6.034 Notes: Section 5.1

Slide 5.1.1
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.
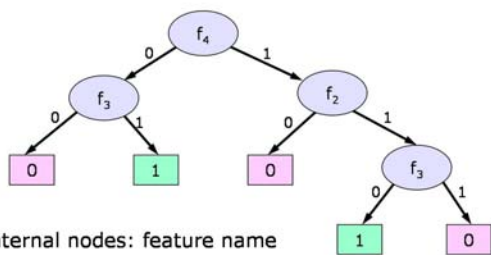
So now we're going to look at an algorithm for learning decision trees. We'll be changing our hypothesis class (sort of), our bias, and our algorithm. We'll continue to assume, for now, that the input features and the output class are boolean. We'll see later that this algorithm can be applied much more broadly.

Slide 5.1.2
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 5.1.3
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.
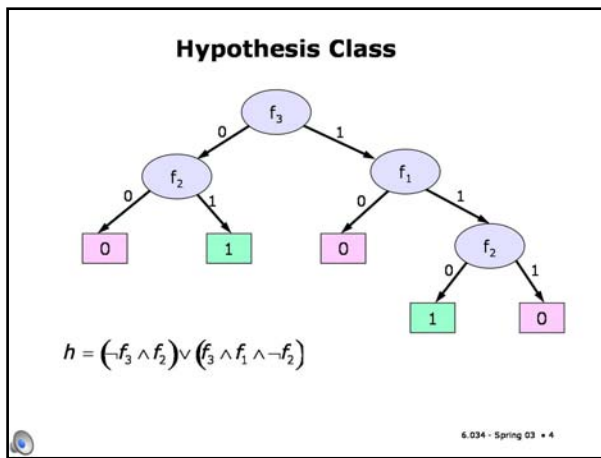
So, in this example, input [0 1 1 0] would generate an output of 0 (because the third element of the input has value 1 and the first has value 0, which takes to a leaf labeled 0).

Hypothesis Class

$$h = \left(\neg f_3 \wedge f_2\right) \vee \left(f_3 \wedge f_1 \wedge \neg f_2\right)$$
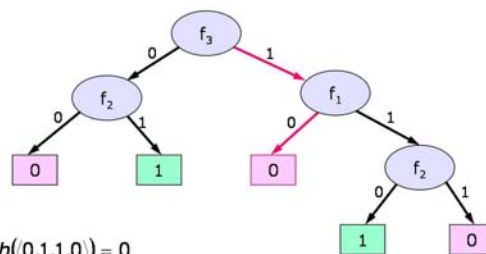
Slide 5.1.4
Decision trees are a way of writing down Boolean expressions. To convert a tree into an expression, you can make each branch of the tree with a 1 at the leaf node into a conjunction, describing the condition that had to hold of the input in order for you to have gotten to that leaf of the 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 5.1.5
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.

Tree Bias

- Both decision trees and DNF with negation can represent any Boolean function. So why bother with trees?
- Because we have a nice algorithm for growing trees that is consistent with a bias for simple trees (few nodes)

Tree Bias

- Both decision trees and DNF with negation can represent any Boolean function. So why bother with trees?
- Because we have a nice algorithm for growing trees that is consistent with a bias for simple trees (few nodes)
- Too hard to find the smallest good tree, so we'll be greedy again
- Have to watch out for overfitting

Slide 5.1.6
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 5.1.7
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.

Trees vs DNF

$$(\neg F \wedge \neg H) \vee (\neg F \wedge H \wedge J) \vee (F \wedge \neg G \wedge K) \vee (F \wedge G)$$

Trees vs DNF

$(F \wedge G) \vee (H \wedge J)$
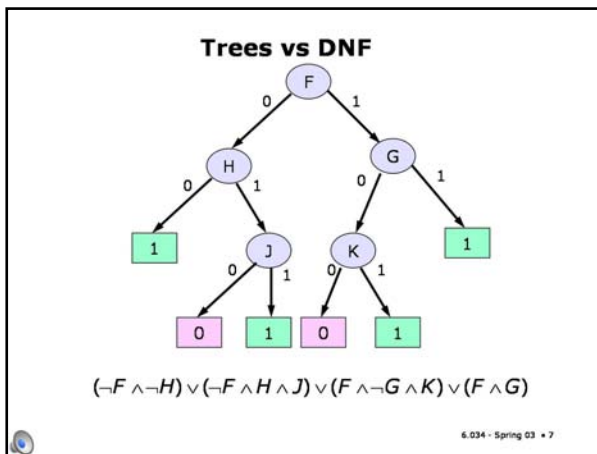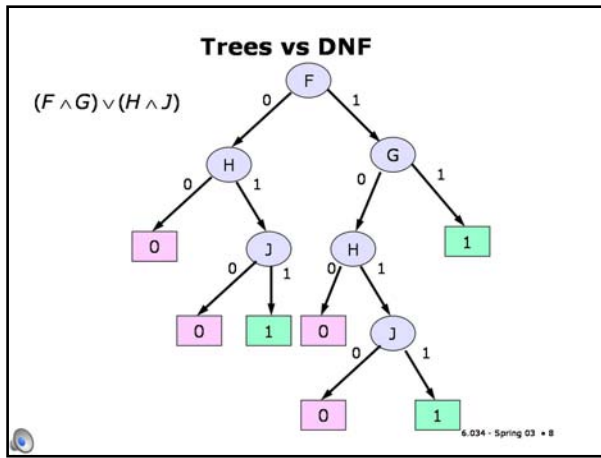
Slide 5.1.8
But here's a case with a very simple DNF expression that requires a large tree to represent it. There's no particular reason to prefer trees over DNF or DNF over trees as a hypothesis class. But the tree-growing algorithm is simple and elegant, so we'll study it.

Slide 5.1.9
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.

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olsen and Stone

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olsen and Stone

```
BuildTree (Data)
```

Slide 5.1.10
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 5.1.11
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.

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olsen and Stone

```
BuildTree (Data)
    if all elements of Data have the same y value, then
        MakeLeafNode(y)
```

Algorithm

- Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olsen and Stone

```
BuildTree (Data)
    if all elements of Data have the same y value, then
        MakeLeafNode(y)
    else
        feature := PickBestFeature(Data)
        MakeInternalNode(feature,
            BuildTree(SelectFalse(Data, feature)),
            BuildTree(SelectTrue(Data, feature)))
```

6.034 - Spring 03 • 12

Slide 5.1.12
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 5.1.13
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.
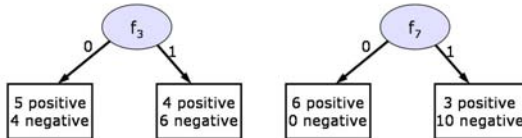
Let's Split

D: 9 positive
10 negative

6.034 - Spring 03 • 13

Let's Split

D: 9 positive
10 negative

f_3

0   1

5 positive
4 negative

4 positive
6 negative

f_7

0   1

6 positive
0 negative

3 positive
10 negative

6.034 - Spring 03 • 14

Slide 5.1.14
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 5.1.15
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
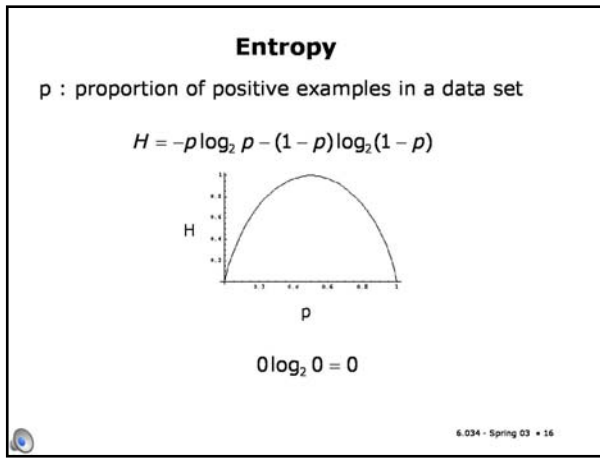
$$- p \log_2 p - (1 - p) \log_2 (1 - p)$$

Entropy

p : proportion of positive examples in a data set

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

6.034 - Spring 03 • 15

Slide 5.1.16

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 5.1.17

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.
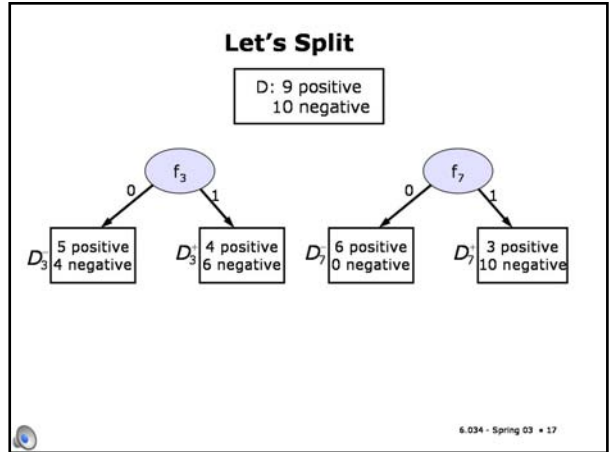
Slide 5.1.18

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.
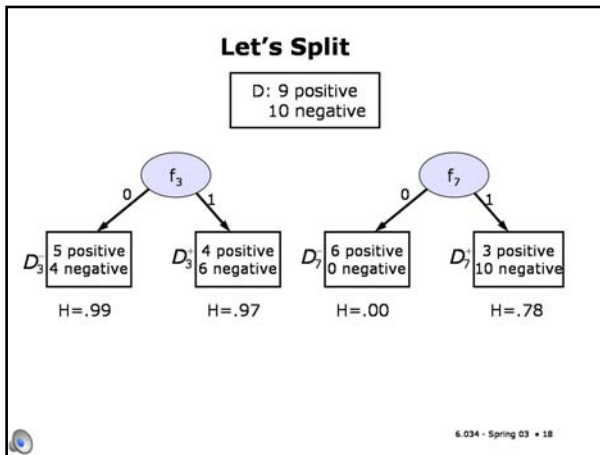
Slide 5.1.19

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 $AE(j) = p_j H(D^+_j) + (1 - p_j)H(D^-_j)$

Let's Split

D: 9 positive
10 negative

f_3 0 / 1

f_7 0 / 1

| 5 positive 4 negative | 4 positive 6 negative | 6 positive 0 negative | 3 positive 10 negative |

H=.99   H=.97   H=.00   H=.78

AE = (9/19)*.99+(10/19)*.97   AE = (6/19)*0+(13/19)*.78
= .98   = .53

6.034 - Spring 03 • 20
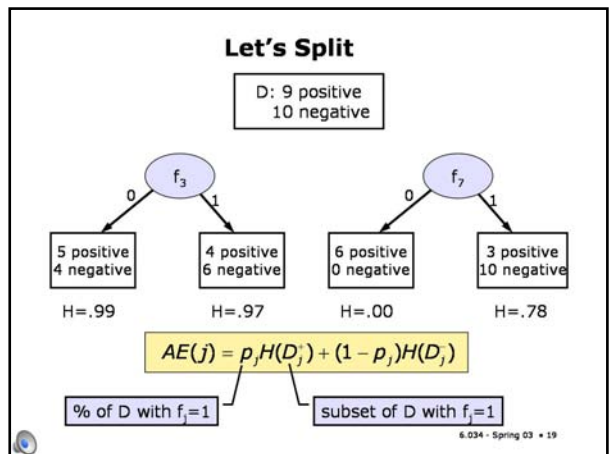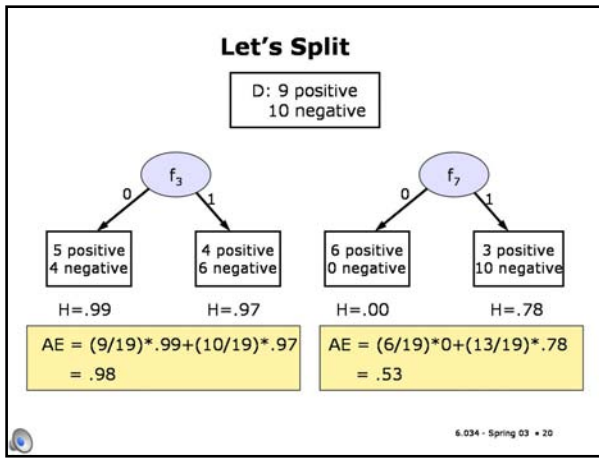
Slide 5.1.20
So, in this example, we can see that the split on f_7 is much more useful, as reflected in the average entropy of its children.

Slide 5.1.21
Going back to our algorithm, then, we'll pick the feature at every step that minimizes the weighted average entropy of the children.

Algorithm

• Developed in parallel in AI by Quinlan and in statistics by Breiman, Friedman, Olshen and Stone

```
BuildTree (Data)
    if all elements of Data have the same y value, then
        MakeLeafNode(y)
    else
        feature := PickBestFeature(Data)
        MakeInternalNode(feature,
            BuildTree(SelectFalse(Data, feature)),
            BuildTree(SelectTrue(Data, feature)))
```

• Best feature minimizes average entropy of data in the children

6.034 - Spring 03 • 21

Stopping

Image removed due to copyright restrictions.

• Stop recursion if data contains only multiple instances of the same x with different y values

6.034 - Spring 03 • 22

Slide 5.1.22
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 5.1.23
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.

Stopping

Image removed due to copyright restrictions.

• Stop recursion if data contains only multiple instances of the same x with different y values
   • Make leaf node with output equal to the y value that occurs in the majority of the cases in the data

6.034 - Spring 03 • 23

Stopping

Image removed due to copyright restrictions ·

- Stop recursion if data contains only multiple instances of the same x with different y values
  - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data
- Consider stopping to avoid overfitting when:

6.034 - Spring 03 · 24

Slide 5.1.24
But growing the tree as far down as we can will often result in overfitting.

Slide 5.1.25
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.

Stopping

Image removed due to copyright restrictions.

- Stop recursion if data contains only multiple instances of the same x with different y values
  - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data
- Consider stopping to avoid overfitting when:
  - entropy of a data set is below some threshold

6.034 - Spring 03 · 25

Stopping

Image removed due to copyright restrictions.

- Stop recursion if data contains only multiple instances of the same x with different y values
  - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data
- Consider stopping to avoid overfitting when:
  - entropy of a data set is below some threshold
  - number elements in a data set is below threshold

6.034 - Spring 03 · 26

Slide 5.1.26
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 5.1.27
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.

Stopping

Image removed due to copyright restrictions.

- Stop recursion if data contains only multiple instances of the same x with different y values
  - Make leaf node with output equal to the y value that occurs in the majority of the cases in the data
- Consider stopping to avoid overfitting when:
  - entropy of a data set is below some threshold
  - number elements in a data set is below threshold
  - best next split doesn't decrease average entropy (but this can get us into trouble)

6.034 - Spring 03 · 27

Simulation

- $H(D) = .92$
- $AE_1=.92$, $AE_2=.92$, $AE_3=.81$, $AE_4=1$

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	0

6.034 - Spring 03 • 28

Slide 5.1.28
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 5.1.29
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).

Simulation

f_3 0 / 1

f_1	f_2	f_3	f_4	y
1	0	0	1	0

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
0	1	1	1	0

6.034 - Spring 03 • 29

Simulation

f_3 0 / 1

0

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
0	1	1	1	0

6.034 - Spring 03 • 30

Slide 5.1.30
So we make it into a leaf with output 0 and consider splitting the data set in the right child.

Slide 5.1.31
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.

Simulation

- $AE_1=.55$, $AE_2=.55$, $AE_4=.95$

f_3 0 / 1

0

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
0	1	1	1	0

6.034 - Spring 03 • 31
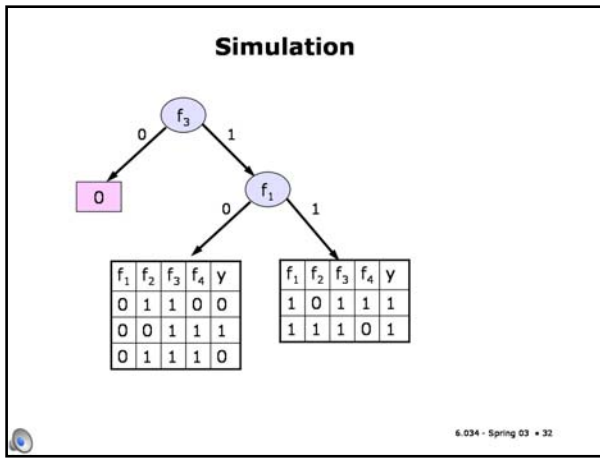
Slide 5.1.32
So, we decide, arbitrarily, to split on feature 1, yielding these sub-problems. All of the examples in the right-hand child have the same output.

Slide 5.1.33
So we can turn it into a leaf with output 1.

In the left child, feature 2 will be the most useful.
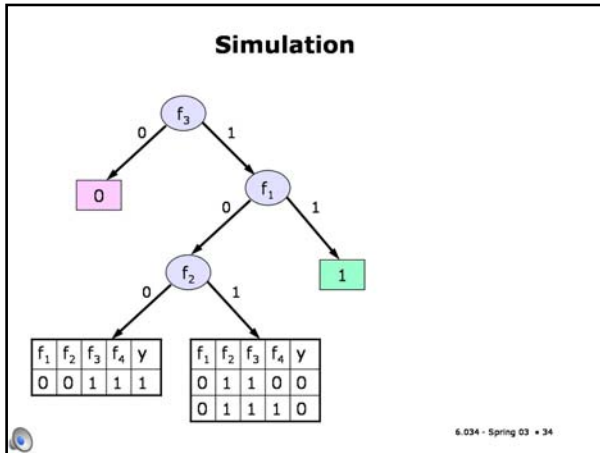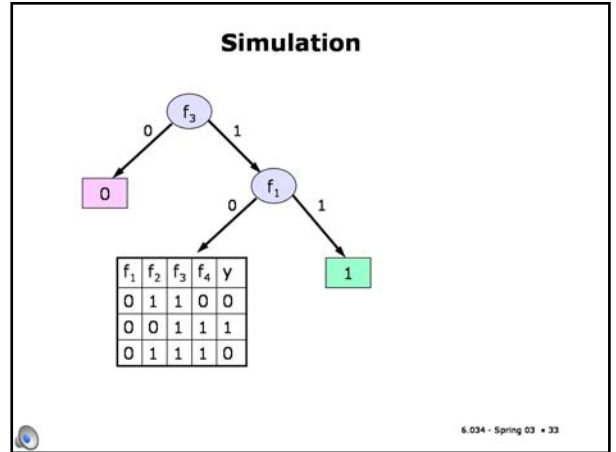
Slide 5.1.34
So we split on it, and now both children are homogeneous (all data points have the same output).

Slide 5.1.35
So we make the leaves and we're done!

Exclusive OR

$$(A \wedge \neg B) \vee (\neg A \wedge B)$$

Slide 5.1.36
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-in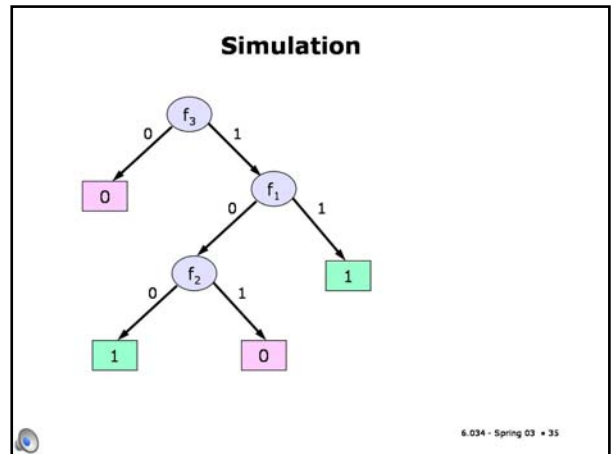put boolean function that has value 1 if one input has value 1 and the other has value 0. If both inputs are 1 or both are 0, then the output is 0. This function is hard to deal with, because neither input feature is, by itself, detectably related to the output. So, local methods that try to add one feature at a time can be easily misled by xor.

Slide 5.1.37
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.

Exclusive OR

$$(A \wedge \neg B) \vee (\neg A \wedge B)$$

- $H(D) = .92$
- $AE_1 = .92$, $AE_2 = .92$,
  $AE_3 = .92$, $AE_4 = .92$

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

Slide 5.1.38
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.

Exclusive OR

f_1 0   1

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 5.1.39
So we split on it, and get homogenous children,

Exclusive OR

f_1 0   1

f_2 0   1

f_1	f_2	f_3	f_4	y
0	0	0	1	1

f_1	f_2	f_3	f_4	y
0	1	1	0	0
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

Exclusive OR
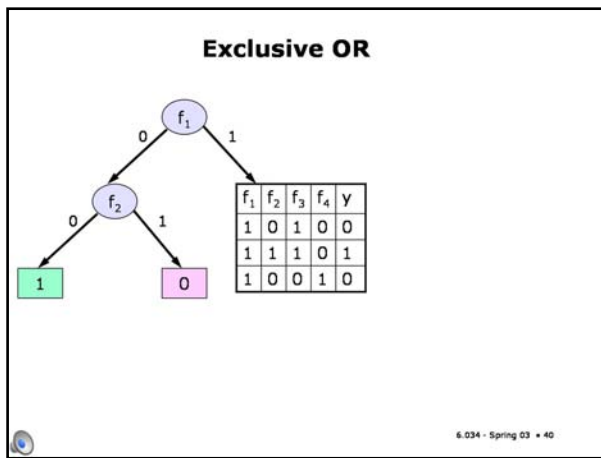
Slide 5.1.40

Which we can replace with leaves.

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

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 5.1.41

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.

Exclusive OR
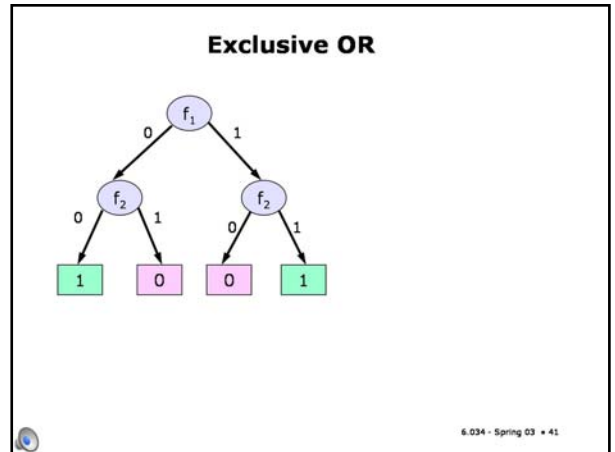
Pruning

- Best way to avoid overfitting and not get tricked by short-term lack of progress
  - Grow tree as far as possible
    - leaves are uniform or contain a single X
  - Prune the tree until it performs well on held-out data
  - Amount of pruning is like epsilon in the DNF algorithm

Slide 5.1.42

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 5.1.43

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.

Congressional Voting

min leaf size = 20
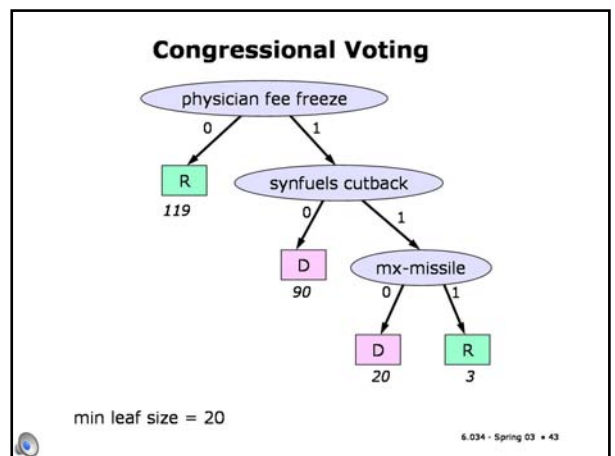
Data Mining

- Making useful predictions in (usually corporate) applications
- Decision trees very popular because
  - easy to implement
  - efficient (even on huge data sets)
  - easy for humans to understand resulting hypotheses

6.034 - Spring 03 • 44

Slide 5.1.44
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 5.2

Slide 5.2.1
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.

Naïve Bayes

- Founded on Bayes' rule for probabilistic inference
- Update probability of hypotheses based on evidence
- Choose hypothesis with the maximum probability after the evidence has been incorporated

Image of Rev. Thomas Bayes removed due to copyright restrictions.

Rev. Thomas Bayes

6.034 - Spring 03 • 1

Naïve Bayes

- Founded on Bayes' rule for probabilistic inference
- Update probability of hypotheses based on evidence
- Choose hypothesis with the maximum probability after the evidence has been incorporated
- Algorithm is particularly useful for domains with **lots** of features

Image of Rev. Thomas Bayes removed due to copyright restrictions.

Rev. Thomas Bayes

6.034 - Spring 03 • 2

Slide 5.2.2
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 5.2.3
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.

Example

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

- $R_1(1,1)=1/5$: fraction of all **positive** examples that have feature 1 **on**
- $R_1(0,1)=4/5$: fraction of all **positive** examples that have feature 1 **off**

6.034 - Spring 03 • 3

Example

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

- $R_1(1,1)=1/5$: fraction of all **positive** examples that have feature 1 **on**
- $R_1(0,1)=4/5$: fraction of all **positive** examples that have feature 1 **off**
- $R_1(1,0)=5/5$: fraction of all **negative** examples that have feature 1 **on**
- $R_1(0,0)=0/5$: fraction of all **negative** examples that have feature 1 **off**

6.034 - Spring 03 • 4

Slide 5.2.4
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 5.2.5
We can compute these values, as shown here, for each of the other features, as well.

Example

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

$R_1(1,1)=1/5$ $R_1(0,1)=4/5$
$R_1(1,0)=5/5$ $R_1(0,0)=0/5$

$R_2(1,1)=1/5$ $R_2(0,1)=4/5$
$R_2(1,0)=2/5$ $R_2(0,0)=3/5$

$R_3(1,1)=4/5$ $R_3(0,1)=1/5$
$R_3(1,0)=1/5$ $R_3(0,0)=4/5$

$R_4(1,1)=2/5$ $R_4(0,1)=3/5$
$R_4(1,0)=4/5$ $R_4(0,0)=1/5$

6.034 - Spring 03 • 5

Prediction

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

6.034 - Spring 03 • 6

Slide 5.2.6
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 5.2.7

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.

Prediction

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

- New x = <0,0,1,1>
- S(1) = $R_1(0,1)*R_2(0,1)*R_3(1,1)*R_4(1,1)$ = .205

6.034 - Spring 03 • 7

Prediction

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

- New x = <0,0,1,1>
- S(1) = $R_1(0,1)*R_2(0,1)*R_3(1,1)*R_4(1,1)$ = .205
- S(0) = $R_1(0,0)*R_2(0,0)*R_3(1,0)*R_4(1,0)$ = 0

6.034 - Spring 03 • 8

Slide 5.2.8

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 5.2.9

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

Prediction

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

- New x = <0,0,1,1>
- S(1) = $R_1(0,1)*R_2(0,1)*R_3(1,1)*R_4(1,1)$ = .205
- S(0) = $R_1(0,0)*R_2(0,0)*R_3(1,0)*R_4(1,0)$ = 0
- S(1) > S(0), so predict class 1

6.034 - Spring 03 • 9

Learning Algorithm

- Estimate from the data, for all j:

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

6.034 - Spring 03 • 10

Slide 5.2.10

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 5.2.11
Now, R_j of 0, 1 is just 1 minus R_j of 1, 1.

Learning Algorithm

- Estimate from the data, for all j:

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

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

6.034 - Spring 03 • 11

Learning Algorithm

- Estimate from the data, for all j:

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

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

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

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

6.034 - Spring 03 • 12

Slide 5.2.12
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 5.2.13
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.

Prediction Algorithm

- Given a new x,

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

6.034 - Spring 03 • 13

Prediction Algorithm

- Given a new x,

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

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

6.034 - Spring 03 • 14

Slide 5.2.14
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 5.2.15
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 5.2.16
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 5.2.17
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:

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

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

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

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

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 5.2.18
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 5.2.19

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.

Prediction with Correction

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

- New $x = <0,0,1,1>$
- $S(1) = R_1(0,1)*R_2(0,1)*R_3(1,1)*R_4(1,1) = .156$
- $S(0) = R_1(0,0)*R_2(0,0)*R_3(1,0)*R_4(1,0) = .017$
- $S(1) > S(0)$, so predict class 1

6.034 - Spring 03 • 19

Hypothesis Space

- Output 1 if

$$\prod_j \alpha_j x_j + (1 - \alpha_j)(1 - x_j) > \prod_j \beta_j x_j + (1 - \beta_j)(1 - x_j)$$

- Depends on parameters $\alpha_1 \dots \alpha_n, \beta_1 \dots \beta_n$ (which we set to be the R_j values)

6.034 - Spring 03 • 20

Slide 5.2.20

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 α_j and β_j for all j.

Slide 5.2.21

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.

Hypothesis Space

- Output 1 if

$$\prod_j \alpha_j x_j + (1 - \alpha_j)(1 - x_j) > \prod_j \beta_j x_j + (1 - \beta_j)(1 - x_j)$$

- Depends on parameters $\alpha_1 \dots \alpha_n, \beta_1 \dots \beta_n$ (which we set to be the R_j values)

- Our method of computing parameters doesn't minimize training set error, but it's fast!

6.034 - Spring 03 • 21

Hypothesis Space

- Output 1 if

$$\prod_j \alpha_j x_j + (1 - \alpha_j)(1 - x_j) > \prod_j \beta_j x_j + (1 - \beta_j)(1 - x_j)$$

- Depends on parameters $\alpha_1 \dots \alpha_n, \beta_1 \dots \beta_n$ (which we set to be the R_j values)

- Our method of computing parameters doesn't minimize training set error, but it's fast!

- Weight of feature j's "vote" in favor of output 1:

$$\log \frac{\alpha_j}{1 - \alpha_j} - \log \frac{\beta_j}{1 - \beta_j}$$

6.034 - Spring 03 • 22

Slide 5.2.22

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 5.2.23

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.

Exclusive Or

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	0	0
1	0	0	1	0
0	1	0	1	0
1	1	1	0	1
0	0	0	1	1

Slide 5.2.24

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.

Exclusive Or

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	0	0
1	0	0	1	0
0	1	0	1	0
1	1	1	0	1
0	0	0	1	1

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

Slide 5.2.25

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.

Exclusive Or

f_1	f_2	f_3	f_4	y
0	1	1	0	0
1	0	1	0	0
1	0	0	1	0
0	1	0	1	0
1	1	1	0	1
0	0	0	1	1

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

- For any new x
- $S(1) = .5 * .5 * .5 * .5 = .0625$
- $S(0) = .5 * .5 * .5 * .5 = .0625$
- We're indifferent between classes

Slide 5.2.26

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).

Congressional Voting

Slide 5.2.27

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.

Congressional Voting

- Accuracy on the congressional voting domain is about 0.91
- Somewhat worse than decision trees (0.95)
- Decision trees can express more complex hypotheses over combinations of attributes

6.034 - Spring 03 • 27

Congressional Voting

- Accuracy on the congressional voting domain is about 0.91
- Somewhat worse than decision trees (0.95)
- Decision trees can express more complex hypotheses over combinations of attributes
- Domain is small enough so that speed is not an issue
- So, prefer trees or DNF in this domain

6.034 - Spring 03 • 28

Slide 5.2.28

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 5.2.29

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.

Congressional Voting

-6.82	physician-fee-freeze
-4.20	el-salvador-aid
-4.20	crime
-3.56	education-spending
3.36	adoption-of-the-budget-resolution
3.25	aid-to-nicaraguan-contras
3.07	mx-missile
-2.51	superfund-right-to-sue
2.40	duty-free-exports
2.14	anti-satellite-test-ban
-2.07	religious-groups-in-schools
2.01	export-administration-act-south-africa
1.66	synfuels-corporation-cutback
1.63	handicapped-infants
-0.17	immigration
-0.08	water-project-cost-sharing

republican
democrat

6.034 - Spring 03 • 29

Probabilistic Inference

6.034 - Spring 03 • 30

Slide 5.2.30

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 5.2.31
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 $Pr(Y = 1 \mid f_1 \ldots f_n)$, 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.

Probabilistic Inference

- Think of features and output as random variables
- Learn $Pr(Y = 1 \mid f_1, \ldots, f_n)$
- Given new example, compute probability it has value 1
- Generate answer 1 if that value is > 0.5, else 0

- Concentrate on estimating this distribution from data

$$Pr(Y = 1 \mid f_1, \ldots, f_n)$$

6.034 - Spring 03 • 31

Bayes' Rule

- Generically:

$$Pr(A \mid B) = Pr(B \mid A)\frac{Pr(A)}{Pr(B)}$$

6.034 - Spring 03 • 32

Slide 5.2.32
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 5.2.33
Applying it to our problem, we get $Pr(Y = 1 \mid f_1 \ldots f_n) = Pr(f_1 \ldots f_n \mid Y = 1) \, Pr(Y = 1) / Pr(f_1 \ldots f_n)$

Bayes' Rule

- Generically:

$$Pr(A \mid B) = Pr(B \mid A)\frac{Pr(A)}{Pr(B)}$$

- Specifically:

$$Pr(Y = 1 \mid f_1 \ldots f_n) = Pr(f_1 \ldots f_n \mid Y = 1)\frac{Pr(Y = 1)}{Pr(f_1 \ldots f_n)}$$

6.034 - Spring 03 • 33

Bayes' Rule

- Generically:

$$Pr(A \mid B) = Pr(B \mid A)\frac{Pr(A)}{Pr(B)}$$

- Specifically:

$$Pr(Y = 1 \mid f_1 \ldots f_n) = Pr(f_1 \ldots f_n \mid Y = 1)\frac{Pr(Y = 1)}{Pr(f_1 \ldots f_n)}$$

independent of Y

6.034 - Spring 03 • 34

Slide 5.2.34
Since the denominator is independent of Y, we can ignore it in figuring out whether $Pr(Y = 1 \mid f_1 \ldots f_n)$ is greater than $Pr(Y = 0 \mid f_1 \ldots f_n)$.

Slide 5.2.35
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.

Bayes' Rule

- Generically:

$$\Pr(A \mid B) = \Pr(B \mid A)\frac{\Pr(A)}{\Pr(B)}$$

- Specifically:

$$\Pr(Y = 1 \mid f_1 \ldots f_n) = \Pr(f_1 \ldots f_n \mid Y = 1)\frac{\Pr(Y = 1)}{\Pr(f_1 \ldots f_n)}$$

independent of Y

prior

6.034 - Spring 03 • 35

Bayes' Rule

- Generically:

$$\Pr(A \mid B) = \Pr(B \mid A)\frac{\Pr(A)}{\Pr(B)}$$

- Specifically:

$$\Pr(Y = 1 \mid f_1 \ldots f_n) = \Pr(f_1 \ldots f_n \mid Y = 1)\frac{\Pr(Y = 1)}{\Pr(f_1 \ldots f_n)}$$

independent of Y

prior

- Concentrate on:

$$\Pr(f_1 \ldots f_n \mid Y = 1)$$

6.034 - Spring 03 • 36

Slide 5.2.36
This will allow us to concentrate on $\Pr(f_1 \ldots f_n \mid Y = 1)$.

Slide 5.2.37
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 $\Pr(f_1 \ldots f_n \mid Y = 1)$ is the product of all the individual conditional probabilities, $\Pr(f_j \mid Y = 1)$.

Why is Bayes Naïve?

- Make a big independence assumption

$$\Pr(f_1 \ldots f_n \mid Y = 1) = \prod_j \Pr(f_j \mid Y = 1)$$

6.034 - Spring 03 • 37

Learning Algorithm

- Estimate from the data, for all j:

$$R(f_j = 1 \mid Y = 1) = \frac{\#(x_j^i = 1 \wedge y^i = 1)}{\#(y^i = 1)}$$

$$R(f_j = 0 \mid Y = 1) = 1 - R(f_j = 1 \mid Y = 1)$$

$$R(f_j = 1 \mid Y = 0) = \frac{\#(x_j^i = 1 \wedge y^i = 0)}{\#(y^i = 0)}$$

$$R(f_j = 0 \mid Y = 0) = 1 - R(f_j = 1 \mid Y = 0)$$

6.034 - Spring 03 • 38

Slide 5.2.38
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 5.2.39

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