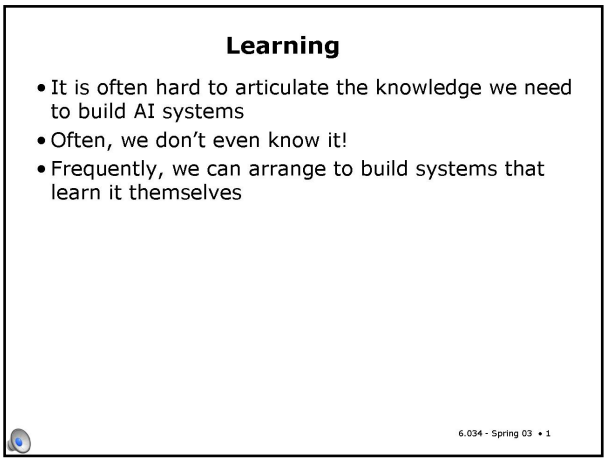
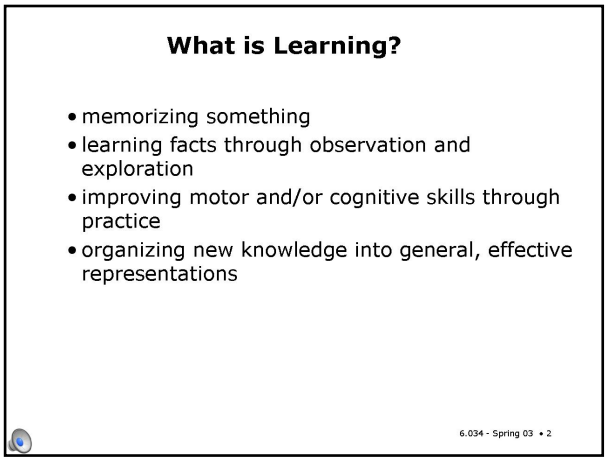
# 6.034 Notes: Section 2.1

**Slide 2.1.1**
So far, we have talked a lot about building systems that have knowledge represented in them explicitly. One way to acquire that knowledge is to build it in by hand. But that can be time-consuming and error prone. And many times, humans just don't have the relevant information available to them.

For instance, you are all experts in visual object recognition. But it's unlikely that you could sit down and write a program that would do the job even remotely as well as you can.

But, in lots of cases, when we don't have direct access to a formal description of a problem, we can learn something about it from examples.



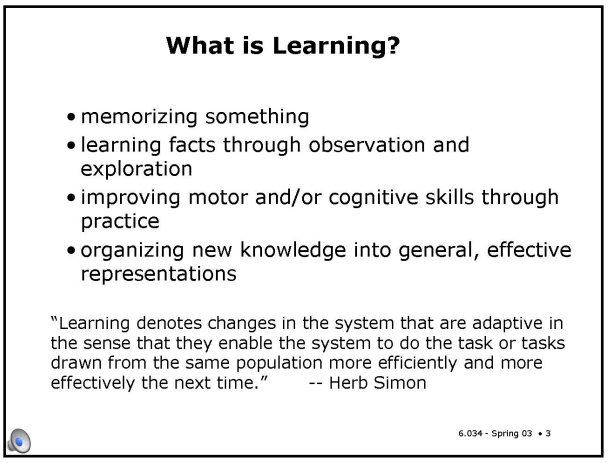

**Slide 2.1.2**
The word "learning" has many different meanings. It is used, at least, to describe

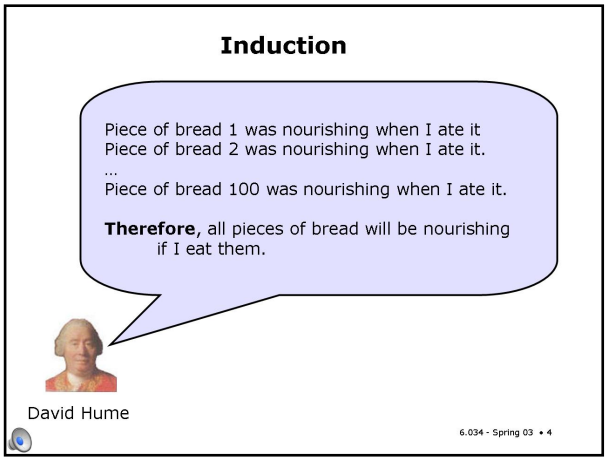
- memorizing something
- learning facts through observation and exploration
- development of motor and/or cognitive skills through practice
- organization of new knowledge into general, effective representations

**Slide 2.1.3**
Herb Simon (an important historical figure in AI and in Economics) gave us this definition of learning:

*Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the task or tasks drawn from the same population more efficiently and more effectively the next time.*

This is not entirely precise, but it gives us the idea that learning systems have to acquire some information from examples of a problem, and perform better because of it (so we wouldn't call a system that simply logs all the images it has ever seen a learning system, because, although, in some sense, it knows a lot, it can't take advantage of its knowledge).

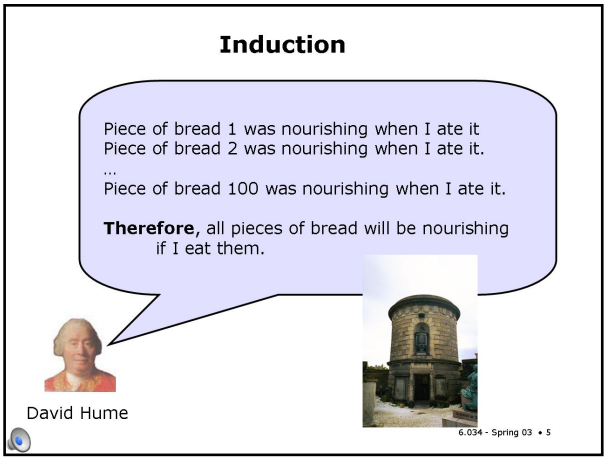


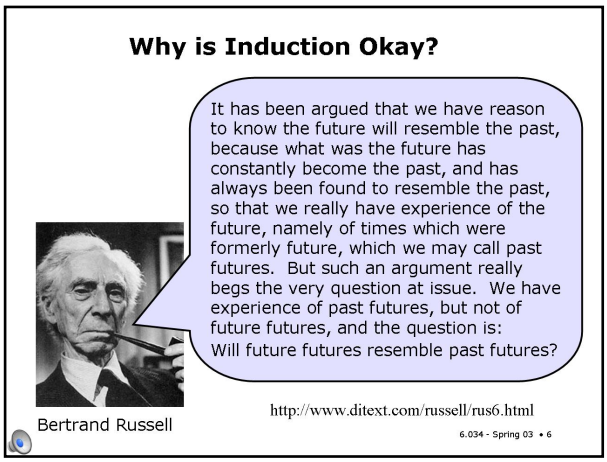



**Slide 2.1.4**
One of the most common kinds of learning is the acquisition of information with the goal of making predictions about the future. But what exactly gives us license to imagine we can predict the future? Lots of philosophers have thought about this problem. David Hume first framed the problem of induction as follows:

- Piece of bread number 1 was nourishing when I ate it.
- Piece of bread number 2 was nourishing when I ate it.
- Piece of bread number 3 was nourishing when I ate it.
- Piece of bread number 100 was nourishing when I ate it.
- *Therefore*, all pieces of bread will be nourishing if I eat them.

**Slide 2.1.5**
And here, for no good reason, is a photo of David Hume's Tomb, just because I saw it once in Edinburgh.




**Slide 2.1.6**
When and why is it okay to apply inductive reasoning??

My favorite quote on the subject is the following, exerpted from Bertrand Russell's "On Induction": (see http://www.ditext.com/russell/rus6.html for the whole thing)

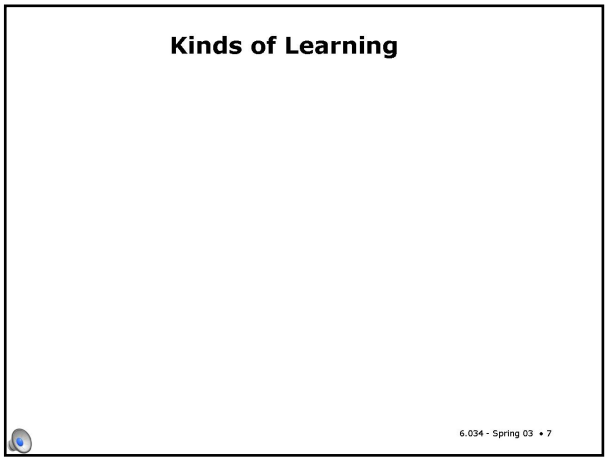
*If asked why we believe the sun will rise tomorrow, we shall naturally answer, 'Because it has always risen every day.' We have a firm belief that it will rise in the future, because it has risen in the past.*

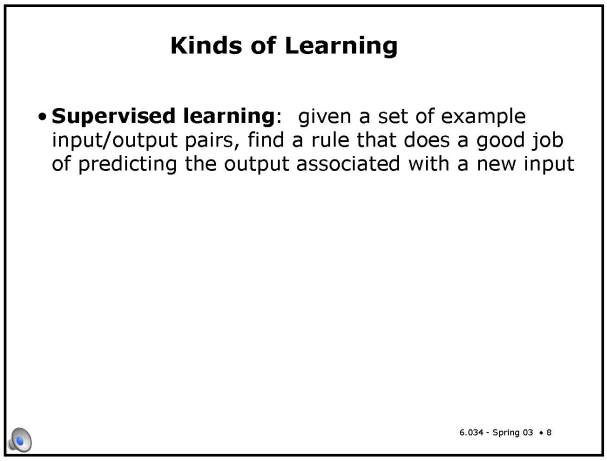
*The real question is: Do any number of cases of a law being fulfilled in the past afford evidence that it will be fulfilled in the future?*

*It has been argued that we have reason to know the future will resemble the past, because what was the future has constantly become the past, and has always been found to resemble the past, so that we really have experience of the future, namely of times which were formerly future, which we may call past futures. But such an argument really begs the very question at issue. We have experience of past futures, but not of future futures, and the question is: Will future futures resemble past futures?*

We won't worry too much about this problem. If induction is not, somehow, justified, then we have no reason to get out of bed in the morning, let alone study machine learning!

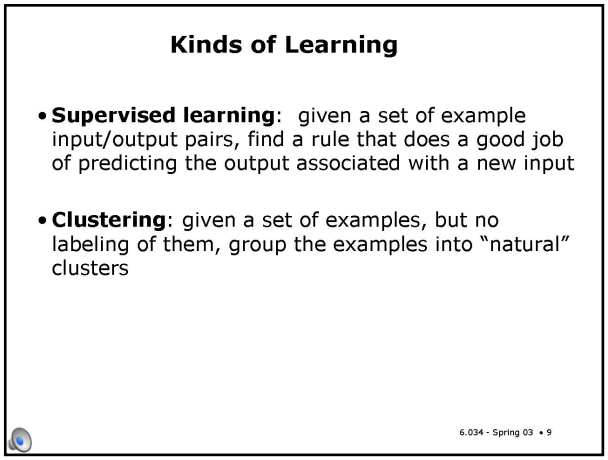
**Slide 2.1.7**
When viewed technically, there are lots of different kinds of machine learning problems. We'll just sketch them out here, so you get an idea of their range.

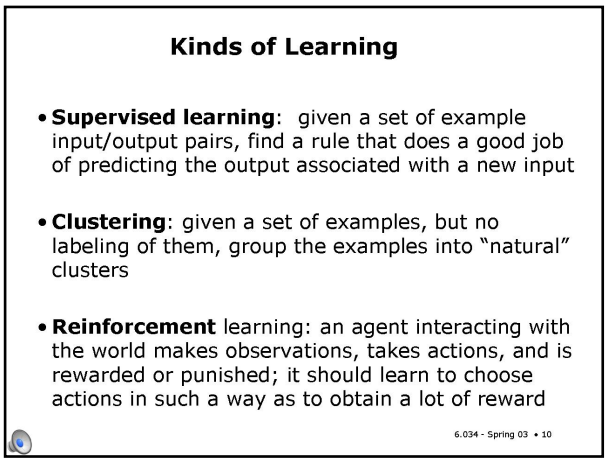





**Slide 2.1.8**
**Supervised learning** is the most common learning problem. Let's say you are given the weights and lengths of a bunch of individual salmon fish, and the weights and lengths of a bunch of individual tuna fish. The job of a supervised learning system would be to find a predictive rule that, given the weight and length of a fish, would predict whether it was a salmon or a tuna.

**Slide 2.1.9**
Another, somewhat less well-specified, learning problem is **clustering**. Now you're given the descriptions of a bunch of different individual animals (or stars, or documents) in terms of a set of features (weight, number of legs, presence of hair, etc), and your job is to divide them into groups, or possibly into a hierarchy of groups that "makes sense". What makes this different from supervised learning is that you're not told in advance what groups the animals should be put into; just that you should find a natural grouping.

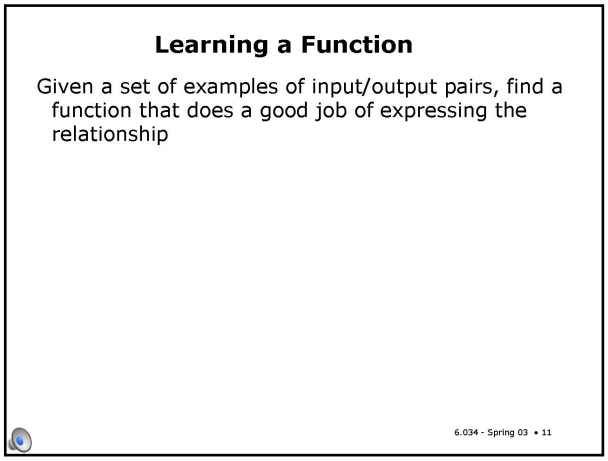


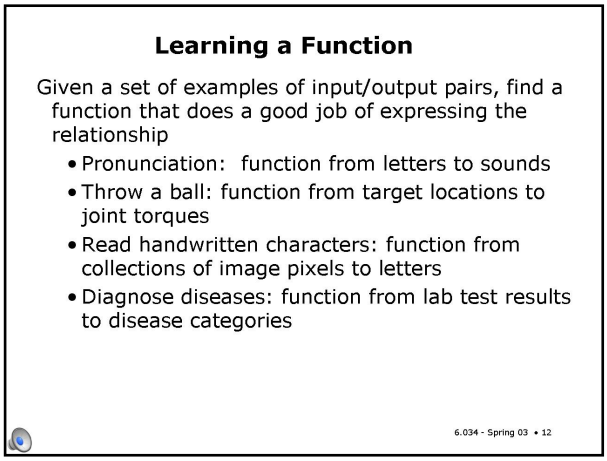




**Slide 2.1.10**
Another learning problem, familiar to most of us, is learning motor skills, like riding a bike. We call this **reinforcement learning**. It's different from supervised learning because no-one explicitly tells you the right thing to do; you just have to try things and see what makes you fall over and what keeps you upright.

Supervised learning is the most straightforward, prevalent, and well-studied version of the learning problem, so we are going to start with it, and spend most of our time on it. Most of the fundamental insights into machine learning can be seen in the supervised case.

**Slide 2.1.11**
One way to think about learning is that you're trying to find the definition of a function, given a bunch of examples of its input and output. This might seem like a pretty narrow definition, but it actually covers a lot of cases.

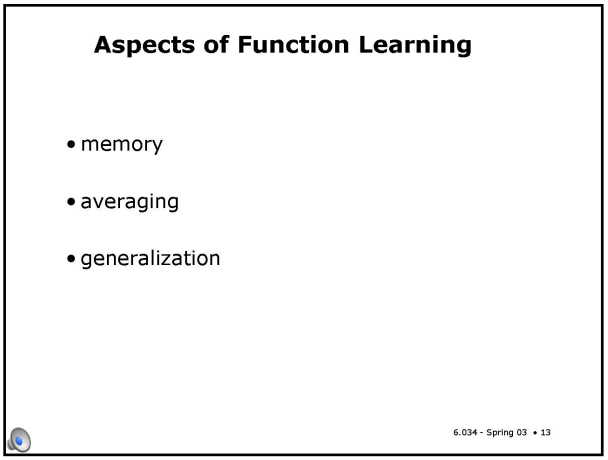


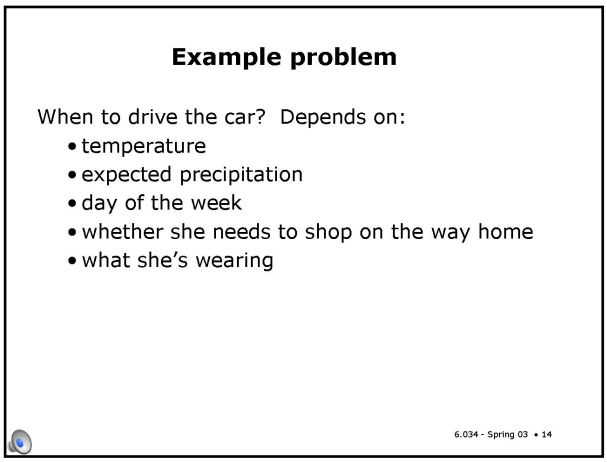



**Slide 2.1.12**

- Learning how to pronounce words can be thought of as finding a function from letters to sounds.
- Learning how to throw a ball can be thought of as finding a function from target locations to joint torques.
- Learning to recognize handwritten characters can be thought of as finding a function from collections of image pixels to letters.
- Learning to diagnose diseases can be thought of as finding a function from lab test results to disease categories.

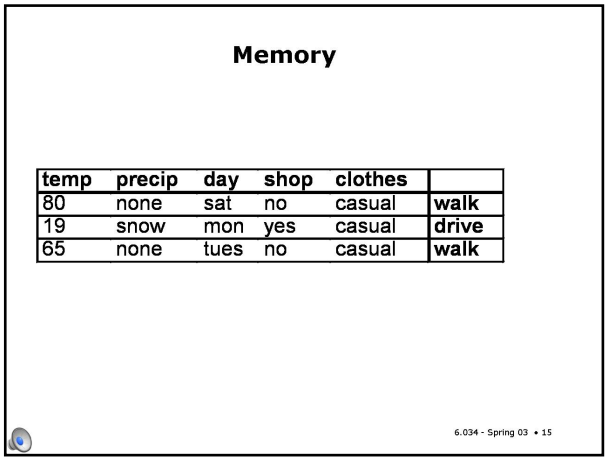
**Slide 2.1.13**
The problem of learning a function from examples, is complicated. You can think of at least three different problems being involved: memory, averaging, and generalization. We'll look at each of these problems, by example.

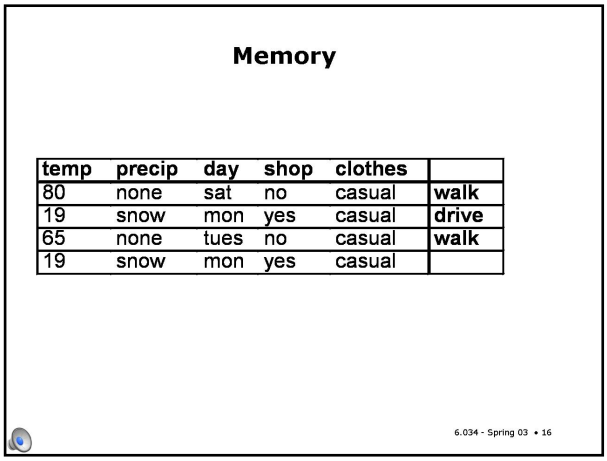



**Slide 2.1.14**
We'll do this in the context of a silly example problem. Imagine that I'm trying predict whether my neighbor is going to drive into work tomorrow, so I can ask for a ride. Whether she drives into work seems to depend on the following attributes of the day: the temperature, what kind of precipitation is expected, the day of the week, whether she needs to shop on the way home, and what she's wearing.

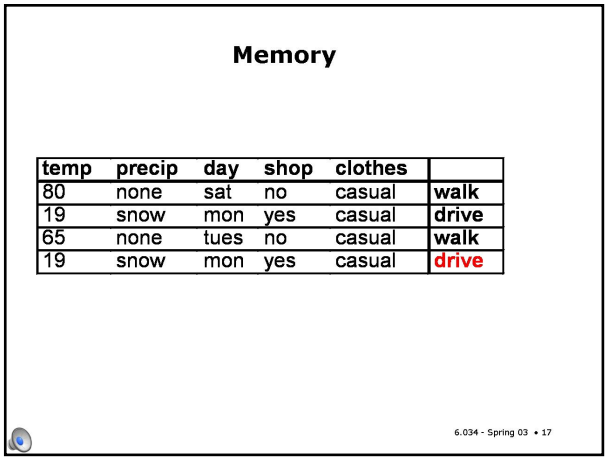
**Slide 2.1.15**
Okay. Let's say we observe our neighbor on three days, which are described in the table, which specifies the properties of the days and whether or not the neighbor walked or drove.

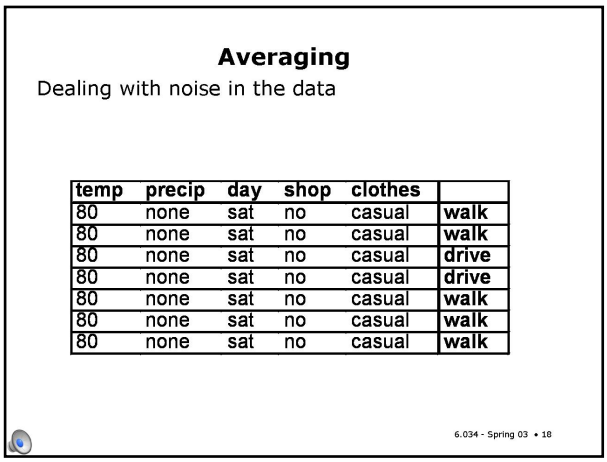





**Slide 2.1.16**
Now, we find ourselves on a snowy 19-degree Monday, when the neighbor is wearing casual clothes and going shopping. Do you think she's going to drive?

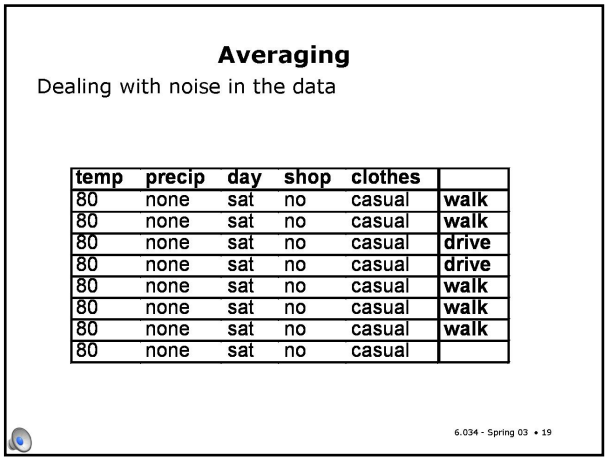
**Slide 2.1.17**
The standard answer in this case is "yes". This day is just like one of the ones we've seen before, and so it seems like a good bet to predict "yes." This is about the most rudimentary form of learning, which is just to memorize the things you've seen before. Still, it can help you do a better job in the future, if those same cases arise again.

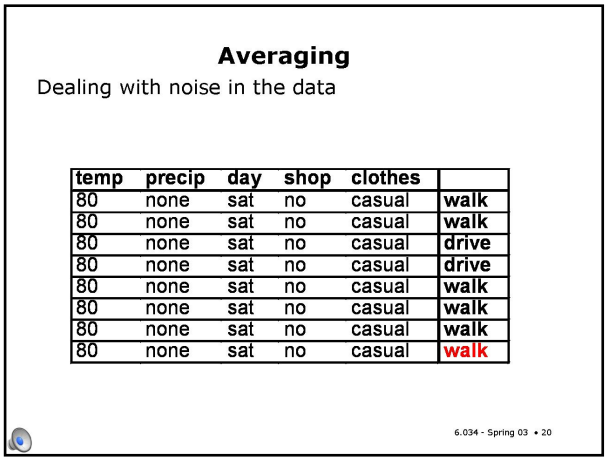



**Slide 2.1.18**
Things are not always as easy as they were in the previous case. What if you get this set of data?

**Slide 2.1.19**
Now, we ask you to predict what's going to happen. You've certainly seen this case before. But the problem is that it has had different answers. Our neighbor is not entirely reliable.

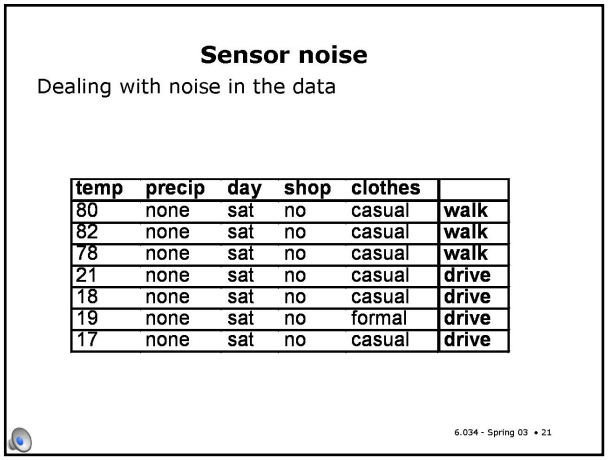


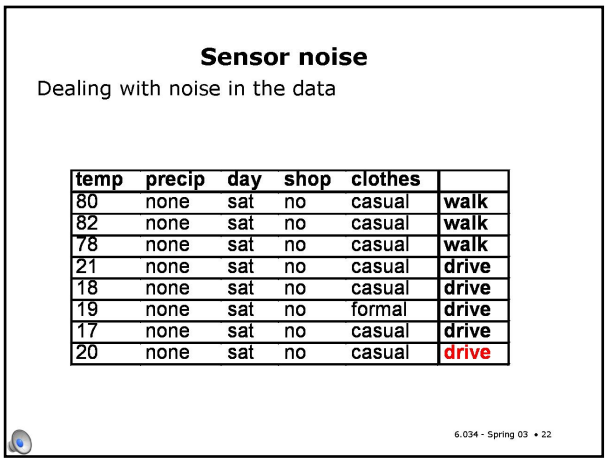



**Slide 2.1.20**
There are a couple of plausible strategies here. One would be to predict the majority outcome. The neighbor walked more times than she drove in this situation, so we might predict "walk". Another option would be to decline to commit to one prediction, but to generate a probability instead. You might answer that you think, with probability 5/7, she will walk.

In this course, we'll give a treatment of machine learning that doesn't really rely on probability. But in many cases, the algorithms and concepts we talk about have probabilistic foundations, and there are many more sophisticated techniques that rely on a deep understanding of probability. So, if you like machine learning, take probability, and then take the machine learning class!

**Slide 2.1.21**
Here's another situation in which noise is an issue. This time, the noise has corrupted our descriptions of the situation (our thermometer is somewhat unreliable, as is our ability to assess the formality of the neighbor's clothing, given her peculiar taste).

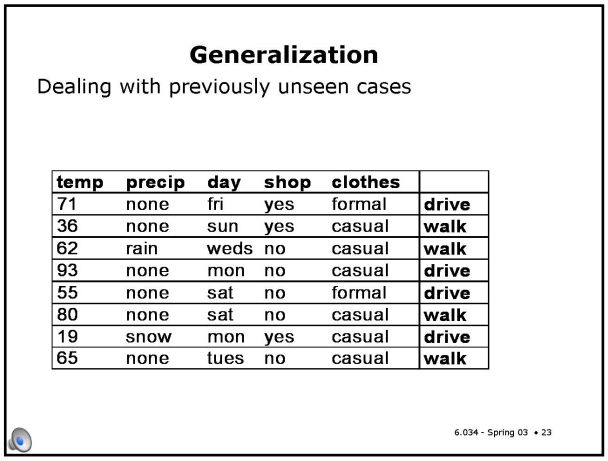


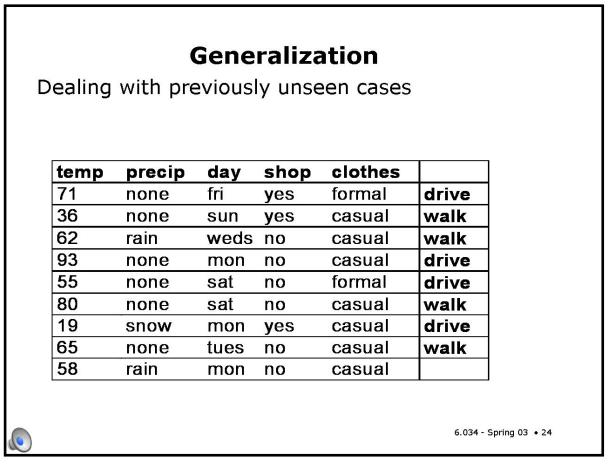




**Slide 2.1.22**
In this case, we'd like to recognize that the day that we're asked to do the prediction for is apparently similar to cases we've seen before. It is ultimately impossible to tell whether this day is *really* different from those other days, or whether our sensors are just acting up. We'll just have to accept that uncertainty.

So, we'll have to treat this as an instance of the more general problem of generalization.

**Slide 2.1.23**
Take a good look at this data set.






**Slide 2.1.24**
Now, consider this day. It's 58 degrees and raining on a Monday. The neighbor is wearing casual clothing and doesn't need to shop. Will she walk or drive?

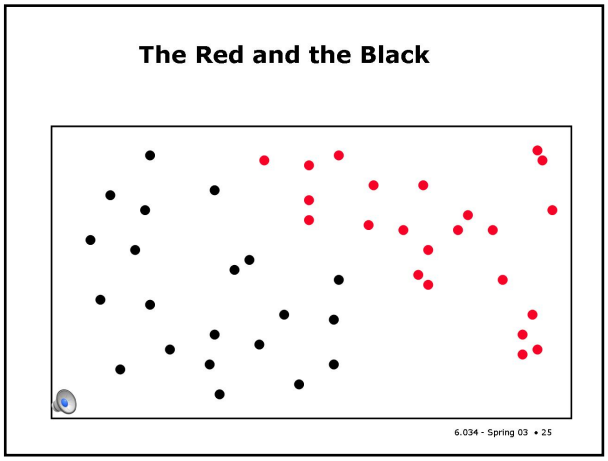
The first thing to observe is that there's no obviously right answer. We have never seen this case before. We could just throw up our hands and decline to make a prediction. But ultimately, we have to decide whether or not to call the neighbor. So, we might fall back on some assumptions about the domain. We might assume, for instance, that there's a kind of *smoothness* property: similar situations will tend to have similar categories.

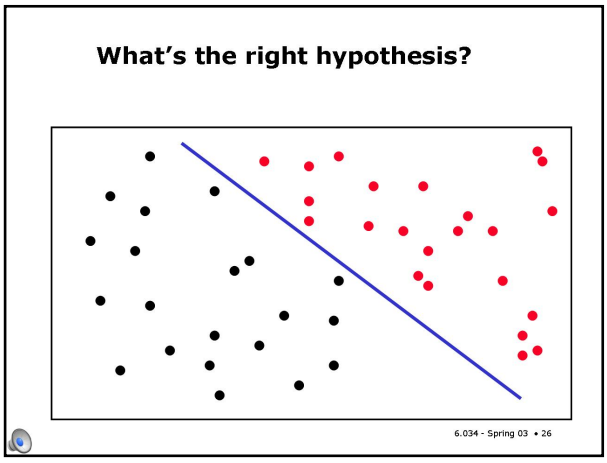
You might plausibly make any of the following arguments:

- She's going to walk because it's raining today and the only other time it rained, she walked.
- She's going to drive because she has always driven on Mondays.
- She's going to walk because she only drives if she is wearing formal clothes, or if the temperature is above 90 or below 20.

The question of which one to choose is hard. It's one of the deep underlying problems of machine learning. We'll look at some examples to try to get more intuition, then come back and revisit it a bit more formally in the next section.

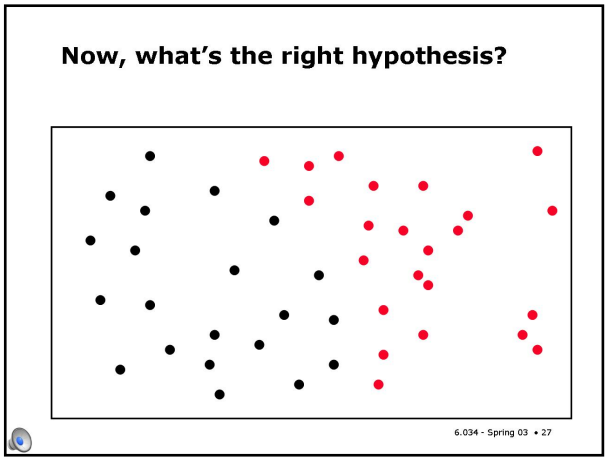
**Slide 2.1.25**
Imagine that you were given all these points, and you needed to guess a function of their x, y coordinates that would have one output for the red ones and a different output for the black ones.

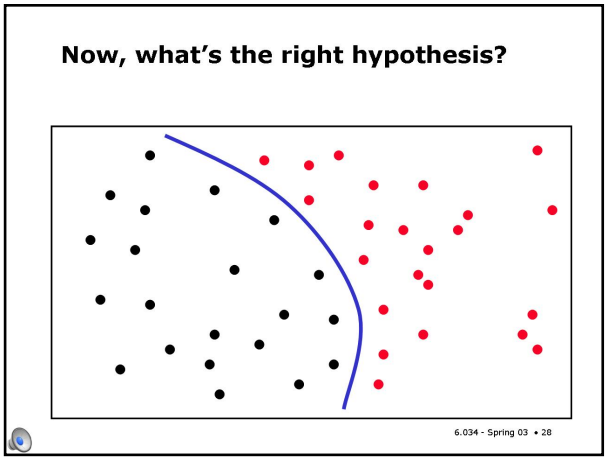



**Slide 2.1.26**
In this case, it seems like you could do pretty well by defining a line that separates the two classes.

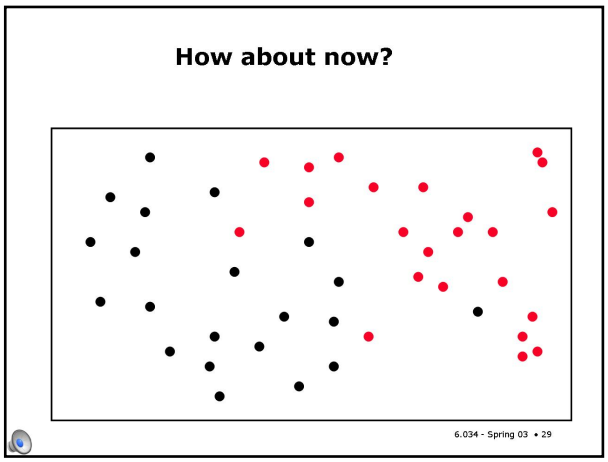
**Slide 2.1.27**
Now, what if we have a slightly different configuration of points? We can't divide them conveniently with a line.

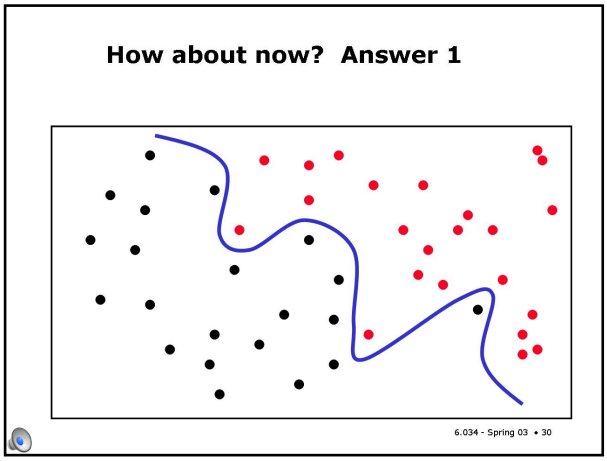



**Slide 2.1.28**
But this parabola-like curve seems like it might be a reasonable separator.

**Slide 2.1.29**
Now, what? This seems much trickier. There are at least a couple of reasonable kinds of answers.

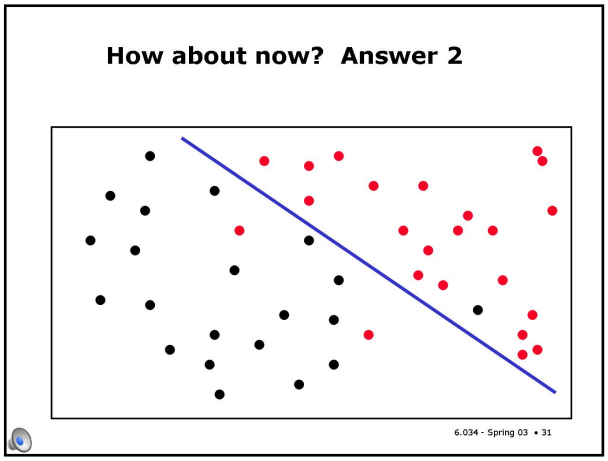


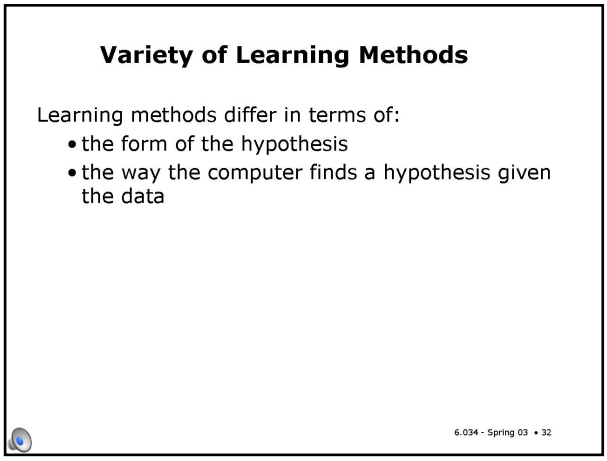




**Slide 2.1.30**
Here's an answer. It successfully separates the reds from the blacks. But there's something sort of unsatisfactory about it. Most people have an intuition that it's too complicated.

**Slide 2.1.31**
Here's another answer. It's not a perfect separator; it gets some of the points wrong. But it's prettier.

In general, we will always be faced with making trade-offs between hypotheses that account for the data perfectly and hypotheses that are, in some sense, simple. There are some mathematical ways to understand these trade-offs; we'll avoid the math, but try to show you, intuitively, how they come up and what to do about them.

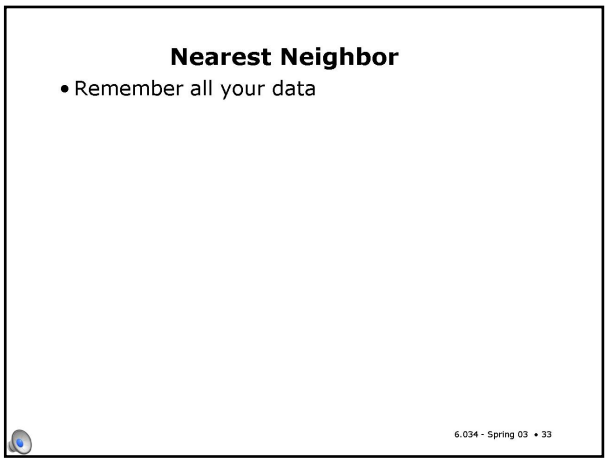


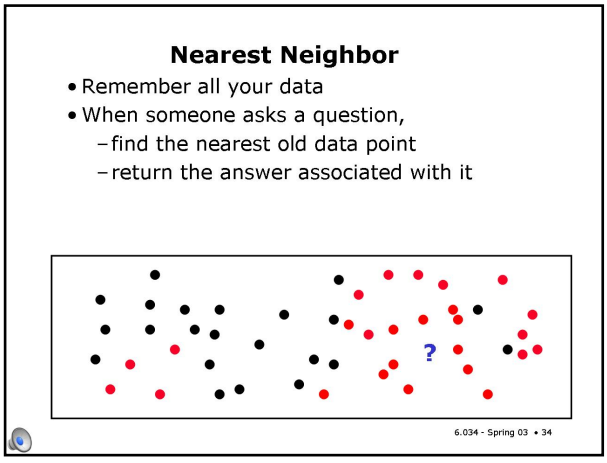




**Slide 2.1.32**
There are lots and lots of different kinds of learning algorithms. I want to show you cartoon versions of a couple of them first, and then we'll get down to the business of understanding them at a detailed algorithmic level.

The learning methods differ both in terms of the kinds of hypotheses they work with and the algorithms they use to find a good hypothesis given the data.

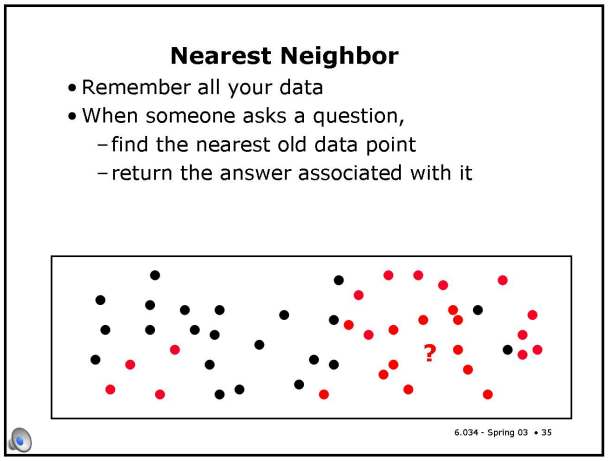
**Slide 2.1.33**
Here's the most simple-minded of all learning algorithms (and my personal favorite (which might imply something about me!)). It's called nearest neighbor (or lazy learning). You simply remember all the examples you've ever seen.

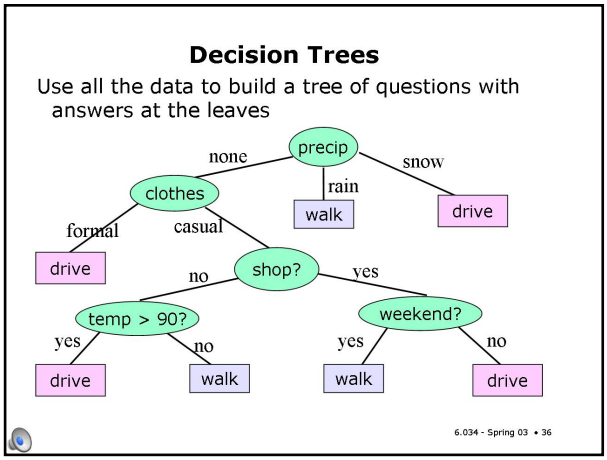



**Slide 2.1.34**
When someone asks you a question, you simply find the existing point that's nearest the one you were asked about, and return its class.

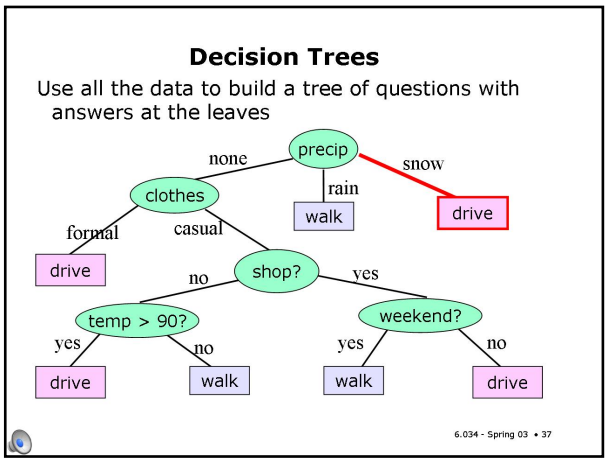
**Slide 2.1.35**
So, in this case, the nearest point to the query is red, so we'd return red.

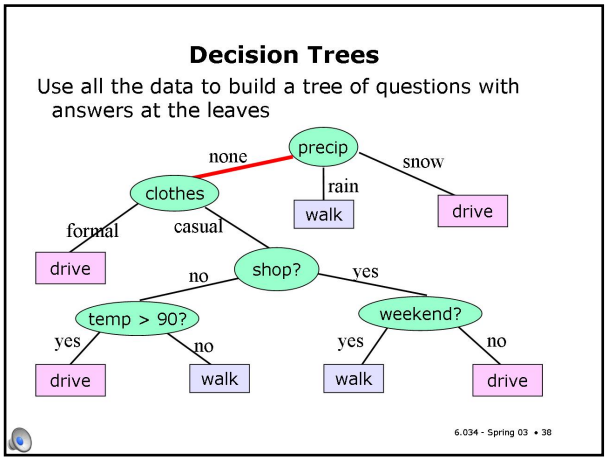



**Slide 2.1.36**
Another well-loved learning algorithm makes hypotheses in the form of decision trees. In a decision tree, each node represents a question, and the arcs represent possible answers. We can use this decision tree to find out what prediction we should make in the drive/walk problem.

**Slide 2.1.37**
We'd start by asking what the current precipitation is. If it's snow, we just stop asking questions and drive.

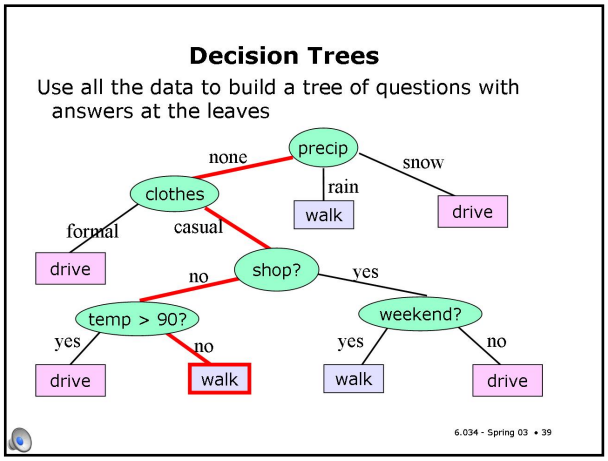


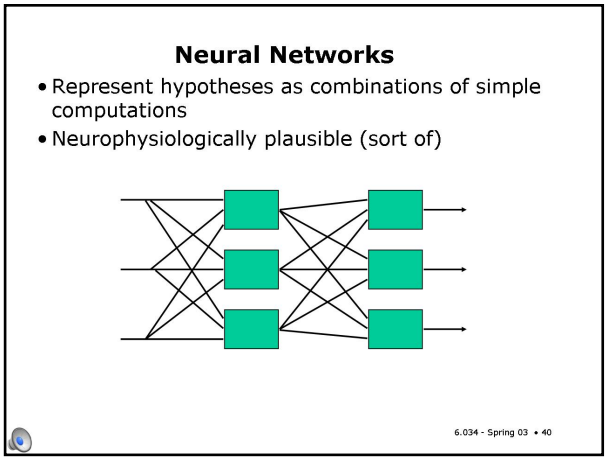




**Slide 2.1.38**
If there's no precipitation, then we have to ask what kind of clothes the neighbor is wearing. If they're formal, she'll drive. If not, we have to ask another question.

**Slide 2.1.39**
We can always continue asking and answering questions until we get to a leaf node of the tree; the leaf will always contain a prediction.

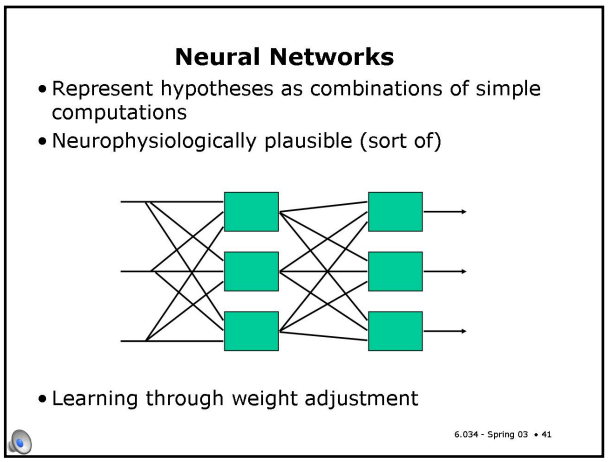
Hypotheses like this are nice for a variety of reasons. One important one is that they're relatively easily interpretable by humans. So, in some cases, we run a learning algorithm on some data and then show the results to experts in the area (astronomers, physicians), and they find that the learning algorithm has found some regularities in their data that are of real interest to them.

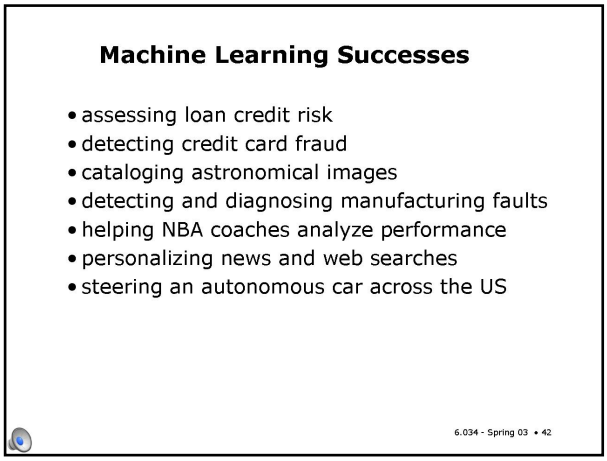



**Slide 2.1.40**
Most people have heard of neural networks. They can represent complicated hypotheses in high-dimensional continuous spaces. They are attractive as a computational model because they are composed of many small computing units. They were motiviated by the structure of neural systems in parts of the brain. Now it is understood that they are not an exact model of neural function, but they have proved to be useful from a purely practical perspective.

**Slide 2.1.41**
In neural networks, the individual units do a simple, fixed computation that combines the values coming into them with "weights" on the arcs. The learning process consists of adjusting the weights on the arcs in reaction to errors made by the network.



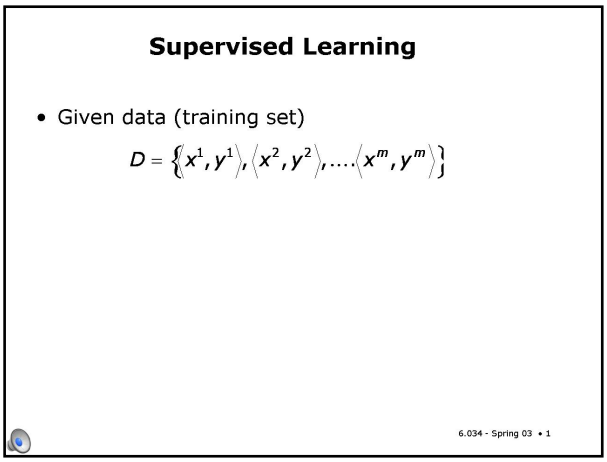

**Slide 2.1.42**
Machine learning methods have been successfully fielded in a variety of applications, including

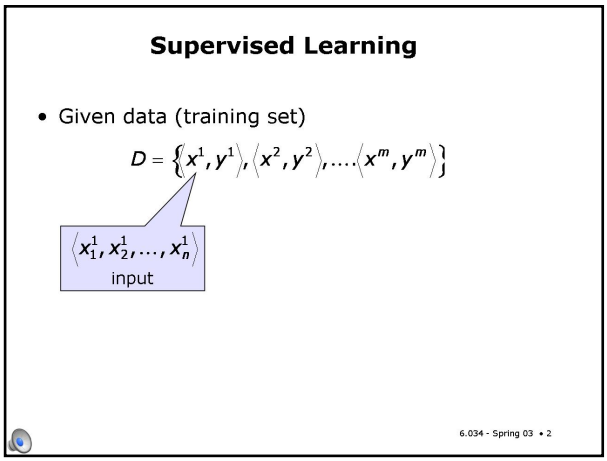
- assessing loan credit risk
- detecting credit card fraud
- cataloging astronomical images
- deciding which catalogs to send to your house
- helping NBA coaches analyze players' performance
- personalizing news and web searches
- steering an autonomous car across the US

Now, we'll roll up our sleeves, and see how all this is done.

# 6.034 Notes: Section 2.2

**Slide 2.2.1**
A supervised learning problem is made up of a number of ingredients. The first is the data (sometimes called the training set). The data, D, is a set of m input-output pairs. We will write the ith element of the data set as $x^i$, $y^i$. In the context of our old running example, an $x^i$ would be a vector of values, one for each of the input features, and a $y^i$ would be an output value (walk or drive).

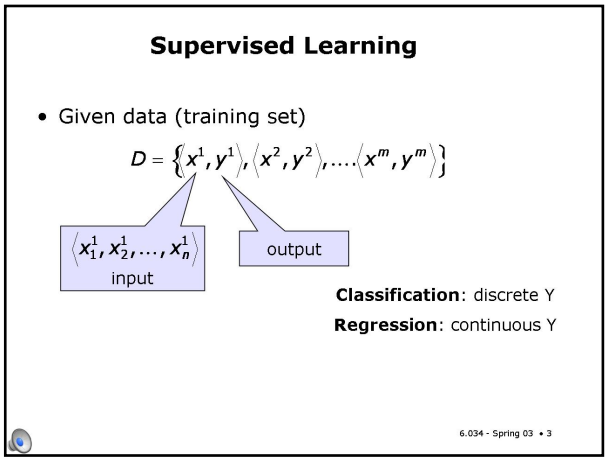


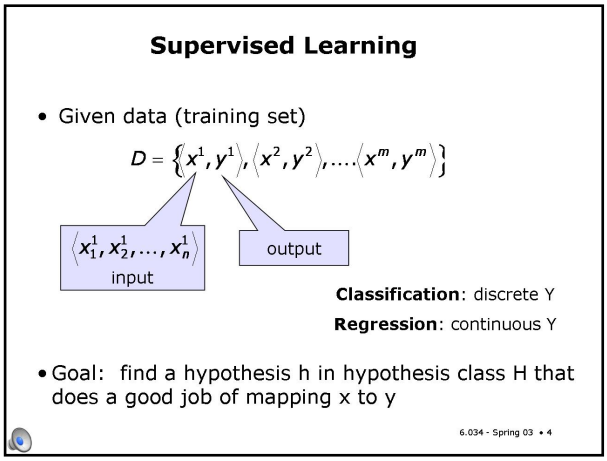




**Slide 2.2.2**
Each $x^i$ is a vector of n values. We'll write $x^i_j$ for the jth feature of the ith input-output pair. We'll consider different kinds of features. Sometimes we'll restrict ourselves to the case where the features are only 0s and 1s. Other times, we'll let them be choosen from a set of discrete elements (like "snow", "rain", "none"). And, still other times, we'll let them be real values, like temperature, or weight.

**Slide 2.2.3**
Similarly, the output, $y^i$, might be a boolean, a member of a discrete set, or a real value. For a particular problem, the $y^i$ will all be of a particular type.

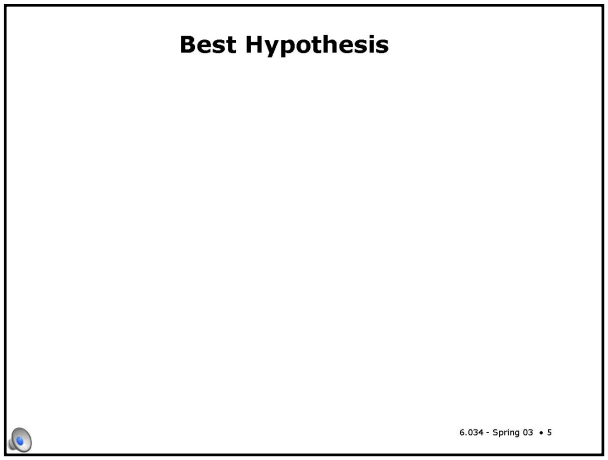
When $y^i$ is a boolean, or a member of a discrete set, we will call the problem a **classification** problem. When $y^i$ is real-valued, we call this a **regression** problem.

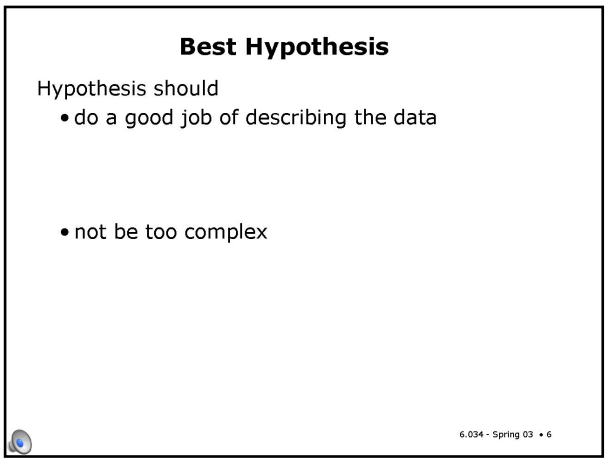





**Slide 2.2.4**
Now, the goal of learning will be to find a hypothesis, h, that does a good job of describing the relationship between the inputs and the outputs. So, a part of the problem specification is capital H, the hypothesis class. H is the set of possible hypotheses that we want our learning algorithm to choose from. It might be something like decision trees with 6 nodes, or lines in two-dimensional space, or neural networks with 3 components.

**Slide 2.2.5**
So, now we have a bunch of data, and a class of possible answers (hypotheses) and we're supposed to return the best one. In order to do that, we need to know exactly what is meant by "best".

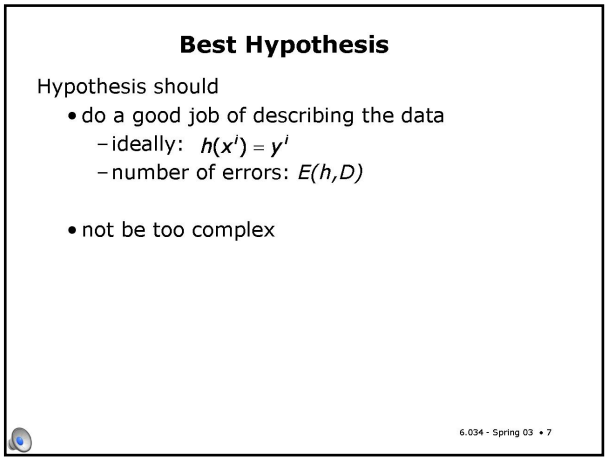


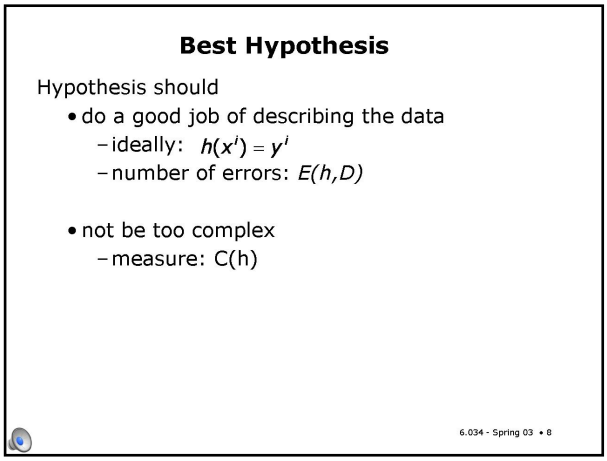




**Slide 2.2.6**
There are typically two components to the notion of best. The hypothesis should do a good job of describing the data and it shouldn't be too complicated.

**Slide 2.2.7**
Ideally, we would like to find a hypothesis h such that, for all data points i, $h(x^i) = y^i$. We will not always be able to (or maybe even want to) achieve this, so perhaps it will only be true for most of the data points, or the equality will be weakened into "not too far from".

We can sometimes develop a measure of the "error" of a hypothesis to the data, written E(h, D). It might be the number of points that are miscategorized, for example.

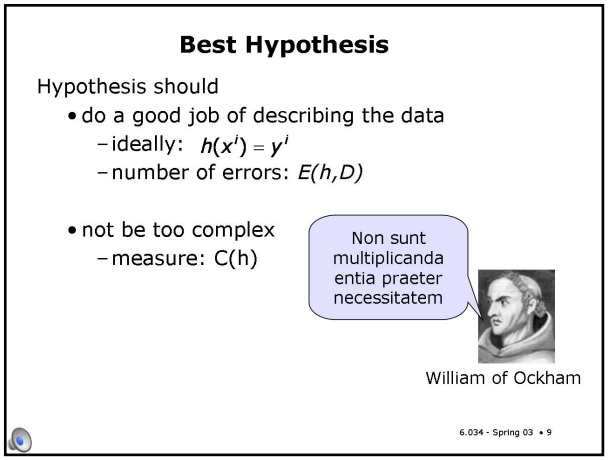


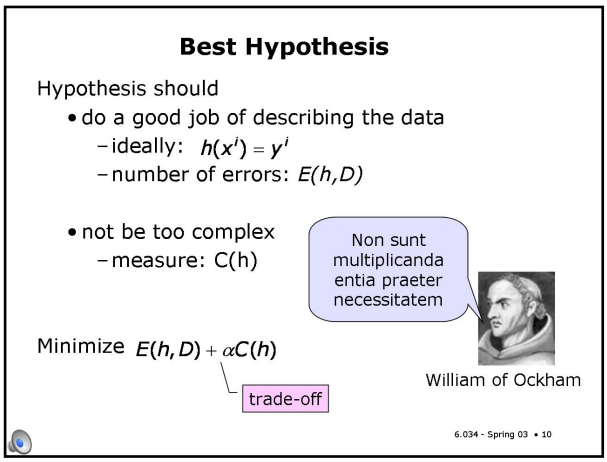



**Slide 2.2.8**
Another issue is the complexity of the hypothesis. We can measure the complexity of a decision tree by the number of nodes it has, or a line by how bendy it is. In general, we'll define a measure of the complexity of hypotheses in H, C(h).

**Slide 2.2.9**
Why do we care about hypothesis complexity? We have an intuitive sense that, all things being equal, simpler hyptheses are better. Why is that? There are lots of statistical and philosophical and information-theoretic arguments in favor of simplicity.

So, for now, let's just make recourse to William of Ockham, a 14th century Franciscan theologian, logician, and heretic. He is famous for "Ockham's Razor", or the principle of parsimony: "Non sunt multiplicanda entia praeter necessitatem." That is, "entities are not to be multiplied beyond necessity". People interpret this to mean that we should always adopt the simplest satisfactory hypothesis.

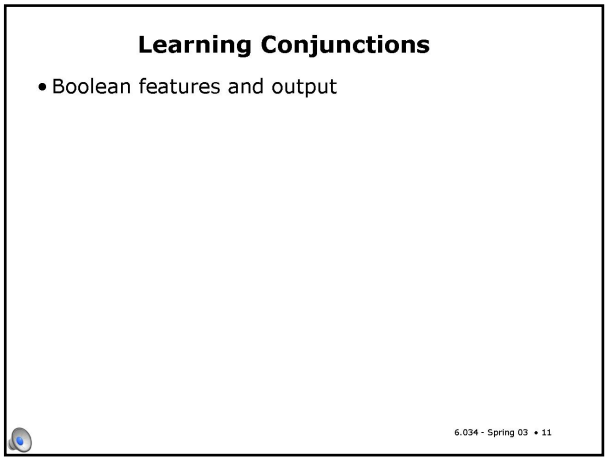


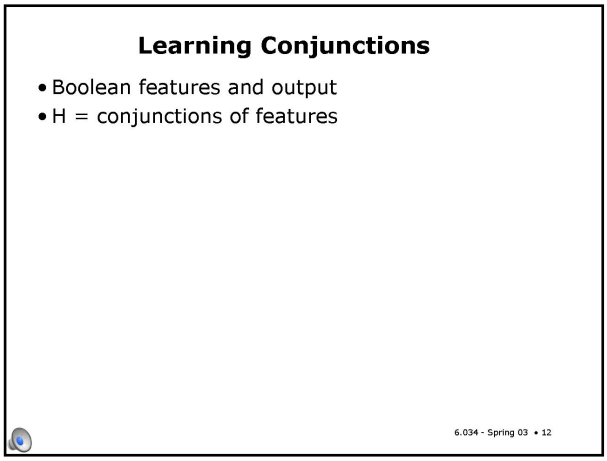




**Slide 2.2.10**
So, given a data set, D, a hypothesis class H, a goodness-of-fit function E, and a complexity measure C, our job will be to find the h in H that minimizes E(h, D) + alpha * C(h), where alpha is a number that we can vary to change how much we value fitting the data well versus having a complex hypothesis.

We won't, in general, be able to do this efficiently, so, as usual, we'll have to make a lot of approximations and short-cuts. In fact, most learning algorithms aren't described exactly this way. But it's the underlying principle behind most of what we do.

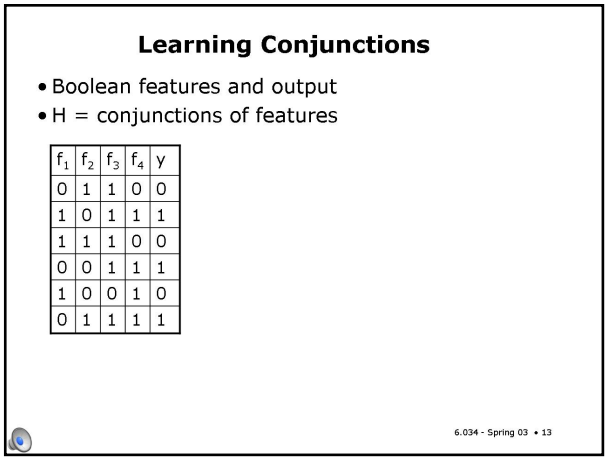
**Slide 2.2.11**
Let's start with a very simple problem, in which all of the input features are Boolean (we'll represent them with 0's and 1's) and the desired output is also Boolean.

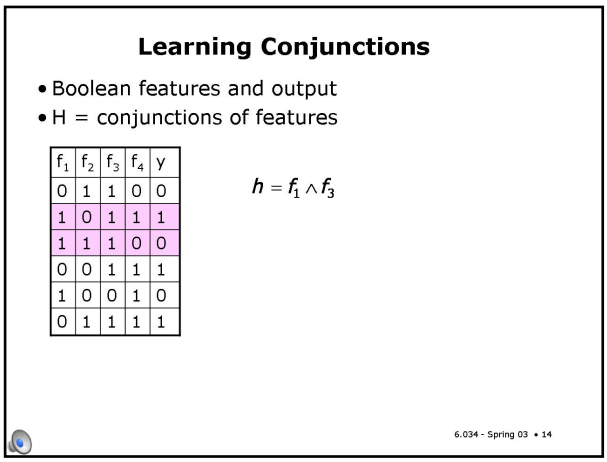





**Slide 2.2.12**
Our hypothesis class will be conjunctions of the input features.

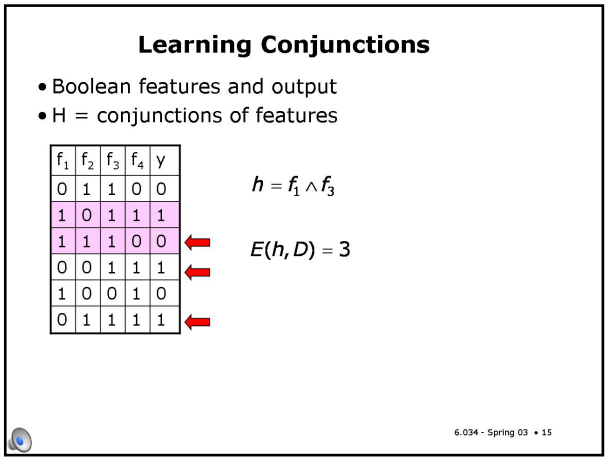
**Slide 2.2.13**
Here's an example data set. It is described using 4 features, which we'll call $f_1$, $f_2$, $f_3$, and $f_4$.

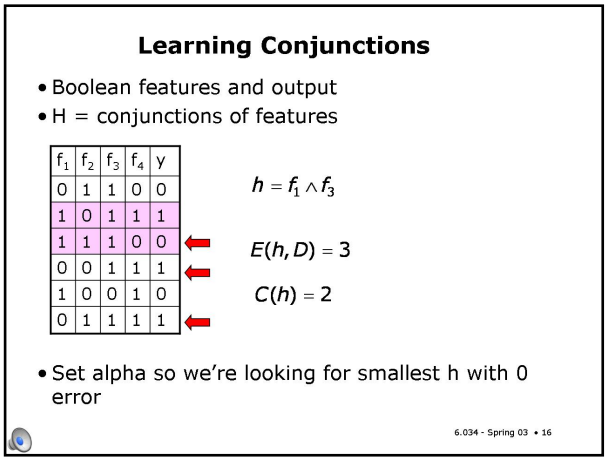



**Slide 2.2.14**
So, to understand the hypothesis space let's consider the hypothesis $f_1 \wedge f_3$. It would be true on examples 2 and 3, and false on all the rest.

**Slide 2.2.15**
We will measure the error of our hypothesis as the number of examples it gets wrong. It marks one negative example as positive, and two positives as negative. So, the error of the hypothesis $f_1 \wedge f_3$ on this data set would be 3.

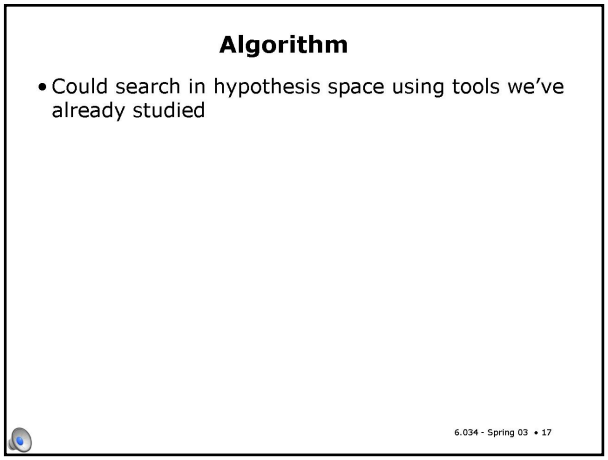


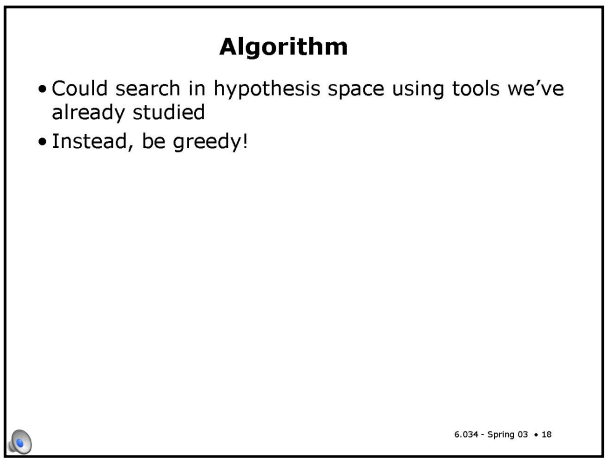



**Slide 2.2.16**
Finally, we'll measure the complexity of our hypothesis by the number of features mentioned in the conjunction. So the hypothesis $f_1$ ^ $f_3$ has complexity 2.

Furthermore, let's assume that alpha is set to be very small, so that our primary goal is to minimize error, but, errors being equal, we'd like to have the smallest conjunction.

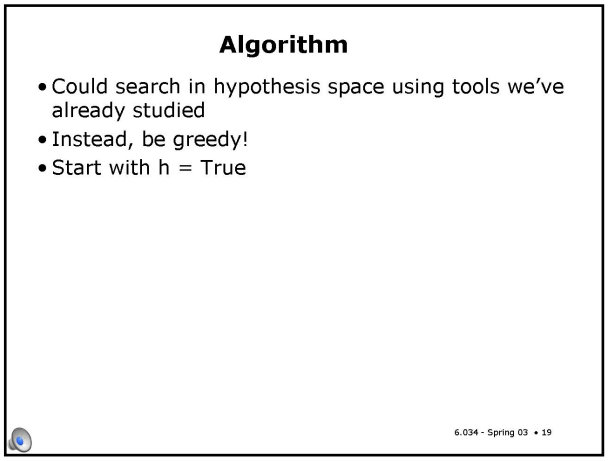
**Slide 2.2.17**
There's now an obvious way to proceed. We could do a general-purpose search in the space of hypotheses, looking for the one that minimizes the cost function. In this case, that might work out okay, since the problem is very small. But in general, we'll want to work in domains with many more features and much more complex hypothesis spaces, making general-purpose search infeasible.




**Slide 2.2.18**
Instead, we'll be greedy! (When I talk about this and other algorithms being greedy, I'll mean something slightly different than what we meant when we talked about "greedy search". That was a particular instance of a more general idea of greedy algorithm. In greedy algorithms, in general, we build up a solution to a complex problem by adding the piece to our solution that looks like it will help the most, based on a partial solution we already have. This will not, typically, result in an optimal solution, but it usually works out reasonably well, and is the only plausible option in many cases (because trying out all possible solutions would be much too expensive)).

**Slide 2.2.19**
We'll start out with our hypothesis set to True (that's the empty conjunction). Usually, it will make some errors. Our goal will be to add as few elements to the conjuction as necessary to make no errors.



## Algorithm

- Could search in hypothesis space using tools we've already studied
- Instead, be greedy!
- Start with h = True
- All errors are on negative examples
- On each step, add conjunct that rules out most new negatives (without excluding positives)

6.034 - Spring 03 • 20

**Slide 2.2.20**
Notice that, because we've started with the hypothesis equal to True, all of our errors are on negative examples. So, one greedy strategy would be to add the feature into our conjunction that rules out as many negative examples as possible without ruling out any positive examples.

**Slide 2.2.21**
Here is pseudo-code for our algorithm. We start with N equal to all the negative examples and the hypothesis equal to true. Then we loop, adding conjuncts that rule out negative examples, until N is empty.

## Pseudo-Code

```
N = negative examples in D
h = True
Loop until N is empty
```

6.034 - Spring 03 • 21

## Pseudo-Code

```
N = negative examples in D
h = True
Loop until N is empty
   For every feature j that does not have value 0 on
    any positive examples
      n_j := number of examples in N
            for which f_j = 0
      j* := j for which n_j is maximized
      h := h ^ f_j*
      N := N - examples in N for which f_j = 0
   If no such feature found, fail
```

6.034 - Spring 03 • 22

**Slide 2.2.22**
Inside the loop, we consider every feature that would not rule out any positive examples. We pick the one that rules out the most negative examples (here's where our greed kicks in), conjoin it to h, and remove the examples that it excludes from N.

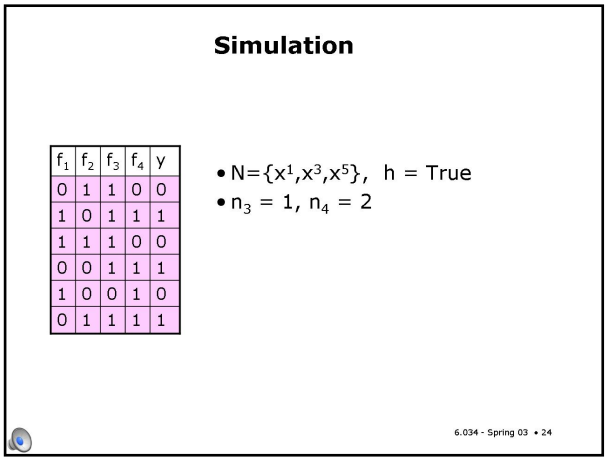
**Slide 2.2.23**
Let's simulate the algorithm on our data. We start with N equal to $x^1$, $x^3$, and $x^5$, which are the negative examples. And h starts as True. We'll cover all the examples that the hypothesis makes true pink.

## Simulation

| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 |

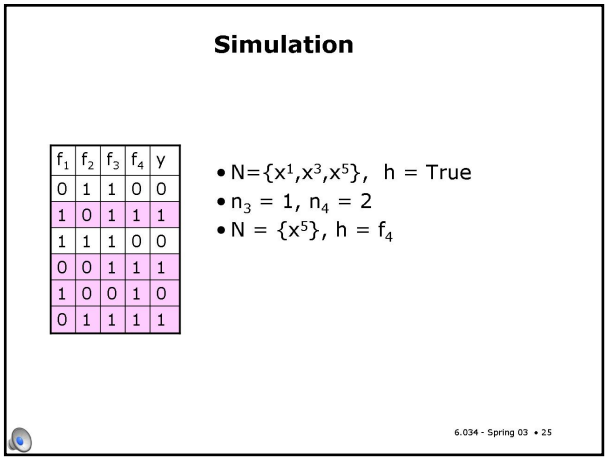
- $N=\{x^1,x^3,x^5\}$, h = True

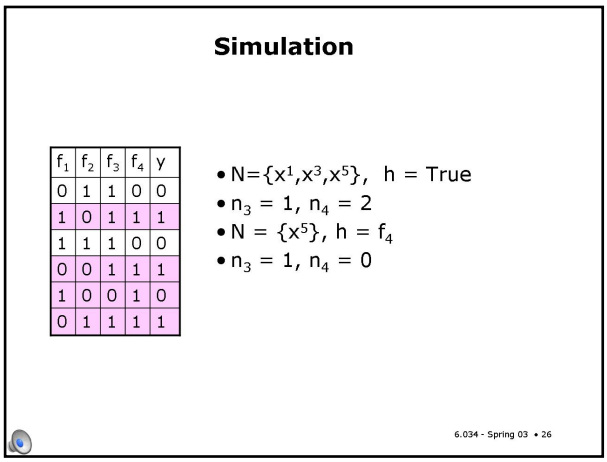
6.034 - Spring 03 • 23




**Slide 2.2.24**
Now, we consider all the the features that would not exclude any positive examples. Those are features $f_3$ and $f_4$. $f_3$ would exclude 1 negative example; $f_4$ would exclude 2. So we pick $f_4$.

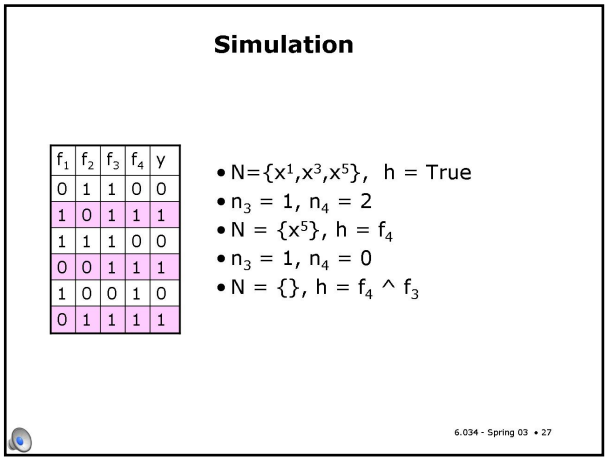
**Slide 2.2.25**
Now we remove the examples from N that are ruled out by $f_4$ and add $f_4$ to h.

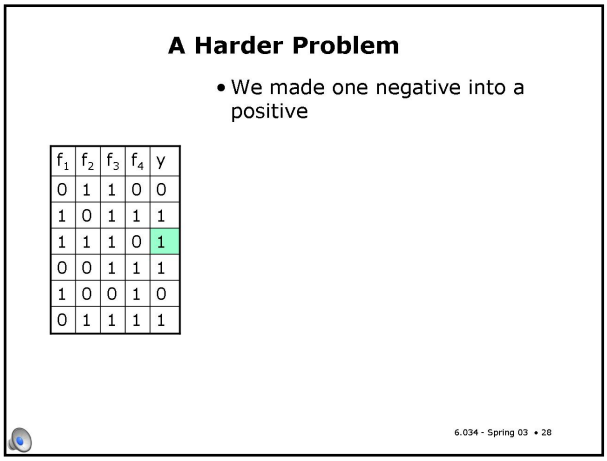



**Slide 2.2.26**
Now, based on the new N, $n_3 = 1$ and $n_4 = 0$. So we pick $f_3$.

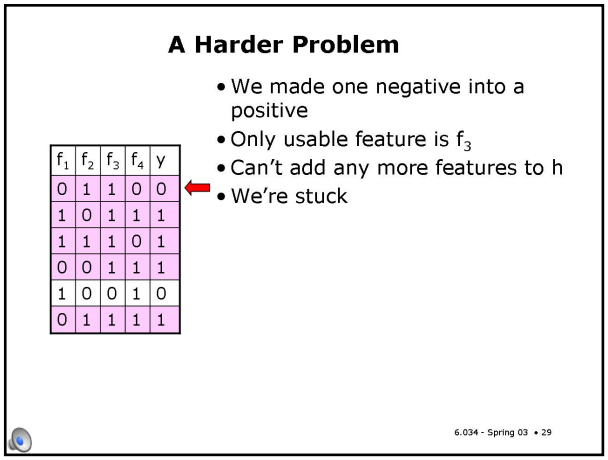
**Slide 2.2.27**
Because $f_3$ rules out the last remaining negative example, we're done!

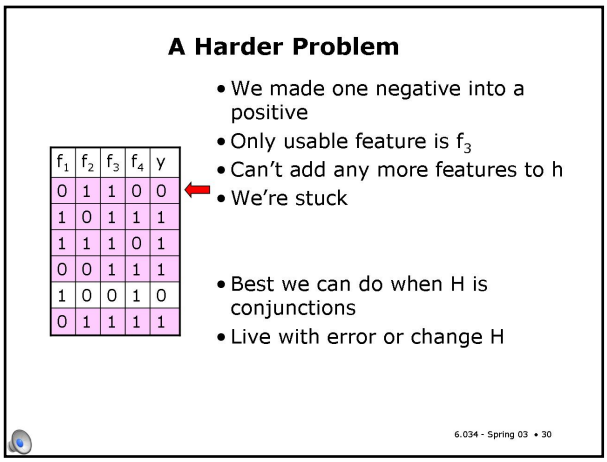





**Slide 2.2.28**
What if we have this data set? In which we just made one negative example into a positive?

**Slide 2.2.29**
If we simulate the algorithm, we will arrive at the situation in which we still have elements in N, but there are no features left that do not exclude positive examples. So we're stuck.



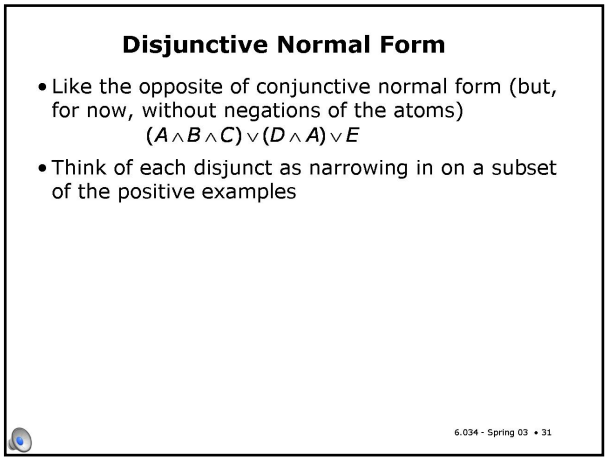




**Slide 2.2.30**
What's going on? The problem is that this hypothesis class simply can't represent the data we have with no errors. So now we have a choice: we can accept the hypothesis we've got, or we can increase the size of the hypothesis class. In practice, which one you should do often depends on knowledge of the domain. But the fact is that pure conjunctions is a very limiting hypothesis class. So let's try something a little fancier.

**Slide 2.2.31**
You can kind of think of a conjunction as narrowing down on a small part of the space. And if all the positive examples can be corralled into one group this way, everything is fine. But for some concepts, it might be necessary to have multiple groups.

This corresponds to a form of Boolean hypothesis that's fairly easy to think about: disjunctive normal form. Disjunctive normal form is kind of like CNF, only this time it has the form (A and B and C) or (D and E and F)

As a hypothesis class, it's like finding multiple groups of positive examples within the negative space.



## Disjunctive Normal Form

- Like the opposite of conjunctive normal form (but, for now, without negations of the atoms)
  $(A \wedge B \wedge C) \vee (D \wedge A) \vee E$
- Think of each disjunct as narrowing in on a subset of the positive examples

| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 |

6.034 - Spring 03 • 32

**Slide 2.2.32**
So, let's look at our harder data set and see if we can find a good hypothesis in DNF.

**Slide 2.2.33**
It's pretty to see that one way to describe it is $f_3$^$f_4$ v $f_1$^$f_2$. Now let's look at an algorithm for finding it.

## Disjunctive Normal Form

- Like the opposite of conjunctive normal form (but, for now, without negations of the atoms)
  $(A \wedge B \wedge C) \vee (D \wedge A) \vee E$
- Think of each disjunct as narrowing in on a subset of the positive examples

| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 |

$(f_3 \wedge f_4) \vee (f_1 \wedge f_3)$

6.034 - Spring 03 • 33

## Learning DNF

- Let H be DNF expressions
- C(h) : number of mentions of features

$C((f_3 \wedge f_4) \vee (f_1 \wedge f_2)) = 4$

6.034 - Spring 03 • 34

**Slide 2.2.34**
We'll let our hypothesis class be expressions in disjunctive normal form. What should our measure of complexity be? One reasonable choice is the total number of mentions of features (so $f_3$^$f_4$ v $f_1$^$f_2$ would have complexity 4).

**Slide 2.2.35**
Now, searching in this space would be much harder than searching in the space of conjunctions. So we'll use another greedy algorithm.

## Learning DNF

- Let H be DNF expressions
- C(h) : number of mentions of features

$C((f_3 \wedge f_4) \vee (f_1 \wedge f_2)) = 4$

- Really hard to search this space, so be greedy again!

6.034 - Spring 03 • 35

## Learning DNF

- Let H be DNF expressions
- C(h) : number of mentions of features

$$C((f_3 \wedge f_4) \vee (f_1 \wedge f_2)) = 4$$

- Really hard to search this space, so be greedy again!
- A conjunction **covers** an example if all of the features mentioned in the conjunction are true in the example

6.034 - Spring 03 • 36

**Slide 2.2.36**
We'll say a conjunction **covers** an example if all of the features mentioned in the conjunction are true in the example.

**Slide 2.2.37**
Here's pseudocode for the algorithm. It has two main loops. The inner loop constructs a conjunction (much like our previous algorithm). The outer loop constructs multiple conjunctions and disjoins them.

The idea is that each disjunct will *cover* or account for some subset of the positive examples. So in the outer loop, we make a conjunction that includes some positive examples and no negative examples, and add it to our hypothesis. We keep doing that until no more positive examples remain to be covered.

## Algorithm

```
P = set of all positive examples
h = False
Loop until P is empty
   r = True
   N = set of all negative examples
   Loop until N is empty
      If all features are in r, fail
      Else, select a feature f_j to add to r
         r = r ^ f_j
         N = N - examples in N for which f_j = 0
   h := h v r
   Covered := examples in P covered by r
   If Covered is empty, fail
   Else P := P - Covered
end
```

6.034 - Spring 03 • 37

## Choosing a Feature

Heuristic: $v_j = n_j^+ / \max(n_j^-, 0.001)$

6.034 - Spring 03 • 38

**Slide 2.2.38**
We also haven't said how to pick which feature to add to r. It's trickier here, because we can't restrict our attention to features that allow us to cover all possible examples. Here's one heuristic for selecting which feature to add next.

**Slide 2.2.39**
Let $v_j$ be $n^+_j$ / $n^-_j$, where $n^+_j$ is the total number of so-far uncovered positive examples that are covered by rule r ^ $f_j$ and $n^-_j$ is the total number of thus-far not-ruled-out negative examples that are covered by rule r ^ $f_j$. The intuition here is that we'd like to add features that cover a lot of positive examples and exclude a lot of negative examples, because that's our overall goal.

There's one additional problem about what to do when $n^-$ is 0. In that case, this is a really good feature, because it covers positives and doesn't admit any negatives. We'll prefer features with zero in the denominator over all others; if we have multiple such features, we'll prefer ones with bigger numerator. To make this fit easily into our framework, if the denominator is zero, we just return as a score 1 million times the numerator.

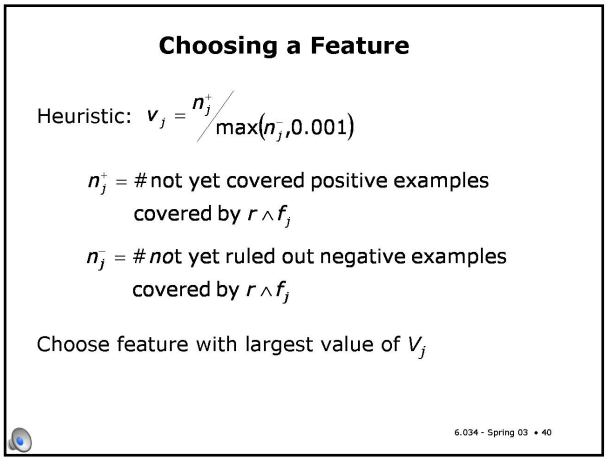
## Choosing a Feature

Heuristic: $v_j = n_j^+ / \max(n_j^-, 0.001)$

$n_j^+$ = # not yet covered positive examples covered by $r \wedge f_j$

$n_j^-$ = # not yet ruled out negative examples covered by $r \wedge f_j$

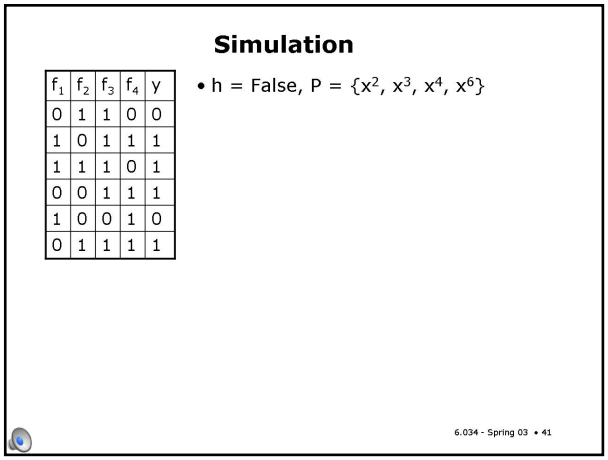
6.034 - Spring 03 • 39

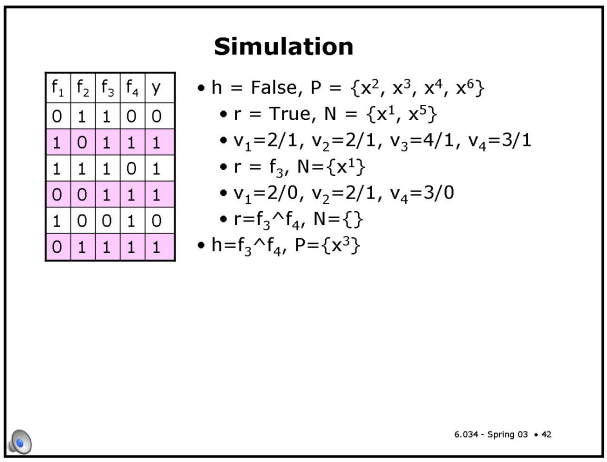



**Slide 2.2.40**
Then we can replace this line in the code with one that says: Select the feature $f_j$ with the highest value of $v_j$ to add to r. And now we have a whole algorithm.

**Slide 2.2.41**
Let's simulate it on our data set. Here's a trace of what happens. It's important to remember that when we compute $v_j$, we consider only positive and negative examples not covered so far.

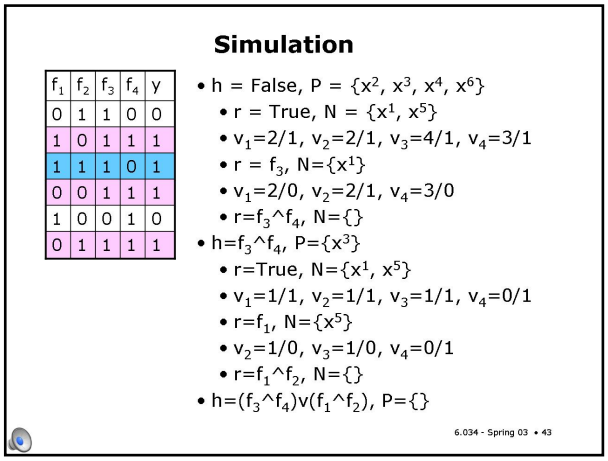
We start with h = false, and so our current hypothesis doesn't make any examples true.



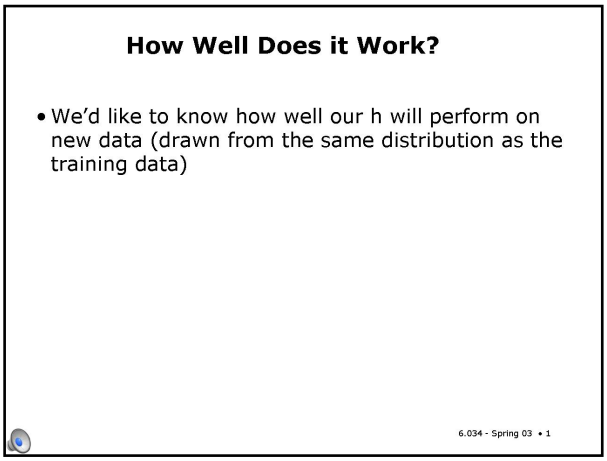

**Slide 2.2.42**
After the first iteration of the inner loop, our hypothesis covers the examples shown in pink. There's still another positive example to get.

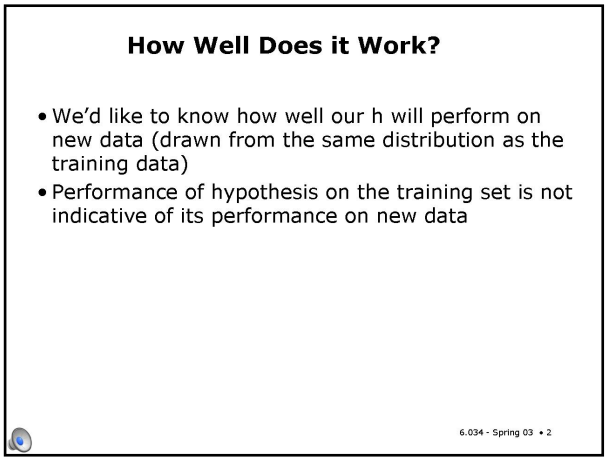
**Slide 2.2.43**
After another iteration, we add a new rule, which covers the example shown in blue. And we're done.



# 6.034 Notes: Section 2.3

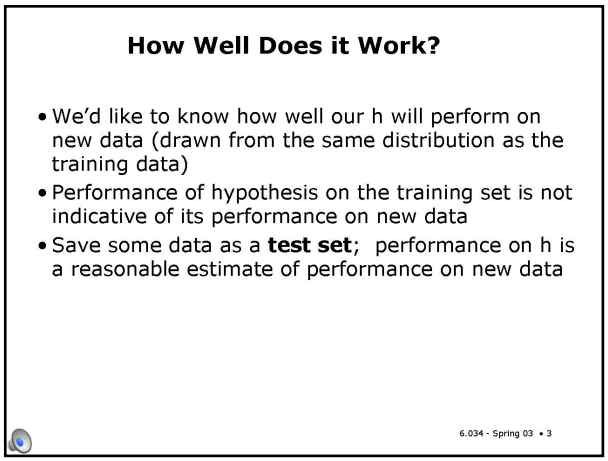
**Slide 2.3.1**
Once our learning algorithm has generated a hypothesis, we'd like to be able to estimate how well it is going to work out in the "real world". What is the real world? Well, we can't expect the hypothesis to perform well in every possible world; we *can* expect it to peform well in the world that generated the data that was used to train it.

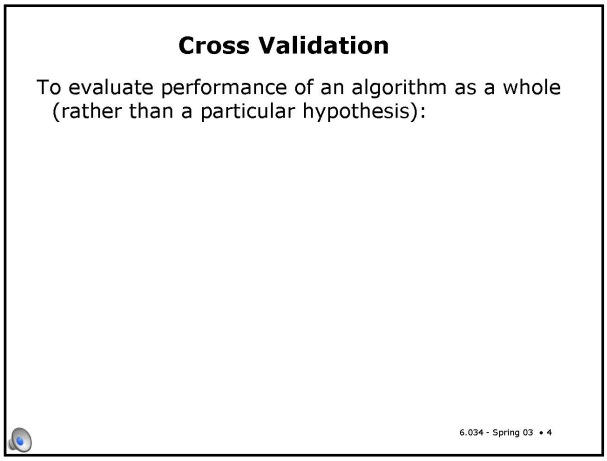



**Slide 2.3.2**
Just by looking at the training data, we can't tell how well the hypothesis will work on unseen data. We have constructed the hypothesis so that it has zero error on the training data. But unless we're wildly optimistic, we can't expect to do so well on new data.

**Slide 2.3.3**
The only way to know how well the hypothesis will do on new data is to save some of the original data in a set called the "validation set" or the "test set". The data in the validation set are not used during training. Instead, they are "held out" until the hypothesis has been derived. Then, we can test the hypothesis on the validation set and see what percentage of the cases it gets wrong. This is called "test set error".



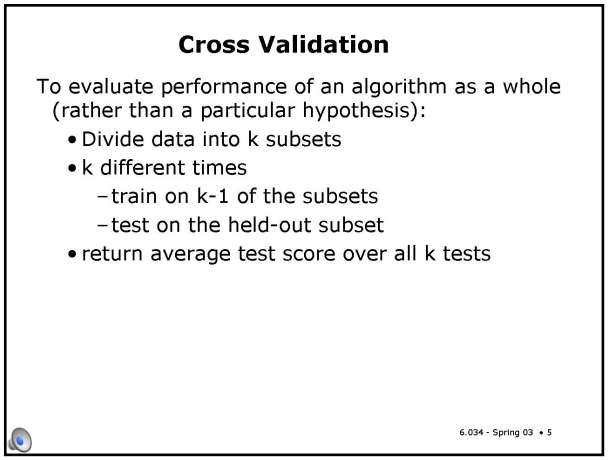


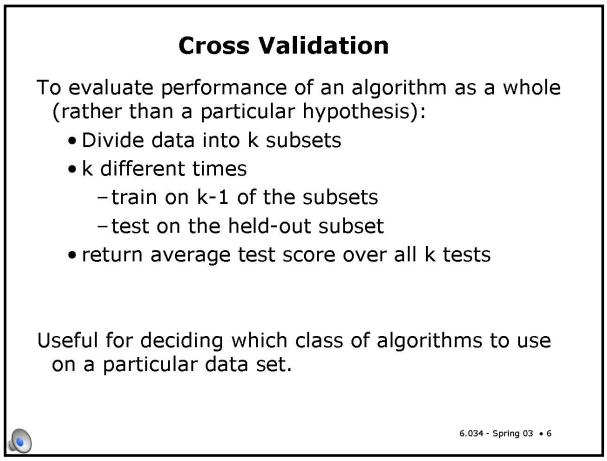

**Slide 2.3.4**
Sometimes you'd like to know how well a particular learning algorithm is working, rather than evaluating a single hypothesis it may have produced. Running the algorithm just once and evaluating its hypothesis only gives you one sample of its behavior, and it might be hard to tell how well it really works from just one sample.

If you have **lots** of data, you can divide it up into batches, and use half of your individual batches to train new hypotheses and the other half of them to evaluate the hypotheses. But you rarely have enough data to be able to do this.

**Slide 2.3.5**
One solution is called cross-validation. You divide your data up into some number (let's say 10 for now) of chunks. You start by feeding chunks 1 through 9 to your learner and evaluating the resulting hypothesis using chunk 10.

Now, you feed all but chunk 9 into the learner and evaluate the resulting hypotheses on chunk 2. And so on.



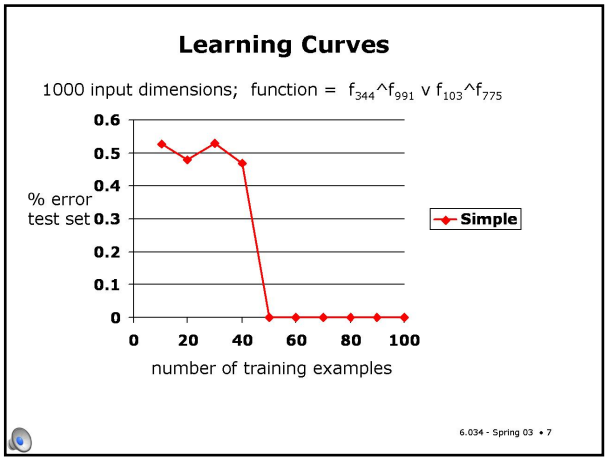



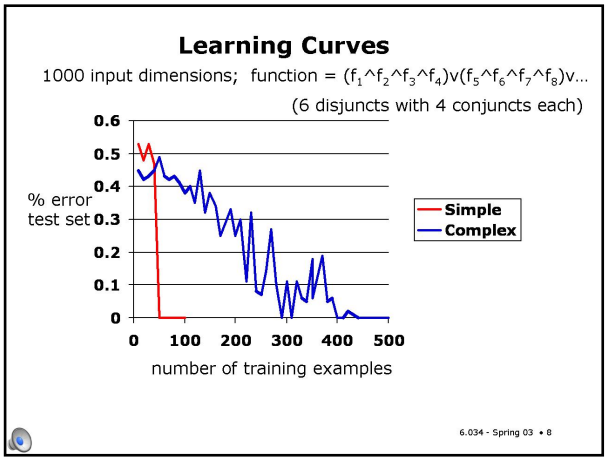

**Slide 2.3.6**
At the end of this process, you will have run the learning algorithm 10 times, getting 10 different hypotheses, and tested each of the hypotheses on a different test set. You can now average the error results from each of these trials to get an aggregate picture of how well your learning algorithm performs. (For those of you who might be statistically sophisticated, you have to be careful with these statistics because the samples are not independent.) Remember that this is a method for comparing algorithms on a data set, not for evaluating the quality of a particular hypothesis.

**Slide 2.3.7**
One interesting thing to see is how well the learning process works as a function of how much data is available to it. I made a learning curve by secretly picking a target function to generate the data, drawing input points uniformly at random and generating their associated y using my secret function. First I generated 10 points and trained a hypothesis on them, then evaluated it. Then I did it for 20 points, and so on.

This is the learning curve for a fairly simply function: $f_{344} \wedge f_{991} \vee f_{103} \wedge f_{775}$, in a domain with 1000 features. The x axis is the amount of training data. The y axis is the number of errors made by the resulting hypothesis, on a test set of data that is drawn uniformly at random, as a percentage of the total size of the test set.

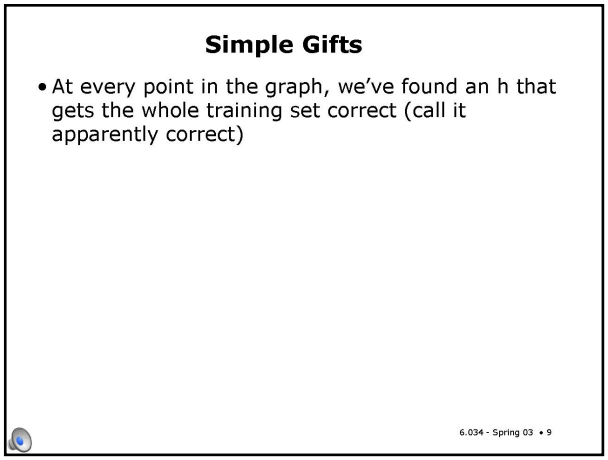
It takes suprisingly few samples before we find the correct hypothesis. With 1000 features, there are $2^{1000}$ possible examples. We are getting the correct hypothesis after seeing a tiny fraction of the whole space.

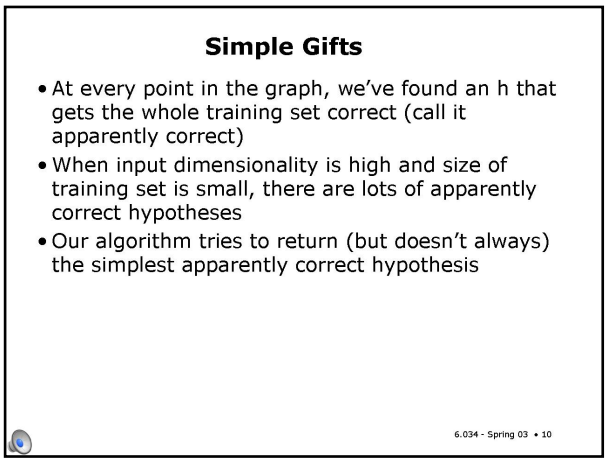





**Slide 2.3.8**
Now, let's look at a much more complex target function. It's $(f_1 \wedge f_2 \wedge f_3 \wedge f_4) \vee (f_5 \wedge f_6 \wedge f_7 \wedge f_8) \vee (f_9 \wedge f_{10} \wedge f_{11} \wedge f_{12}) \vee (f_{13} \wedge f_{14} \wedge f_{15} \wedge f_{16}) \vee (f_{17} \wedge f_{18} \wedge f_{19} \wedge f_{20}) \vee (f_{21} \wedge f_{22} \wedge f_{23} \wedge f_{24})$. It takes a lot more data to learn. Why is that?

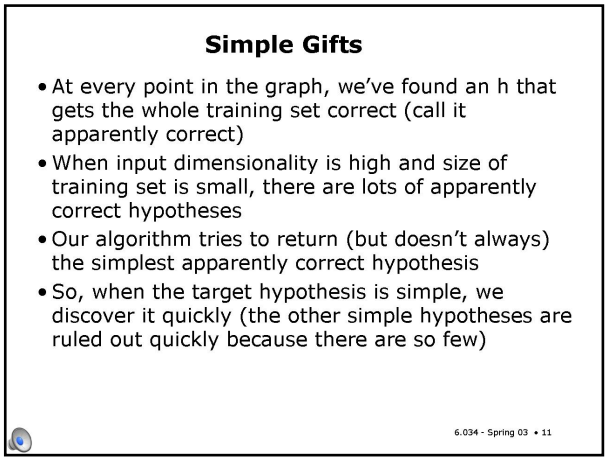
**Slide 2.3.9**
First, it's important to remember that, at every point in the graph, the learning algorithm has found a hypothesis that gets all of the training data correct.

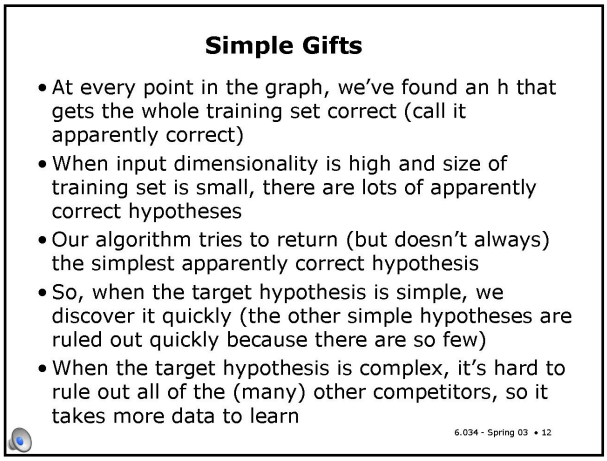



**Slide 2.3.10**
When we have just a little bit of data, there are many possible functions that agree with it all. How does our algorithm choose which answer to give? It tries to give a simple one. So, we would say that the algorithm has a "bias" in favor of simple hypotheses.

**Slide 2.3.11**
So, when the target hypothesis is simple, we discover it quickly (the other simple hypotheses are ruled out with just a little bit of data, because there are so few of them).

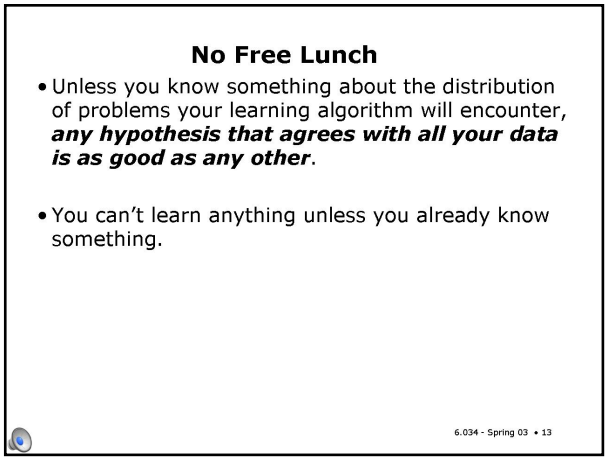


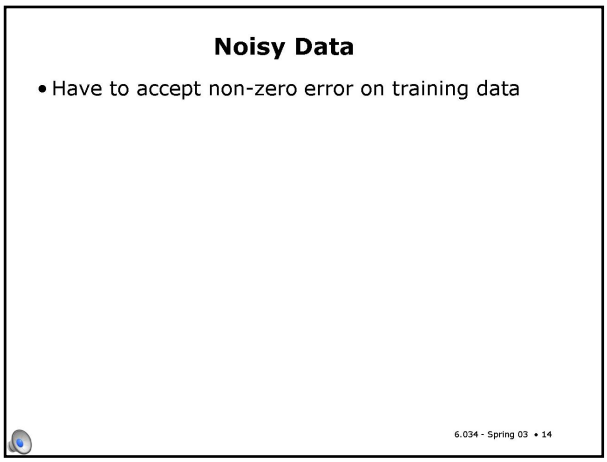



**Slide 2.3.12**
If we thought our problems were going to have complex answers, then we might think about biasing our algorithm to prefer more complex answers. But then we start to see the wisdom of Ockham. There are **lots** more complex hypotheses than there are simple ones. So there will be huge numbers of complex hypotheses that agree with the data and no basis to choose among them, unless we know something about the domain that generated our data (maybe some features are thought, in advance, to be more likely to have an influence on the outcome, for example).

**Slide 2.3.13**
There is, in fact, a theorem in machine learning called the "No Free Lunch" theorem. It basically states that, unless you know something about the process that generated your data, any hypothesis that agrees with all the data you've seen so far is as good a guess as any other one.

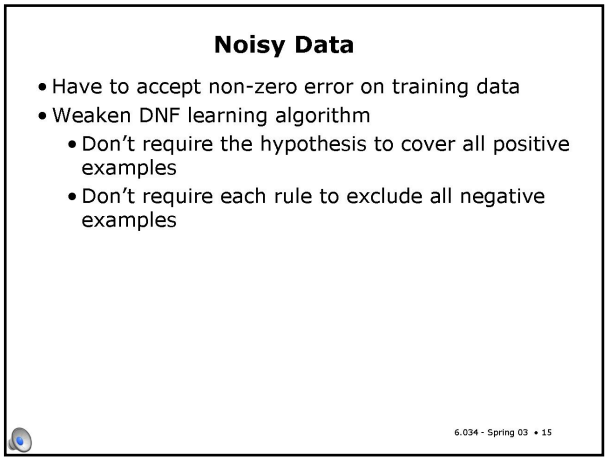
The picture will get a bit more complicated when we look at data that are noisy.

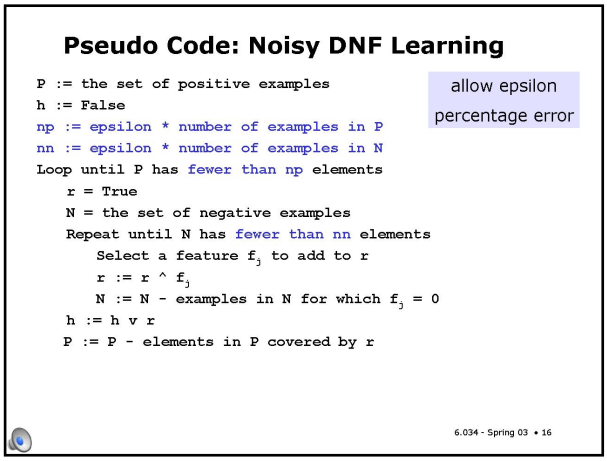



**Slide 2.3.14**
It is actually very rare to encounter a machine learning problem in which there is a deterministic function that accounts for y's dependence on x in all cases. It is much more typical to imagine that the process that generates the x's and y's is probabilistic, possibly corrupting measurements of the features, or simply sometimes making different decisions in the same circumstances.

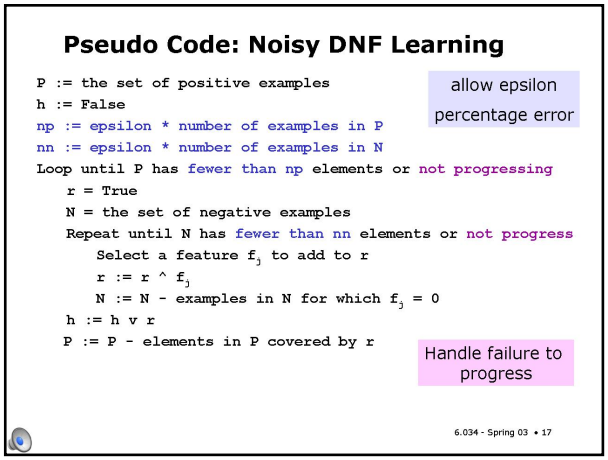
**Slide 2.3.15**
We can easily handle noise in the framework we have introduced. We left the door open for it, so to speak, in our formulation of the problem we were trying to solve. We want to find a hypothesis h that minimizes error plus complexity. When there's noise, we'll have to accept that our hypothesis will probably have non-zero error on the training set; and we'll have to be very careful about how we weight these two criteria, as we will see.

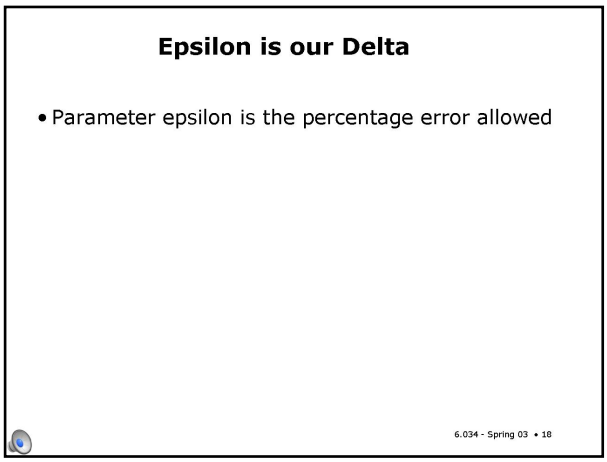





**Slide 2.3.16**
We can use our DNF learning algorithm almost unchanged. But, instead of requiring each rule to exclude **all** negative examples, and requiring the whole hypothesis to cover **all** positive examples, we will only require them to cover some percentage of them. Here is the pseudocode, with the changes highlighted in blue.

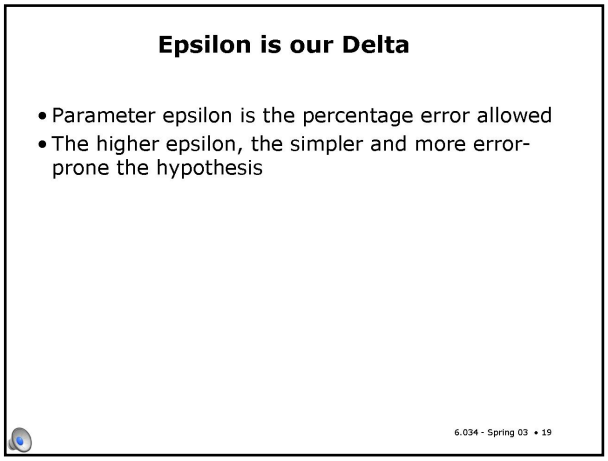
**Slide 2.3.17**
We also have to add further conditions for stopping both loops. If there is no feature that can be added that reduces the size of N, then the inner loop must terminate. If the inner loop cannot generate a rule that reduces the size of P, then the outer loop must terminate and fail.

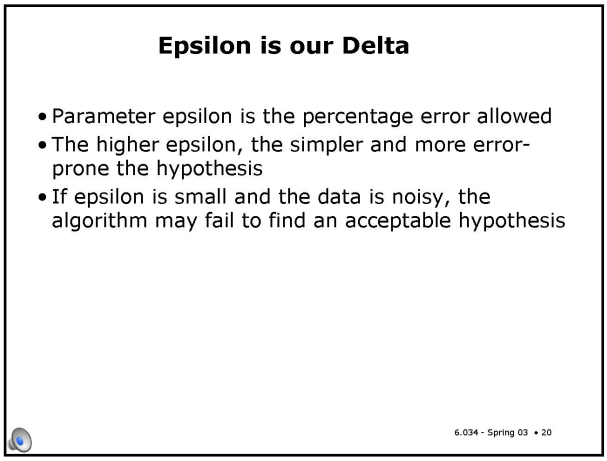



**Slide 2.3.18**
Now we have a new parameter, epsilon, which is the percentage of examples we're allowed to get wrong. (Well, we'll actually get more wrong, because each disjunct is allowed to get epsilon percent of the negatives wrong).

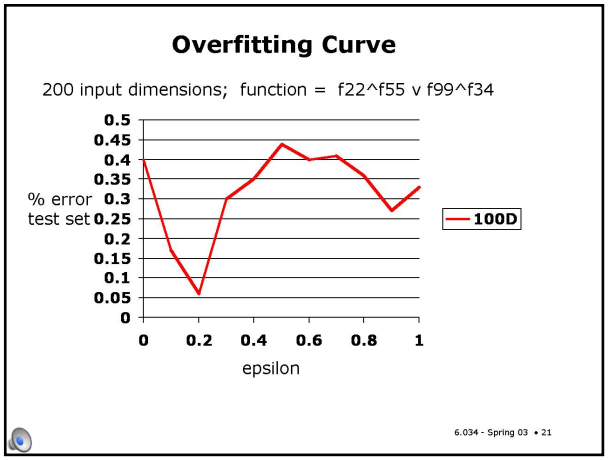
**Slide 2.3.19**
Although it is not exactly equivalent to manipulating the alpha parameter in our ideal criterion, it is a way to trade off accuracy for complexity. The higher the epsilon, the simpler and more error-prone our hypothesis.

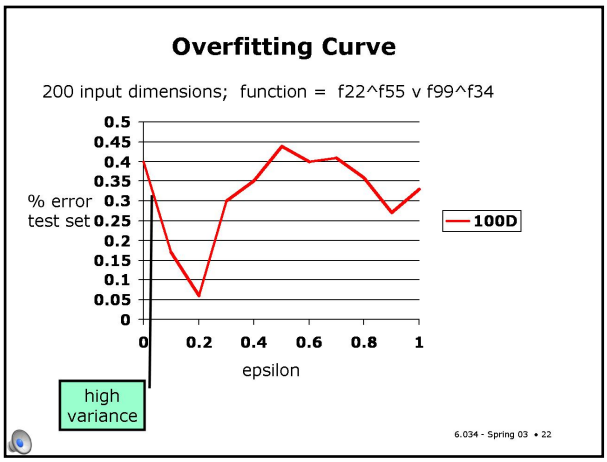





**Slide 2.3.20**
If epsilon is small and the data is noisy, the algorithm may fail to find an acceptable hypothesis.

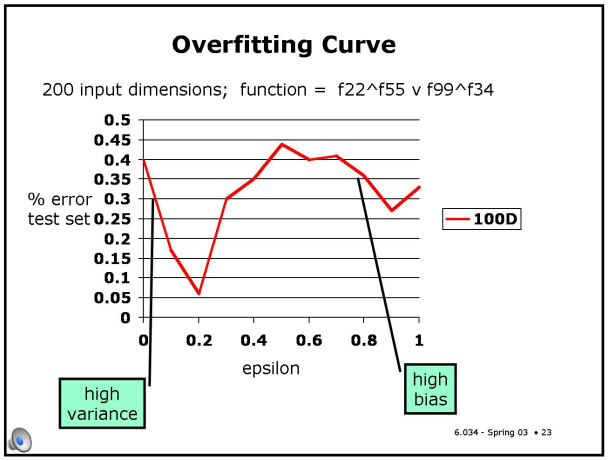
**Slide 2.3.21**
Let's see what happens in a moderately noisy domain. This curve shows the problem of learning our easy DNF function in a domain with 100 features, on a training set of size 100. The data has 10 percent noise, which means that, in 10 percent of the cases, we expect the output set to be the opposite of the one specified by the target function. The x axis is epsilon, ranging from 0 to 1. At setting 0, it is the same as our original algorithm. At setting 1, it's willing to make an arbitrary number of errors, and will always return the hypothesis False. The y axis is percentage error on a freshly drawn set of testing data. Note that, because there is 10 percent error in the data (training as well as testing), our very best hope is to generate a hypothesis that has 10 percent error on average on testing data.

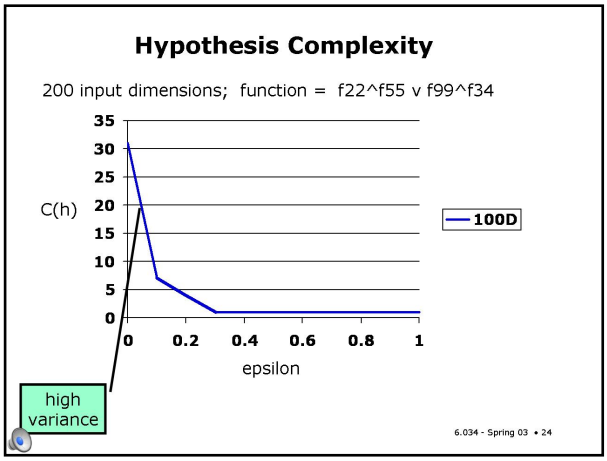



**Slide 2.3.22**
We get the best performance for a value of epsilon that's between 0 and 1. What's the matter with setting it to some value near 0? We run into the problem of **overfitting**. In overfitting, we work very hard to reduce the error of our hypothesis on the training set. But we may have spent a lot of effort modeling the noise in the data, and perform poorly on the test set. That's what is happening here. It is said, in this case, that our algorithm has high **variance**. Another way to think about the problem is to see that if we get another noisy data set generated by the same target function, we are likely to get a wildly different answer. By turning epsilon up a bit, we generate simpler hypotheses that are not so susceptible to variations in the data set, and so can get better generalization performance.

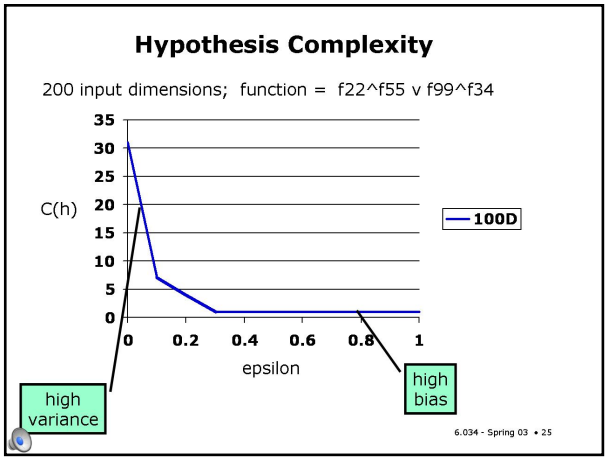
**Slide 2.3.23**
Of course, turning epsilon up too high will keep us from building a hypothesis of sufficient complexity. Then we'd do poorly because we are unable to even represent the right answer. It is said, in this case, that our algorithm has high **bias**.

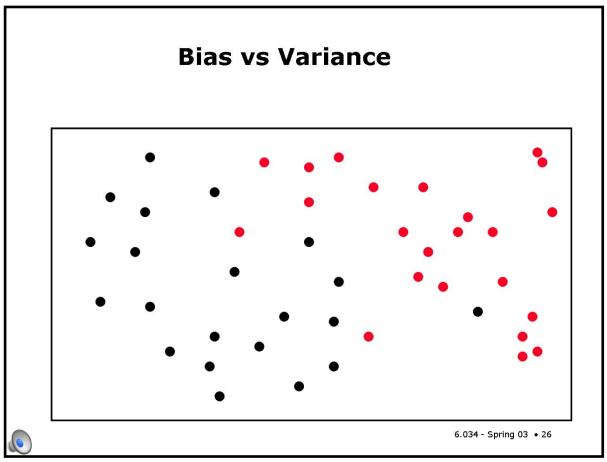





**Slide 2.3.24**
It's also instructive to see how the complexity of the resulting hypothesis varies as a function of epsilon. When epsilon is 0, we are asking for a hypothesis with zero error on the training set. We are able to find one, but it's very complex (31 literals). This is clearly a high variance situation; that hypothesis is completely influenced by the particular training set and would change radically on a newly drawn training set (and therefore, have high error on a testing set, as we saw on the previous slide).

**Slide 2.3.25**
As epsilon increases, we rapidly go down to a complexity of 1, which is incapable of representing the target hypothesis. The low-point in the error curve was at epsilon = 0.2, which has a complexity of 4 here. And it's no coincidence that the target concept also has a complexity of 4.

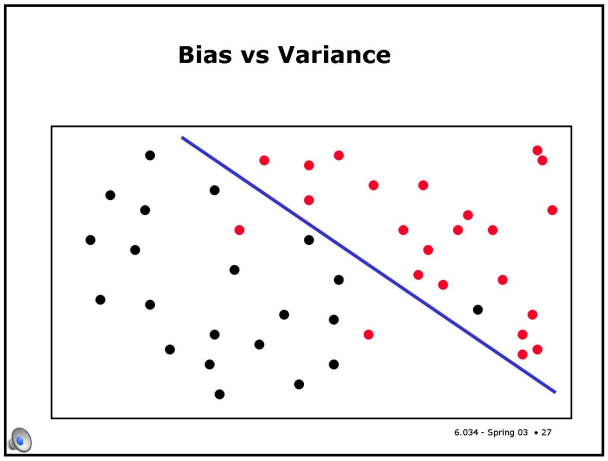


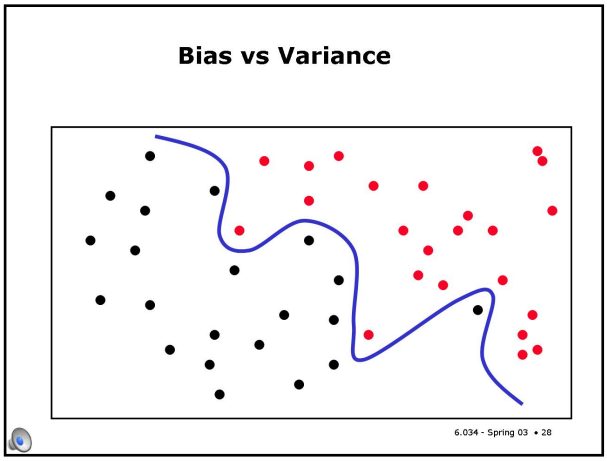




**Slide 2.3.26**
This problem is called the bias/variance trade-off. Because this is such an important idea in machine learning, let's look at it one more time in the context of our old pictures of dots on the plane.

**Slide 2.3.27**
Here's some noisy data.

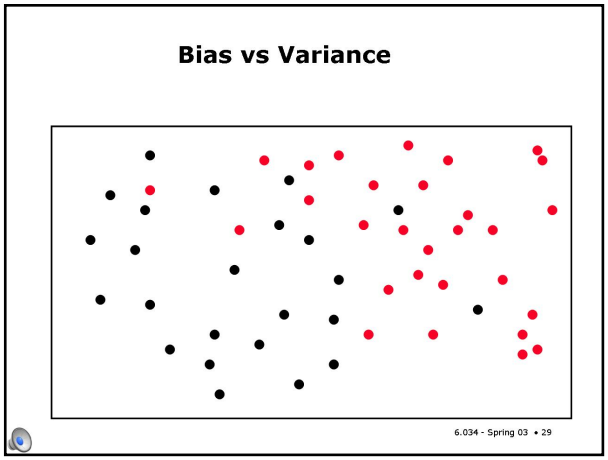
If we separate it with a line, we have low complexity, but lots of error.

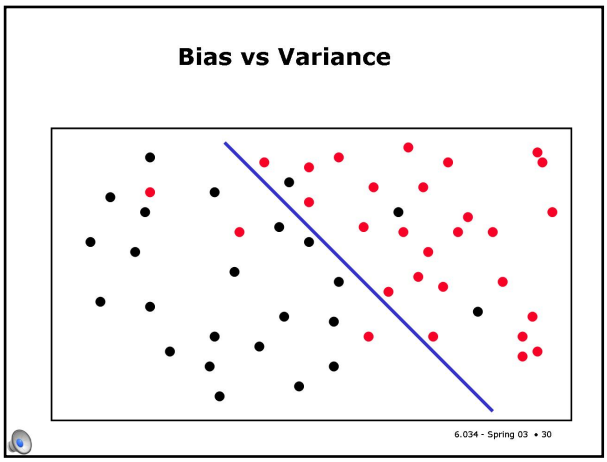





**Slide 2.3.28**
If we separate it with a squiggly snake, we can reduce the error but add a lot of complexity.

**Slide 2.3.29**
So, what, exactly, is so bad about complexity? Well, let's draw a bunch more points from the same underlying distribution and add them to our data set.

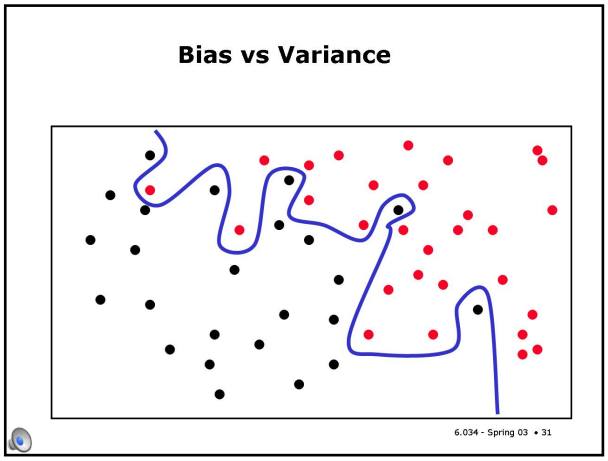


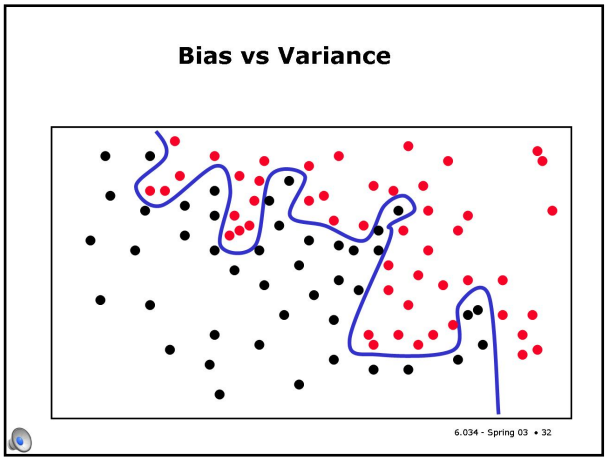




**Slide 2.3.30**
We wouldn't need to change our line hypothesis very much to do our best to accomodate them.

**Slide 2.3.31**
But we'd have to change the snake hypothesis radically. This is what it means to have high variance.

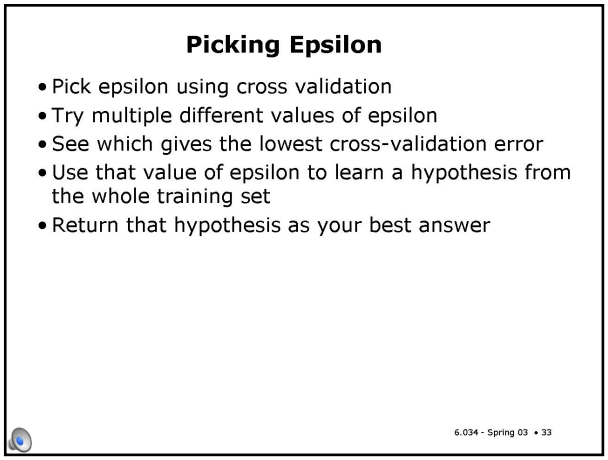
The line, although it was making mistakes on the original data set, performs better on unseen data.

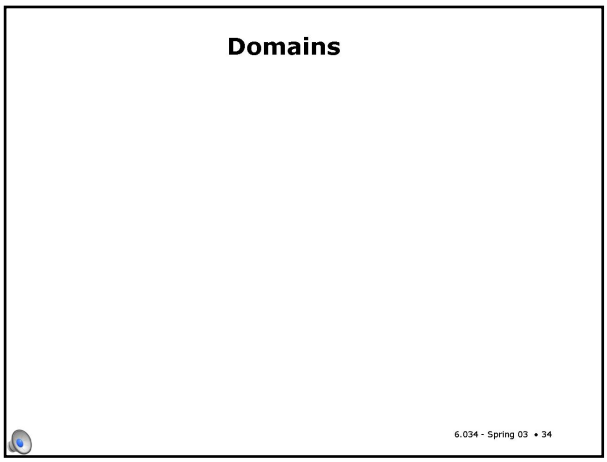





**Slide 2.3.32**
In practice, it is common to allow hypotheses of increased complexity as the amount of available training data increases. So, in this situation, we might feel justified in choosing a moderately complex hypothesis.

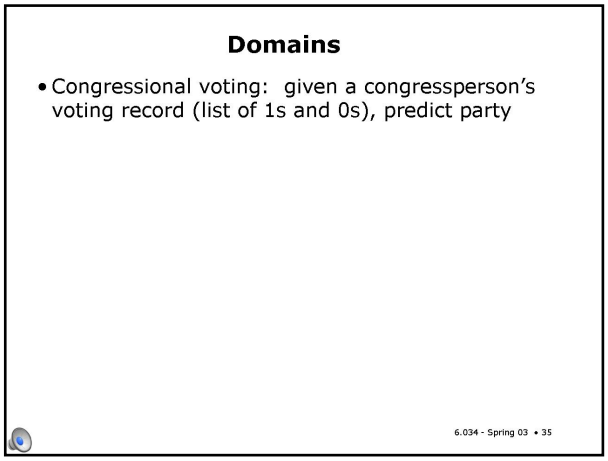
**Slide 2.3.33**
So, given a data set, how should we choose epsilon? This is where cross-validation can come in handy. You select a number of different values of epsilon, and use cross-validation to estimate the error when using that epsilon. Then, you pick the epsilon value with the least estimated error, and train your learning algorithm with all the available data using that value of epsilon. The resulting hypothesis is probably the best you can do (given what you know).

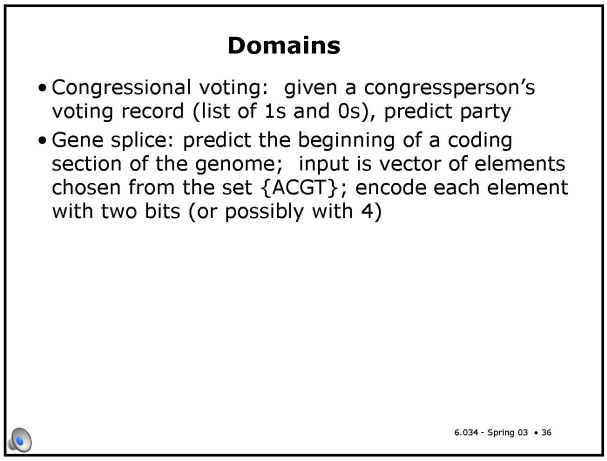



**Slide 2.3.34**
So, is this algorithm actually good for anything? Yes. There are lots of domains that can be encoded as vectors of binary features with a binary classification. Here are some examples.

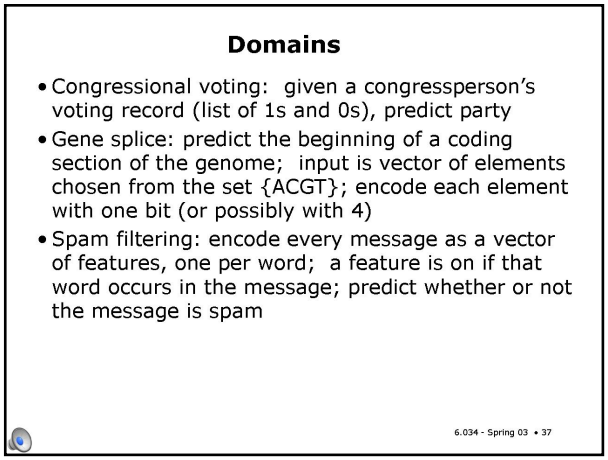
**Slide 2.3.35**
Congressional voting: given a congressperson's voting record, where the features are the individual's votes on various bills, can you predict whether they are a republican or democrat?

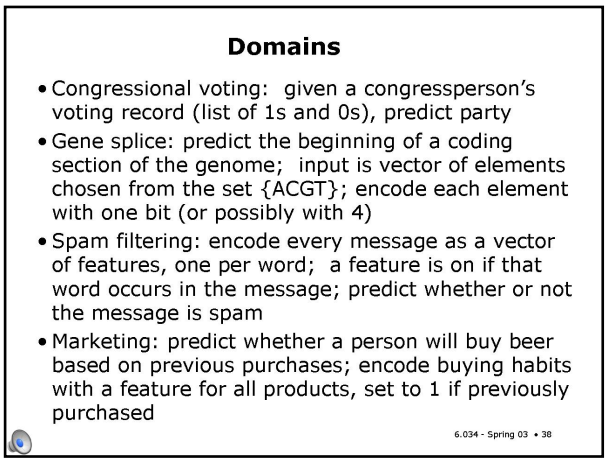





**Slide 2.3.36**
Gene splicing: given a sequence of bases (A, C, G, or T), predict whether the mid-point is a splice junction between a coding segment of DNA and a non-coding segment. In order to apply our algorithms to this problem, we would have to change the representation somewhat. It would be easy to take each base and represent it as two binary values; A is 00, C is 01, G is 10 and T is 11. Even though it seems inefficient, for learning purposes, it is often better to represent discrete elements using a unary code; so A would be 0001, C would be 0010, G would be 0100 and T would be 1000.

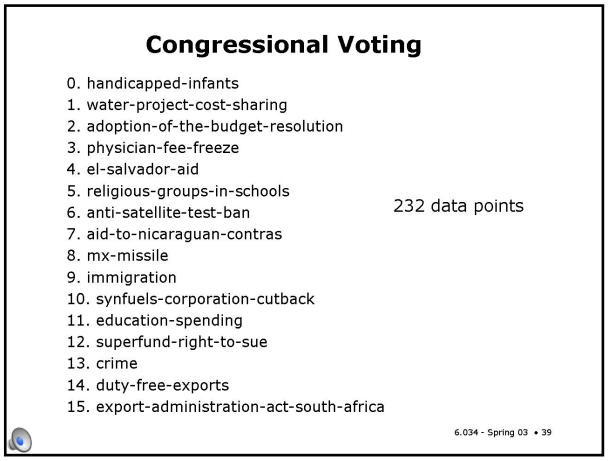
**Slide 2.3.37**
Spam filtering: is this email message spam? Represent a document using a really big feature space, with one feature for each possible word in the dictionary (leaving out some very common ones, like "a" and "the"). The feature for a word has value 1 if that word exists in the document and 0 otherwise.

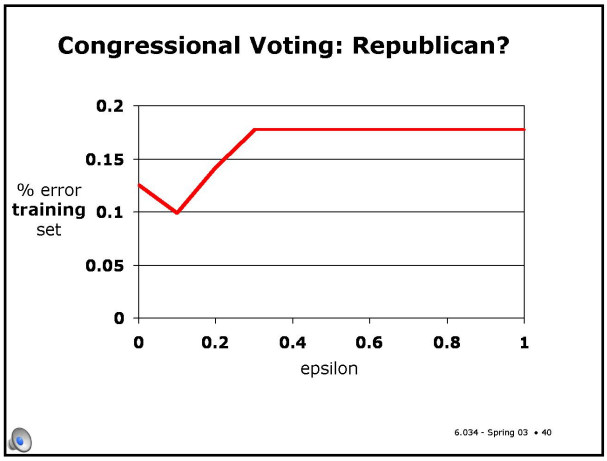



**Slide 2.3.38**
Marketing: is this grocery-store customer likely to buy beer? Represent a person's buying history using a big feature space, with one feature for each product in the supermarket. The feature for a product has value 1 if the person has ever bought that product and 0 otherwise.

**Slide 2.3.39**
Just for fun, we ran the DNF learning algorithm on the congressional voting data. A lot of the records had missing data (those congress folks are always out in the hallways talking instead of in there voting); we just deleted those records (though in a machine learning class we would study better ways of handling missing data items). There were 232 training examples, each of which had 15 features. The features were the individuals' votes on these bills (unfortunately, we don't know any details about the bills, but the names are evocative).

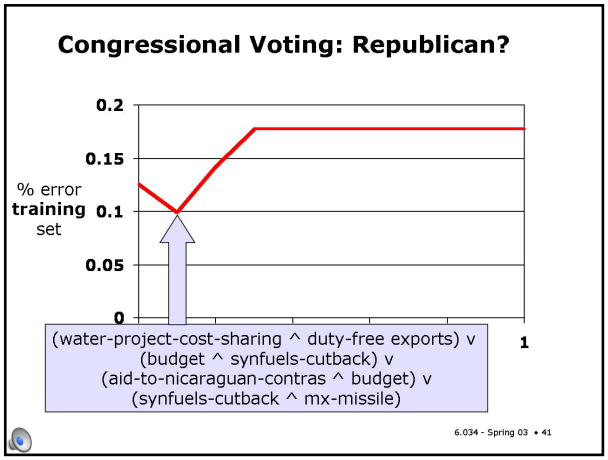


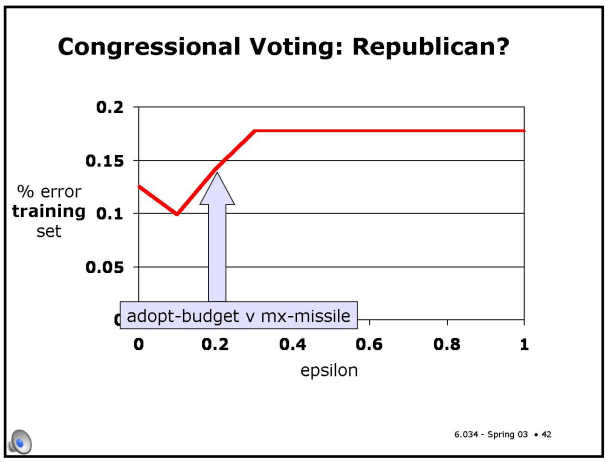



**Slide 2.3.40**
We are trying to predict whether a person is a Republican, based on their voting history during part of 1984. Here is a plot of the **training set error** as a function of epsilon. What I should really do is cross-validation, but instead I just ran the algorithm on the whole data set and I'm plotting the error on the training set.

Something sort of weird is going on. Training set error should get smaller as epsilon decreases. Why do we have it going up for epsilon equal zero? It's because of the greediness of our algorithm. We're not actually getting the best hypothesis in that case.

**Slide 2.3.41**
Here's the hypothesis we get at this point. It's kind of complicated (but mentions a lot of Republican things, like aid to the Nicaraguan contras and the mx-missile).

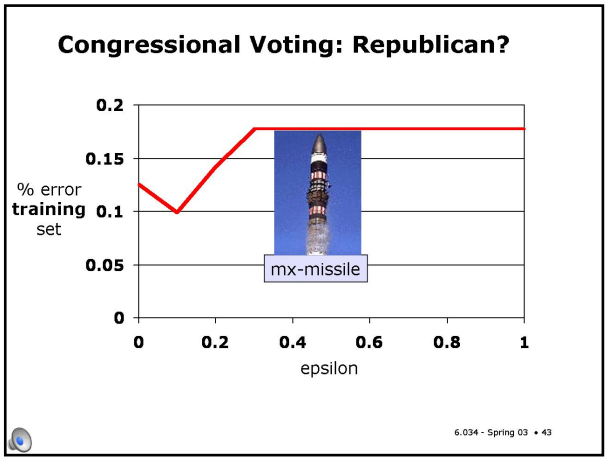



**Slide 2.3.42**
Here's a simpler hypothesis (with slightly worse error).

**Slide 2.3.43**
And here's the best hypothesis of complexity 1 for predicting whether someone is a Republican: did they vote "yes" for the MX missile?

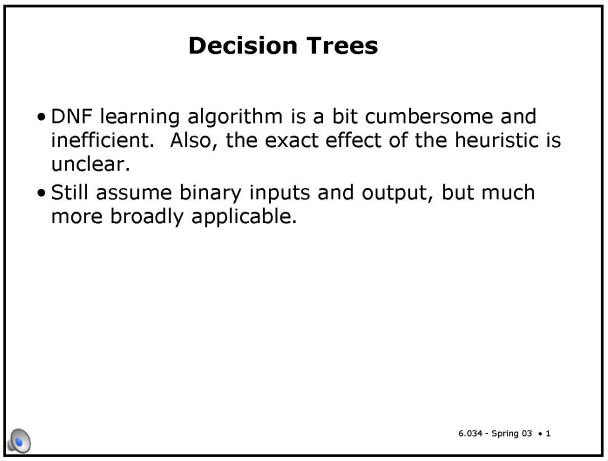
And, just because you're all too young to remember this, a little history. The MX missile, also called the "Peacekeeper" was our last big nuclear missile project. It's 71 feet long, with 10 warheads, and launched from trains. It was politically very contentious. Congress killed it at least twice before it was finally approved.

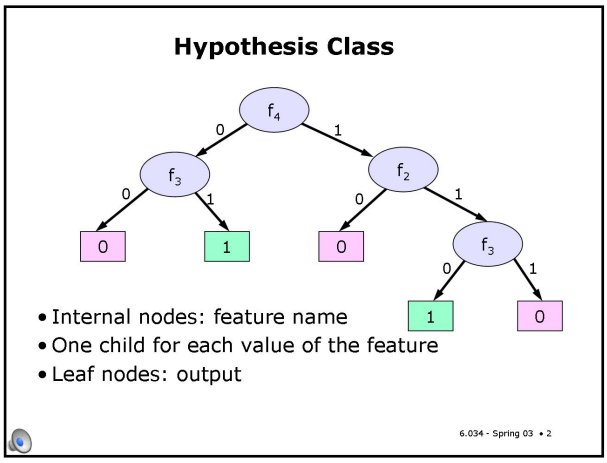


# 6.034 Notes: Section 2.4

**Slide 2.4.1**
The learning algorithm for DNF that we saw last time is a bit complicated and can be inefficient. It's also not clear that we're making good decisions about which attributes to add to a rule, especially when there's noise.

So now we're going to look at an algorithm for learning decision trees. We'll be changing our hypothesis class (sort of), our bias, and our algorithm. We'll continue to assume, for now, that the input features and the output class are boolean. We'll see later that this algorithm can be applied much more broadly.



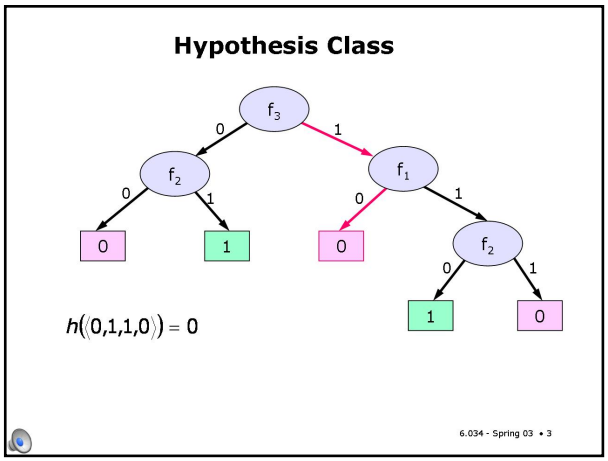



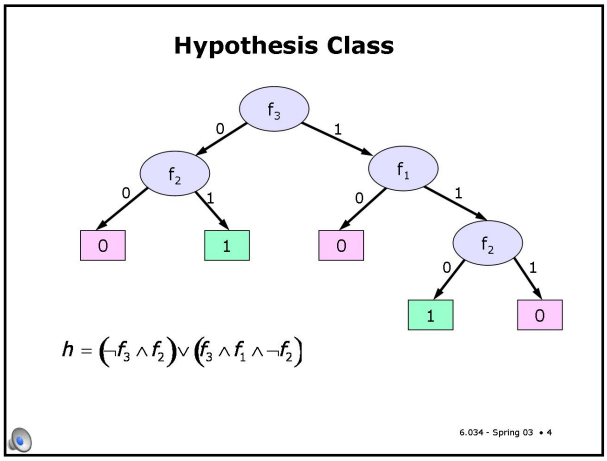

**Slide 2.4.2**
Our hypothesis class is going to be decision trees. A decision tree is a tree (big surprise!). At each internal (non-leaf) node, there is the name of a feature. There are two arcs coming out of each node, labeled 0 and 1, standing for the two possible values that the feature can take on.

Leaf nodes are labeled with decisions, or outputs. Since our y's are Boolean, the leaves are labeled with 0 or 1.

**Slide 2.4.3**
Trees represent Boolean functions from x's (vectors of feature values) to Booleans. To compute the output for a given input vector $x^i$, we start at the root of the tree. We look at the feature there, let's say it's feature j, and then look to see what the value of $x^i_j$ is. Then we go down the arc corresponding to that value. If we arrive at another internal node, we look up that feature value, follow the correct arc, and so on. When we arrive at a leaf node, we take the label we find there and generate that as an output.

So, in this example, input [0 1 1 0] would generate an output of 0 (because the third element of the input has value 1 and the first has value 0, which takes to a leaf labeled 0).

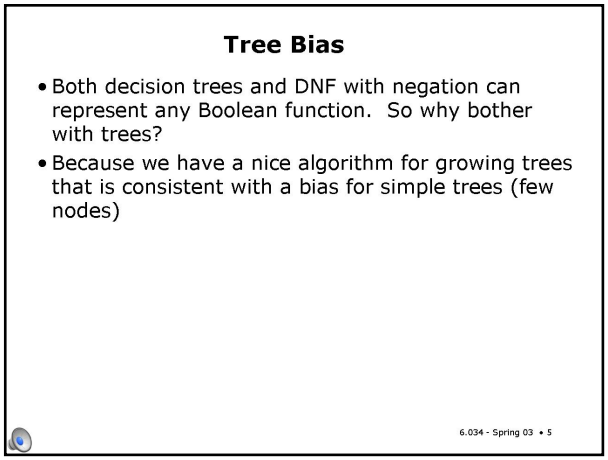


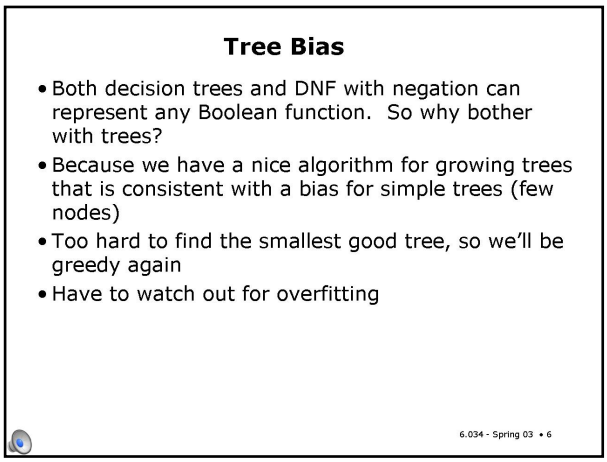



**Slide 2.4.4**
Decision trees are a way of writing down Boolean expressions. To convert a tree into an expression, you can make each branch of the tree with a 1 at the leaf node into a conjunction, describing the condition that had to hold of the input in order for you to have gotten to that leaf of the tree. Then, you take this collection of expressions (one for each leaf) and disjoin them.

So, our example tree describes the boolean function **not $f_3$ and $f_2$ or $f_3$ and $f_1$ and not $f_2$**.

**Slide 2.4.5**
If we allow negations of primitive features in DNF, then both DNF and decision trees are capable of representing any boolean function. Why, then, should we bother to change representations in this way? The answer is that we're going to define our bias in a way that is natural for trees, and use an algorithm that is tailored to learning functions in the tree representation.

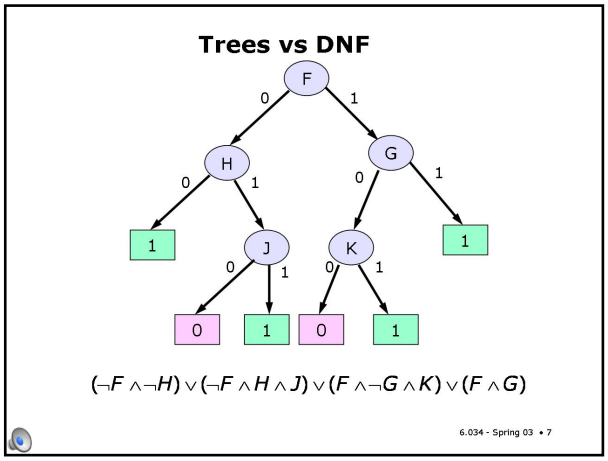


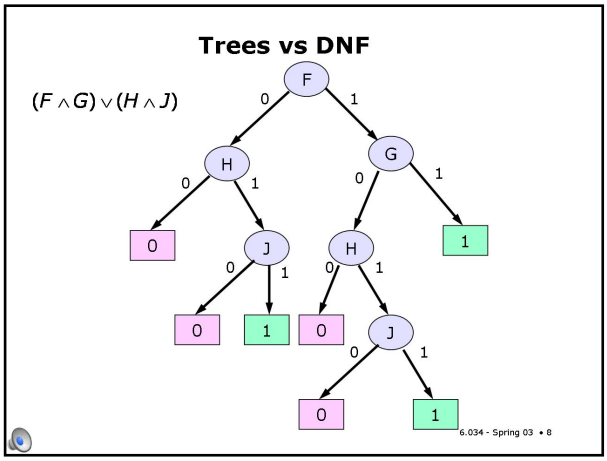




**Slide 2.4.6**
Application of Ockham's razor will lead us to prefer trees that are small, measured by the number of nodes. As before, it will be computationally intractable to find the minimum-sized tree that satisfies our error criteria. So, as before, we will be greedy, growing the tree in a way that seems like it will make it best on each step.

We'll consider a couple of methods for making sure that our resulting trees are not too large, so we can guard against overfitting.

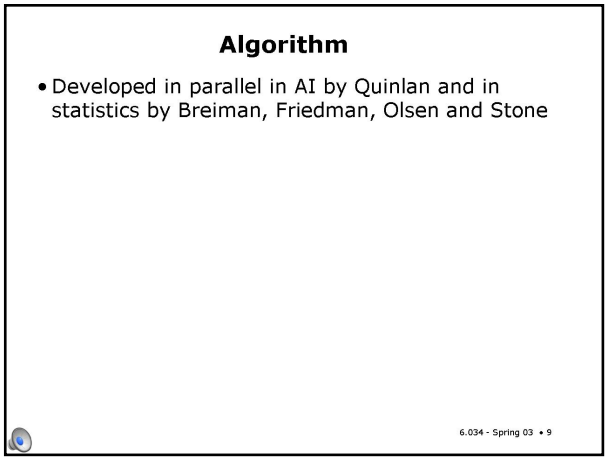
**Slide 2.4.7**
It's interesting to see that a bias for small trees is different from a bias for small DNF expressions. Consider this tree. It corresponds to a very complex function in DNF.

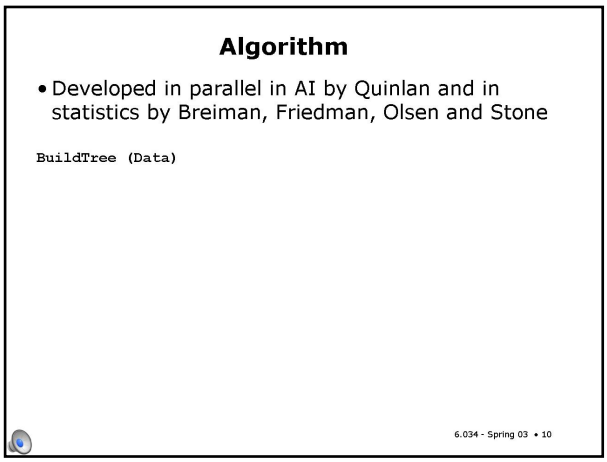





**Slide 2.4.8**
But here's a case with a very simple DNF expression that requires a large tree to represent it. There's no particular reason to prefer trees over DNF or DNF over trees as a hypothesis class. But the tree-growing algorithm is simple and elegant, so we'll study it.

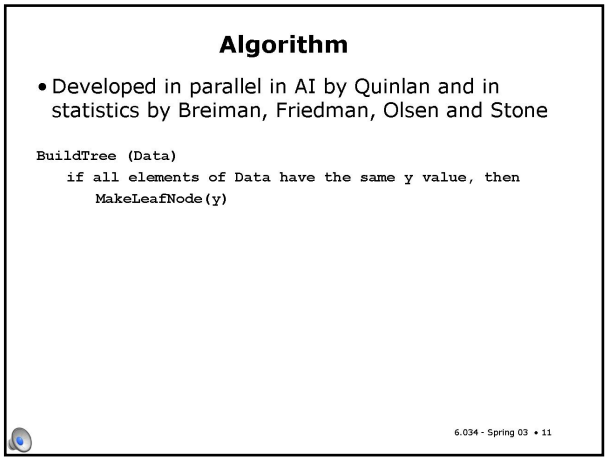
**Slide 2.4.9**
The idea of learning decision trees and the algorithm for doing so was, interestingly, developed independently by researchers in statistics and researchers in AI at about the same time around 1980.

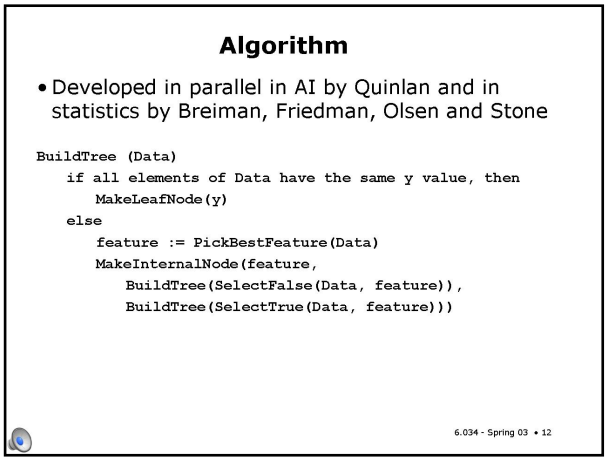



**Slide 2.4.10**
We will build the tree from the top down. Here is pseudocode for the algorithm. It will take as input a data set, and return a tree.

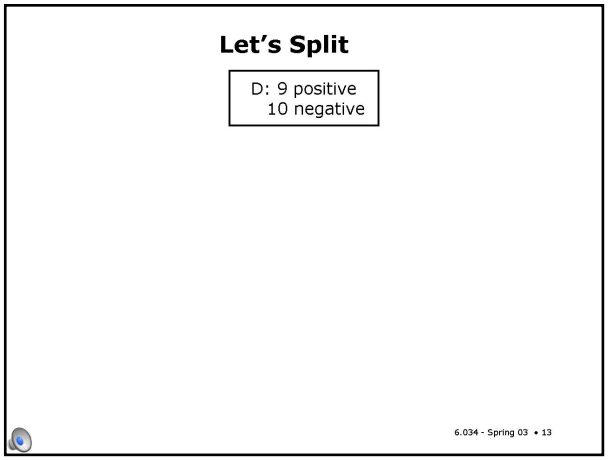
**Slide 2.4.11**
We first test to see if all the data elements have the same y value. If so, we simply make a leaf node with that y value and we're done. This is the base case of our recursive algorithm.

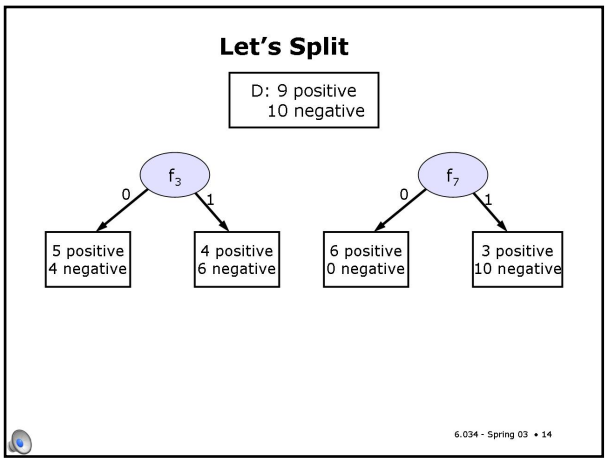





**Slide 2.4.12**
If we have a mixture of different y values, we choose a feature to use to make a new internal node. Then we divide the data into two sets, those for which the value of the feature is 0 and those for which the value is 1. Finally, we call the algorithm recursively on each of these data sets. We use the selected feature and the two recursively created subtrees to build our new internal node.

**Slide 2.4.13**
So, how should we choose a feature to split the data? Our goal, in building this tree, is to separate the negative instances from the positive instances with the fewest possible tests. So, for instance, if there's a feature we could pick that has value 0 for all the positive instances and 1 for all the negative instances, then we'd be delighted, because that would be the last split we'd have to make. On the other hand, a feature that divides the data into two groups that have the same proportion of positive and negative instances as we started with wouldn't seem to have helped much.

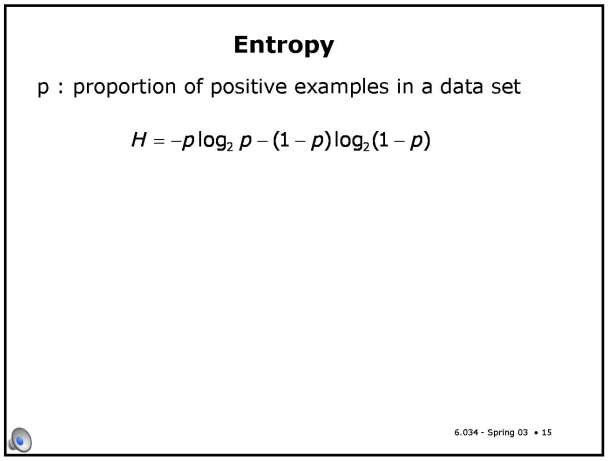


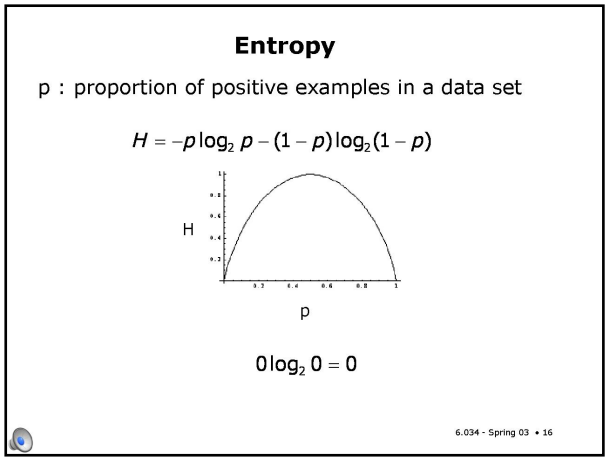




**Slide 2.4.14**
In this example, it looks like the split based on $f_7$ will be more helpful. To formalize that intuition, we need to develop a measure of the degree of uniformity of the subsets of the data we'd get by splitting on a feature.

**Slide 2.4.15**
We'll start by looking at a standard measure of disorder, used in physics and information theory, called **entropy**. We'll just consider it in the binary case, for now. Let p be the proportion of positive examples in a data set (that is, the number of positive examples divided by the total number of examples). Then the entropy of that data set is

$$- p \log_2 p - (1 - p) \log_2 (1 - p)$$



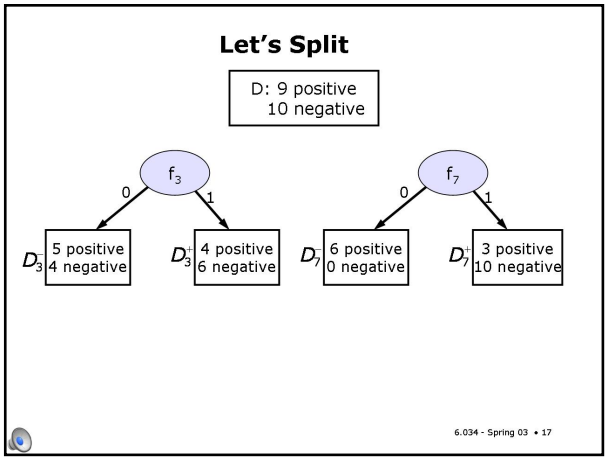


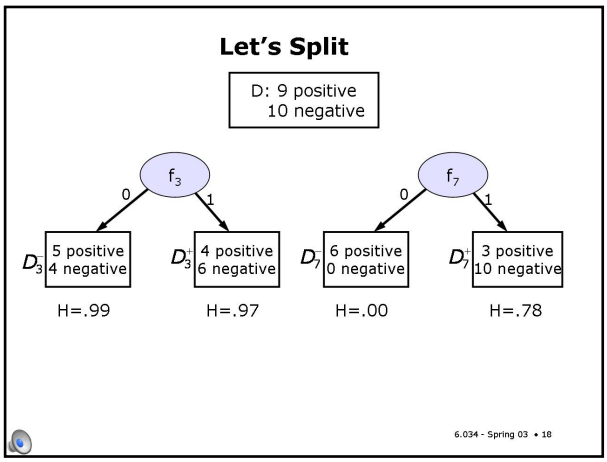

**Slide 2.4.16**
Here's a plot of the entropy as a function of p. When p is 0 or 1, then the entropy is 0. We have to be a little bit careful here. When p is 1, then 1 log 1 is clearly 0. But what about when p is 0? Log 0 is negative infinity. But 0 wins the battle, so 0 log 0 is also 0.

So, when all the elements of the set have the same value, either 0 or 1, then the entropy is 0. There is no disorder (unlike in my office!).

The entropy function is maximized when p = 0.5. When p is one half, the set is as disordered as it can be. There's no basis for guessing what the answer might be.

**Slide 2.4.17**
When we split the data on feature j, we get two data sets. We'll call the set of examples for which feature j has value 1 $D^+_j$ and those for which j has value 0 $D^-_j$.

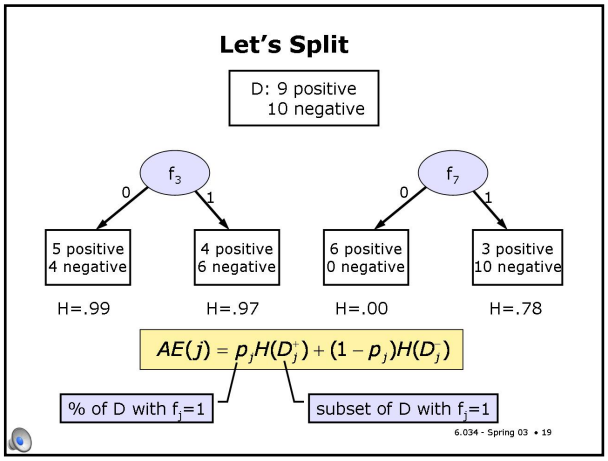


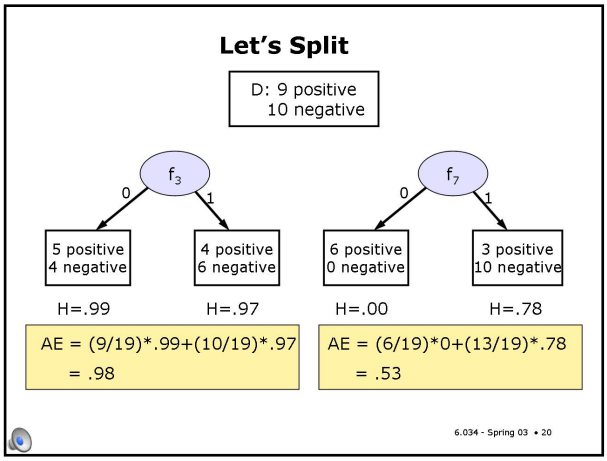




**Slide 2.4.18**
We can compute the entropy for each of these subsets. For some crazy reason, people usually use the letter H to stand for entropy. We'll follow suit.

**Slide 2.4.19**
Now, we have to figure out how to combine these two entropy values into a measure of how good splitting on feature j is. We could just add them together, or average them. But what if have this situation, in which there's one positive example in one data set and 100 each of positive and negative examples in the other? It doesn't really seem like we've done much good with this split, but if we averaged the entropies, we'd get a value of 1/4.

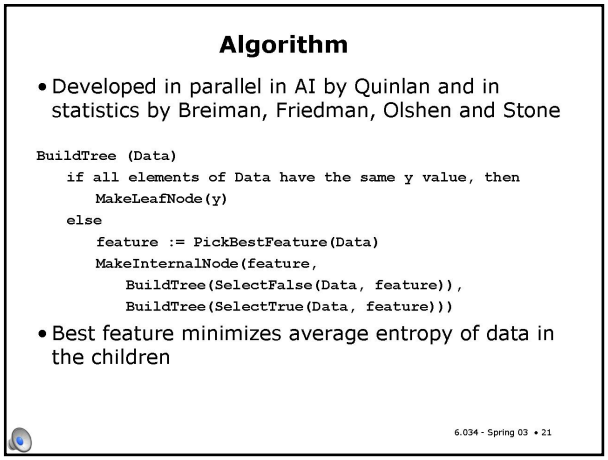
So, to keep things fair, we'll use a weighted average to combine the entropies of the two sets. Let $p_j$ be the proportion of examples in the data set D for which feature j has value 1. We'll compute a weighted **average entropy** for splitting on feature j as $AE(j) = p_j H(D^+_j) + (1 - p_j)H(D^-_j)$

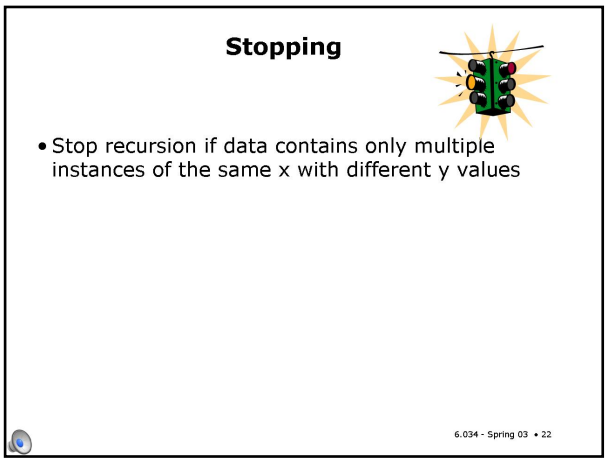





**Slide 2.4.20**
So, in this example, we can see that the split on $f_7$ is much more useful, as reflected in the average entropy of its children.

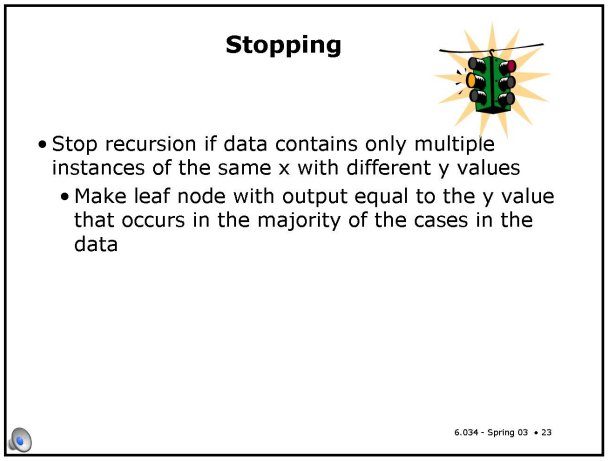
**Slide 2.4.21**
Going back to our algorithm, then, we'll pick the feature at every step that minimizes the weighted average entropy of the children.

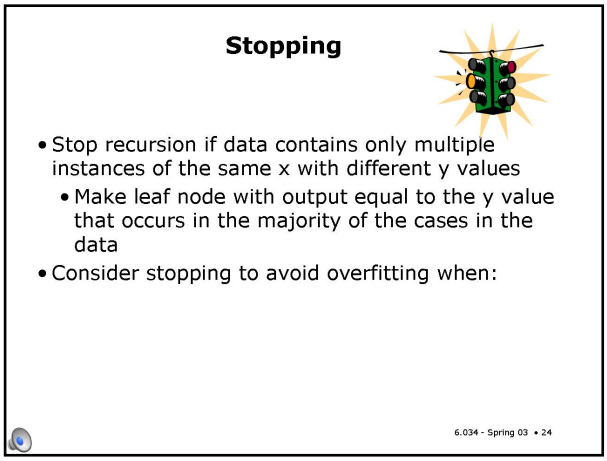



**Slide 2.4.22**
As usual, when there is noise in the data, it's easy to overfit. We could conceivably grow the tree down to the point where there's a single data point in each leaf node. (Or maybe not: in fact, if have two data points with the same x values but different y values, our current algorithm will never terminate, which is certainly a problem). So, at the very least, we have to include a test to be sure that there's a split that makes both of the data subsets non-empty. If there is not, we have no choice but to stop and make a leaf node.

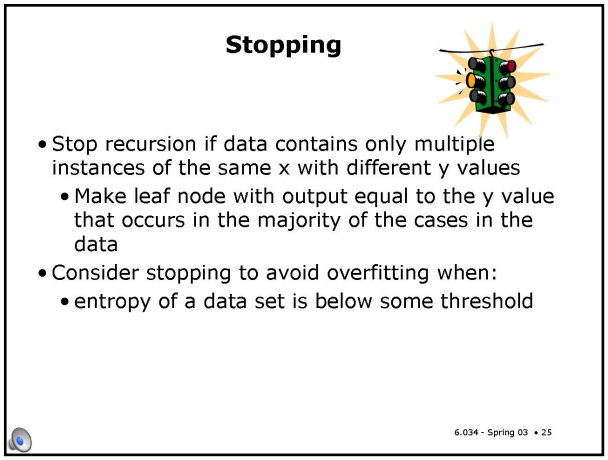
**Slide 2.4.23**
What should we do if we have to stop and make a leaf node when the data points at that node have different y values? Choosing the majority y value is the best strategy. If there are equal numbers of positive and negative points here, then you can just pick a y arbitrarily.

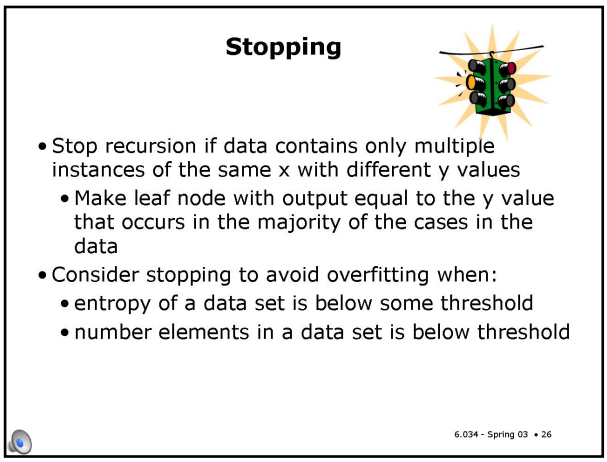





**Slide 2.4.24**
But growing the tree as far down as we can will often result in overfitting.

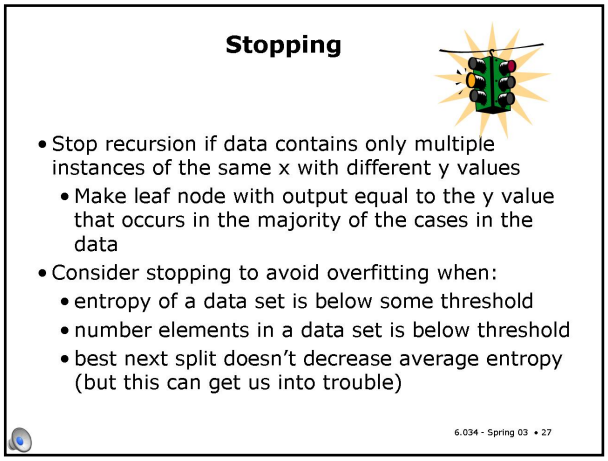
**Slide 2.4.25**
The simplest solution is to change the test on the base case to be a threshold on the entropy. If the entropy is below some value epsilon, we decide that this leaf is close enough to pure.

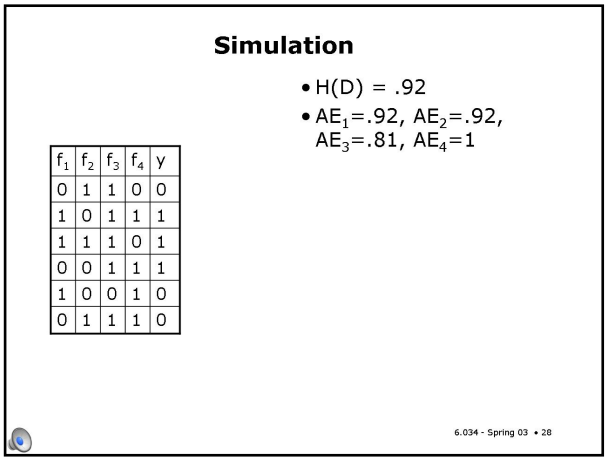



**Slide 2.4.26**
Another simple solution is to have a threshold on the size of your leaves; if the data set at some leaf has fewer than that number of elements, then don't split it further.

**Slide 2.4.27**
Another possible method is to only split if the split represents a real improvement. We can compare the entropy at the current node to the average entropy for the best attribute. If the entropy is not significantly decreased, then we could just give up.

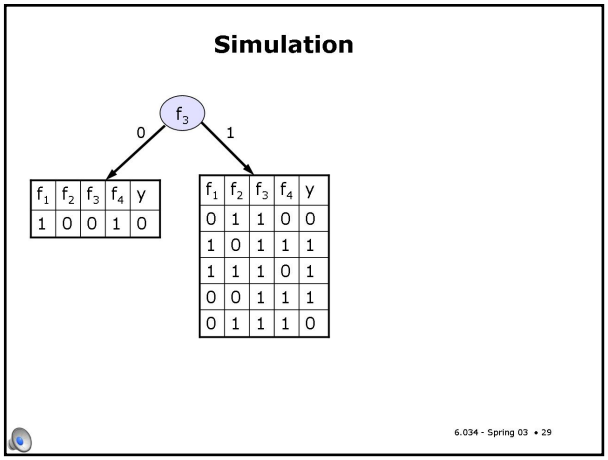


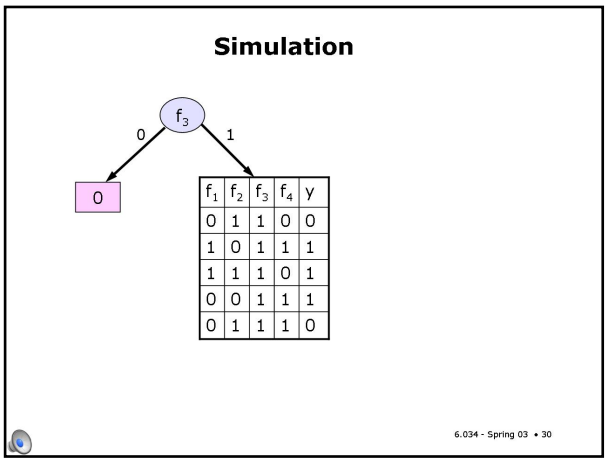



**Slide 2.4.28**
Let's see how our tree-learning algorithm behaves on the example we used to demonstrate the DNF-learning algorithm. Our data set has a starting entropy of .92. Then, we can compute, for each feature, what the average entropy of the children would be if we were to split on that feature.

In this case, the best feature to split on is $f_3$.

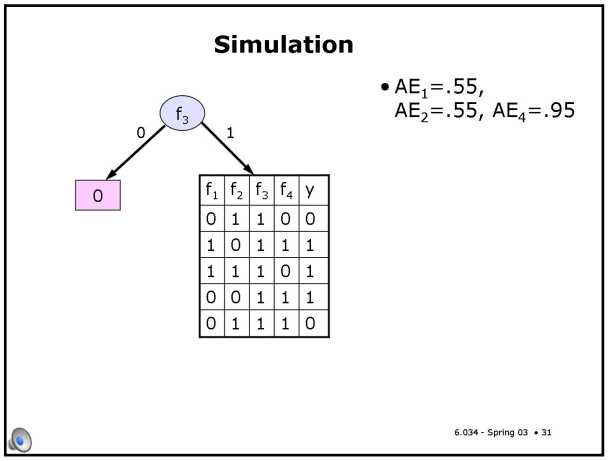
**Slide 2.4.29**
So, if we make that split, we have these two data sets. The one on the left has only a single output (in fact, only a single point).

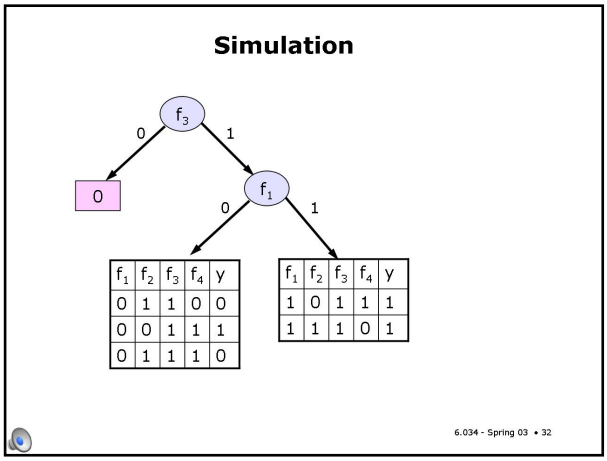



**Slide 2.4.30**
So we make it into a leaf with output 0 and consider splitting the data set in the right child.

**Slide 2.4.31**
The average entropies of the possible splits are shown here. Features 1 and 2 are equally useful, and feature 4 is basically no help at all.

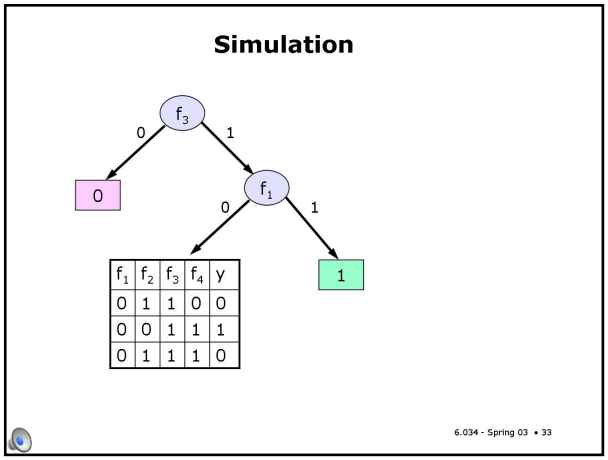


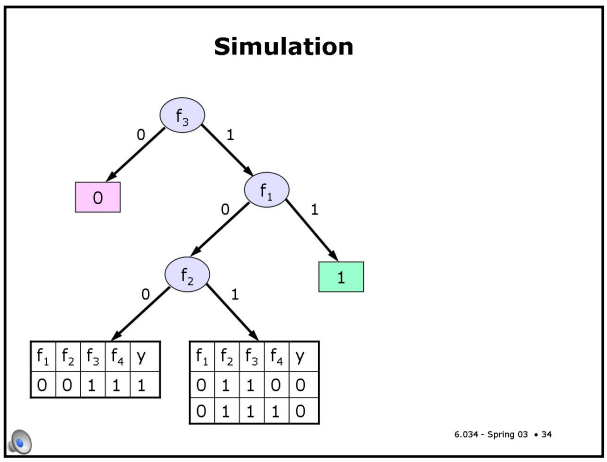



**Slide 2.4.32**
So, we decide, arbitrarily, to split on feature 1, yielding these sub-problems. All of the examples in the right-hand child have the same output.

**Slide 2.4.33**
So we can turn it into a leaf with output 1.

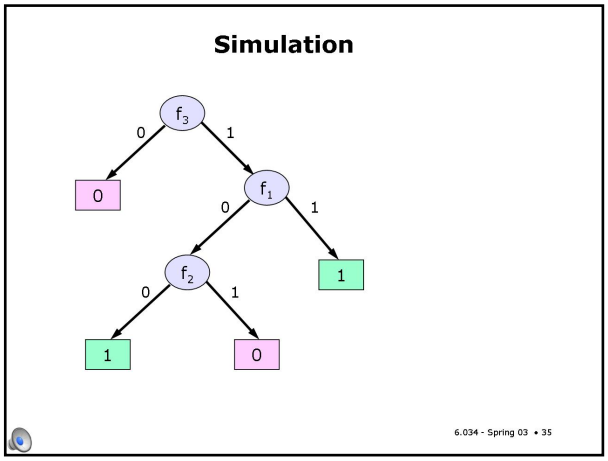
In the left child, feature 2 will be the most useful.

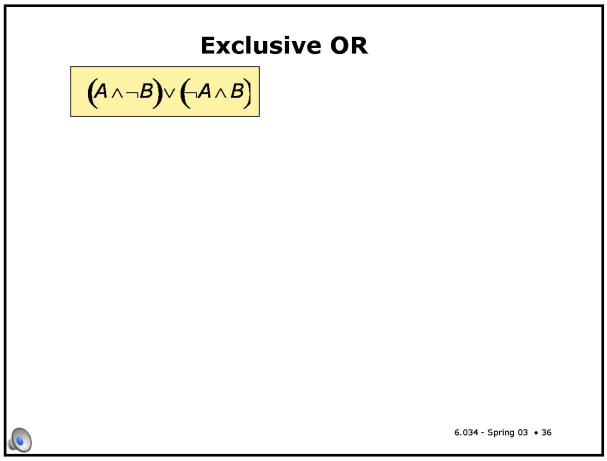



**Slide 2.4.34**
So we split on it, and now both children are homogeneous (all data points have the same output).

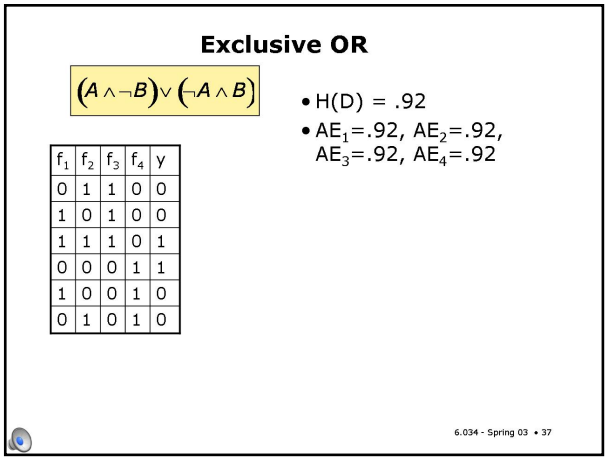
**Slide 2.4.35**
So we make the leaves and we're done!



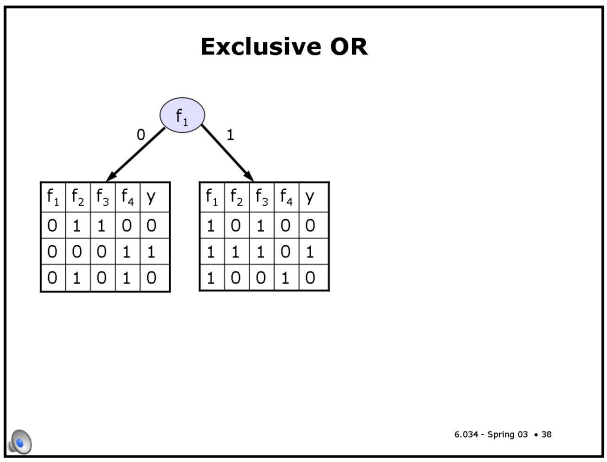



**Slide 2.4.36**
One class of functions that often haunts us in machine learning are those that are based on the "exclusive OR". Exclusive OR is the two-input boolean function that has value 1 if one input has value 1 and the other has value 0. If both inputs are 1 or both are 0, then the output is 0. This function is hard to deal with, because neither input feature is, by itself, detectably related to the output. So, local methods that try to add one feature at a time can be easily misled by xor.

**Slide 2.4.37**
Let's look at a somewhat tricky data set. The data set has entropy .92. Furthermore, no matter what attribute we split on, the average entropy is .92. If we were using the stopping criterion that says we should stop when there is no split that improves the average entropy, we'd stop now.




| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |

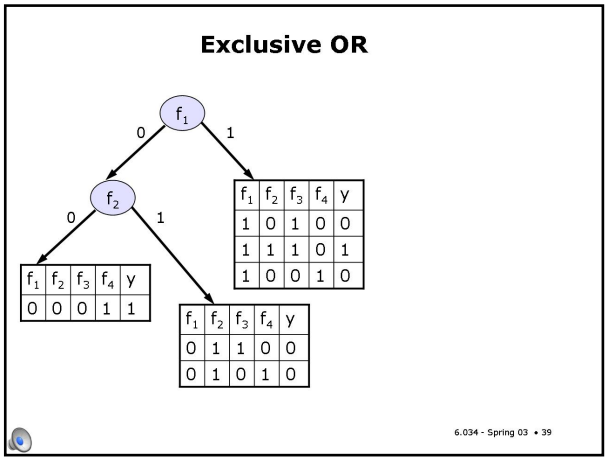



| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 |

| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |

**Slide 2.4.38**
But let's go ahead and split on feature 1. Now, if we look at the left-hand data set, we'll see that feature 2 will have an average entropy of 0.

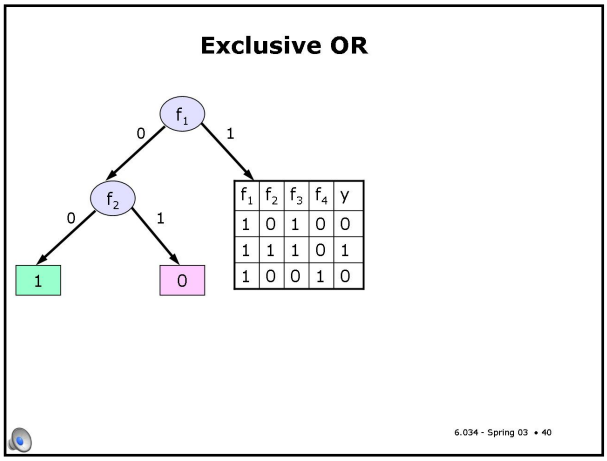
**Slide 2.4.39**
So we split on it, and get homogenous children,




| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |

| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |

| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |

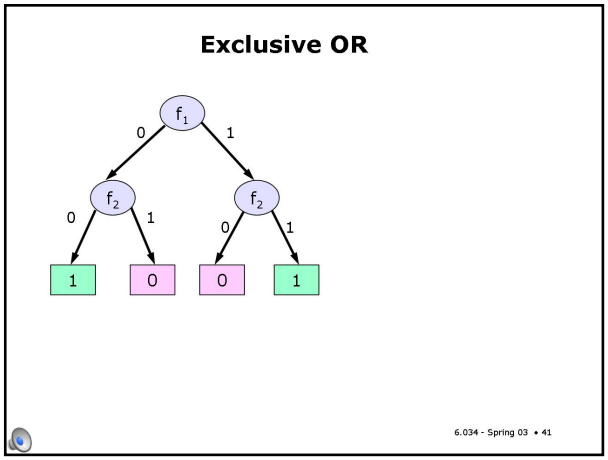


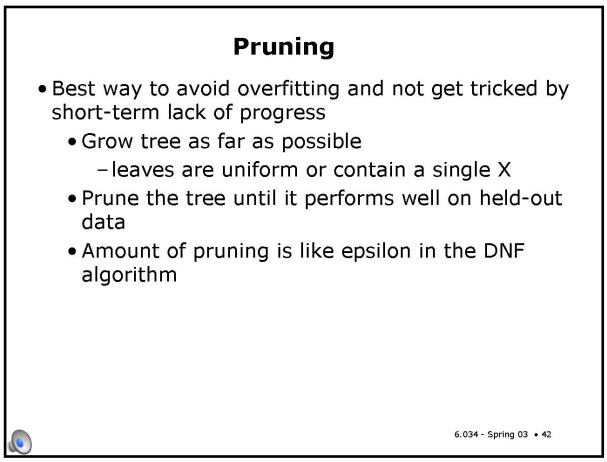

**Slide 2.4.40**
Which we can replace with leaves.

Now, it's also easy to see that feature 2 will again be useful here,

**Slide 2.4.41**
And we go straight to leaves on this side.

So we can see that, although no single feature could reduce the average entropy of the child data sets, features 1 and 2 were useful in combination.

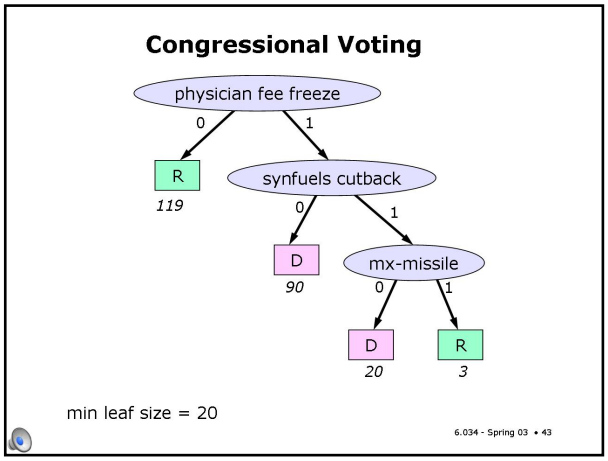


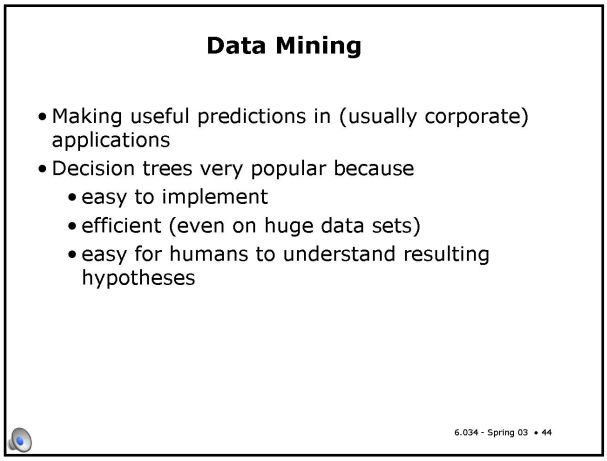




**Slide 2.4.42**
Most real decision-tree building systems avoid this problem by building the tree down significantly deeper than will probably be useful (using something like an entropy cut-off or even getting down to a single data-point per leaf). Then, they prune the tree, using cross-validation to decide what an appropriate pruning depth is.

**Slide 2.4.43**
We ran a simple version of the tree-learning program on the congressional voting database. Instead of pruning, it has a parameter on the minimum leaf size. If it reaches a node with fewer than that number of examples, it stops and makes a leaf with the majority output.

Here's the tree we get when the minimum leaf size of 20. This problem is pretty easy, so this small tree works very well. If we grow bigger trees, we don't get any real increase in accuracy.





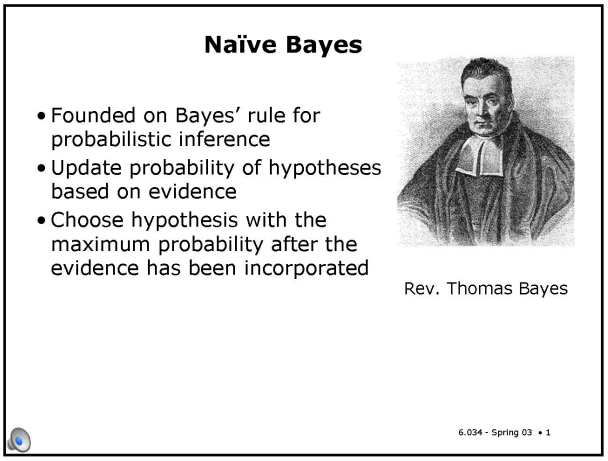

**Slide 2.4.44**
Because decision tree learning is easy to implement and relatively computationally efficient, and especially because the hypotheses are easily understandable by humans, this technology is very widely used.

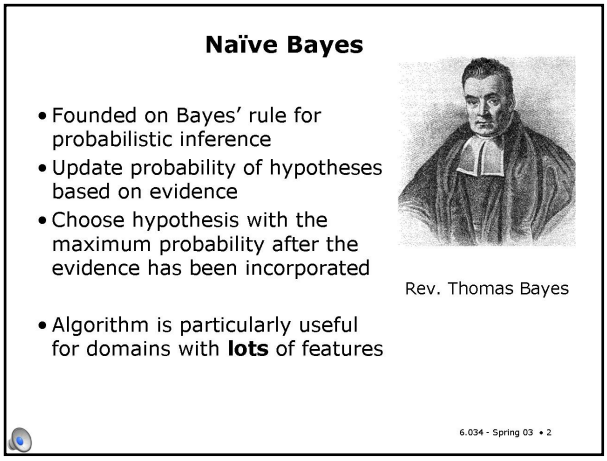
The field of data mining is the application of learning algorithms to problems of interest in many industries. Decision trees is one of the principle methods used in data mining.

In the next few sections, we'll see how to broaden the applicability of this and other algorithms to domains with much richer kinds of inputs and outputs.

# 6.034 Notes: Section 2.5

**Slide 2.5.1**
Let's look at one more algorithm, which is called naive Bayes. It's named after the Reverend Thomas Bayes, who developed a very important theory of probabilistic reasoning.

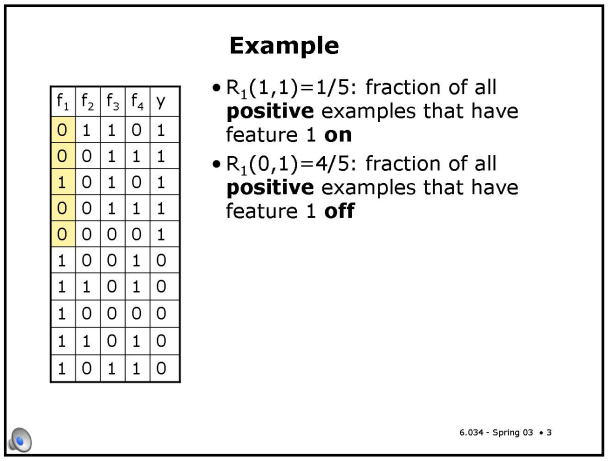


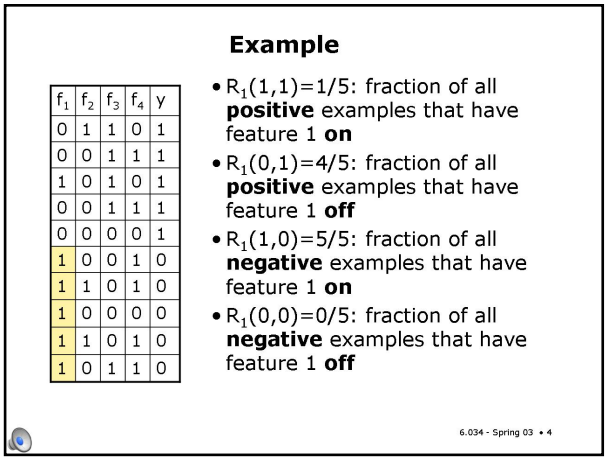




**Slide 2.5.2**
It's widely used in applications with lots of features. It was derived using a somewhat different set of justifications than the ones we've given you. We'll start by going through the algorithm, and at the end I'll go through its probabilistic background. Don't worry if you don't follow it exactly. It's just motivational, but it should make sense to anyone who has studied basic probability.

**Slide 2.5.3**
Let's start by looking at an example data set. We're going to try to characterize, for each feature individually, how it is related to the class of an example.

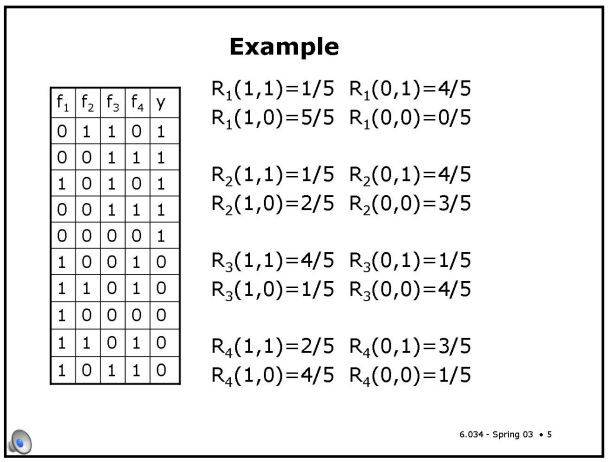
First, we look at the positive examples, and count up what fraction of them have feature 1 on and what fraction have feature 1 off. We'll call these fractions $R_1(1, 1)$ and $R_1(0,1)$. We can see here that most positive examples have this feature 1 off.

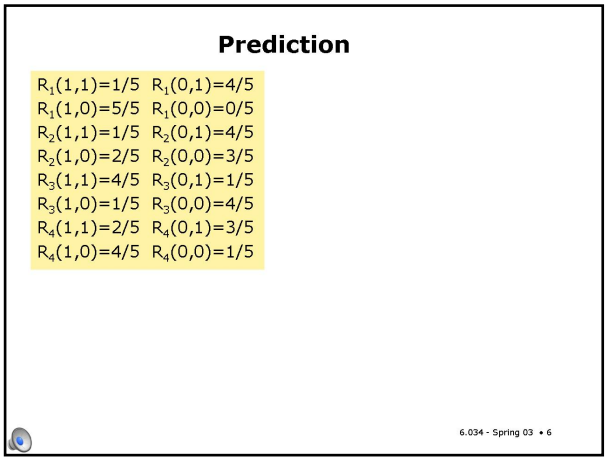



**Slide 2.5.4**
Now, we look at the negative examples, and figure out what fraction of negative examples have feature 1 on and what fraction have it off. We call these fractions $R_1(1, 0)$ and $R_1(0, 0)$. Here we see that **all** negative examples have this feature on.

**Slide 2.5.5**
We can compute these values, as shown here, for each of the other features, as well.

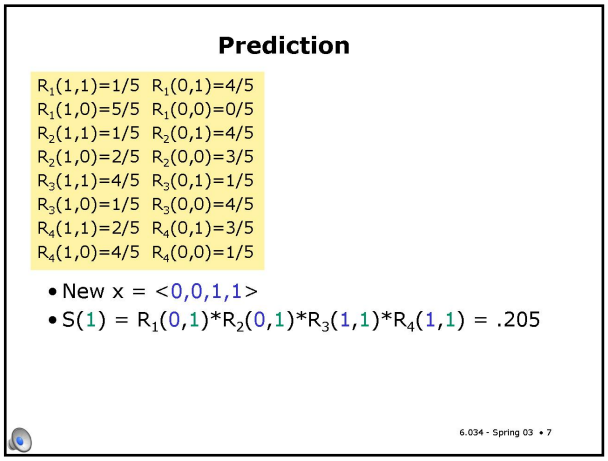


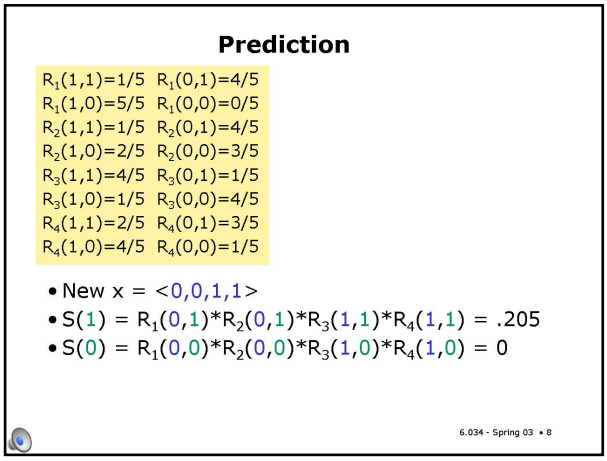




**Slide 2.5.6**
These R values actually represent our hypothesis in a way we'll see more clearly later. But that means that, given a new input x, we can use the R values to compute an output value Y.

**Slide 2.5.7**
Imagine we get a new x = <0, 0, 1, 1>. We start out by computing a "score" for this example being a positive example. We do that by multiplying the positive R values, one for each feature. So, our x has feature 1 equal to 0, so we use R1 of 0, 1. It has feature 2 equal to zero, so we use $R_2$ of 0, 1. It has feature 3 equal to 1, so we use $R_3$ of 1, 1. And so on. I've shown the feature values in blue to make it clear which arguments to the R functions they're responsible for. Similarly, I've shown the 1's that come from the fact that we're computing the positive score in green.

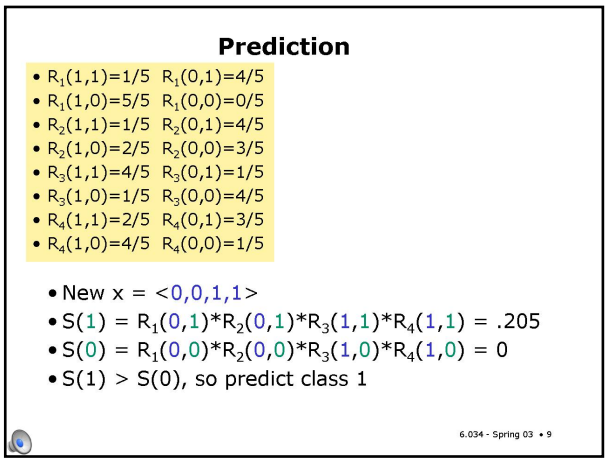
Each of the factors in the score represents the degree to which this feature tends to have this value in positive examples. Multiplied all together, they give us a measure of how likely it is that this example is positive.

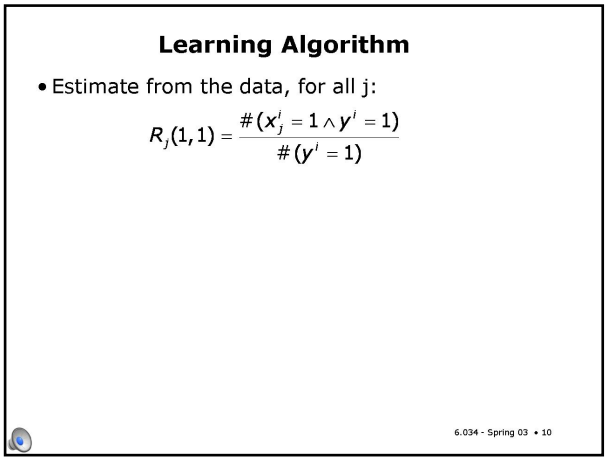



**Slide 2.5.8**
We can do the same thing to compute a score for x being a negative example. Something pretty radical happens here, because we have $R_1$ of 0, 0 equal to 0. We've never seen a negative example with feature 1 off, so we have concluded, essentially, that it's impossible for that to happen. Thus, because our x has feature 1 equal to 0, we think it's impossible for x to be a negative example.

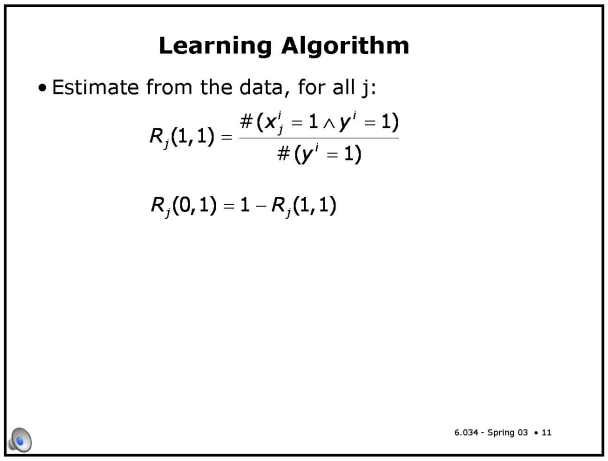
**Slide 2.5.9**
Finally, we compare score 1 to score 0, and generate output 1 because score 1 is larger than score 0.

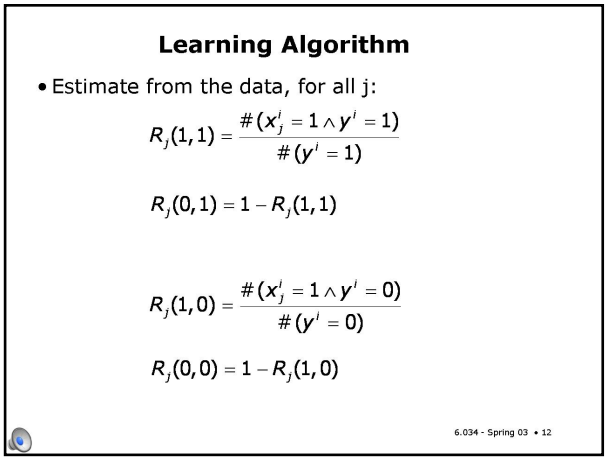



**Slide 2.5.10**
Here's the learning algorithm written out just a little bit more generally. To compute $R_j$ of 1, 1, we just count, in our data set, how many examples there have been in which feature j has had value 1 and the output was also 1, and divide that by the total number of samples with output 1.

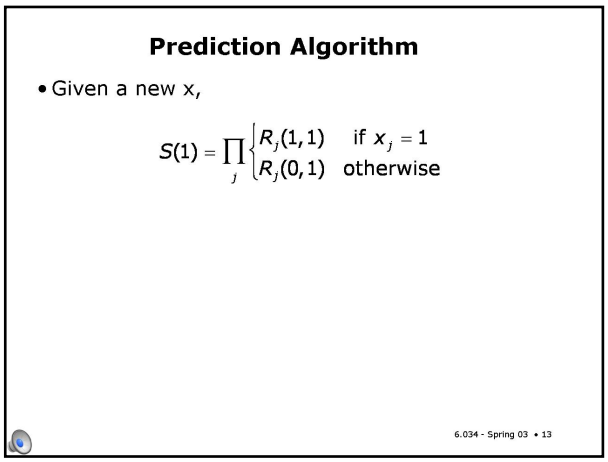
**Slide 2.5.11**
Now, $R_j$ of 0, 1 is just 1 minus $R_j$ of 1, 1.

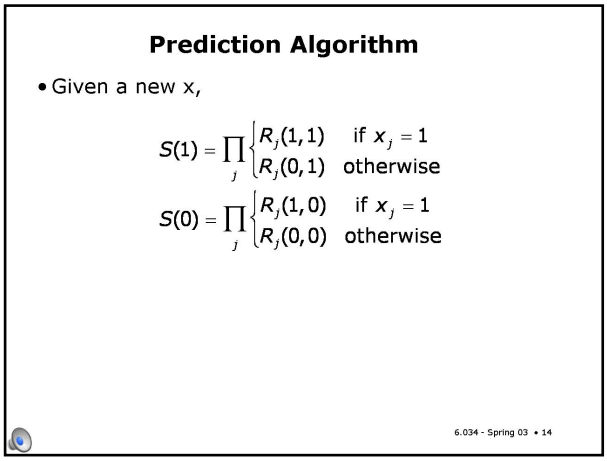



**Slide 2.5.12**
Similarly, $R_j$ of 1, 0 is the number of examples in which feature j had value 1 and the output was 0, divided the total number of examples with output 0. And $R_j$ of 0, 0 is just 1 minus $R_j$ of 1, 0.

**Slide 2.5.13**
Now, given a new example, x, let the score for class 1, S(1), be the product, over all j, of $R_j$ of 1,1 if $x_j$ = 1 and $R_j$ of 0, 1 otherwise.




**Slide 2.5.14**
Similarly, S(0) is the product, over all j, of $R_j$ of 1, 0 if $x_j$ = 1 and $R_j$ of 0,0 otherwise.

**Slide 2.5.15**
If S(1) is greater than S(0), then we'll predict that Y = 1, else 0.

## Prediction Algorithm

- Given a new x,

$$S(1) = \prod_j \begin{cases} R_j(1,1) & \text{if } x_j = 1 \\ R_j(0,1) & \text{otherwise} \end{cases}$$

$$S(0) = \prod_j \begin{cases} R_j(1,0) & \text{if } x_j = 1 \\ R_j(0,0) & \text{otherwise} \end{cases}$$

- Output 1 if $S(1) > S(0)$

6.034 - Spring 03 • 15

## Prediction Algorithm

- Given a new x,

$$\log S(1) = \sum_j \begin{cases} \log R_j(1,1) & \text{if } x_j = 1 \\ \log R_j(0,1) & \text{otherwise} \end{cases}$$

$$\log S(0) = \sum_j \begin{cases} \log R_j(1,0) & \text{if } x_j = 1 \\ \log R_j(0,0) & \text{otherwise} \end{cases}$$

- Output 1 if $\log S(1) > \log S(0)$

Better to add logs than to multiply small probabilities

6.034 - Spring 03 • 16

**Slide 2.5.16**
We can run into problems of numerical precision in our calculations if we multiply lots of probabilities together, because the numbers will rapidly get very small. One standard way to deal with this is to take logs everywhere. Now, we'll output 1 if the log of the score for 1 is greater than the log of score 0. And the log of a product is the sum of the logs of the factors.

**Slide 2.5.17**
In our example, we saw that if we had never seen a feature take value 1 in a positive example, our estimate for how likely that would be to happen in the future was 0. That seems pretty radical, especially when we only have had a few examples to learn from. There's a standard hack to fix this problem, called the "Laplace correction". When counting up events, we add a 1 to the numerator and a 2 to the denominator.

If we've never seen any positive instances, for example, our R (1,1) values would be 1/2, which seems sort of reasonable in the absence of any information. And if we see lots and lots of examples, this 1 and 2 will be washed out, and we'll converge to the same estimate that we would have gotten without the correction.

There's a beautiful probabilistic justification for what looks like an obvious hack. But, sadly, it's beyond the scope of this class.

## Laplace Correction

- Avoid getting 0 or 1 as an answer:

$$R_j(1,1) = \frac{\#(x_j^i = 1 \wedge y^i = 1) + \mathbf{1}}{\#(y^i = 1) + \mathbf{2}}$$

$$R_j(0,1) = 1 - R_j(1,1)$$

$$R_j(1,0) = \frac{\#(x_j^i = 1 \wedge y^i = 0) + \mathbf{1}}{\#(y^i = 0) + \mathbf{2}}$$

$$R_j(0,0) = 1 - R_j(1,0)$$

6.034 - Spring 03 • 17

## Example with Correction

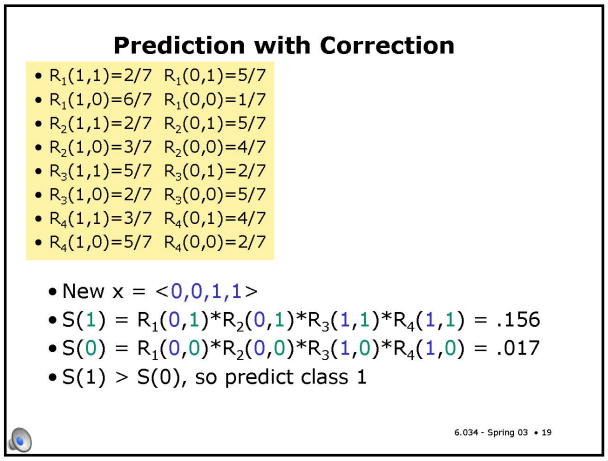
| $f_1$ | $f_2$ | $f_3$ | $f_4$ | y |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |

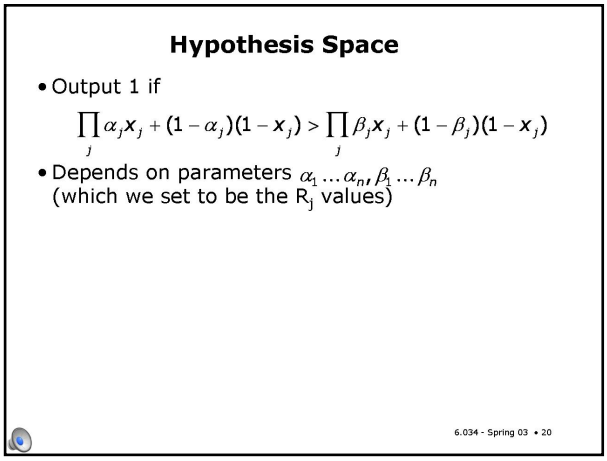
- $R_1(1,1)=2/7$ $R_1(0,1)=5/7$
- $R_1(1,0)=6/7$ **$R_1(0,0)=1/7$**
- $R_2(1,1)=2/7$ $R_2(0,1)=5/7$
- $R_2(1,0)=3/7$ $R_2(0,0)=4/7$
- $R_3(1,1)=5/7$ $R_3(0,1)=2/7$
- $R_3(1,0)=2/7$ $R_3(0,0)=5/7$
- $R_4(1,1)=3/7$ $R_4(0,1)=4/7$
- $R_4(1,0)=5/7$ $R_4(0,0)=2/7$

6.034 - Spring 03 • 18

**Slide 2.5.18**
Here's what happens to our original example if we use the Laplace correction. Notably, $R_1$ of 0, 0 is now 1/7 instead of 0, which is less dramatic.

**Slide 2.5.19**
And so, when it comes time to make a prediction, the score for answer 0 is no longer 0. We think it's possible, but unlikely, that this example is negative. So we still predict class 1.

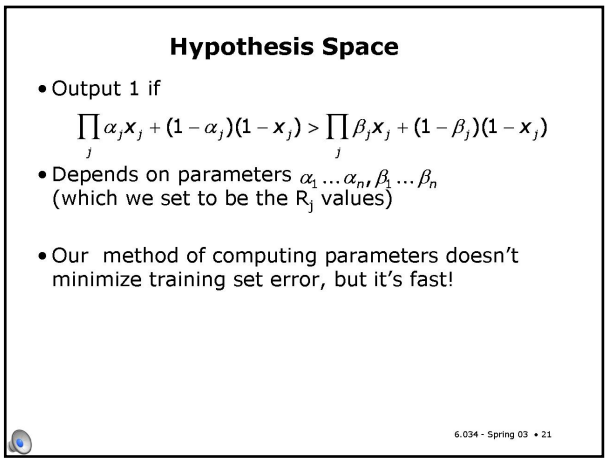


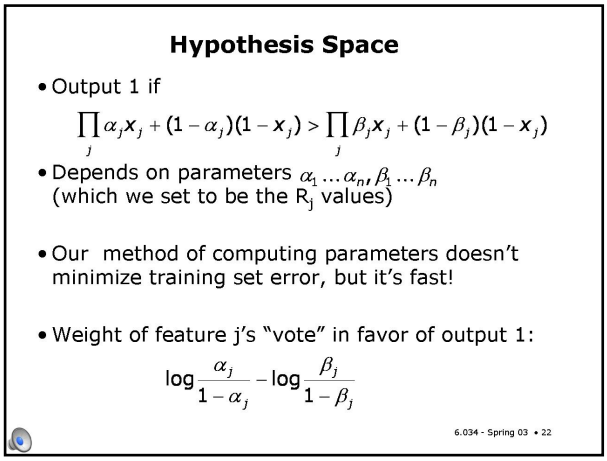




**Slide 2.5.20**
What's the story of this algorithm in terms of hypothesis space? We've fixed the spaces of hypotheses to have the form shown here. This is a very restricted form. But it is still a big (infinite, in fact) hypothesis space, because we have to pick the actual values of the coefficients $alpha_j$ and $beta_j$ for all j.

**Slide 2.5.21**
All of our bias is in the form of the hypothesis. We've restricted it significantly, so we would now like to choose the alpha's and beta's in such a way as to minimize the error on the training set. For somewhat subtle technical reasons (ask me and I'll tell you), our choice of the R scores for the alpha's and beta's doesn't exactly minimize error on the training set. But it usually works pretty well.

The main reason we like this algorithm is that it's easy to train. One pass through the data and we can compute all the parameters. It's especially useful in things like text categorization, where there are huge numbers of attributes and we can't possibly look at them many times.

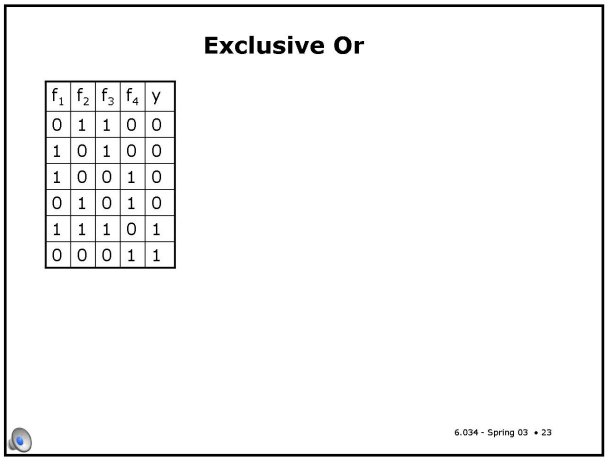


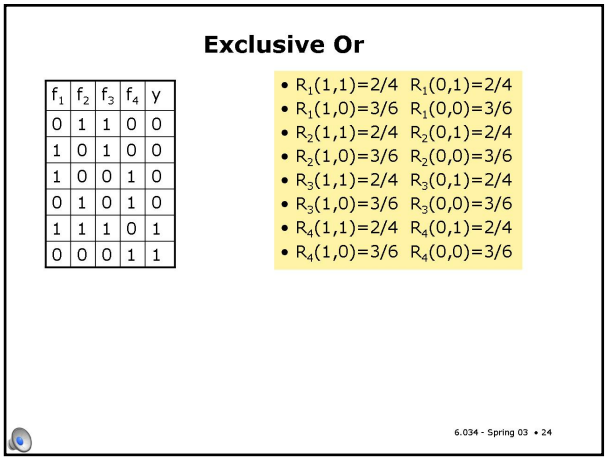




**Slide 2.5.22**
One possible concern about this algorithm is that it's hard to interpret the hypotheses you get back. With DNF or decision trees, it's easy for a human to understand what features are playing an important role, for example.

In naive Bayes, all the features are playing some role in the categorization. You can think of each one as casting a weighted vote in favor an answer of 1 versus 0. The weight of each feature's vote is this expression. The absolute value of this weight is a good indication of how important a feature is, and its sign tells us whether that feature is indicative of output 1 (when it's positive) or output 0 (when it's negative).

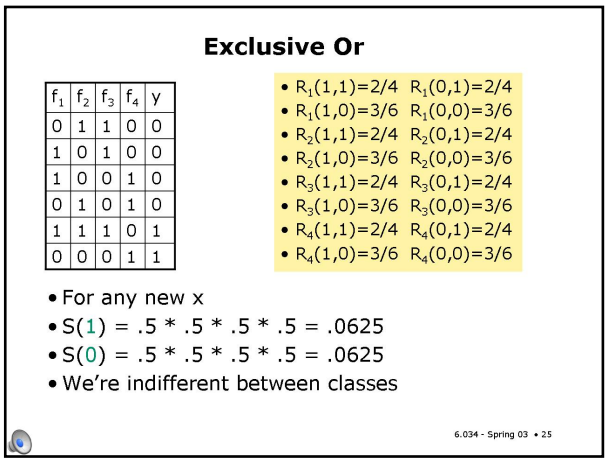
**Slide 2.5.23**
This algorithm makes a fundamental assumption that we can characterize the influence of each feature on the class independently, and then combine them through multiplication. This assumption isn't always justified. We'll illustrate this using our old nemesis, exclusive or. Here's the data set we used before.

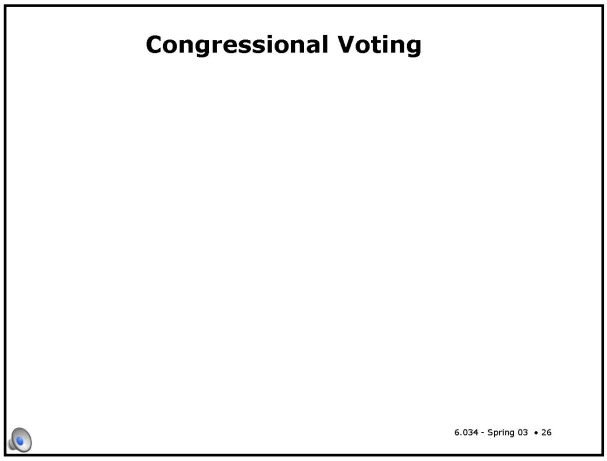



**Slide 2.5.24**
Here are the R values obtained via counting and the Laplace correction. They're all equal to 1/2, because no feature individually gives information about whether the example is positive or negative.

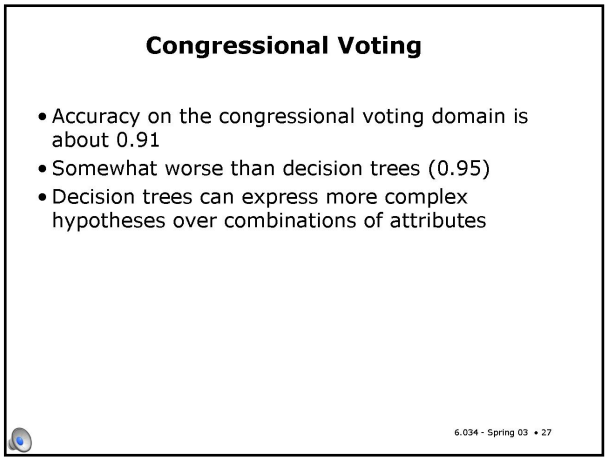
**Slide 2.5.25**
Sure enough, when we compute the scores for any new example, we get the same result, giving us no basis at all for predicting the output.

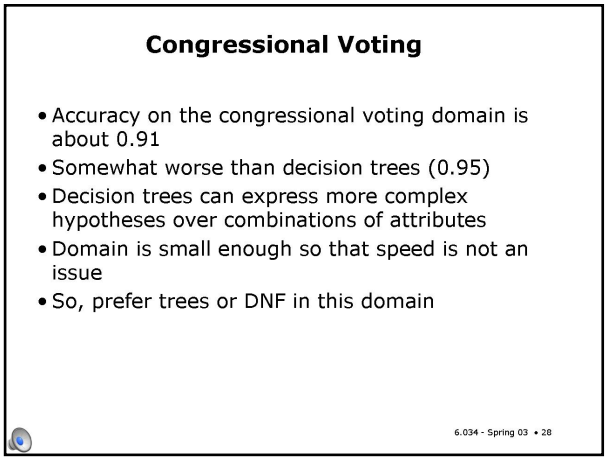



**Slide 2.5.26**
Now we show the results of applying naive Bayes to the congressional voting domain. In this case, we might expect the independence assumption to be reasonably well satisfied (a congressperson probably doesn't decide on groups of votes together, unless there are deals being made).

**Slide 2.5.27**
Using cross-validation, we determined that the accuracy of hypotheses generated by naive Bayes was approximately 0.91. This is not as good as that of decision trees, which had an accuracy of about 0.95. This result is not too surprising: decision trees can express more complex hypotheses that consider combinations of attributes.

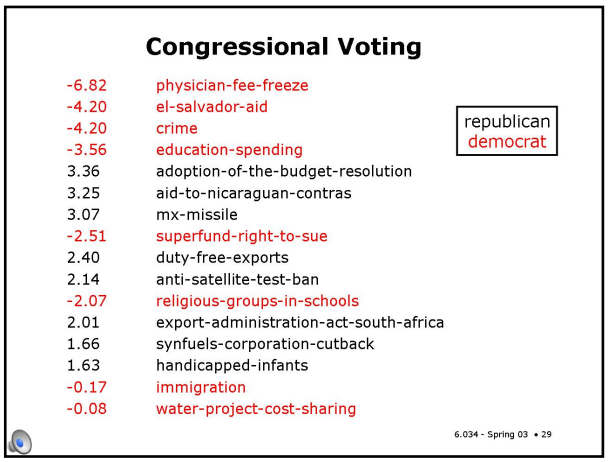


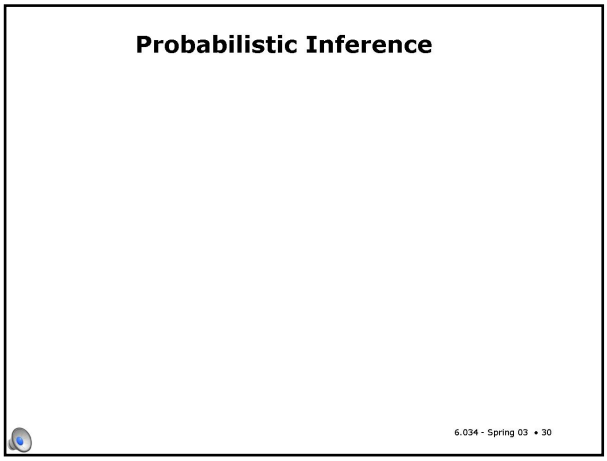




**Slide 2.5.28**
This domain is small enough that the efficiency of naive Bayes doesn't really matter. So we would prefer to use trees or DNF on this problem.

**Slide 2.5.29**
It's interesting to look at the weights found for the various attributes. Here, I've sorted the attributes according to magnitude of the weight assigned to them by naive Bayes. The positive ones (colored black) vote in favor of the output being 1 (republican); the negative ones (colored red) vote against the the output being 1 (and therefore in favor of democrat).

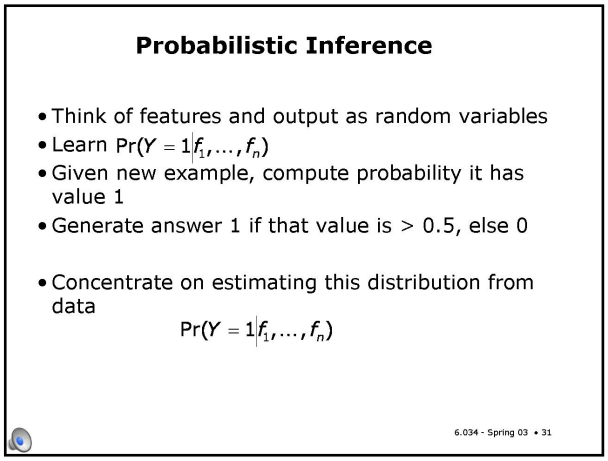
The results are consistent with the answers we've gotten from the other algorithms. The most diagnostic single issue seems to be voting on whether physician fees should be frozen; that is a strong indicator of being a democrat. The strongest indicators of being republican are accepting the budget, aid to the contras, and support of the mx missile.

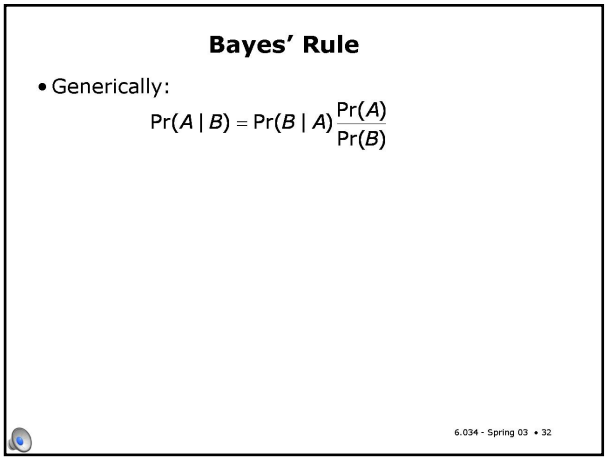



**Slide 2.5.30**
Now we'll look briefly at the probabilistic justification for the algorithm. If you don't follow it, don't worry. But if you've studied probability before, this ought to provide some useful intuition.

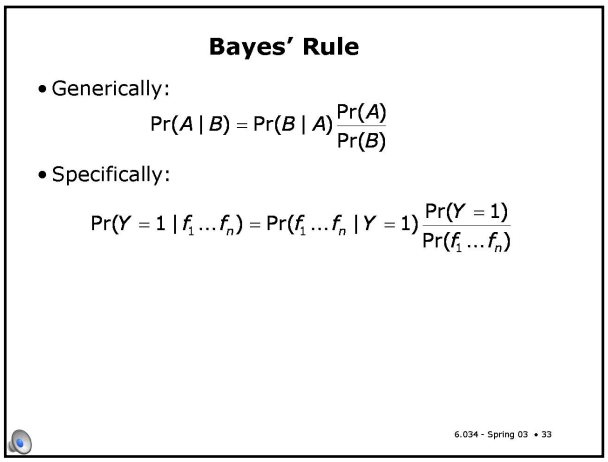
**Slide 2.5.31**
One way to think about the problem of deciding what class a new item belongs to is to think of the features and the output as random variables. If we knew Pr(Y = 1 | $f_1$ ... $f_n$), then when we got a new example, we could compute the probability that it had a Y value of 1, and generate the answer 1 if the probability was over 0.5. So, we're going to concentrate on coming up with a way to estimate this probability.

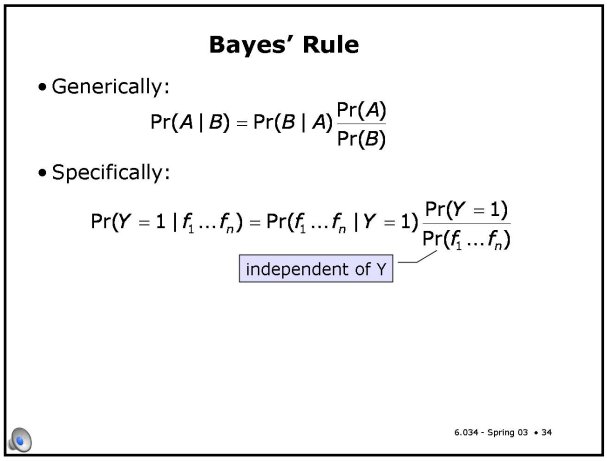



**Slide 2.5.32**
Bayes' rule gives us a way to take the conditional probability Pr(A|B) and express it in terms of Pr(B|A) and the marginals Pr(A) and Pr(B).

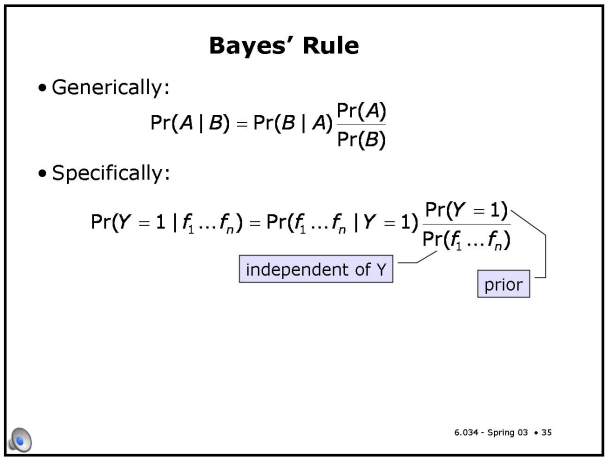
**Slide 2.5.33**
Applying it to our problem, we get Pr(Y = 1 | $f_1$ ... $f_n$) = Pr($f_1$ .. $f_n$ | Y = 1) Pr(Y = 1) / Pr($f_1$ .. $f_n$)

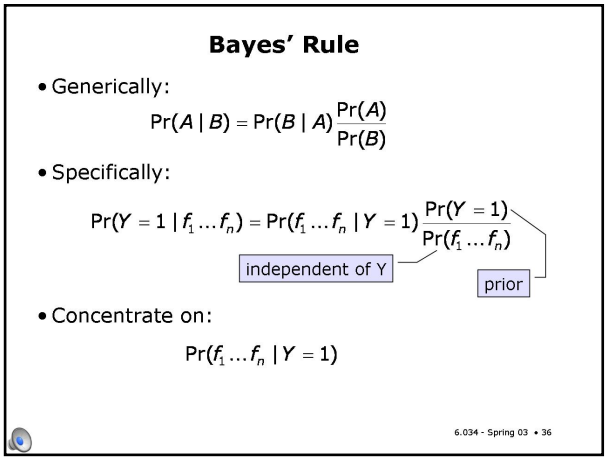



**Slide 2.5.34**
Since the denominator is independent of Y, we can ignore it in figuring out whether Pr(Y = 1 | $f_1$ ... $f_n$) is greater than Pr(Y = 0 | $f_1$ ... $f_n$).

**Slide 2.5.35**
The term Pr(Y = 1) is often called the **prior**. It's a way to build into our decision-making a previous belief about the proportion of things that are positive. For now, we'll just assume that it's 0.5 for both positive and negative classes, and ignore it.

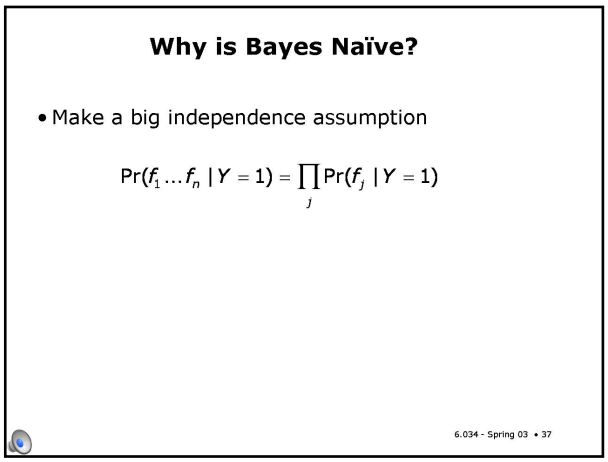


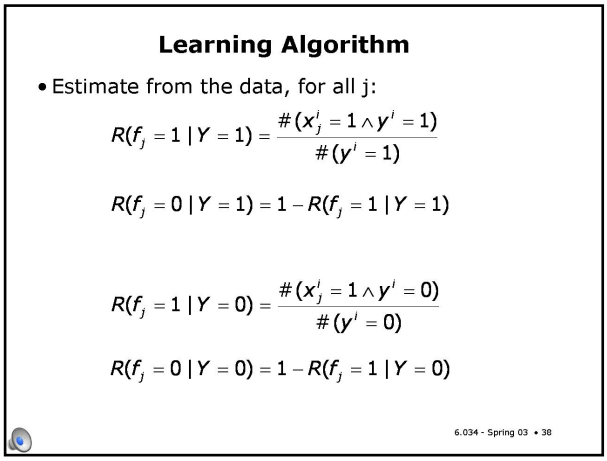




**Slide 2.5.36**
This will allow us to concentrate on $\Pr(f_1 \ldots f_n \mid Y = 1)$.

**Slide 2.5.37**
The algorithm is called **naive** Bayes because it makes a big assumption, which is that it can be broken down into a product like this. A probabilist would say that we are assuming that the features are conditionally independent given the class.

So, we're assuming that $\Pr(f_1 .. f_n \mid Y = 1)$ is the product of all the individual conditional probabilities, $\Pr(f_j \mid Y = 1)$.




**Slide 2.5.38**
Here is our same learning algorithm (without the Laplace correction, for simplicity), expressed in probabilistic terms.

We can think of the R values as estimates of the underlying conditional probabilities, based on the training set as a statistical sample drawn from those distributions.

**Slide 2.5.39**
And here's the prediction algorithm, just written out using probabilistic notation. The S is also an estimated probability. So we predict output 1 just when we think it's more likely that our new x would have come from class 1 than from class 0.

Now, we'll move on to considering the situation in which the inputs and outputs of the learning process can be real-valued.

## Prediction Algorithm

- Given a new x,

$$S(x_1 \ldots x_n \mid Y = 1) = \prod_j \begin{cases} R(f_j = 1 \mid Y = 1) & \text{if } x_j = 1 \\ R(f_j = 0 \mid Y = 1) & \text{otherwise} \end{cases}$$

$$S(x_1 \ldots x_n \mid Y = 0) = \prod_j \begin{cases} R(f_j = 1 \mid Y = 0) & \text{if } x_j = 1 \\ R(f_j = 0 \mid Y = 0) & \text{otherwise} \end{cases}$$

- Output 1 if

$$S(x_1 \ldots x_n \mid Y = 1) > S(x_1 \ldots x_n \mid Y = 0)$$

6.034 - Spring 03 • 39