$ percent of 100 (as suggested above), $C_{30}^{100} \approx 3 \times 10^{25}$.

Leave-one-out cross-validation [edit]

Leave-*one*-out cross-validation (**LOOCV**) is a particular case of leave-*p*-out cross-validation with $p = 1$. The process looks similar to jackknife; however, with cross-validation you compute a statistic on the left-out sample(s), while with jackknifing you compute a statistic from the kept samples only.

LOO cross-validation does not have the same problem of excessive compute time as general LpO cross-validation because $C_1^n = n$.

Non-exhaustive cross-validation [edit]

Non-exhaustive cross validation methods do not compute all ways of splitting the original sample. Those methods are approximations of leave-*p*-out cross-validation.

k-fold cross-validation [edit]

In *k*-fold cross-validation, the original sample is randomly partitioned into *k* equal sized subsamples. Of the *k* subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k - 1$ subsamples are used as training data. The cross-validation process is then repeated *k* times (the *folds*), with each of the *k* subsamples used exactly once as the validation data. The *k* results from the folds can then be averaged to produce a single estimation. The advantage of this method over repeated random sub-sampling (see below) is that all observations are used for both training and validation, and each observation is used for validation exactly once. 10-fold cross-validation is commonly used,[7] but in general *k* remains an unfixed parameter.

For example, setting $k = 2$ results in 2-fold cross-validation. In 2-fold cross-validation, we randomly shuffle the dataset into two sets d_0 and d_1, so that both sets are equal size (this is usually implemented by shuffling the data array and then splitting it in two). We then train on d_0 and validate on d_1, followed by training on d_1 and validating on d_0.

When $k = n$ (the number of observations), the *k*-fold cross-validation is exactly the leave-one-out cross-validation.

In *stratified* *k*-fold cross-validation, the folds are selected so that the mean response value is approximately equal in all the folds. In the case of a dichotomous classification, this means that each fold contains roughly the same proportions of the two types of class labels.

Homework for next class

- Work on final project and finishing homework 4.
- HW4 due 12/1.
- Next lecture: Continue machine learning (regression, clustering, start deep learning).