CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

Schedule

- 12/5, 12/7: Machine learning (classification, regression, clustering, deep learning(neural networks))
- 12/7: Project presentations and class project due
  - Project code due tonight 12/5 at 12am on Moodle.
- Final exam on 12/14
  - 12pm in GL-139
- Evaluation
  - Log on to MyFIU portal at https://my.fiu.edu.
  - Click on SPOTs.
  - Select the course from the list of SPOTs.
  - Click on the instructor's name.
  - You will now be on the form and can share your perceptions and type comments.

Announcements

- HW4 (final homework assignment) was due Friday 12/1 on Moodle
- Project final paper due 12/7: 2-4 page paper in pdf
- Project presentations: ~2-3 minutes each
  - Ok for just one student to present from a group of 2-3

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.

Machine learning

- 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.

Machine learning

- 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.

Machine learning

- 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

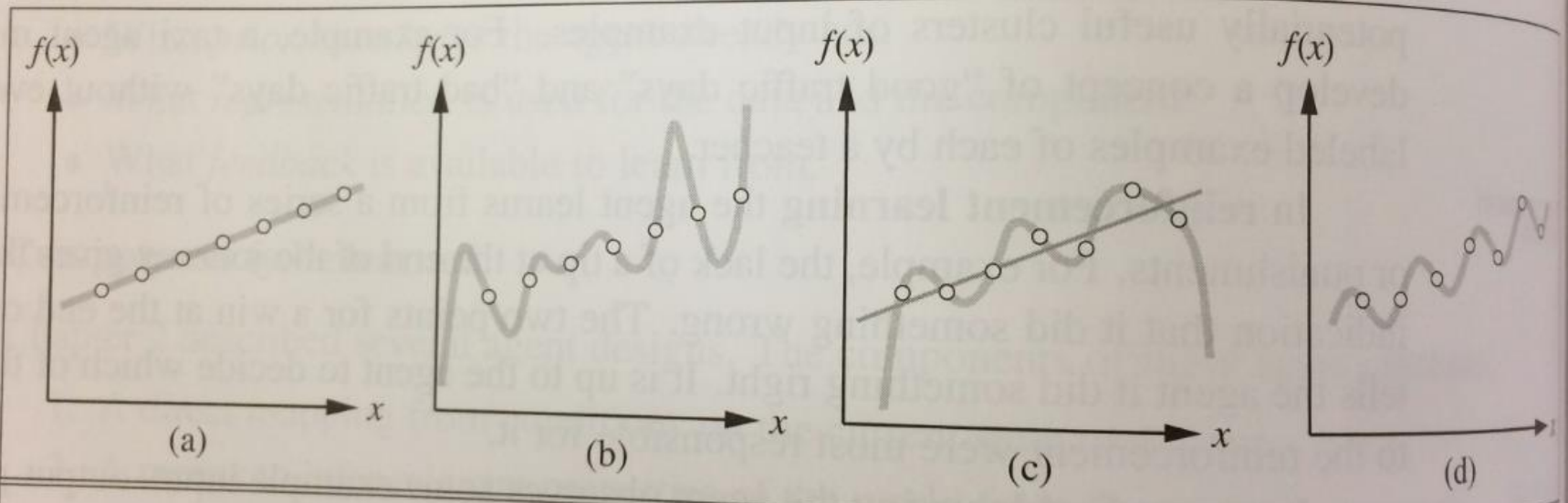


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data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

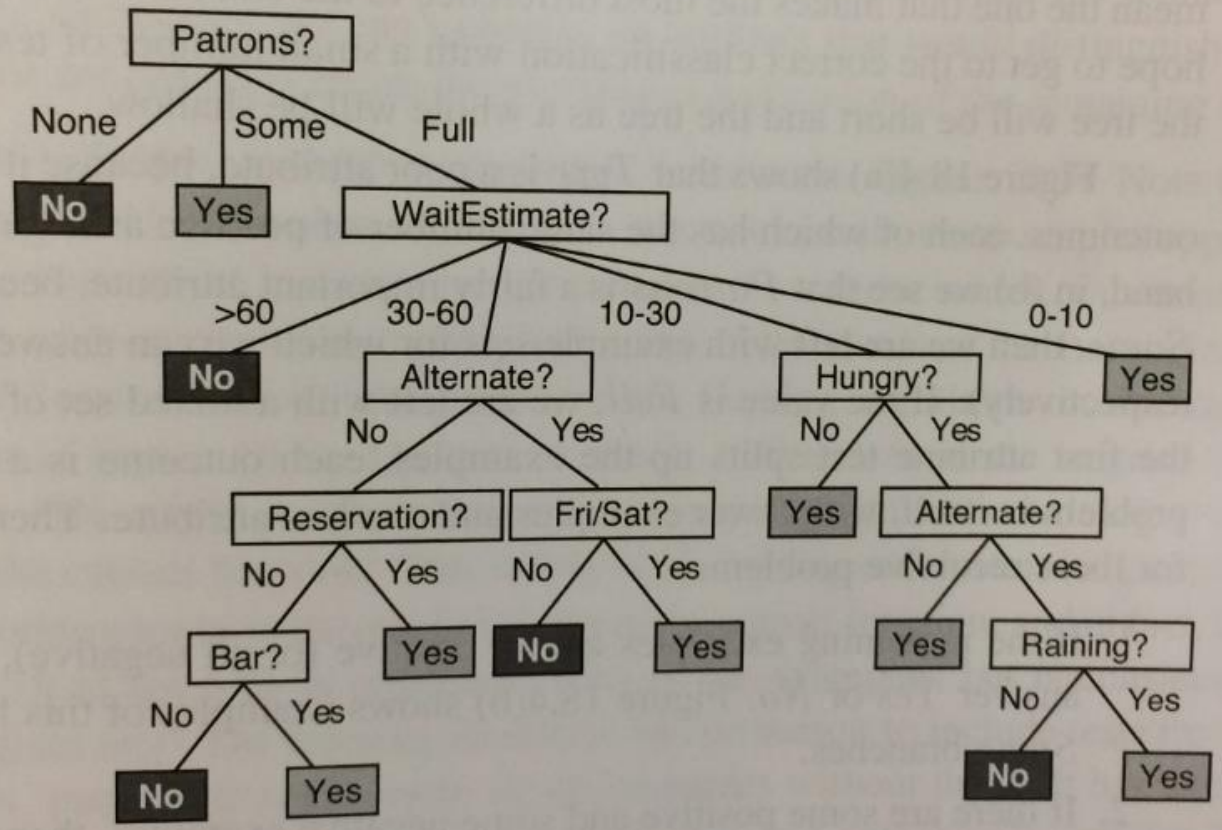


Figure 18.2 A decision tree for deciding whether to wait for a table.

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

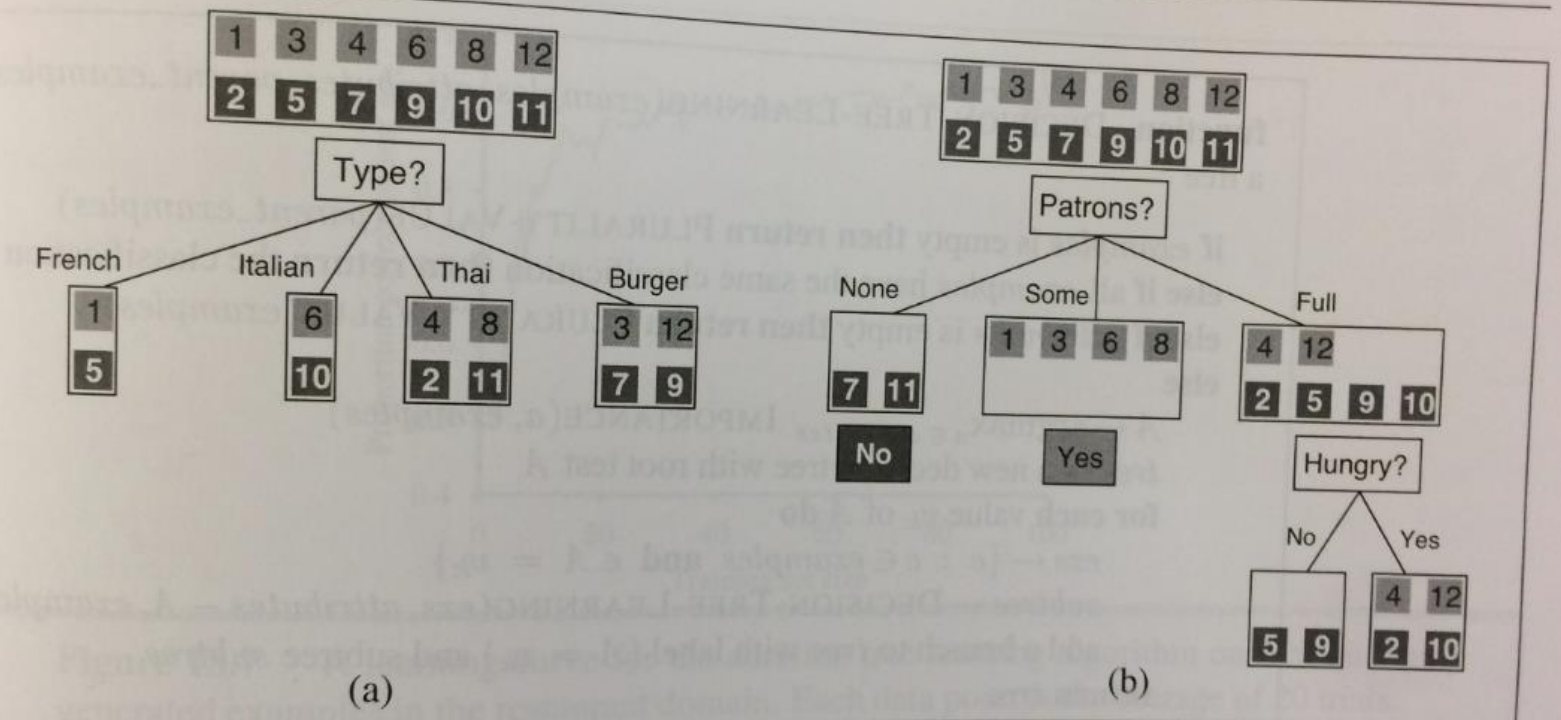


Figure 18.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of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

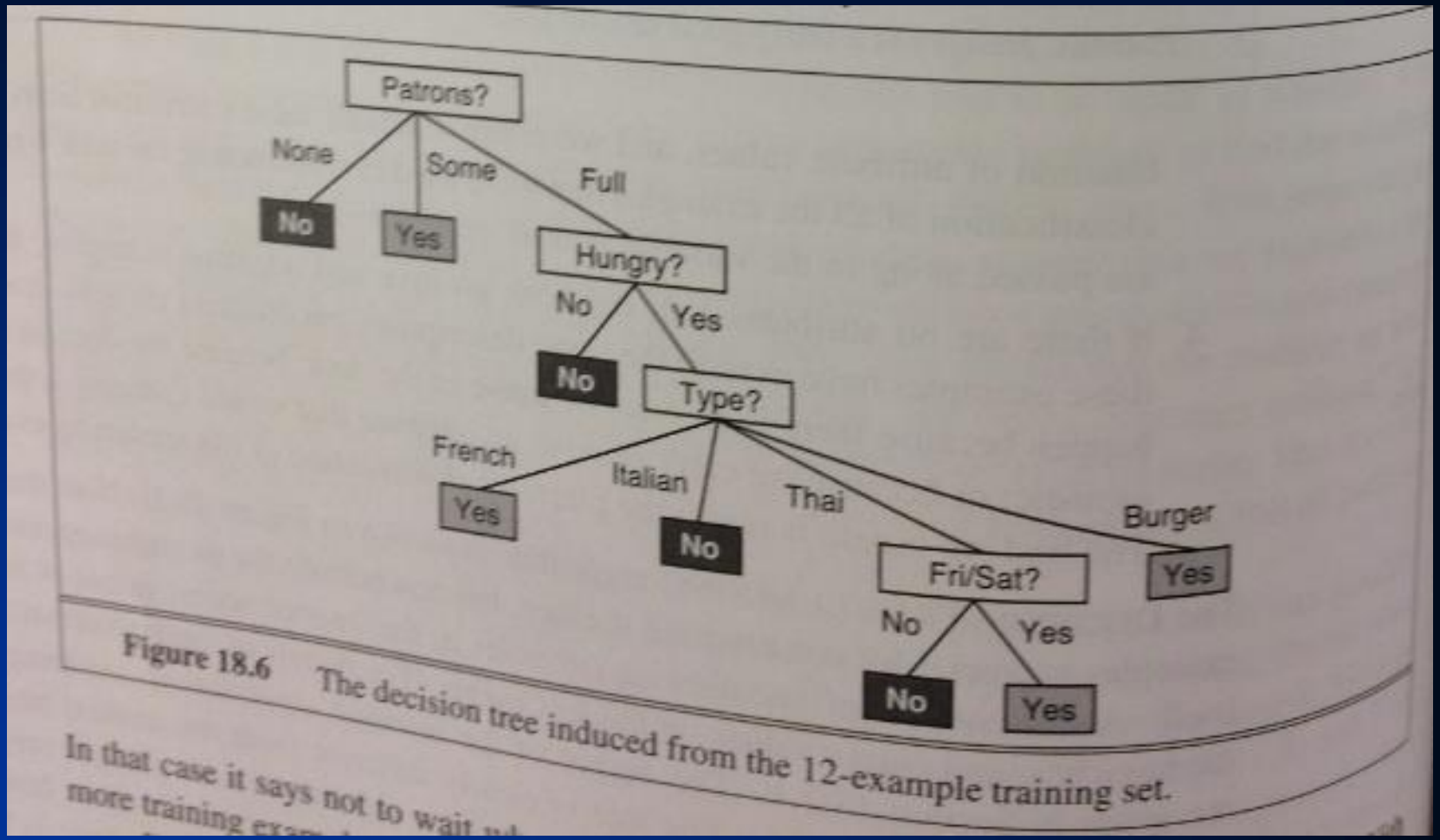


Figure 18.6 The decision tree induced from the 12-example training set.

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

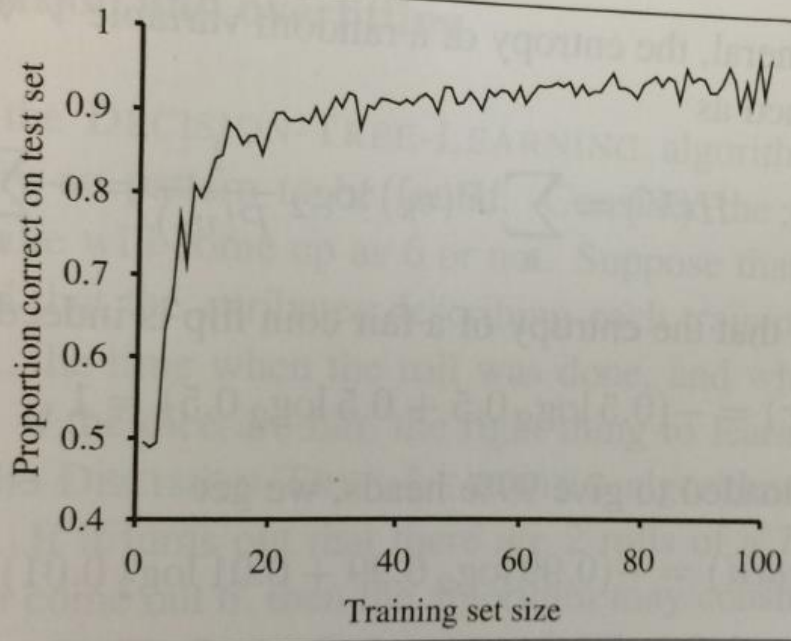


Figure 18.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

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.

Generalization and overfitting

- On some problems, the DECISION-TREE-LEARNING algorithm will generate a large tree when there is actually no pattern to be found. Consider the problem of trying to predict whether the roll of a die will come up as 6 or not. Suppose that experiments are carried out with various dice and that the attributes describing each training example include the color of the die, its weight, the time when the roll was done, and whether the experiments had their fingers crossed. If the dice are fair, the right thing to learn is a tree with a single node that says “no.” But the DECISION-TREE-LEARNING algorithm will seizes on any pattern it can find in the input. If it turns out that there are 2 rolls of a 7-gram blue die with fingers crossed and they both come out 6, then the algorithm may construct a path that predicts 6 in that case. This problem is called **overfitting**. A general phenomenon, overfitting occurs with all types of learners, even when the target function is not at all random. In Figure 18.1(b) and (c) we saw polynomial functions overfitting the data. Overfitting becomes more likely as the hypothesis space and the number of input attributes grows, and less likely as we increase the number of training examples.

Generalization and overfitting

- 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 DECISION-TREE-LEARNING. 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.

Generalization and overfitting

- 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 that each have roughly the same proportion of positive examples as the whole set, p/(p+n), and so the information gain will be close to zero. Thus, the information gain is a good clue to irrelevance. Now the question is, how large a gain should we require in order to split on a particular attribute?

Generalization and overfitting

- 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.
  - Can use various statistical tests to perform the pruning, such as χ^2-test.

Generalization and overfitting

- With pruning, noise in the examples can be tolerated. Errors in the example's label (e.g., an example (x,Yes) that should be (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.

Decision tree extensions

- Decision tree methodology can be extended in many directions to deal with:
  - Missing data
  - Multivalued attributes
  - Continuous and integer-valued input attributes
  - Continuous-valued output attributes (**regression tree** rather than classification tree. A regression tree has at each leaf a linear function of some subset of numerical attributes, rather than a single value).

Broad applicability of decision trees

- 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 usually the first method tried when a classification method is to be extracted from a data set. One important property of decision trees is that it is possible for a human to understand the reason for the output of the learning algorithm. (Indeed, this is a *legal requirement* for financial decisions that are subject to anti-discrimination laws.) This is a property not shared by some other representations, such as neural networks.

Evaluating and choosing the best hypothesis

- We want to learn a hypothesis that fits the future data best. To make that precise we need to define "future data" and "best." We make the **stationary assumption**: that there is a probability distribution over examples that remains stationary over time. Each example data point (before we see it) is a random variable E_j whose observed value $e_j = (x_j, y_j)$ is sampled from that distribution, and is independent of the previous examples:
  - $P(E_j | E_{j-1}, E_{j-2}, \ldots) = P(E_j)$
- And each example has an identical prior probability distribution:
  - $P(E_j) = P(E_{j-1}) = P(E_{j-2}) = \ldots$
- Examples that satisfy these assumptions are called *independent and identically distributed* or **i.i.d**. An i.i.d. assumption connects the past to the future; without some such connection, all bets are off—the future could be anything.

Choosing the best hypothesis

- The next step is to define “best fit.” We define the **error rate** of a hypothesis as the proportion of mistakes it makes—the proportion of times that h(x)!=y for an (x,y) example. Now, just because a hypothesis h has low error rate on the training set does not mean that it will generalize well. A professor knows that an exam will not accurately evaluate students if they have already seen the exam questions. Similarly, to get an accurate evaluation of a hypothesis, we need to test it on a set of examples it has not seen yet. The simplest approach is the learning algorithm produces h and a test set on which the accuracy of h is evaluated. This method, sometimes called **holdout cross-validation**, has the disadvantage that it fails to use all the available data; if we use half the data for the test set, then we are only training on half the data, and we may get a poor hypothesis. On the other hand, if we reserve only 10% of the data for the test set, then we may, by statistical chance, get a poor estimate of the actual accuracy.

Cross validation

- We can squeeze more out of the data and still get an accurate estimate using a technique called k-**fold-cross-validation**. The idea is that each example serves double duty—as training data and test data. First we split the data into k equal subsets. We then perform k rounds of learning; on each round 1/k of the data is held out as a test set and the remaining examples are used as training data. The average test set score of the k rounds should then be a better estimate than a single score. Popular values for k are 5 and 10—enough to give an estimate that is statistically likely to be accurate, at a cost of 5 to 10 times longer computation time. The extreme is k = n, also known as **leave-one-out cross-validation** or **LOOCV.**

Cross validation

- Despite the best efforts of statistical methodologists, users frequently invalidate their results by inadvertently **peeking** at the test data. Peeking can happen like this: A learning algorithm has various "knobs" that can be twiddled to tune its behavior—for example, various different criteria for choosing the next attribute in decision tree learning. The researcher generates hypotheses for various different settings of the knobs, measures their error rates on the test set, and reports the error rate of the best hypothesis. Alas, peeking has occurred! The reason is that the hypothesis was selected *on the basis of its test set error rate*, so information about the test set has leaked into the learning algorithm.

Evaluating the best hypothesis

- Peeking is a consequence of using test-set performance to both *choose* a hypothesis and *evaluate* it. The way to avoid this is to *really* hold the test set out—lock it away until you are completely done with learning and simply wish to obtain an independent evaluation of the final hypothesis. (And then, if you don't like the results … you have to obtain, and lock away, a completely new test set if you want to go back and find a better hypothesis.) If the test set is locked away, but you still want to measure performance on unseen data as a way of selecting a good hypothesis, then divide the available data (without the test set) into a training set and a **validation** set. This can be used to find a good tradeoff between hypothesis complexity and goodness of fit.

Model selection: Complexity versus goodness of fit

- Earlier we showed that higher-degree polynomials can fit the training data better, but when the degree is too high they will overfit, and perform poorly on validation data. Choosing the degree of the polynomial is an instance of the problem of **model selection**. You can think of the task of finding the best hypothesis as two tasks: model selection defines the hypothesis space and then **optimization** finds the best hypothesis within that space. We now explain how to select among models that are parameterized by *size*. For example, with polynomials we have *size* = 1 for linear functions, size = 2 for quadratics, etc. For decision trees, the size could be the number of nodes in the tree. In all cases we want to find the value of the *size* parameter that best balances underfitting and overfitting to give the bets test set accuracy.

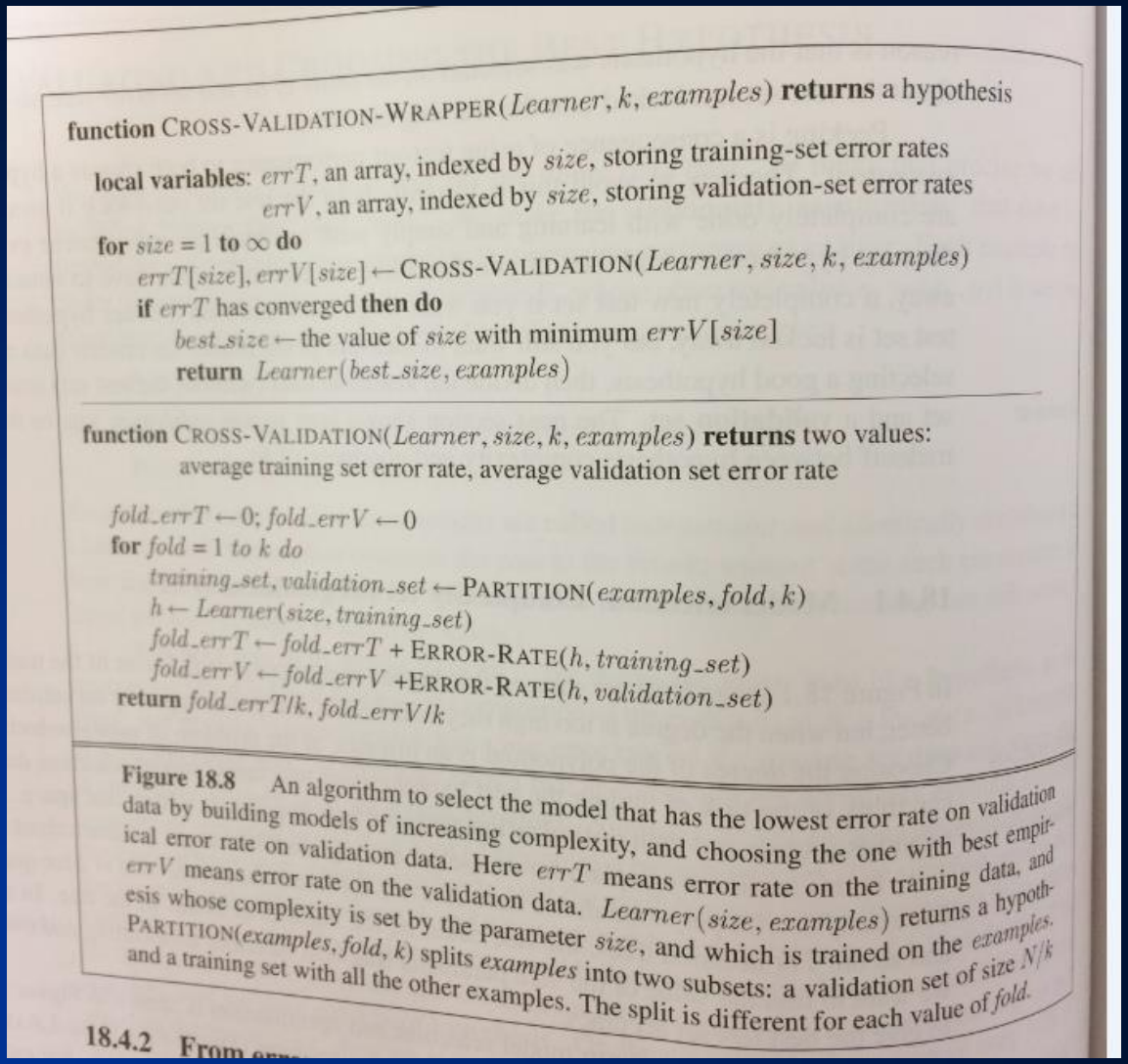
Model selection

- An algorithm to perform model selection and optimization is shown in Figure 18.8. It is a **wrapper** that takes a learning algorithm as an argument (DECISION-TREE-LEARNING, for example). The wrapper enumerates models according to a parameter, *size*. For each size, it uses cross validation on *Learner* to compute the average error rate on the training and test sets. We start with the smallest, simplest models (which probably underfit the data), and iterate, considering more complex models at each step, until the models start to overfit. In Figure 18.9 we see typical curves: the training set error decreases monotonically (although there may in general be slight random variation), while the validation set error decreases at first, and then increases when the model begins to overfit. The cross-validation procedure picks the value of *size* with the lowest validation set error; the bottom of the U-shaped curve. We then generate a hypothesis of that *size*, using all the data (without holding out any of it). Finally, of course, we should evaluate the returned hypothesis on a separate test set.

Model selection

- This approach requires that the learning algorithm accept a parameter, *size*, and deliver a hypothesis of that size. As we said, for decision tree learning the size can be the number of nodes (or depth of the tree). We can modify DECISION-TREE-LEARNER so that it takes the number of nodes as an input, builds the tree breadth-first rather than depth-first (but at each level it still chooses the highest gain attribute first), and stops when it reaches the desired number of nodes.

Algorithm for model selection



function CROSS-VALIDATION-WRAPPER(*Learner*, *k*, *examples*) **returns** a hypothesis

  local variables: *errT*, an array, indexed by *size*, storing training-set error rates
  errV, an array, indexed by *size*, storing validation-set error rates

  for *size* = 1 **to** ∞ **do**
    errT[*size*], *errV*[*size*] ← CROSS-VALIDATION(*Learner*, *size*, *k*, *examples*)
    if *errT* has converged **then do**
      best_size ← the value of *size* with minimum *errV*[*size*]
      return *Learner*(*best_size*, *examples*)

function CROSS-VALIDATION(*Learner*, *size*, *k*, *examples*) **returns** two values:
  average training set error rate, average validation set error rate

  fold_errT ← 0; *fold_errV* ← 0
  for *fold* = 1 *to k do*
    training_set, *validation_set* ← PARTITION(*examples*, *fold*, *k*)
    h ← *Learner*(*size*, *training_set*)
    fold_errT ← *fold_errT* + ERROR-RATE(*h*, *training_set*)
    fold_errV ← *fold_errV* +ERROR-RATE(*h*, *validation_set*)
  return *fold_errT*/*k*, *fold_errV*/*k*

Figure 18.8 An algorithm to select the model that has the lowest error rate on validation data by building models of increasing complexity, and choosing the one with best empirical error rate on validation data. Here *errT* means error rate on the training data, and *errV* means error rate on the validation data. *Learner*(*size*, *examples*) returns a hypothesis whose complexity is set by the parameter *size*, and which is trained on the *examples*. PARTITION(*examples*, *fold*, *k*) splits *examples* into two subsets: a validation set of size *N*/*k* and a training set with all the other examples. The split is different for each value of *fold*.

18.4.2 From

Error rates on training and validation data

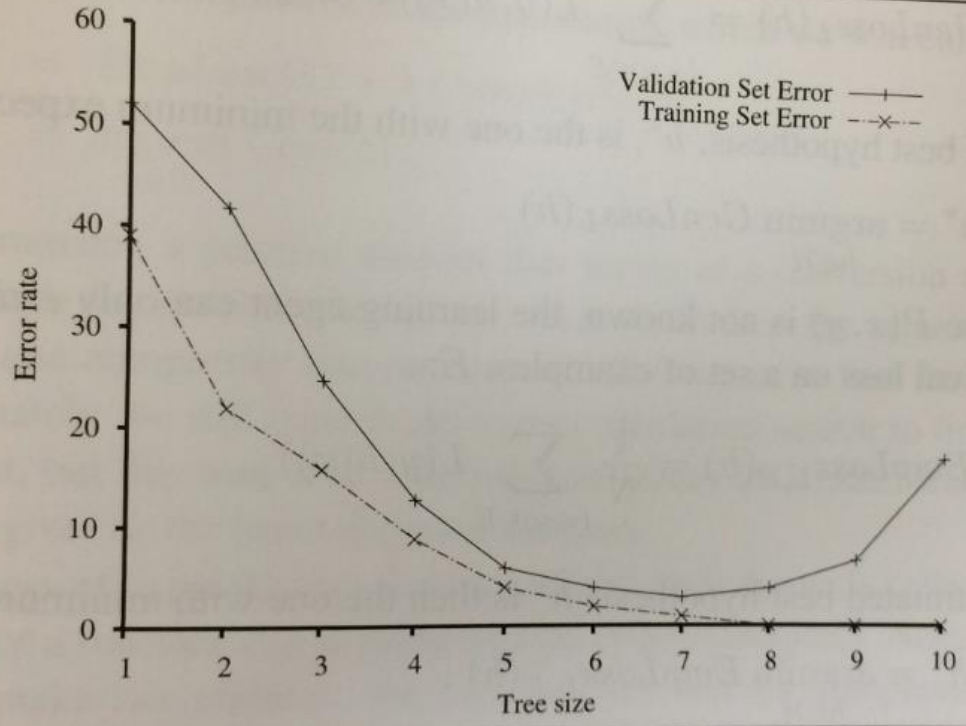


Figure 18.9 Error rates on training data (lower, dashed line) and validation data (upper, solid line) for different size decision trees. We stop when the training set error rate asymptotes, and then choose the tree with minimal error on the validation set; in this case the tree of size 7 nodes.

Regression and classification with linear models

- Now it is time to move on from decision trees to a different hypothesis space, one that has been used for hundreds of years: the class of **linear functions** of continuous-valued inputs. We'll start with the simplest case: regression with a univariate linear function, otherwise known as "fitting a straight line." This can be extended to the multivariate case as well.

Regression

- A univariate linear function (a straight line) with input x and output y has the form $y = w_1x + w_0$, where 20 and 21 are real-valued coefficients to be learned. We use the letter w because we think of the coefficients as **weights**; the value of y is changed by changing the relative weight of one term or another. We'll define w to be the vector $[w_0,w_1]$ and define $h_w(x) = w_1x + w_0$. Figure 18.13(a) shows an example of a training set of n points in the x,y plane, each point representing the size in square feet and the price of a house offered for sale. The task of finding the hw that best fits these data is called **linear regression**. To fit a line to the data, all we have to do is find the values of the weights $[w_0,w_1]$ that minimize the empirical loss. It is traditional to use the squared loss function, L2, summed over all the training examples.

Linear regression

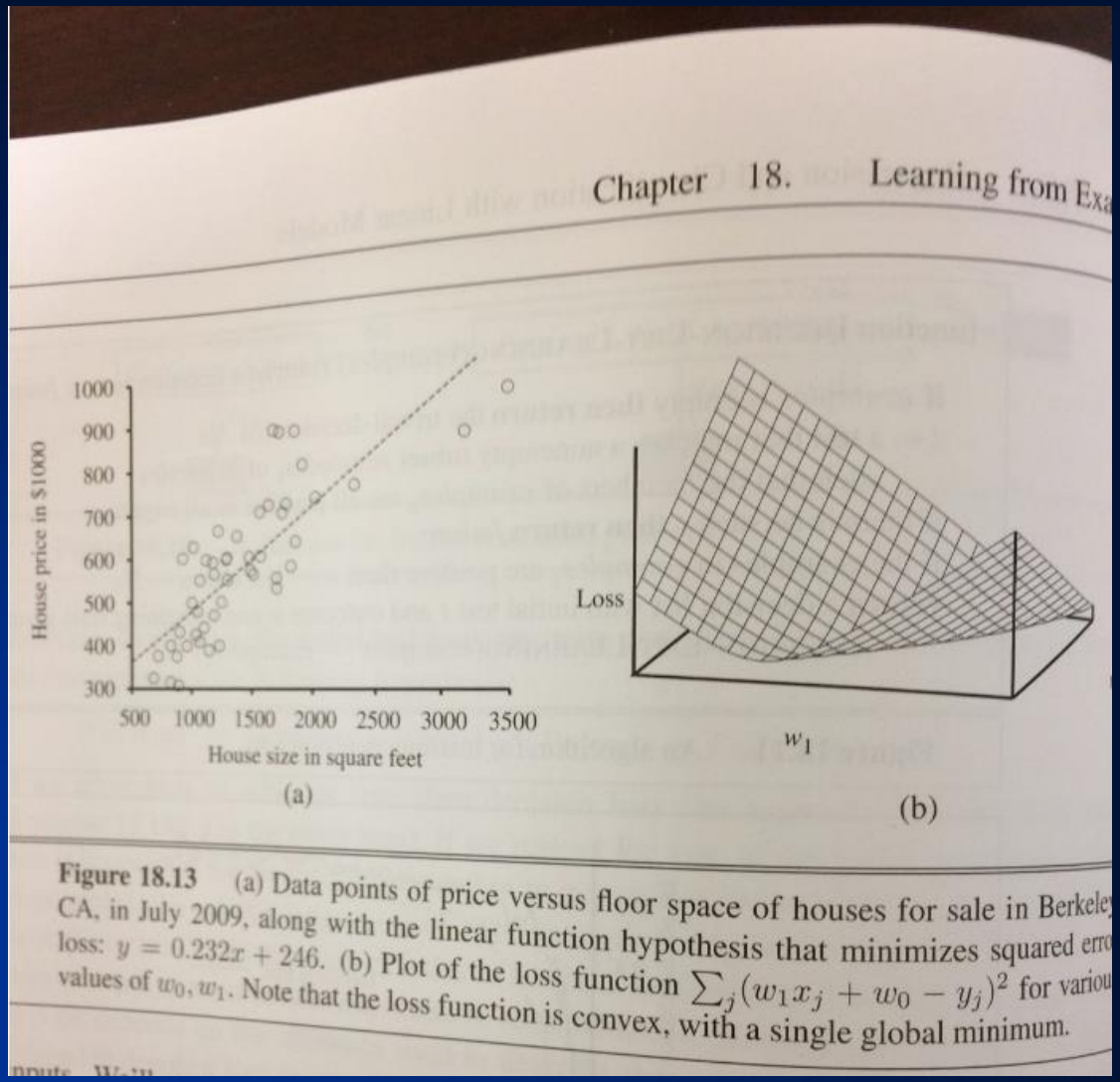


Figure 18.13 (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

Linear regression

$$Loss(h_{\mathbf{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\mathbf{w}}(x_j)) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(x_j))^2 = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 .$$

We would like to find $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} Loss(h_{\mathbf{w}})$. The sum $\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 . \quad (18.2)$$

These equations have a unique solution:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; \quad w_0 = (\sum y_j - w_1(\sum x_j))/N . \quad (18.3)$$

Linear regression

The general problem [edit]

β_j that are determined to give the best fit.

Consider an overdetermined system

$$\sum_{j=1}^{n} X_{ij}\beta_j = y_i, \ (i = 1, 2, \ldots, m),$$

of *m* linear equations in *n* unknown coefficients, $\beta_1, \beta_2, \ldots, \beta_n$, with $m > n$. (Note: for a linear model as above, not all of $\boldsymbol{X}$ contains information on the data points. The first column is populated with ones, $X_{i1} = 1$, only the other columns contain actual data, and *n* = number of regressors + 1.) This can be written in matrix form as

$$\mathbf{X}\boldsymbol{\beta} = \mathbf{y},$$

where

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1n} \\ X_{21} & X_{22} & \cdots & X_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{m1} & X_{m2} & \cdots & X_{mn} \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}.$$

Such a system usually has no solution, so the goal is instead to find the coefficients $\boldsymbol{\beta}$ which fit the equations "best",in the sense of solving the quadratic minimization problem

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} S(\boldsymbol{\beta}),$$

where the objective function *S* is given by

$$S(\boldsymbol{\beta}) = \sum_{i=1}^{m} \left| y_i - \sum_{j=1}^{n} X_{ij}\beta_j \right|^2 = \left\| \mathbf{y} - \mathbf{X}\boldsymbol{\beta} \right\|^2.$$

A justification for choosing this criterion is given in properties below. This minimization problem has a unique solution, provided that the *n* columns of the matrix $\mathbf{X}$ are linearly independent, given by solving the normal equations

$$(\mathbf{X}^{\mathrm{T}}\mathbf{X})\hat{\boldsymbol{\beta}} = \mathbf{X}^{\mathrm{T}}\mathbf{y}.$$

The matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is known as the Gramian matrix of $\mathbf{X}$, which possesses several nice properties such as being a positive semi-definite matrix, and the matrix $\mathbf{X}^{\mathrm{T}}\mathbf{y}$ is known as the moment matrix of regressand by regressors.[1] Finally, $\hat{\boldsymbol{\beta}}$ is the coefficient vector of the least-squares hyperplane, expressed as

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}.$$

Regularization

- **Regularization** can be used (essentially adding a parameter with a penalty term, e.g., for ridge regression) in order to reduce overfitting.

Supervised vs. unsupervised learning

- Supervised learning is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way (see inductive bias).
  - Includes classification (e.g., for decision trees) and regression.

Bias-variance tradeoff for supervised learning

- A first issue is the tradeoff between bias and variance. Imagine that we have available several different, but equally good, training data sets. A learning algorithm is biased for a particular input x if, when trained on each of these data sets, it is systematically incorrect when predicting the correct output for x. A learning algorithm has high variance for a particular input x if it predicts different output values when trained on different training sets. The prediction error of a learned classifier is related to the sum of the bias and the variance of the learning algorithm. Generally, there is a tradeoff between bias and variance. A learning algorithm with low bias must be "flexible" so that it can fit the data well. But if the learning algorithm is too flexible, it will fit each training data set differently, and hence have high variance. A key aspect of many supervised learning methods is that they are able to adjust this tradeoff between bias and variance (either automatically or by providing a bias/variance parameter that the user can adjust).

Bias-variance tradeoff

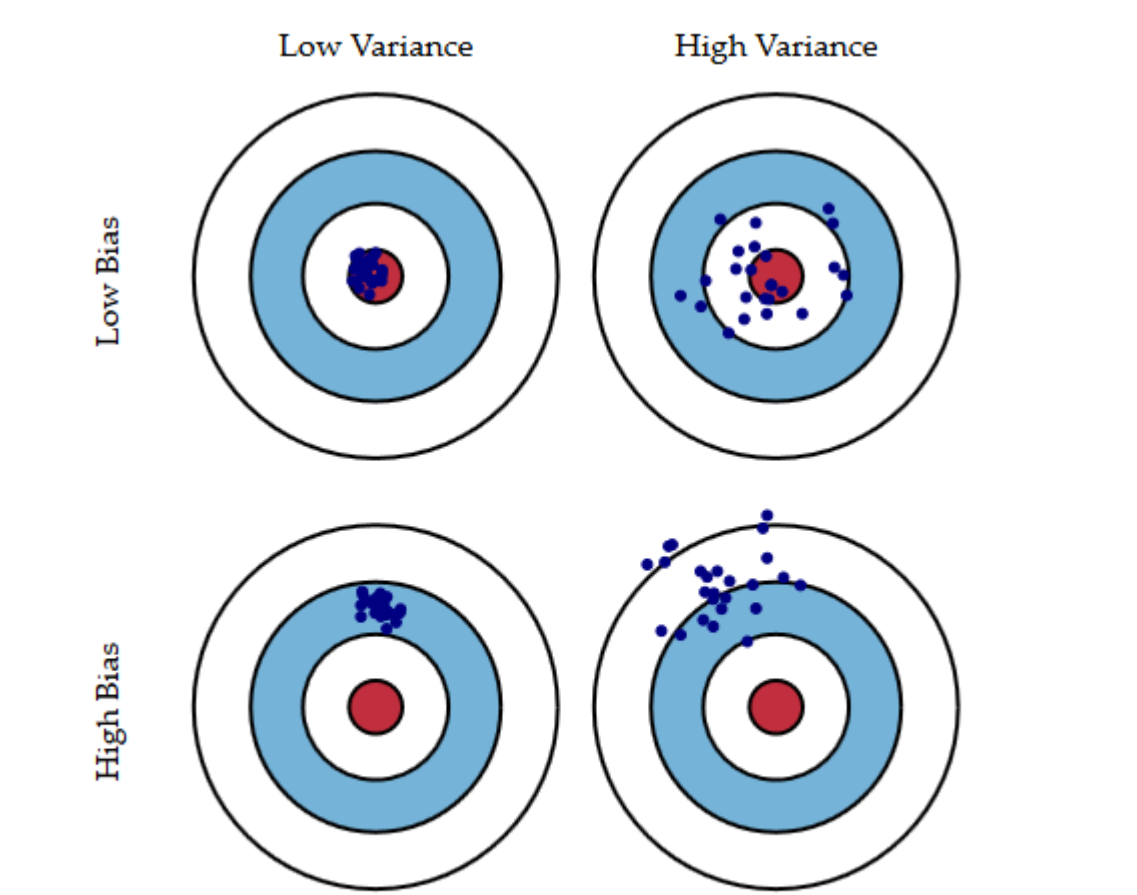


Fig. 1 Graphical illustration of bias and variance.

Supervised learning approaches

- We covered decision trees and regression
- Boosting
- Naïve Bayes classifier (e.g., for natural language processing)
- Nearest neighbor algorithm (“k-nn”)
- Maximum entropy classifier
- Support vector machines
- Random forests
- Many others. Python has built-in libraries and packages for the main ones.

Unsupervised learning

- Unsupervised machine learning is the machine learning task of inferring a function to describe hidden structure from "unlabeled" data (a classification or categorization is not included in the observations). Since the examples given to the learner are unlabeled, there is no evaluation of the accuracy of the structure that is output by the relevant algorithm—which is one way of distinguishing unsupervised learning from supervised learning and reinforcement learning.
- Approaches to unsupervised learning include:
  - Clustering
  - Anomaly detection
  - Neural networks
  - Approaches for latent variable modeling (e.g., expectation-maximization (EM) algorithm)

Clustering for hurricane classification

Saffir–Simpson scale

Category	Wind speeds
Five	≥70 m/s, ≥137 knots, ≥157 mph, ≥252 km/h
Four	58–70 m/s, 113–136 knots, 130–156 mph, 209–251 km/h
Three	50–58 m/s, 96–112 knots, 111–129 mph, 178–208 km/h
Two	43–49 m/s, 83–95 knots, 96–110 mph, 154–177 km/h
One	33–42 m/s, 64–82 knots, 74–95 mph, 119–153 km/h

Related classifications

Tropical storm	18–32 m/s, 34–63 knots, 39–73 mph, 63–118 km/h
Tropical depression	≤17 m/s, ≤33 knots, ≤38 mph, ≤62 km/h

Hurricane AI

- The Saffir–Simpson hurricane wind scale (SSHWS), formerly the Saffir–Simpson hurricane scale (SSHS), classifies hurricanes – Western Hemisphere tropical cyclones that exceed the intensities of tropical depressions and tropical storms – into five categories distinguished by the intensities of their sustained winds. To be classified as a hurricane, a tropical cyclone must have maximum sustained winds of at least 74 mph (33 m/s; 64 kn; 119 km/h) (Category 1). The highest classification in the scale, Category 5, consists of storms with sustained winds exceeding 156 mph (70 m/s; 136 kn; 251 km/h).
- The classifications can provide some indication of the potential damage and flooding a hurricane will cause upon landfall.
- Officially, the Saffir–Simpson hurricane wind scale is used only to describe hurricanes forming in the Atlantic Ocean and northern Pacific Ocean east of the International Date Line. Other areas use different scales to label these storms, which are called "cyclones" or "typhoons", depending on the area.
- There is some criticism of the SSHS for not taking rain, storm surge, and other important factors into consideration, but SSHS defenders say that part of the goal of SSHS is to be straightforward and simple to understand.

Hurricane AI

- The scale was developed in 1971 by civil engineer Herbert Saffir and meteorologist Robert Simpson, who at the time was director of the U.S. National Hurricane Center (NHC).[1] The scale was introduced to the general public in 1973,[2] and saw widespread use after Neil Frank replaced Simpson at the helm of the NHC in 1974.[3]
- The initial scale was developed by Saffir, a structural engineer, who in 1969 went on commission for the United Nations to study low-cost housing in hurricane-prone areas.[4] While performing the study, Saffir realized there was no simple scale for describing the likely effects of a hurricane. Mirroring the utility of the Richter magnitude scale in describing earthquakes, he devised a 1–5 scale based on wind speed that showed expected damage to structures. Saffir gave the scale to the NHC, and Simpson added the effects of storm surge and flooding.

AI clustering

- **Cluster analysis** or **clustering** is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression, and computer graphics.

Clustering

- Cluster analysis itself is not one specific algorithm, but the general task to be solved. It can be achieved by various algorithms that differ significantly in their notion of what constitutes a cluster and how to efficiently find them. Popular notions of clusters include groups with small distances among the cluster members, dense areas of the data space, intervals or particular statistical distributions. Clustering can therefore be formulated as a multi-objective optimization problem. The appropriate clustering algorithm and parameter settings (including values such as the distance function to use, a density threshold or the number of expected clusters) depend on the individual data set and intended use of the results. Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.

K-means clustering algorithm

Algorithms [edit]

Standard algorithm [edit]

The most common algorithm uses an iterative refinement technique. Due to its ubiquity it is often called the ***k*-means algorithm**; it is also referred to as **Lloyd's algorithm**, particularly in the computer science community.

Given an initial set of k means $m_1^{(1)},\ldots,m_k^{(1)}$ (see below), the algorithm proceeds by alternating between two steps:[6]

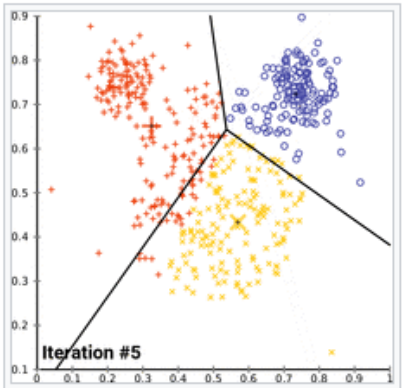
Assignment step: Assign each observation to the cluster whose mean has the least squared Euclidean distance, this is intuitively the "nearest" mean.[7] (Mathematically, this means partitioning the observations according to the Voronoi diagram generated by the means).

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \leq j \leq k \right\},$$

where each x_p is assigned to exactly one $S^{(t)}$, even if it could be assigned to two or more of them.

Update step: Calculate the new means to be the centroids of the observations in the new clusters.

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$



Convergence of k-means

The algorithm has converged when the assignments no longer change. There is no guarantee that the optimum is found using this algorithm.[8]

The algorithm is often presented as assigning objects to the nearest cluster by distance. Using a different distance function other than (squared) Euclidean distance may stop the algorithm from converging.[*citation needed*] Various modifications of k-means such as spherical k-means and k-medoids have been proposed to allow using other distance measures.

Other clustering approaches

- K-medoids
- K-medians
- Different initialization methods, e.g., kmeans++
  - Typically initial cluster means are chosen at random

Homework for next class

- Finish final project
- Next lecture: Wrap up machine learning (neural networks (deep learning))
- Project presentations
  - 2-3 minutes each