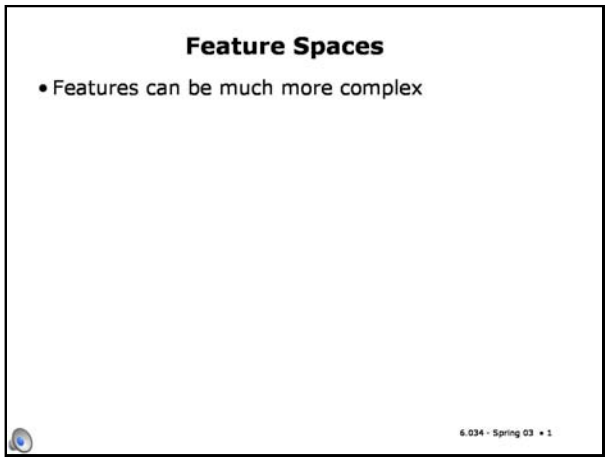
6.034 Notes: Section 10.1

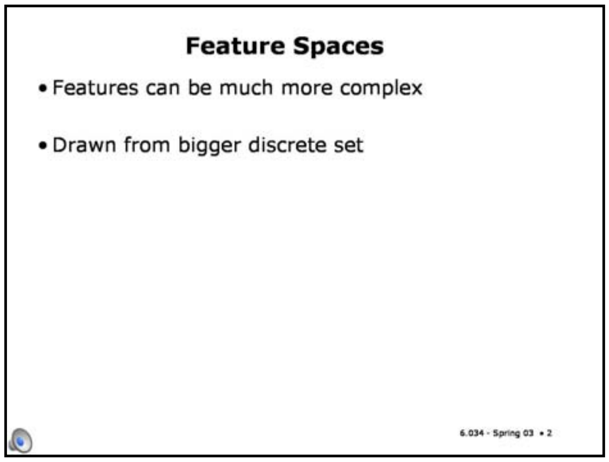
Slide 10.1.1
So far, we've only talked about binary features. But real problems are typically characterized by much more complex features.



Feature Spaces

- Features can be much more complex

6.034 - Spring 03 • 1



Feature Spaces

- Features can be much more complex
- Drawn from bigger discrete set

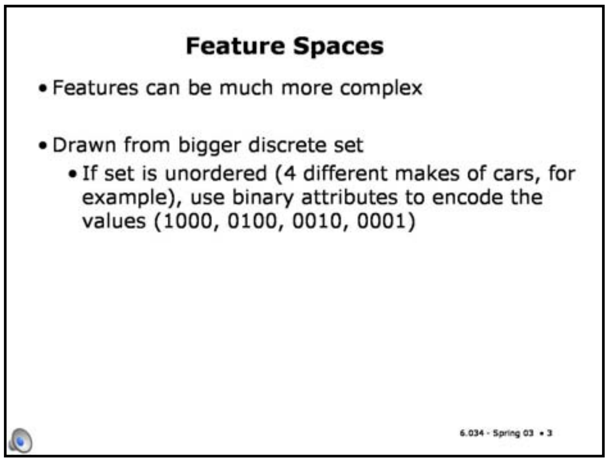
6.034 - Spring 03 • 2

Slide 10.1.2
Some features can take on values in a discrete set that has more than two elements. Examples might be the make of a car, or the age of a person.

Slide 10.1.3
When the set doesn't have a natural order (actually, when it doesn't have a natural distance between the elements), then the easiest way to deal with it is to convert it into a bunch of binary attributes.

Your first thought might be to convert it using binary numbers, so that if you have four elements, you can encode them as 00, 01, 10, and 11. Although that could work, it makes hard work for the learning algorithm, which, in order to select out a particular value in the set will have to do some hard work to decode the bits in these features.

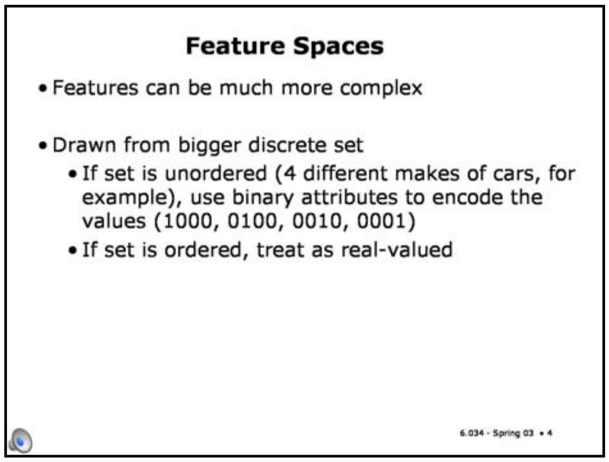
Instead, we typically make it easier on our algorithms by encoding such sets in unary, with one bit per element in the set. Then, for each value, we turn on one bit and set the rest to zero. So, we could encode a four-item set as 1000, 0100, 0010, 0001.



Feature Spaces

- Features can be much more complex
- Drawn from bigger discrete set
  - If set is unordered (4 different makes of cars, for example), use binary attributes to encode the values (1000, 0100, 0010, 0001)

6.034 - Spring 03 • 3



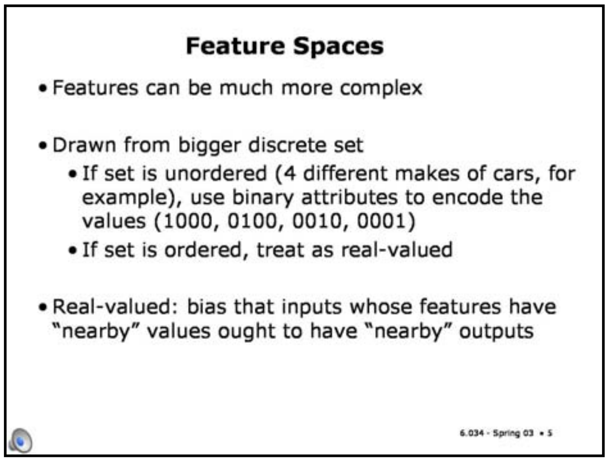
Feature Spaces

- Features can be much more complex
- Drawn from bigger discrete set
  - If set is unordered (4 different makes of cars, for example), use binary attributes to encode the values (1000, 0100, 0010, 0001)
  - If set is ordered, treat as real-valued

6.034 - Spring 03 • 4

Slide 10.1.4
On the other hand, when the set has a natural order, like someone's age, or the number of bedrooms in a house, it can usually be treated as if it were a real-valued attribute using methods we're about to explore.

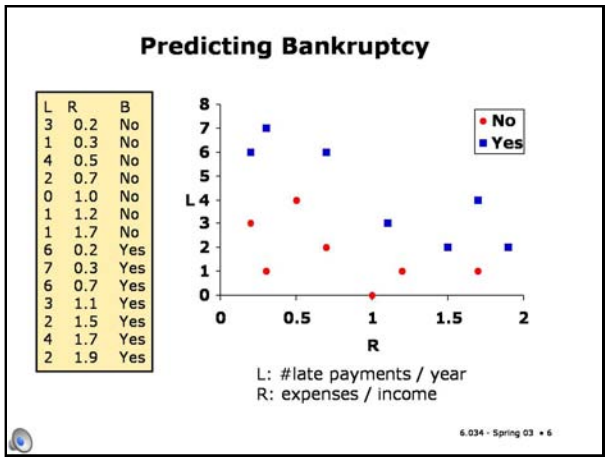
Slide 10.1.5
We'll spend this segment and the next looking at methods for dealing with real-valued attributes. The main goal will be to take advantage of the notion of distance between values that the reals affords us in order to build in a very deep bias that inputs whose features have "nearby" values ought, in general, to have "nearby" outputs.



Feature Spaces

- Features can be much more complex
- Drawn from bigger discrete set
  - If set is unordered (4 different makes of cars, for example), use binary attributes to encode the values (1000, 0100, 0010, 0001)
  - If set is ordered, treat as real-valued
- Real-valued: bias that inputs whose features have "nearby" values ought to have "nearby" outputs

6.034 - Spring 03 • 5



Predicting Bankruptcy

L	R	B
3	0.2	No
1	0.3	No
4	0.5	No
2	0.7	No
0	1.0	No
1	1.2	No
1	1.7	No
6	0.2	Yes
7	0.3	Yes
6	0.7	Yes
3	1.1	Yes
2	1.5	Yes
4	1.7	Yes
2	1.9	Yes

L: #late payments / year
R: expenses / income

6.034 - Spring 03 • 6

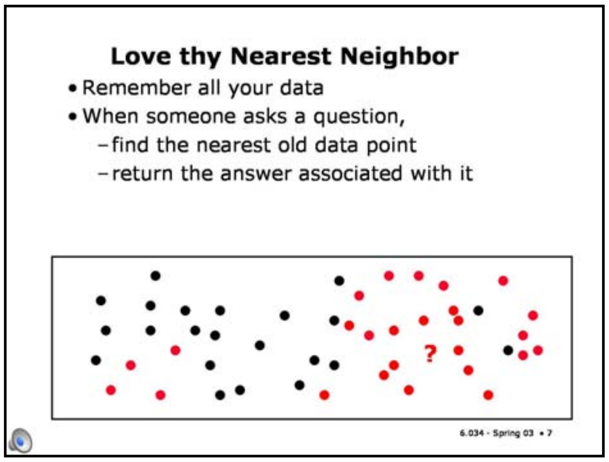
Slide 10.1.6
We'll use the example of predicting whether someone is going to go bankrupt. It only has two features, to make it easy to visualize.

One feature, L, is the number of late payments they have made on their credit card this year. This is a discrete value that we're treating as a real.

The other feature, R, is the ratio of their expenses to their income. The higher it is, the more likely you'd think the person would be to go bankrupt.

We have a set of examples of people who did, in fact go bankrupt, and a set who did not. We can plot the points in a two-dimensional space, with a dimension for each attribute. We've colored the "positive" (bankrupt) points blue and the negative points red.

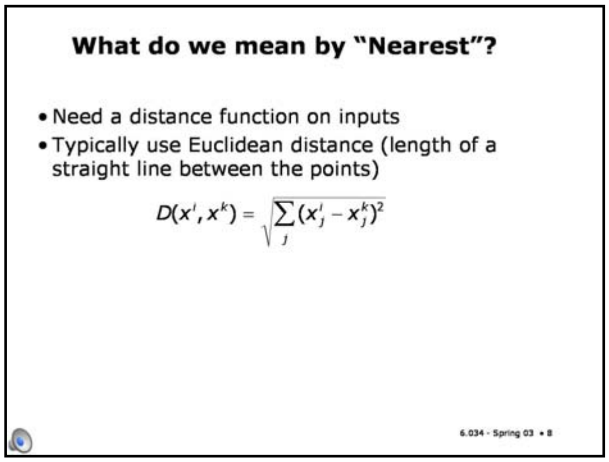
Slide 10.1.7
We took a brief look at the nearest neighbor algorithm in the first segment on learning. The idea is that you remember all the data points you've ever seen and, when you're given a query point, you find the old point that's nearest to the query point and predict its y value as your output.



Love thy Nearest Neighbor

- Remember all your data
- When someone asks a question,
  - find the nearest old data point
  - return the answer associated with it

6.034 - Spring 03 • 7



What do we mean by "Nearest"?

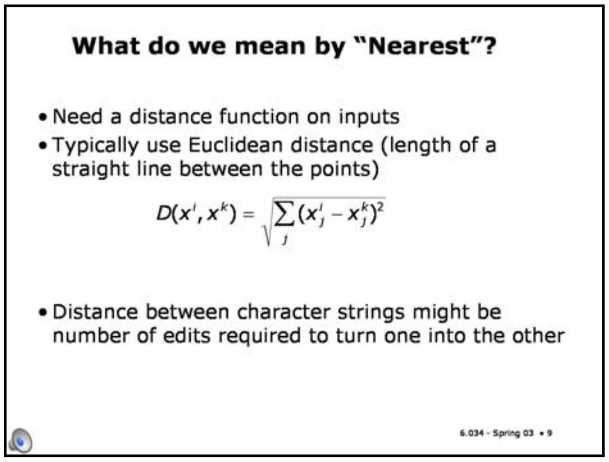
- Need a distance function on inputs
- Typically use Euclidean distance (length of a straight line between the points)

$$D(x^i, x^k) = \sqrt{\sum_j (x_j^i - x_j^k)^2}$$

6.034 - Spring 03 • 8

Slide 10.1.8
In order to say what point is nearest, we have to define what we mean by "near". Typically, we use Euclidean distance between two points, which is just the square root of the sum of the squared differences between corresponding feature values.

Slide 10.1.9
In other machine learning applications, the inputs can be something other than fixed-length vectors of numbers. We can often still use nearest neighbor, with creative use of distance metrics. The distance between two DNA strings, for example, might be the number of single-character edits required to turn one into the other.



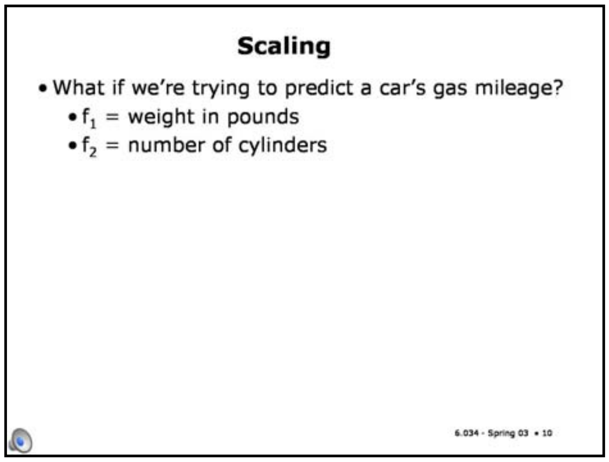
What do we mean by "Nearest"?

- Need a distance function on inputs
- Typically use Euclidean distance (length of a straight line between the points)

$$D(x^i, x^k) = \sqrt{\sum_j (x_j^i - x_j^k)^2}$$

- Distance between character strings might be number of edits required to turn one into the other

6.034 - Spring 03 • 9



Scaling

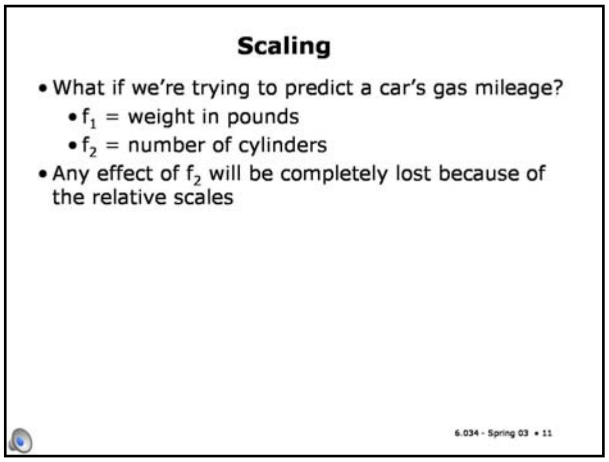
- What if we're trying to predict a car's gas mileage?
  - f_1 = weight in pounds
  - f_2 = number of cylinders

6.034 - Spring 03 • 10

Slide 10.1.10
The naive Euclidean distance isn't always appropriate, though.

Consider the case where we have two features describing a car. One is its weight in pounds and the other is the number of cylinders. The first will tend to have values in the thousands, whereas the second will have values between 4 and 8.

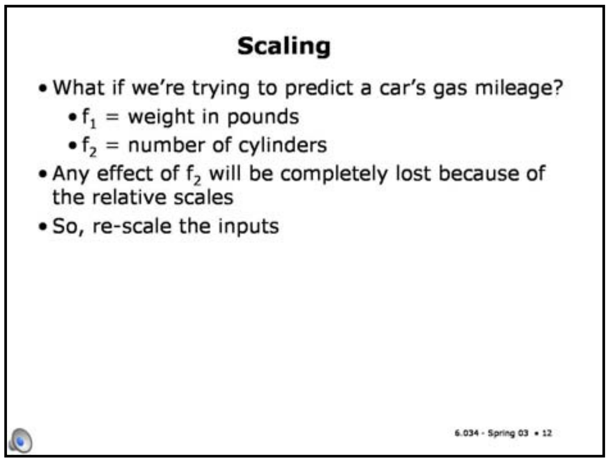
Slide 10.1.11
If we just use Euclidean distance in this space, the number of cylinders will have essentially no influence on nearness. A difference of 4 pounds in a car's weight will swamp a difference between 4 and 8 cylinders.



Scaling

- What if we're trying to predict a car's gas mileage?
  - f_1 = weight in pounds
  - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales

6.034 - Spring 03 • 11



Scaling

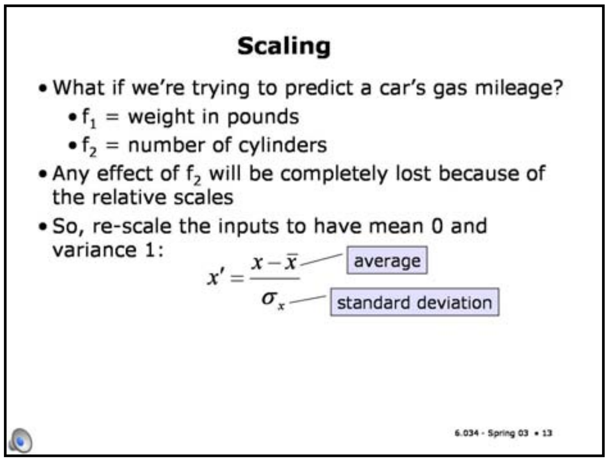
- What if we're trying to predict a car's gas mileage?
  - f_1 = weight in pounds
  - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales
- So, re-scale the inputs

6.034 - Spring 03 • 12

Slide 10.1.12
One standard method for addressing this problem is to re-scale the features.

In the simplest case, you might, for each feature, compute its range (the difference between its maximum and minimum values). Then scale the feature by subtracting the minimum value and dividing by the range. All features values would be between 0 and 1.

Slide 10.1.13
A somewhat more robust method (in case you have a crazy measurement, perhaps due to a noise in a sensor, that would make the range huge) is to scale the inputs to have 0 mean and standard deviation 1. If you haven't seen this before, it means to compute the average value of the feature, x-bar, and subtract it from each feature value, which will give you features all centered at 0. Then, to deal with the range, you compute the standard deviation (which is the square root of the variance, which we'll talk about in detail in the segment on regression) and divide each value by that. This transformation, called normalization, puts all of the features on about equal footing.



Scaling

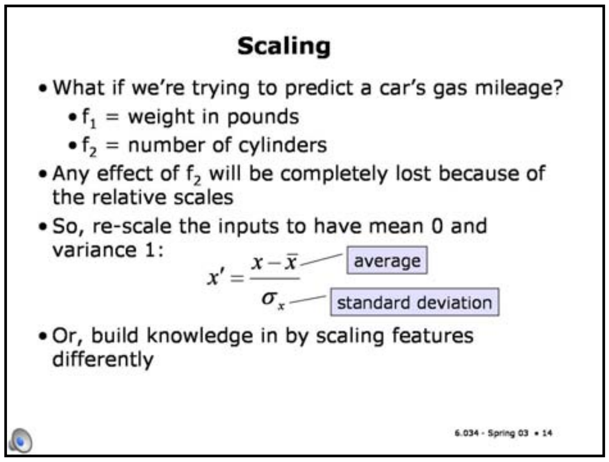
- What if we're trying to predict a car's gas mileage?
  - f_1 = weight in pounds
  - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales
- So, re-scale the inputs to have mean 0 and variance 1:

$$x' = \frac{x - \bar{x}}{\sigma_x}$$

average

standard deviation

6.034 - Spring 03 • 13



Scaling

- What if we're trying to predict a car's gas mileage?
  - f_1 = weight in pounds
  - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales
- So, re-scale the inputs to have mean 0 and variance 1:

$$x' = \frac{x - \bar{x}}{\sigma_x}$$

average

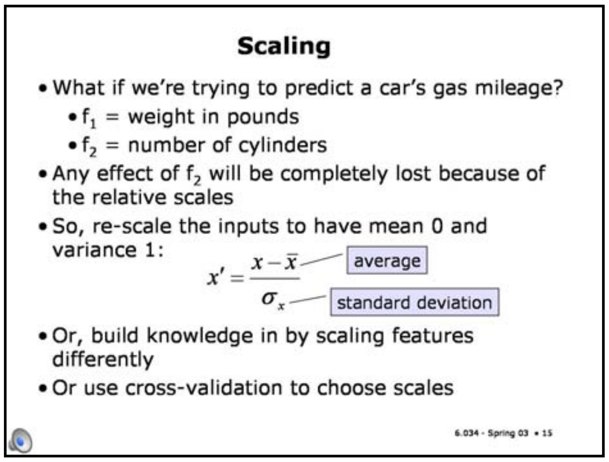
standard deviation

- Or, build knowledge in by scaling features differently

6.034 - Spring 03 • 14

Slide 10.1.14
Of course, you may not want to have all your features on equal footing. It may be that you happen to know, based on the nature of the domain, that some features are more important than others. In such cases, you might want to multiply them by a weight that will increase their influence in the distance calculation.

Slide 10.1.15
Another popular, but somewhat advanced, technique is to use cross validation and gradient descent to choose weightings of the features that generate the best performance on the particular data set.



Scaling

- What if we're trying to predict a car's gas mileage?
  - f_1 = weight in pounds
  - f_2 = number of cylinders
- Any effect of f_2 will be completely lost because of the relative scales
- So, re-scale the inputs to have mean 0 and variance 1:

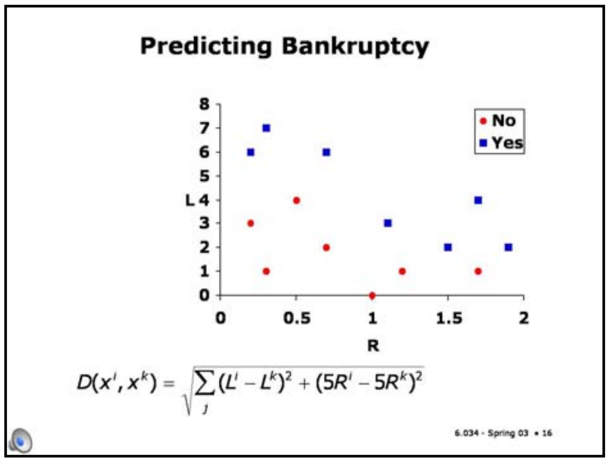
$$x' = \frac{x - \bar{x}}{\sigma_x}$$

average

standard deviation

- Or, build knowledge in by scaling features differently
- Or use cross-validation to choose scales

6.034 - Spring 03 • 15



Predicting Bankruptcy

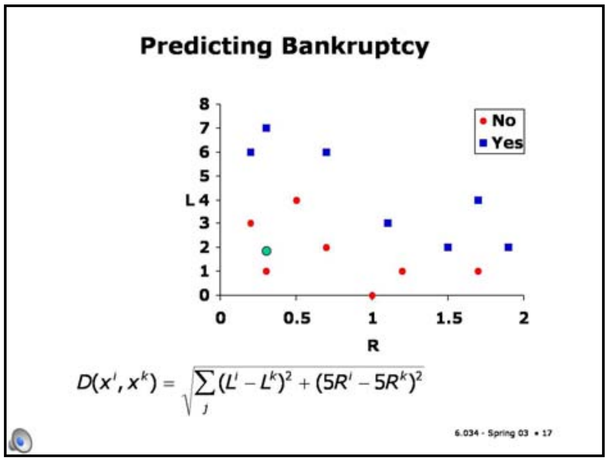
$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

6.034 - Spring 03 • 16

Slide 10.1.16
Okay. Let's see how nearest neighbor works on our bankruptcy example. Let's say we've thought about the domain and decided that the R feature (ratio between expenses and income) needs to be scaled up by 5 in order to be appropriately balanced against the L feature (number of late payments).

So we'll use Euclidian distance, but with the R values multiplied by 5 first. We've scaled the axes on the slide so that the two dimensions are graphically equal. This means that locus of points at a particular distance d from a point on our graph will appear as a circle.

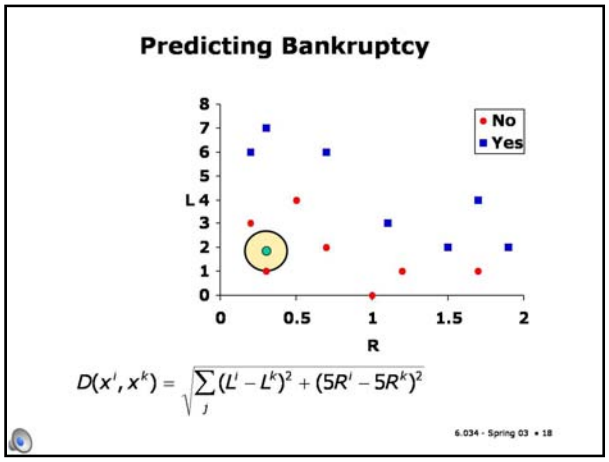
Slide 10.1.17
Now, let's say we have a new person with R equal 0.3 and L equal to 2. What y value should we predict?



Predicting Bankruptcy

$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

6.034 - Spring 03 • 17



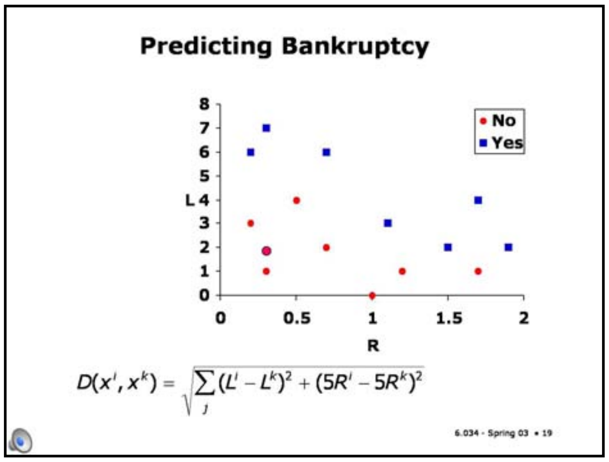
Predicting Bankruptcy

$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

6.034 - Spring 03 • 18

Slide 10.1.18
We look for the nearest point, which is the red point at the edge of the yellow circle. The fact that there are no old points in the circle means that this red point is indeed the nearest neighbor of our query point.

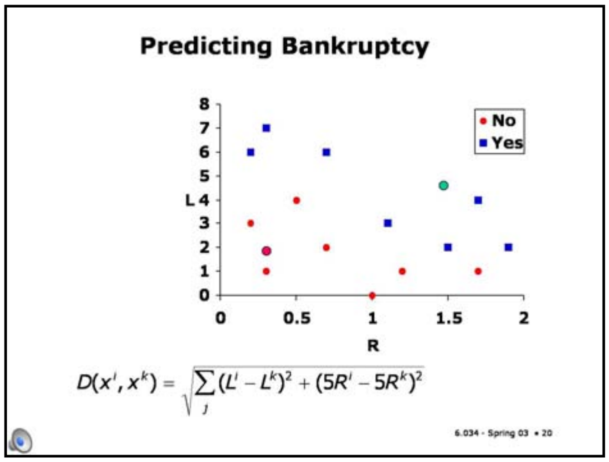
Slide 10.1.19
And so our answer would be "no".



Predicting Bankruptcy

$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

6.034 - Spring 03 • 19



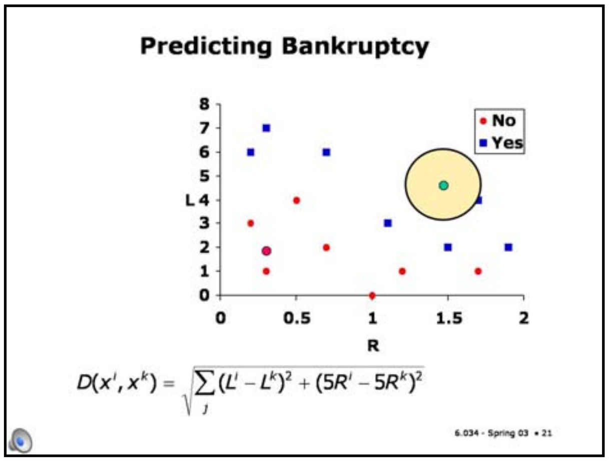
Predicting Bankruptcy

$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

6.034 - Spring 03 • 20

Slide 10.1.20
Similarly, for another query point,

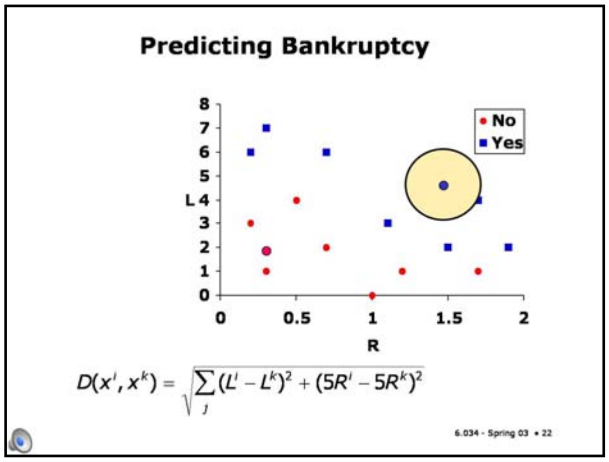
Slide 10.1.21
we find the nearest neighbor, which has output "yes"



Predicting Bankruptcy

$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

6.034 - Spring 03 • 21



Predicting Bankruptcy

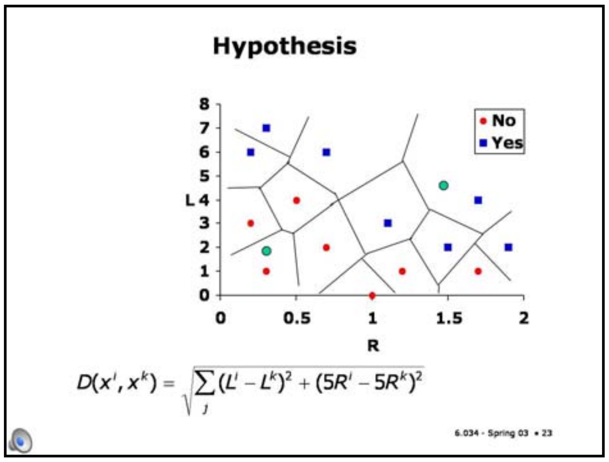
$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

6.034 - Spring 03 • 22

Slide 10.1.22
and generate "yes" as our prediction.

Slide 10.1.23
So, what is the hypothesis of the nearest neighbor algorithm? It's sort of different from our other algorithms, in that it isn't explicitly constructing a description of a hypothesis based on the data it sees.

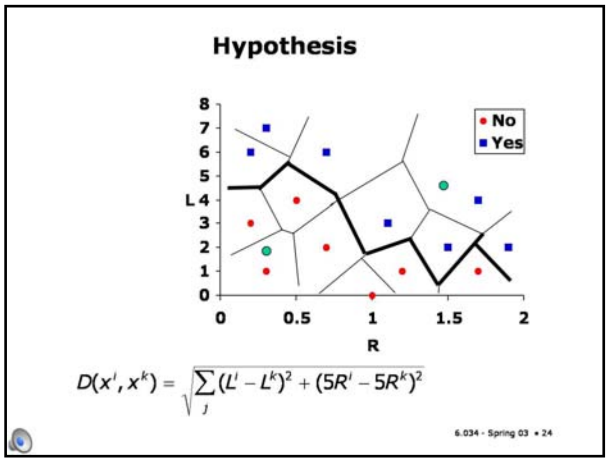
Given a set of points and a distance metric, you can divide the space up into regions, one for each point, which represent the set of points in space that are nearer to this designated point than to any of the others. In this figure, I've drawn a (somewhat inaccurate) picture of the decomposition of the space into such regions. It's called a "Voronoi partition" of the space.



Hypothesis

$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

6.034 - Spring 03 • 23



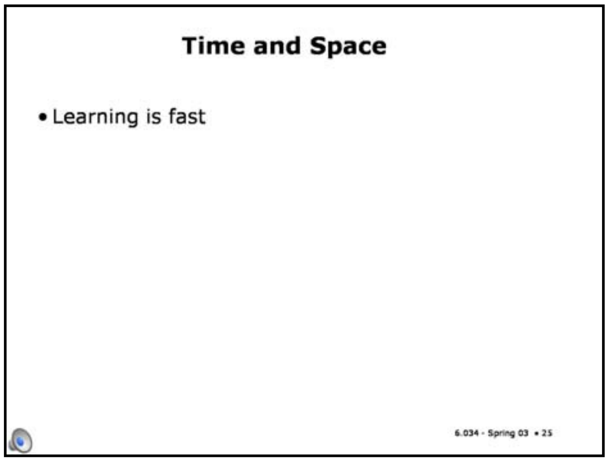
Hypothesis

$$D(x^i, x^k) = \sqrt{\sum_j (L^i - L^k)^2 + (5R^i - 5R^k)^2}$$

Slide 10.1.24
Now, we can think of our hypothesis as being represented by the edges in the Voronoi partition that separate a region associated with a positive point from a region associated with a negative one. In our example, that generates this bold boundary.

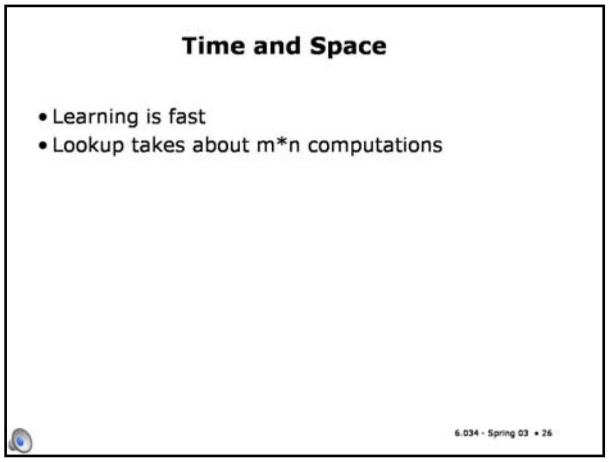
It's important to note that we never explicitly compute this boundary; it just arises out of the "nearest neighbor" query process.

Slide 10.1.25
It's useful to spend a little bit of time thinking about how complex this algorithm is. Learning is very fast. All you have to do is remember all the data you've seen!



Time and Space

- Learning is fast

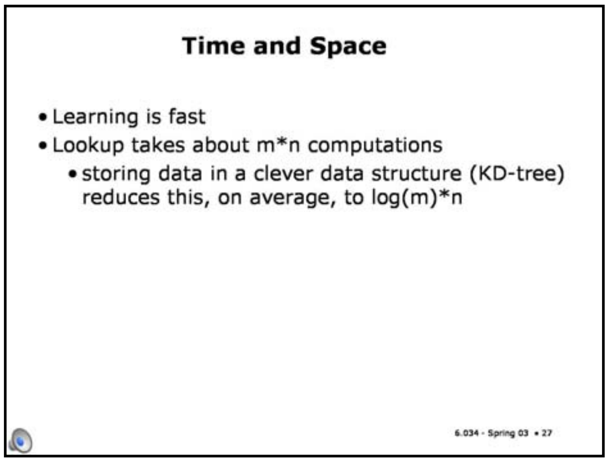


Time and Space

- Learning is fast
- Lookup takes about m*n computations

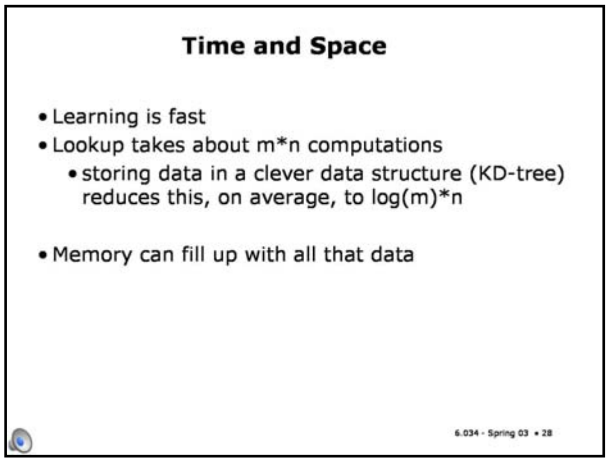
Slide 10.1.26
What takes longer is answering a query. Naively, you have to, for each point in your training set (and there are m of them) compute the distance to the query point (which takes about n computations, since there are n features to compare). So, overall, this takes about m * n time.

Slide 10.1.27
It's possible to organize your data into a clever data structure (one such structure is called a K-D tree). It will allow you to find the nearest neighbor to a query point in time that's, on average, proportional to the log of m, which is a huge savings.



Time and Space

- Learning is fast
- Lookup takes about m*n computations
  - storing data in a clever data structure (KD-tree) reduces this, on average, to log(m)*n



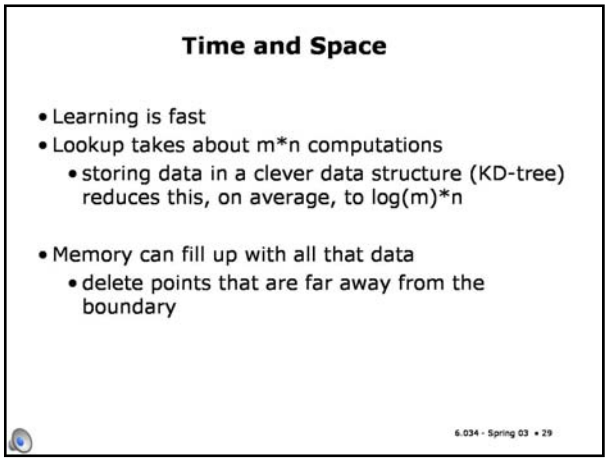
Time and Space

- Learning is fast
- Lookup takes about m*n computations
  - storing data in a clever data structure (KD-tree) reduces this, on average, to log(m)*n
- Memory can fill up with all that data

6.034 - Spring 03 • 28

Slide 10.1.28
Another issue is memory. If you gather data over time, you might worry about your memory filling up, since you have to remember it all.

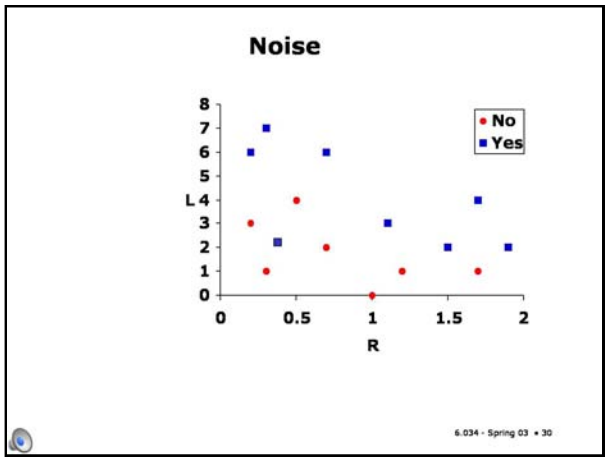
Slide 10.1.29
There are a number of variations on nearest neighbor that allow you to forget some of the data points; typically the ones that are most forgettable are those that are far from the current boundary between positive and negative.



Time and Space

- Learning is fast
- Lookup takes about m*n computations
  - storing data in a clever data structure (KD-tree) reduces this, on average, to log(m)*n
- Memory can fill up with all that data
  - delete points that are far away from the boundary

6.034 - Spring 03 • 29



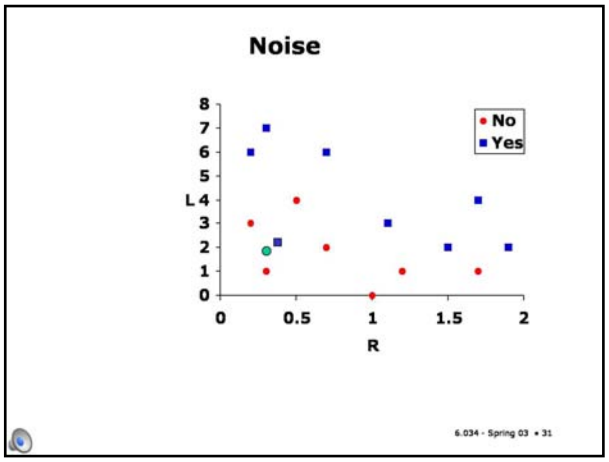
Noise

6.034 - Spring 03 • 30

Slide 10.1.30
In our example so far, there has not been much (apparent) noise; the boundary between positives and negatives is clean and simple. Let's now consider the case where there's a blue point down among the reds. Someone with an apparently healthy financial record goes bankrupt.

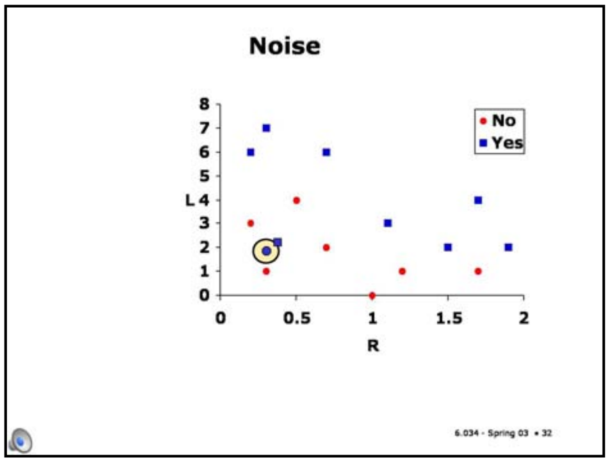
There are, of course, two ways to deal with this data point. One is to assume that it is not noise; that is, that there is some regularity that makes people like this one go bankrupt in general. The other is to say that this example is an "outlier". It represents an unusual case that we would prefer largely to ignore, and not to incorporate it into our hypothesis.

Slide 10.1.31
So, what happens in nearest neighbor if we get a query point next to this point?



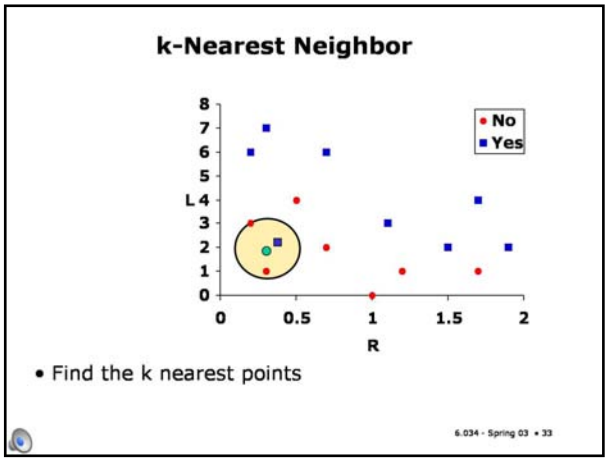
Noise

6.034 - Spring 03 • 31

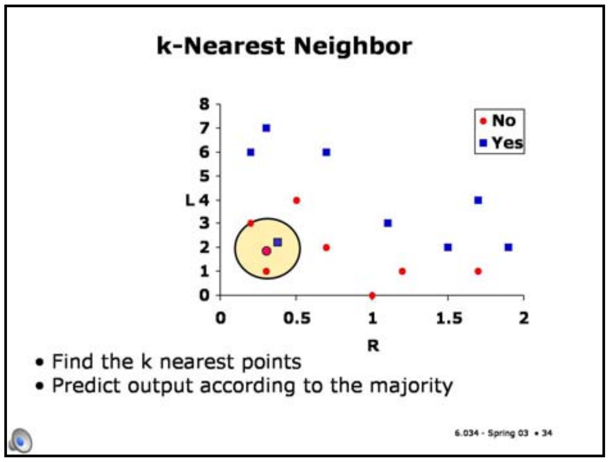


Slide 10.1.32
We find the nearest neighbor, which is a "yes" point, and predict the answer "yes". This outcome is consistent with the first view; that is, that this blue point represents some important property of the problem.

Slide 10.1.33
But if we think there might be noise in the data, we can change the algorithm a bit to try to ignore it. We'll move to the k-nearest neighbor algorithm. It's just like the old algorithm, except that when we get a query, we'll search for the k closest points to the query points. And we'll generate, as output, the output associated with the majority of the k closest elements.


k-Nearest Neighbor

- Find the k nearest points

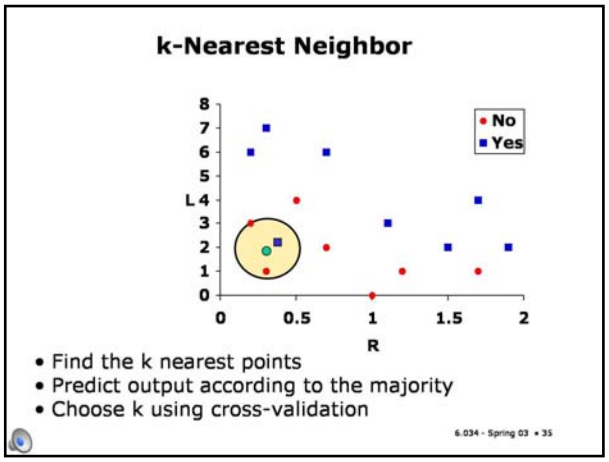

k-Nearest Neighbor

- Find the k nearest points
- Predict output according to the majority

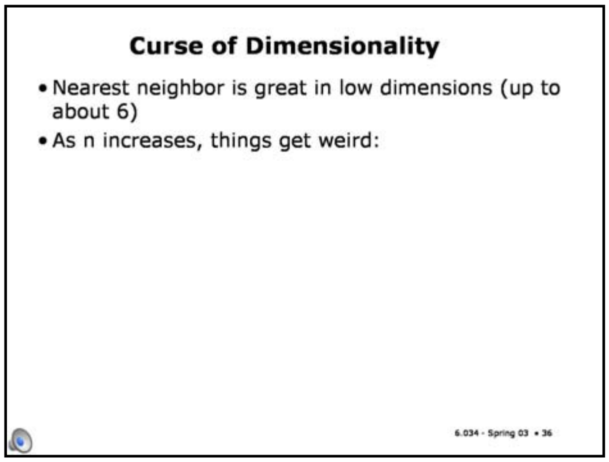
Slide 10.1.34
In this case, we've chosen k to be 3. The three closest points consist of two "no"s and a "yes", so our answer would be "no".

Slide 10.1.35
It's not entirely obvious how to choose k. The smaller the k, the more noise-sensitive your hypothesis is. The larger the k, the more "smeared out" it is. In the limit of large k, you would always just predict the output value that's associated with the majority of your training points. So, k functions kind of like a complexity-control parameter, exactly analogous to epsilon in DNF and min-leaf-size in decision trees. With smaller k, you have high variance and risk overfitting; with large k, you have high bias and risk not being able to express the hypotheses you need.

It's common to choose k using cross-validation.


k-Nearest Neighbor

- Find the k nearest points
- Predict output according to the majority
- Choose k using cross-validation



Curse of Dimensionality

- Nearest neighbor is great in low dimensions (up to about 6)
- As n increases, things get weird:

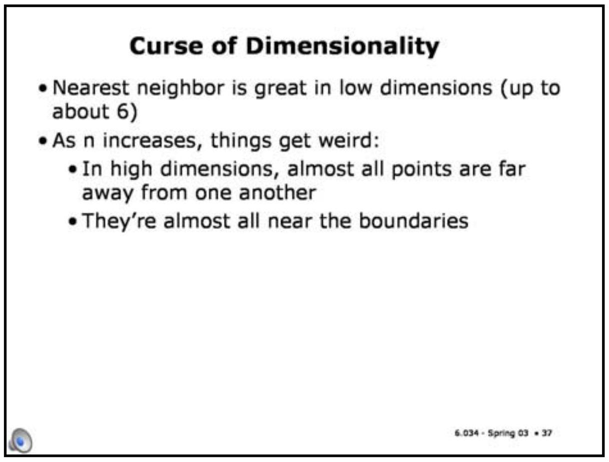
6.034 - Spring 03 • 36

Slide 10.1.36
Nearest neighbor works very well (and is often the method of choice) for problems in relatively low-dimensional real-valued spaces.

But as the dimensionality of a space increases, its geometry gets weird. Here are some suprising (to me, at least) facts about high-dimensional spaces.

Slide 10.1.37
In high dimensions, almost all points are far away from one another.

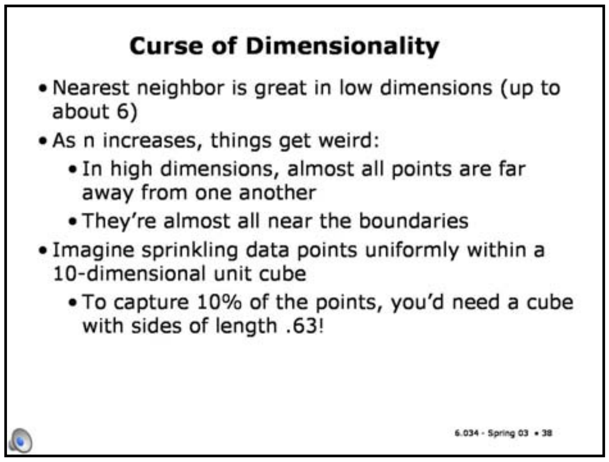
If you make a cube or sphere in high dimensions, then almost all the points within that cube or sphere are near the boundaries.



Curse of Dimensionality

- Nearest neighbor is great in low dimensions (up to about 6)
- As n increases, things get weird:
  - In high dimensions, almost all points are far away from one another
  - They're almost all near the boundaries

6.034 - Spring 03 • 37



Curse of Dimensionality

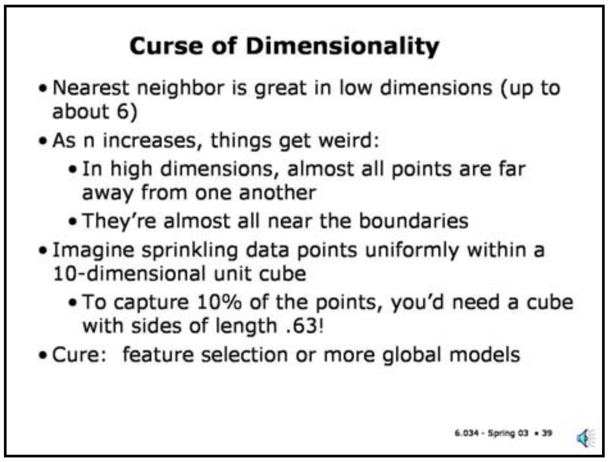
- Nearest neighbor is great in low dimensions (up to about 6)
- As n increases, things get weird:
  - In high dimensions, almost all points are far away from one another
  - They're almost all near the boundaries
- Imagine sprinkling data points uniformly within a 10-dimensional unit cube
  - To capture 10% of the points, you'd need a cube with sides of length .63!

6.034 - Spring 03 • 38

Slide 10.1.38
Imagine sprinkling data points uniformly within a 10-dimensional unit cube (cube whose sides are of length 1).

To capture 10% of the points, you'd need a cube with sides of length .63!

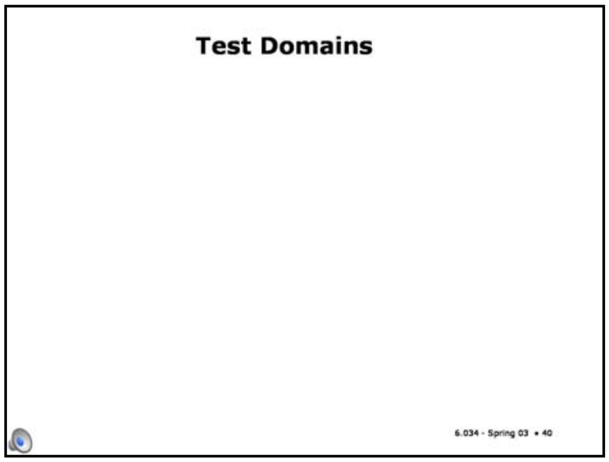
Slide 10.1.39
All this means that the notions of nearness providing a good generalization principle, which are very effective in low-dimensional spaces, become fairly ineffective in high-dimensional spaces. There are two ways to handle this problem. One is to do "feature selection", and try to reduce the problem back down to a lower-dimensional one. The other is to fit hypotheses from a much smaller hypothesis class, such as linear separators, which we will see in the next chapter.



Curse of Dimensionality

- Nearest neighbor is great in low dimensions (up to about 6)
- As n increases, things get weird:
  - In high dimensions, almost all points are far away from one another
  - They're almost all near the boundaries
- Imagine sprinkling data points uniformly within a 10-dimensional unit cube
  - To capture 10% of the points, you'd need a cube with sides of length .63!
- Cure: feature selection or more global models

6.034 - Spring 03 • 39

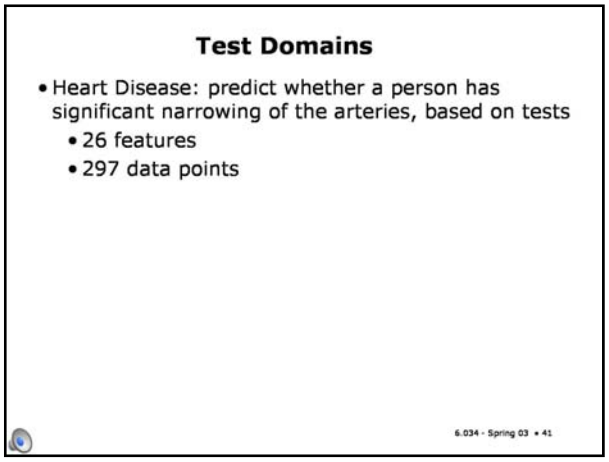


Test Domains

6.034 - Spring 03 • 40

Slide 10.1.40
We'll look at how nearest neighbor performs on two different test domains.

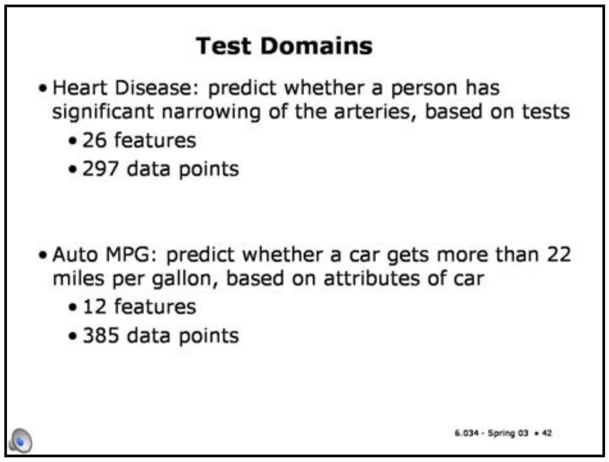
Slide 10.1.41
The first domain is predicting whether a person has heart disease, represented by a significant narrowing of the arteries, based on the results of a variety of tests. This domains has 297 different data points, each of which is characterized by 26 features. A lot of these features are actually boolean, which means that although the dimensionality is high, the curse of dimensionality, which really only bites us badly in the case of real-valued features, doesn't cause too much problem.



Test Domains

- Heart Disease: predict whether a person has significant narrowing of the arteries, based on tests
  - 26 features
  - 297 data points

6.034 - Spring 03 • 41



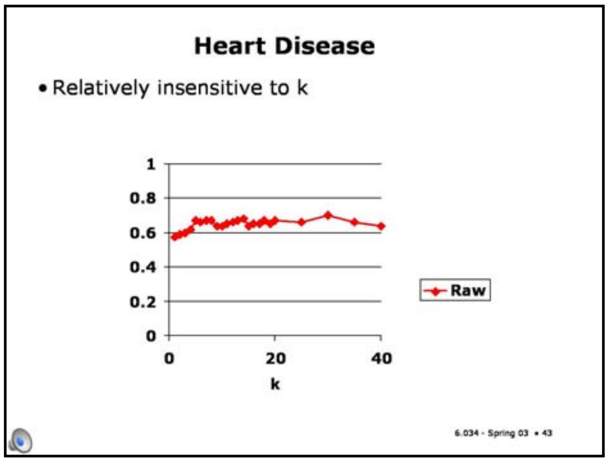
Test Domains

- Heart Disease: predict whether a person has significant narrowing of the arteries, based on tests
  - 26 features
  - 297 data points
- Auto MPG: predict whether a car gets more than 22 miles per gallon, based on attributes of car
  - 12 features
  - 385 data points

6.034 - Spring 03 • 42

Slide 10.1.42
In the second domain, we're trying to predict whether a car gets more than 22 miles-per-gallon fuel efficiency. We have 385 data points, characterized by 12 features. Again, a number of the features are binary.

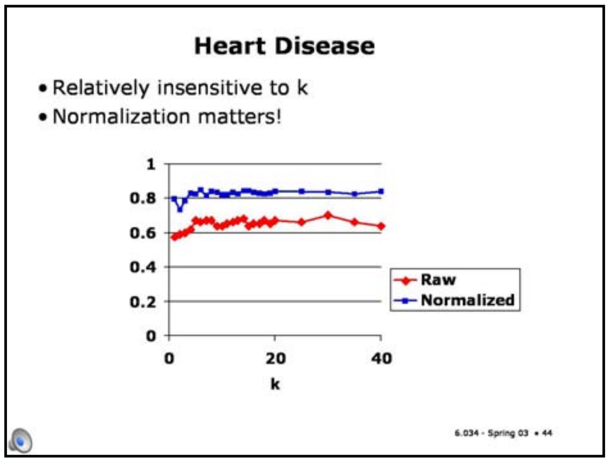
Slide 10.1.43
Here's a graph of the cross-validation accuracy of nearest neighbor on the heart disease data, shown as a function of k. Looking at the data, we can see that the performance is relatively insensitive to the choice of k, though it seems like maybe it's useful to have k be greater than about 5.



Heart Disease

- Relatively insensitive to k

6.034 - Spring 03 • 43

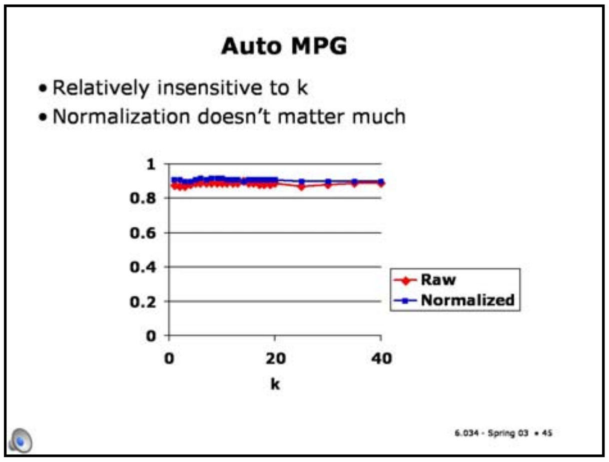


Heart Disease

- Relatively insensitive to k
- Normalization matters!

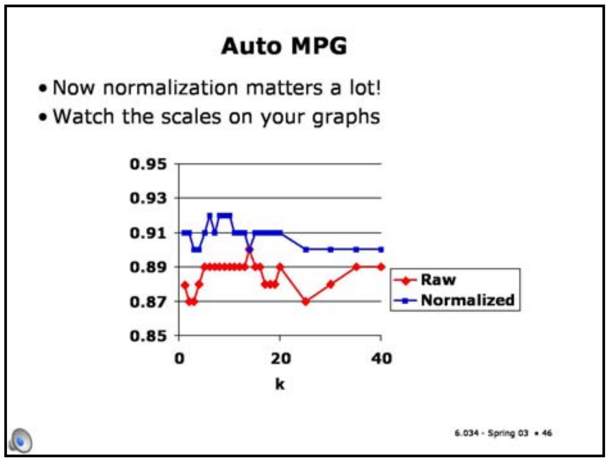
Slide 10.1.44
The red curve is the performance of nearest neighbor using the features directly as they are measured, without any scaling. We then normalized all of the features to have mean 0 and standard deviation 1, and re-ran the algorithm. You can see here that it makes a noticable increase in performance.

Slide 10.1.45
We ran nearest neighbor with both normalized and un-normalized inputs on the auto-MPG data. It seems to perform pretty well in all cases. It is still relatively insensitive to k, and normalization only seems to help a tiny amount.



Auto MPG

- Relatively insensitive to k
- Normalization doesn't matter much



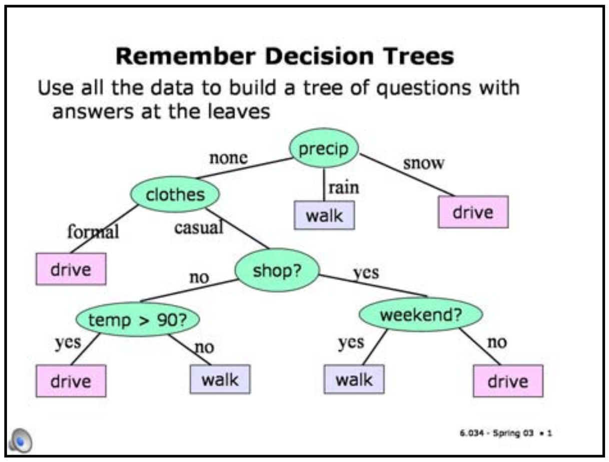
Auto MPG

- Now normalization matters a lot!
- Watch the scales on your graphs

Slide 10.1.46
Watch out for tricky graphing! It's always possible to make your algorithm look much better than the other leading brand (as long as it's a little bit better), by changing the scale on your graphs. The previous graph had a scale of 0 to 1. This graph has a scale of 0.85 to 0.95. Now the normalized version looks much better! Be careful of such tactics when you read other peoples' papers; and certainly don't practice them in yours.

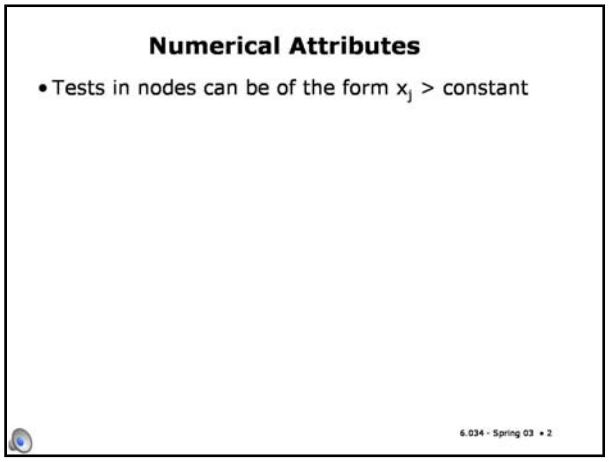
6.034 Notes: Section 10.2

Slide 10.2.1
Now, let's go back to decision trees, and see if we can apply them to problems where the inputs are numeric.



Remember Decision Trees

Use all the data to build a tree of questions with answers at the leaves



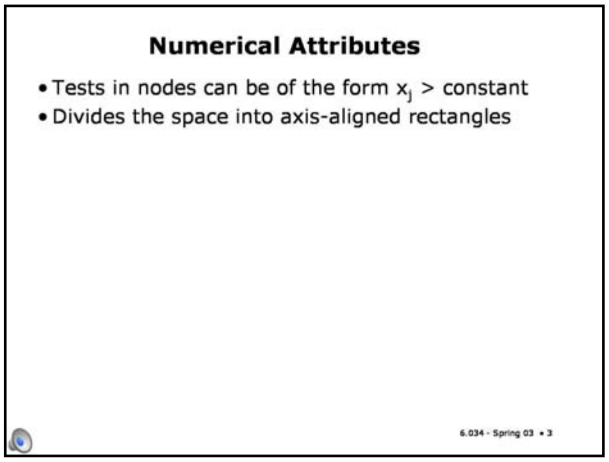
Numerical Attributes

- Tests in nodes can be of the form x_j > constant

6.034 - Spring 03 • 2

Slide 10.2.2
When we have features with numeric values, we have to expand our hypothesis space to include different tests on the leaves. We will allow tests on the leaves of a decision tree to be comparisons of the form $x_j > c$, where c is a constant.

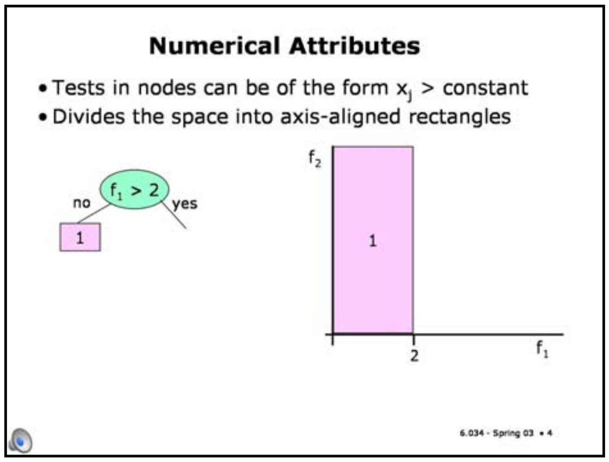
Slide 10.2.3
This class of splits allows us to divide our feature-space into a set of exhaustive and mutually exclusive hyper-rectangles (that is, rectangles of potentially high dimension), with one rectangle for each leaf of the tree. So, each rectangle will have an output value (1 or 0) associated with it. The set of rectangles and their output values constitutes our hypothesis.



Numerical Attributes

- Tests in nodes can be of the form x_j > constant
- Divides the space into axis-aligned rectangles

6.034 - Spring 03 • 3

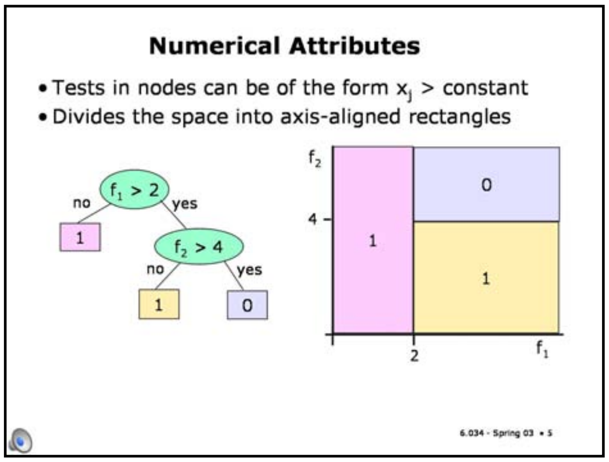


Numerical Attributes

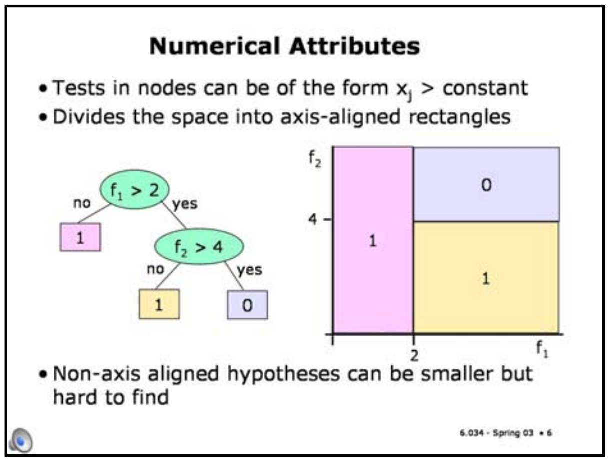
- Tests in nodes can be of the form x_j > constant
- Divides the space into axis-aligned rectangles

Slide 10.2.4
So, in this example, at the top level, we split the space into two parts, according to whether feature 1 has a value greater than 2. If not, then the output is 1.

Slide 10.2.5
If f1 is greater than 2, then we have another split, this time on whether f2 is greater than 4. If it is, the answer is 0, otherwise, it is 1. You can see the corresponding rectangles in the two-dimensional feature space.


Numerical Attributes

- Tests in nodes can be of the form x_j > constant
- Divides the space into axis-aligned rectangles

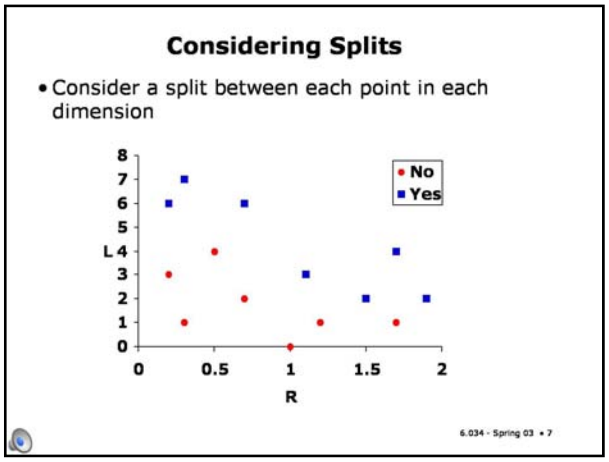

Numerical Attributes

- Tests in nodes can be of the form x_j > constant
- Divides the space into axis-aligned rectangles
- Non-axis aligned hypotheses can be smaller but hard to find

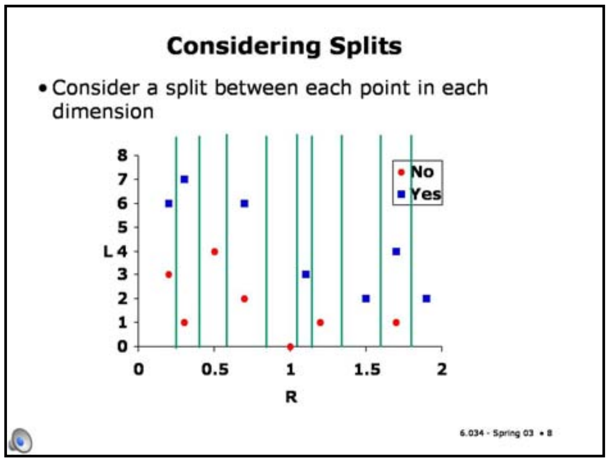
Slide 10.2.6
This class of hypotheses is fairly rich, but it can be hard to express some concepts.

There are fancier versions of numeric decision trees that allow splits to be arbitrary hyperplanes (allowing us, for example, to make a split along a diagonal line in the 2D case), but we won't pursue them in this class.

Slide 10.2.7
The only thing we really need to do differently in our algorithm is to consider splitting between each data point in each dimension.


Considering Splits

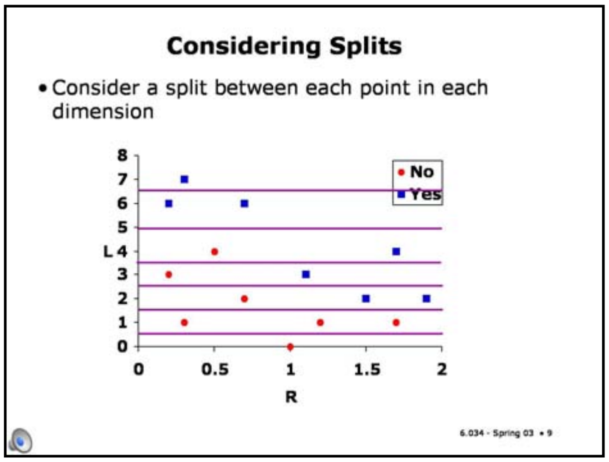
- Consider a split between each point in each dimension


Considering Splits

- Consider a split between each point in each dimension

Slide 10.2.8
So, in our bankruptcy domain, we'd consider 9 different splits in the R dimension (in general, you'd expect to consider m - 1 splits, if you have m data points; but in our data set we have some examples with equal R values).

Slide 10.2.9
And there are another 6 possible splits in the L dimension (because L is an integer, really, there are lots of duplicate L values).

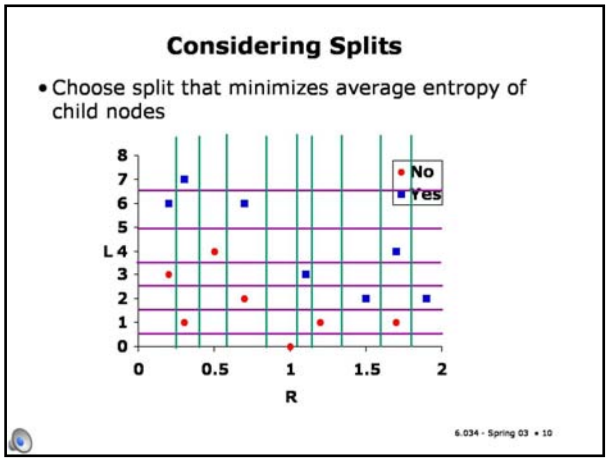


Considering Splits

- Consider a split between each point in each dimension

6.034 - Spring 03 • 9

Slide 10.2.10
All together, this is a lot of possible splits! As before, when building a tree, we'll choose the split that minimizes the average entropy of the resulting child nodes.



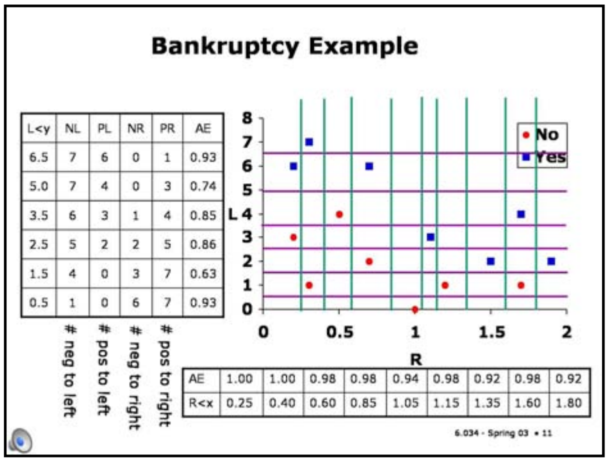
Considering Splits

- Choose split that minimizes average entropy of child nodes

6.034 - Spring 03 • 10

Slide 10.2.11
Let's see what actually happens with this algorithm in our bankruptcy domain.

We consider all the possible splits in each dimension, and compute their average entropies.



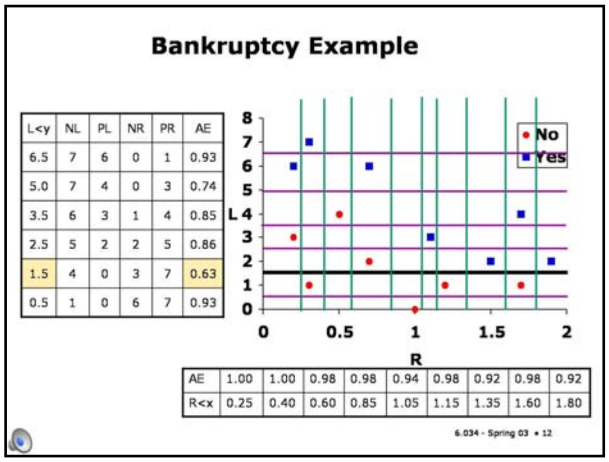
Bankruptcy Example

L<y	NL	PL	NR	PR	AE
6.5	7	6	0	1	0.93
5.0	7	4	0	3	0.74
3.5	6	3	1	4	0.85
2.5	5	2	2	5	0.86
1.5	4	0	3	7	0.63
0.5	1	0	6	7	0.93

neg to left
pos to left
neg to right
pos to right

AE	1.00	1.00	0.98	0.98	0.94	0.98	0.92	0.98	0.92
R<x	0.25	0.40	0.60	0.85	1.05	1.15	1.35	1.60	1.80

6.034 - Spring 03 • 11



Bankruptcy Example

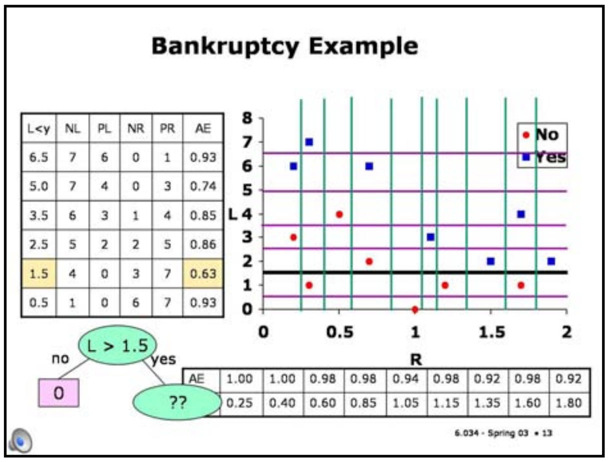
L<y	NL	PL	NR	PR	AE
6.5	7	6	0	1	0.93
5.0	7	4	0	3	0.74
3.5	6	3	1	4	0.85
2.5	5	2	2	5	0.86
1.5	4	0	3	7	0.63
0.5	1	0	6	7	0.93

AE	1.00	1.00	0.98	0.98	0.94	0.98	0.92	0.98	0.92
R<x	0.25	0.40	0.60	0.85	1.05	1.15	1.35	1.60	1.80

6.034 - Spring 03 • 12

Slide 10.2.12
Splitting in the L dimension at 1.5 will do the best job of reducing entropy, so we pick that split.

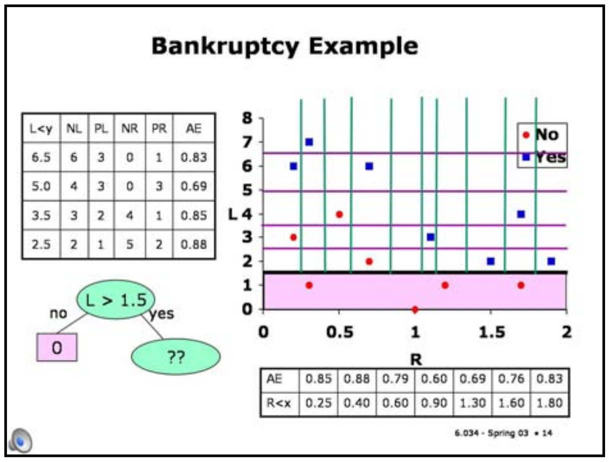
Slide 10.2.13
And we see that, conveniently, all the points with L not greater than 1.5 are of class 0, so we can make a leaf there.



Bankruptcy Example

L<y	NL	PL	NR	PR	AE
6.5	7	6	0	1	0.93
5.0	7	4	0	3	0.74
3.5	6	3	1	4	0.85
2.5	5	2	2	5	0.86
1.5	4	0	3	7	0.63
0.5	1	0	6	7	0.93

AE	1.00	1.00	0.98	0.98	0.94	0.98	0.92	0.98	0.92
	0.25	0.40	0.60	0.85	1.05	1.15	1.35	1.60	1.80



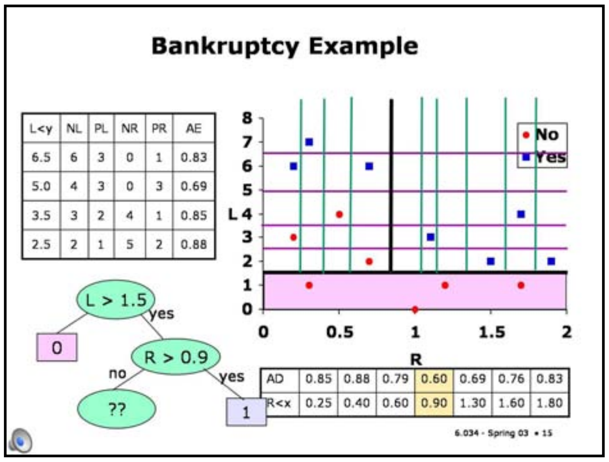
Bankruptcy Example

L<y	NL	PL	NR	PR	AE
6.5	6	3	0	1	0.83
5.0	4	3	0	3	0.69
3.5	3	2	4	1	0.85
2.5	2	1	5	2	0.88

AE	0.85	0.88	0.79	0.60	0.69	0.76	0.83
R<x	0.25	0.40	0.60	0.90	1.30	1.60	1.80

Slide 10.2.14
Now, we consider all the splits of the remaining part of the space. Note that we have to recalculate all the average entropies again, because the points that fall into the leaf node are taken out of consideration.

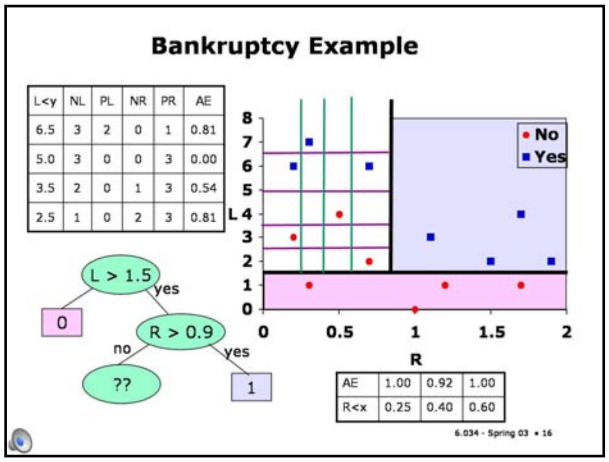
Slide 10.2.15
Now the best split is at R > 0.9. And we see that all the points for which that's true are positive, so we can make another leaf.



Bankruptcy Example

L<y	NL	PL	NR	PR	AE
6.5	6	3	0	1	0.83
5.0	4	3	0	3	0.69
3.5	3	2	4	1	0.85
2.5	2	1	5	2	0.88

AD	0.85	0.88	0.79	0.60	0.69	0.76	0.83
R<x	0.25	0.40	0.60	0.90	1.30	1.60	1.80



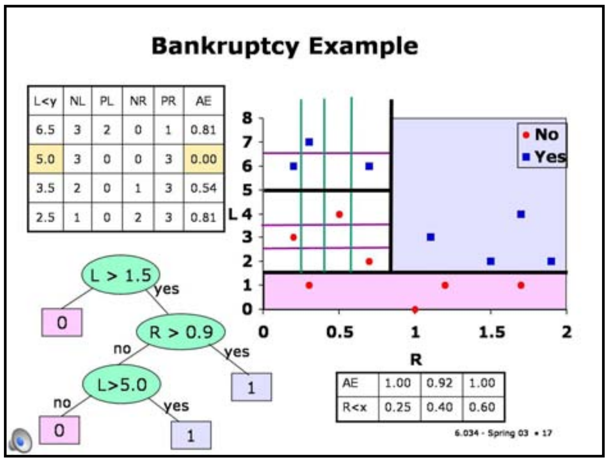
Bankruptcy Example

L<y	NL	PL	NR	PR	AE
6.5	3	2	0	1	0.81
5.0	3	0	0	3	0.00
3.5	2	0	1	3	0.54
2.5	1	0	2	3	0.81

AE	1.00	0.92	1.00
R<x	0.25	0.40	0.60

Slide 10.2.16
Again we consider all possible splits of the points that fall down the other branch of the tree.

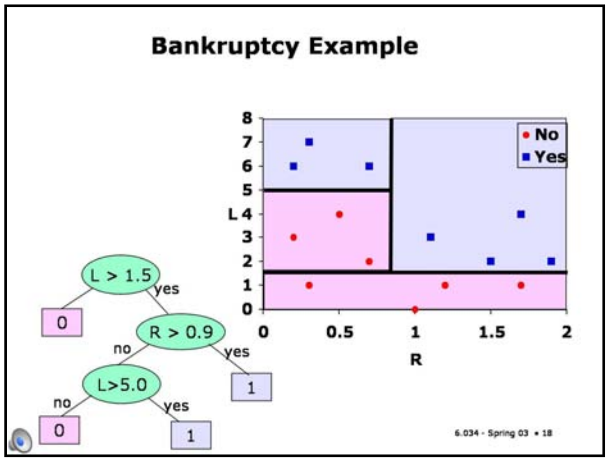
Slide 10.2.17
And we find that splitting on L > 5.0 gives us two homogenous leaves.



Bankruptcy Example

L<y	NL	PL	NR	PR	AE
6.5	3	2	0	1	0.81
5.0	3	0	0	3	0.00
3.5	2	0	1	3	0.54
2.5	1	0	2	3	0.81

AE	1.00	0.92	1.00
R<x	0.25	0.40	0.60



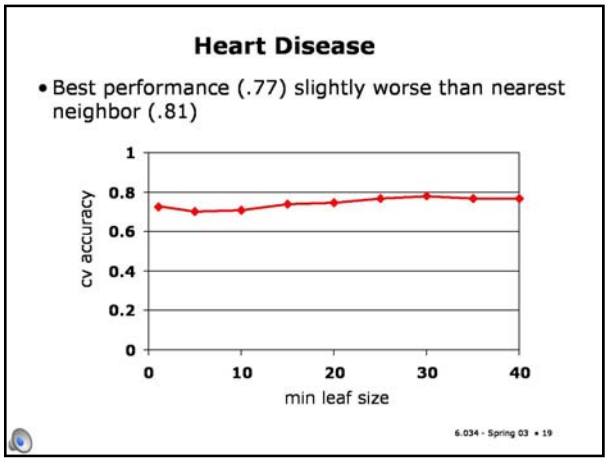
Bankruptcy Example

Slide 10.2.18
So, we finish with this tree, which happens to have zero error on our data set.

Of course, all of the issues that we talked about before with boolean attributes apply here: in general, you'll want to stop growing the tree (or post-prune it) in order to avoid overfitting.

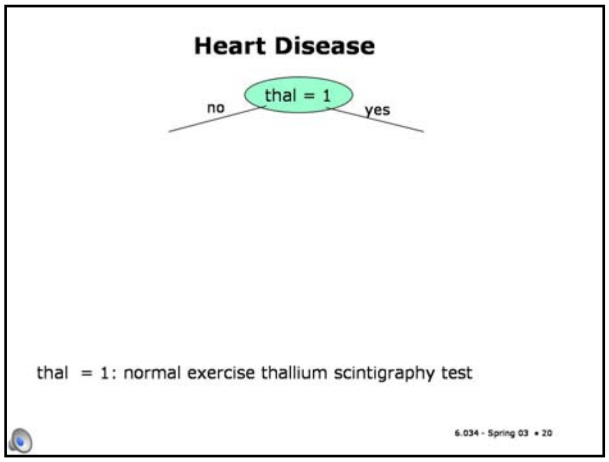
Slide 10.2.19
We ran this decision-tree algorithm on the heart-disease data set. This graph shows the cross-validation accuracy of the hypotheses generated by the decision-tree algorithm as a function of the min-leaf-size parameter, which stops splitting when the number of examples in a leaf gets below the specified size.

The best performance of this algorithm is about .77, which is slightly worse than the performance of nearest neighbor.



Heart Disease

- Best performance (.77) slightly worse than nearest neighbor (.81)



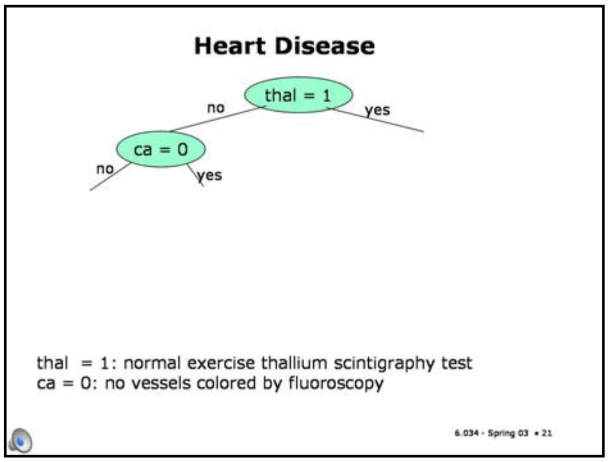
Heart Disease

thal = 1: normal exercise thallium scintigraphy test

Slide 10.2.20
But performance isn't everything. One of the nice things about the decision tree algorithm is that we can interpret the hypothesis we get out. Here is an example decision tree resulting from the learning algorithm.

I'm not a doctor (and I don't even play one on TV), but the tree at least kind of makes sense. The top-level split is on whether a certain kind of stress test, called "thal" comes out normal.

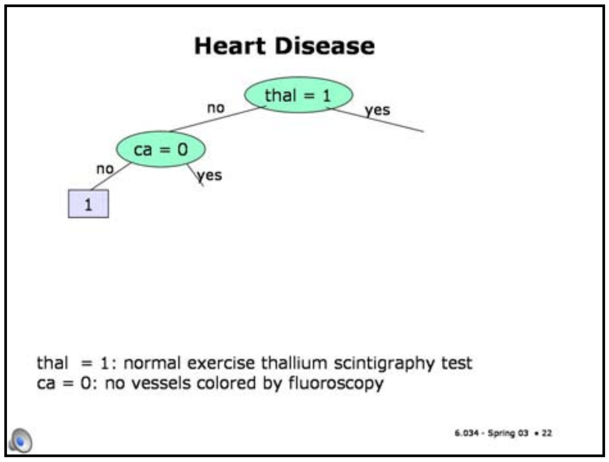
Slide 10.2.21
If thal is not normal, then we look at the results of the "ca" test. This test has as results numbers 0 through 3, indicating how many blood vessels were shown to be blocked in a different test. We chose to code this feature with 4 binary attributes.



Heart Disease

thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy

6.034 - Spring 03 • 21



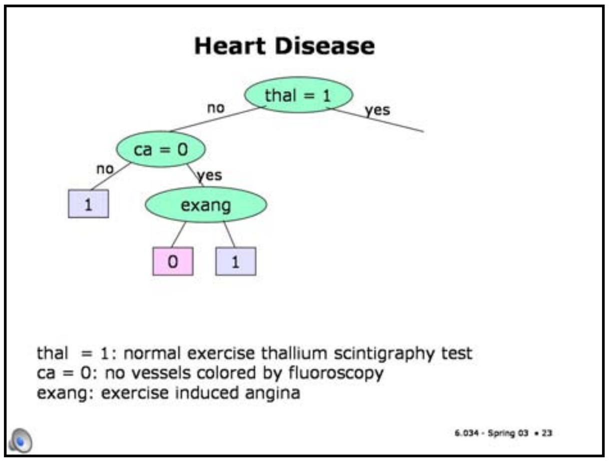
Heart Disease

thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy

6.034 - Spring 03 • 22

Slide 10.2.22
So "ca = 0" is false if 1 or more blood vessels appeared to be blocked. If that's the case, we assert that the patient has heart disease.

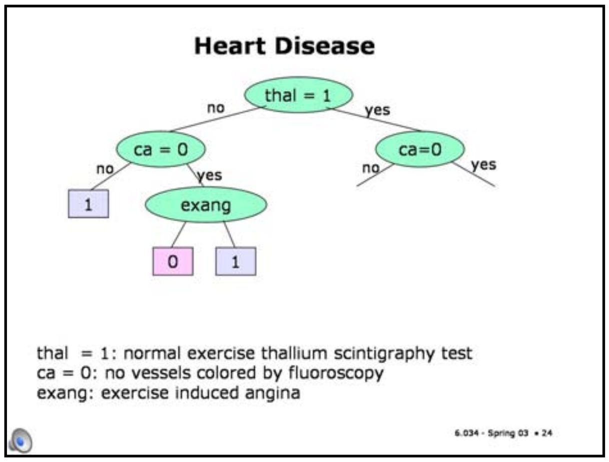
Slide 10.2.23
Now, if no blood vessels appeared to be blocked, we ask whether the patient is having exercise-induced angina (chest pain) or not. If not, we say they don't have heart disease; if so, we say they do.



Heart Disease

thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy
exang: exercise induced angina

6.034 - Spring 03 • 23



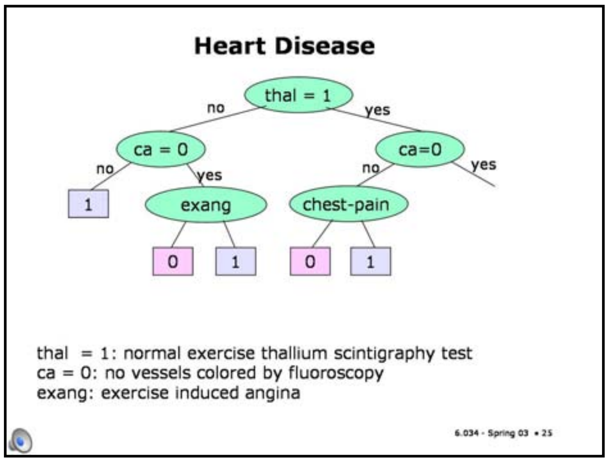
Heart Disease

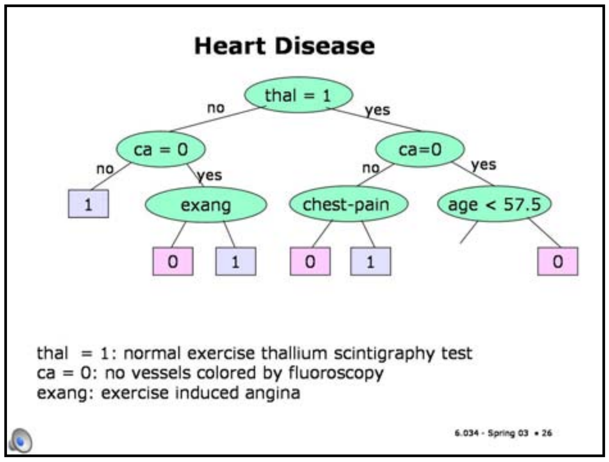
thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy
exang: exercise induced angina

6.034 - Spring 03 • 24

Slide 10.2.24
Now, over on the other side of the tree, where the first test was normal, we also look at the results of the ca test.

Slide 10.2.25
If it doesn't have value 0 (that is one or more vessels appear blocked), then we ask whether they have chest pain (presumably this is resting, not exercise-induced chest pain), and that determines the output.

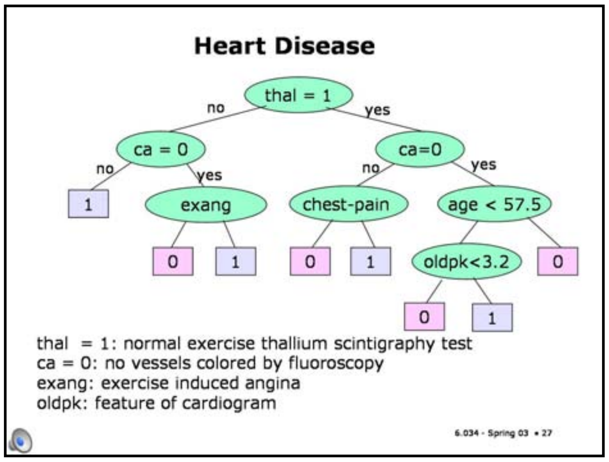

Heart Disease
thal = 1
no
yes
ca = 0
ca=0
no
yes
no
yes
1
exang
chest-pain
0
1
0
1
thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy
exang: exercise induced angina
6.034 - Spring 03 • 25

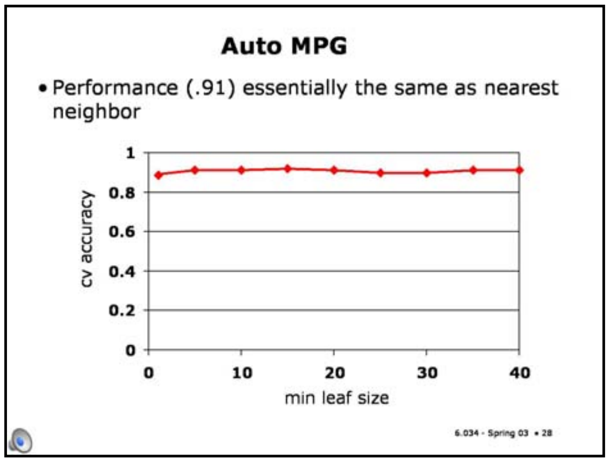

Heart Disease
thal = 1
no
yes
ca = 0
ca=0
no
yes
no
yes
1
exang
chest-pain
age < 57.5
0
1
0
1
0
thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy
exang: exercise induced angina
6.034 - Spring 03 • 26

Slide 10.2.26
If no blood vessels appear to be blocked, we consider the person's age. If they're less than 57.5, then we declare them to be heart-disease free. Whew!

Slide 10.2.27
If they're older than 57.5, then we examine some technical feature of the cardiogram, and let that determine the output.

Hypotheses like this are very important in real domains. A hospital would be much more likely to base or change their policy for admitting emergency-room patients who seem to be having heart problems based on a hypothesis that they can see and interpret rather than based on the sort of numerical gobbledigook that comes out of nearest neighbor or naive Bayes.

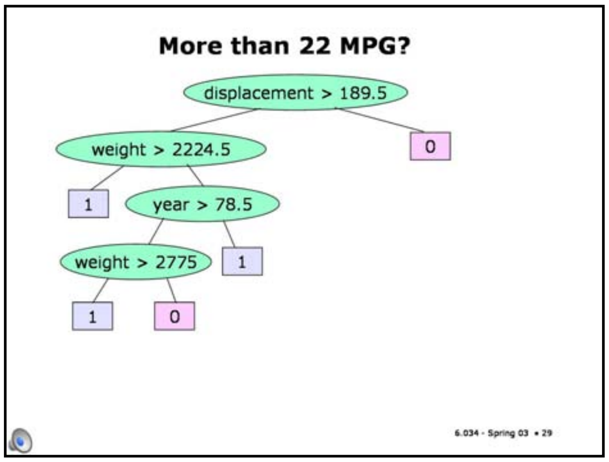

Heart Disease
thal = 1
no
yes
ca = 0
ca=0
no
yes
no
yes
1
exang
chest-pain
age < 57.5
0
1
0
1
oldpk<3.2
0
0
1
thal = 1: normal exercise thallium scintigraphy test
ca = 0: no vessels colored by fluoroscopy
exang: exercise induced angina
oldpk: feature of cardiogram
6.034 - Spring 03 • 27



Slide 10.2.28
We also ran the decision-tree algorithm on the Auto MPG data. We got essentially the same performance as nearest neighbor, and a strong insensitivity to leaf size.

Slide 10.2.29
Here's a sample resulting decision tree. It seems pretty reasonable. If the engine is big, then we're unlikely to have good gas mileage. Otherwise, if the weight is low, then we probably do have good gas mileage. For a low-displacement, heavy car, we consider the model-year. If it's newer than 1978.5 (this is an old data set!) then we predict it will have good gas mileage. And if it's older, then we make a final split based on whether or not it's really heavy.

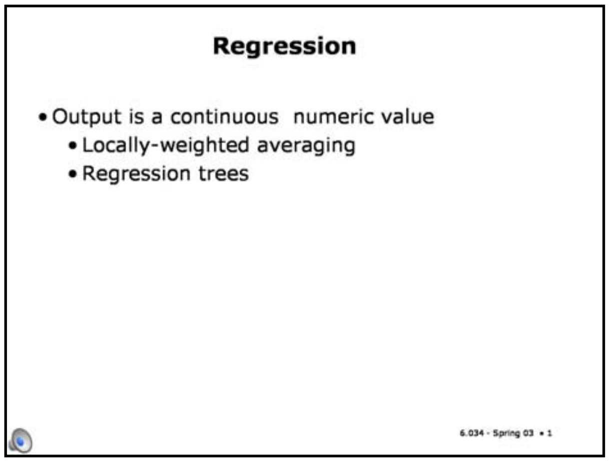
It's also possible to apply naive bayes to problems with numeric attributes, but it's hard to justify without recourse to probability, so we'll skip it. %To do: %- add a slide showing how one non-isothetic split would do the job, %but it requires a lot of rectangles.



More than 22 MPG?

6.034 Notes: Section 10.3

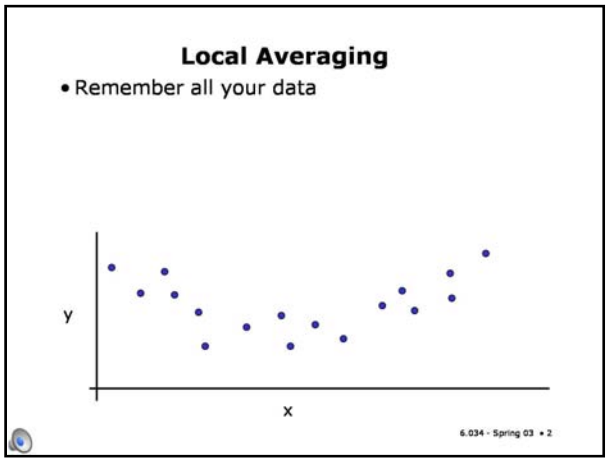
Slide 10.3.1
So far, we've spent all of our time looking at classification problems, in which the y values are either 0 or 1. Now we'll briefly consider the case where the y's are numeric values. We'll see how to extend nearest neighbor and decision trees to solve regression problems.



Regression

- Output is a continuous numeric value
  - Locally-weighted averaging
  - Regression trees

6.034 - Spring 03 • 1

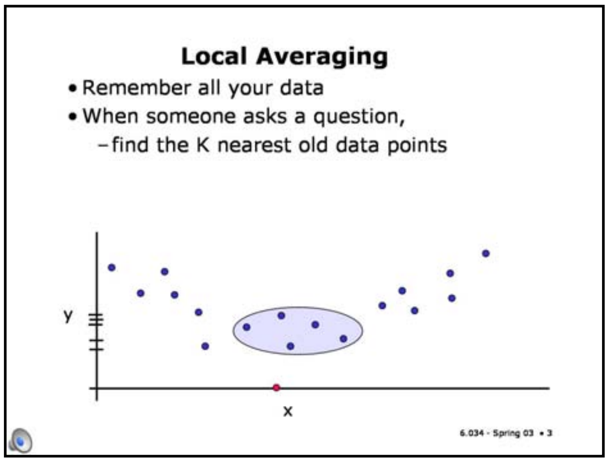


Local Averaging

- Remember all your data

Slide 10.3.2
The simplest method for doing regression is based on nearest neighbor. As in nearest neighbor, you remember all your data.

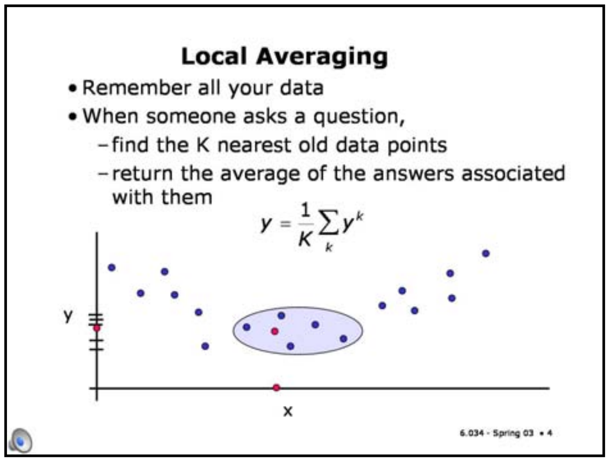
Slide 10.3.3
When you get a new query point x, you find the k nearest points.



Local Averaging

- Remember all your data
- When someone asks a question,
  - find the K nearest old data points

y
x
6.034 - Spring 03 • 3



Local Averaging

- Remember all your data
- When someone asks a question,
  - find the K nearest old data points
  - return the average of the answers associated with them

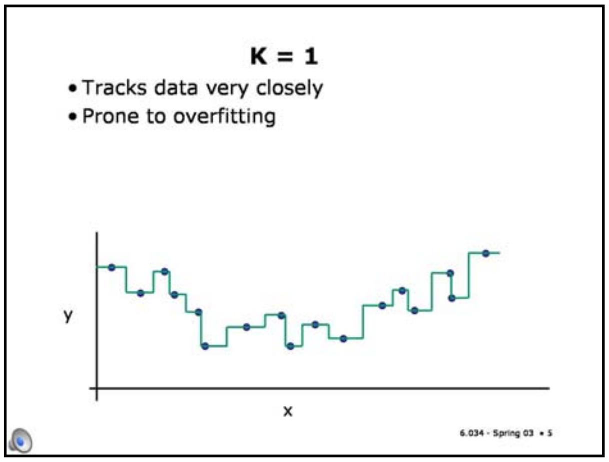
$$y = \frac{1}{K}\sum_k y^k$$

y
x
6.034 - Spring 03 • 4

Slide 10.3.4
Then average their y values and return that as your answer.

Of course, I'm showing this picture with a one-dimensional x, but the idea applies for higher-dimensional x, with the caveat that as the dimensionality of x increases, the curse of dimensionality is likely to be upon us.

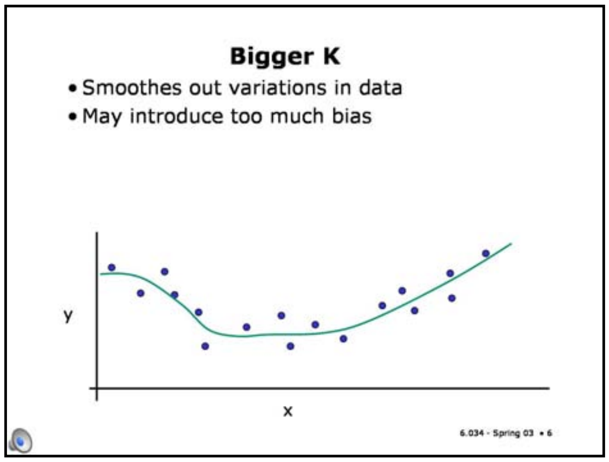
Slide 10.3.5
When k = 1, this is like fitting a piecewise constant function to your data. It will track your data very closely, but, as in nearest neighbor, have high variance and be prone to overfitting.



K = 1

- Tracks data very closely
- Prone to overfitting

y
x
6.034 - Spring 03 • 5



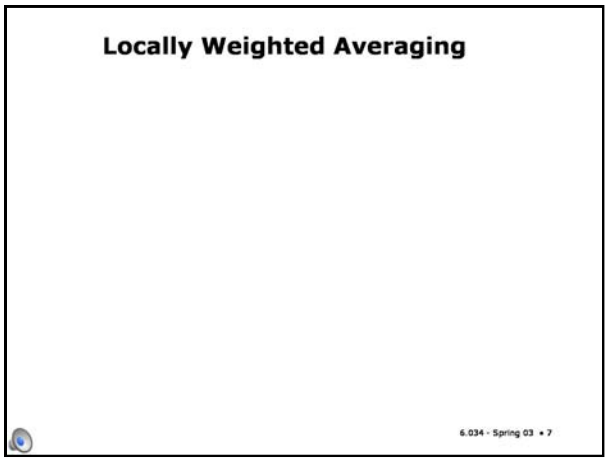
Bigger K

- Smoothes out variations in data
- May introduce too much bias

y
x
6.034 - Spring 03 • 6

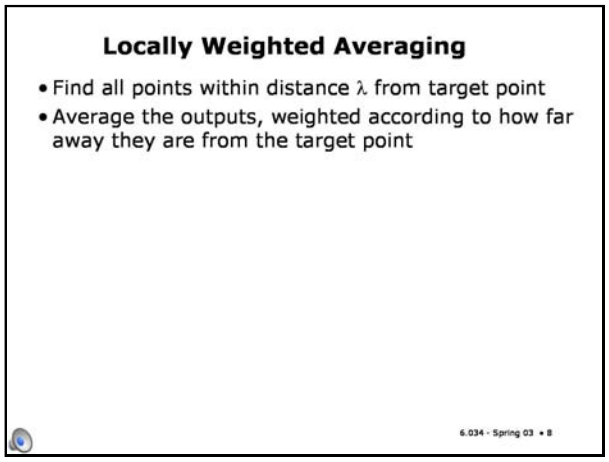
Slide 10.3.6
When k is larger, variations in the data will be smoothed out, but then there may be too much bias, making it hard to model the real variations in the function.

Slide 10.3.7
One problem with plain local averaging, especially as k gets large, is that we are letting all k neighbors have equal influence on the predicting the output of the query point. In locally weighted averaging, we still average the y values of multiple neighbors, but we weight them according to how close they are to the target point. That way, we let nearby points have a larger influence than farther ones.



Locally Weighted Averaging

6.034 - Spring 03 • 7



Locally Weighted Averaging

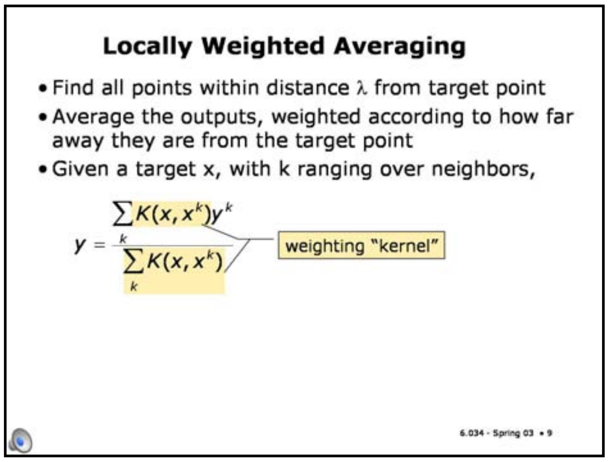
- Find all points within distance λ from target point
- Average the outputs, weighted according to how far away they are from the target point

6.034 - Spring 03 • 8

Slide 10.3.8
The simplest way to describe locally weighted averaging involves finding all points that are within a distance lambda from the target point, rather than finding the k nearest points. We'll describe it this way, but it's not too hard to go back and reformulate it to depend on the k nearest.

Slide 10.3.9
Rather than committing to the details of the weighting function right now, let's just assume that we have a "kernel" function K, which takes the query point and a training point, and returns a weight, which indicates how much influence the y value of the training point should have on the predicted y value of the query point.

Then, to compute the predicted y value, we just add up all of the y values of the points used in the prediction, multiplied by their weights, and divide by the sum of the weights.



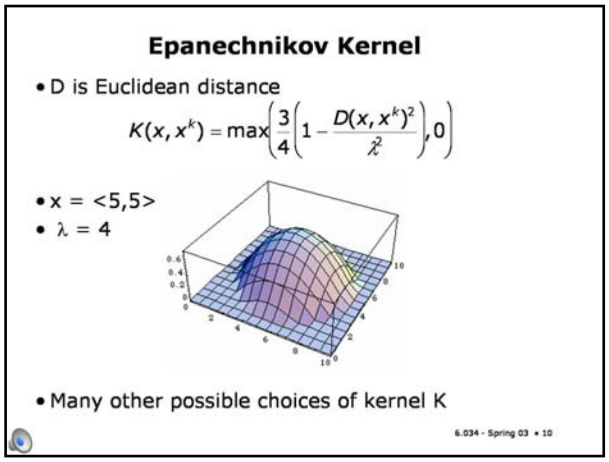
Locally Weighted Averaging

- Find all points within distance λ from target point
- Average the outputs, weighted according to how far away they are from the target point
- Given a target x, with k ranging over neighbors,

$$y = \frac{\sum_k K(x,x^k)y^k}{\sum_k K(x,x^k)}$$

weighting "kernel"

6.034 - Spring 03 • 9

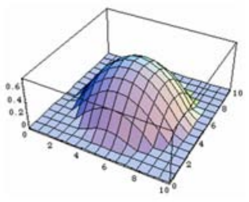


Epanechnikov Kernel

- D is Euclidean distance

$$K(x,x^k) = \max\left(\frac{3}{4}\left(1 - \frac{D(x,x^k)^2}{\lambda^2}\right), 0\right)$$

- x = <5,5>
- λ = 4



- Many other possible choices of kernel K

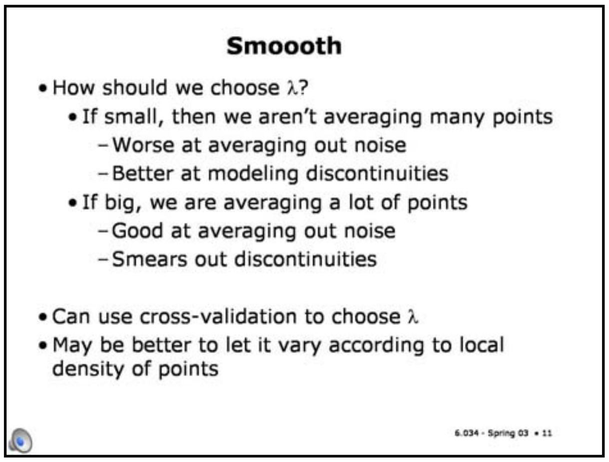
6.034 - Spring 03 • 10

Slide 10.3.10
Here is one popular kernel, which is called the Epanechnikov kernel (I like to say that word!). You don't have to care too much about it; but see that it gives high weight to points that are near the query point (5,5 in this graph) and decreasing weights out to distance lambda.

There are lots of other kernels which have various plusses and minuses, but the differences are too subtle for us to bother with at the moment.

Slide 10.3.11
As usual, we have the same issue with lambda here as we have had with epsilon, min-leaf-size, and k. If it's too small, we'll have high variance; if it's too big, we'll have high bias. We can use cross-validation to choose.

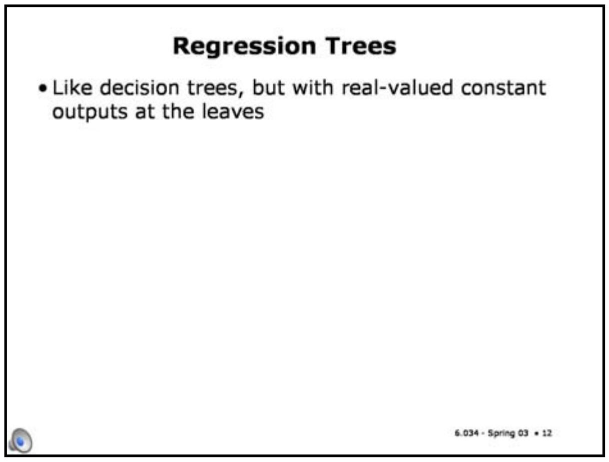
In general, it's better to convert the algorithm to use k instead of lambda (it just requires making the lambda parameter in the kernel be the distance to the farthest of the k nearest neighbors). This means that we're always averaging the same number of points; so in regions where we have a lot of data, we'll look more locally, but in regions where the training data is sparse, we'll cast a wider net.



Smoooth

- How should we choose λ?
  - If small, then we aren't averaging many points
    - Worse at averaging out noise
    - Better at modeling discontinuities
  - If big, we are averaging a lot of points
    - Good at averaging out noise
    - Smears out discontinuities
- Can use cross-validation to choose λ
- May be better to let it vary according to local density of points

6.034 - Spring 03 • 11



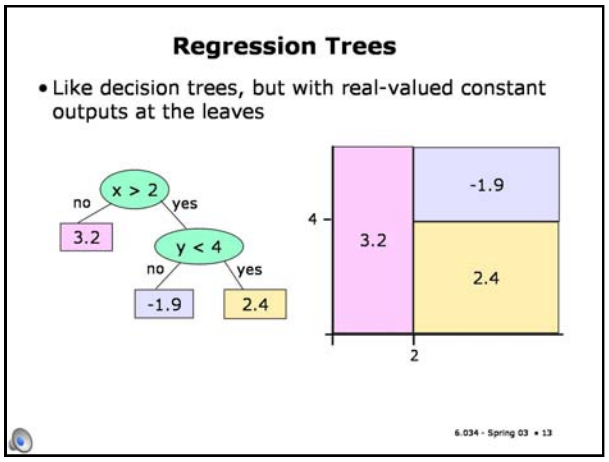
Regression Trees

- Like decision trees, but with real-valued constant outputs at the leaves

6.034 - Spring 03 • 12

Slide 10.3.12
Now we'll take a quick look at regression trees, which are like decision trees, but which have numeric constants at the leaves rather than booleans.

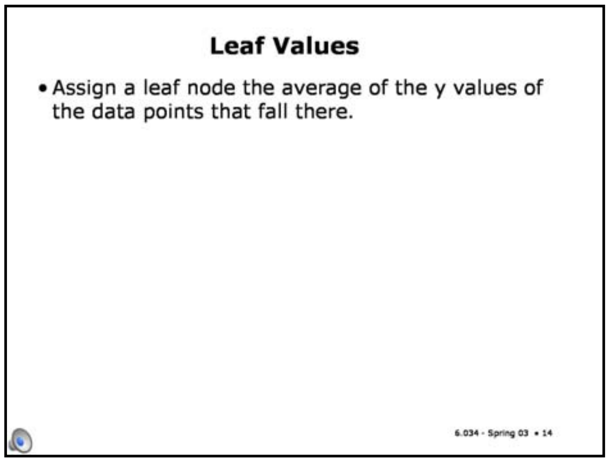
Slide 10.3.13
Here's an example regression tree. It has the same kinds of splits as a regular tree (in this case, with numeric features), but what's different are the labels of the leaves.



Regression Trees

- Like decision trees, but with real-valued constant outputs at the leaves

6.034 - Spring 03 • 13



Leaf Values

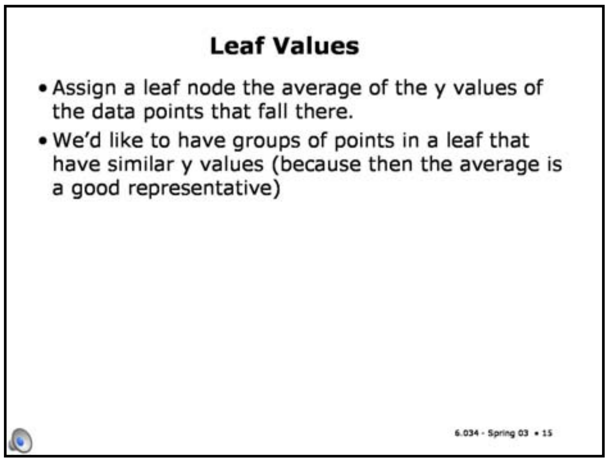
- Assign a leaf node the average of the y values of the data points that fall there.

6.034 - Spring 03 • 14

Slide 10.3.14
Let's start by thinking about how to assign a value to a leaf, assuming that multiple training points are in the leaf and we have decided, for whatever reason, to stop spliting.

In the boolean case, we used the majority output value as the value for the leaf. In the numeric case, we'll use the average output value. It makes sense, and besides there's a hairy statistical argument in favor of it, as well.

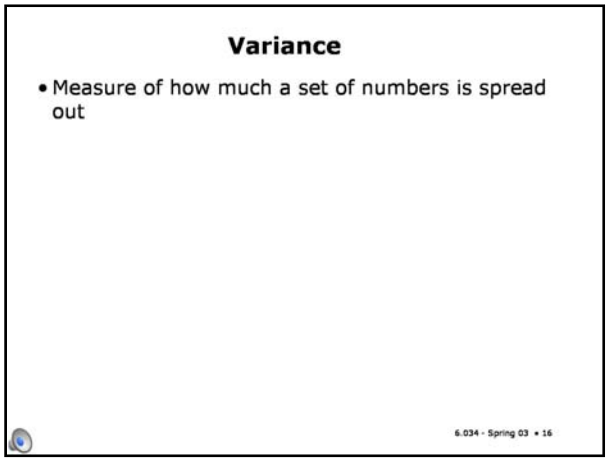
Slide 10.3.15
So, if we're going to use the average value at a leaf as its output, we'd like to split up the data so that the leaf averages are not too far away from the actual items in the leaf.



Leaf Values

- Assign a leaf node the average of the y values of the data points that fall there.
- We'd like to have groups of points in a leaf that have similar y values (because then the average is a good representative)

6.034 - Spring 03 • 15



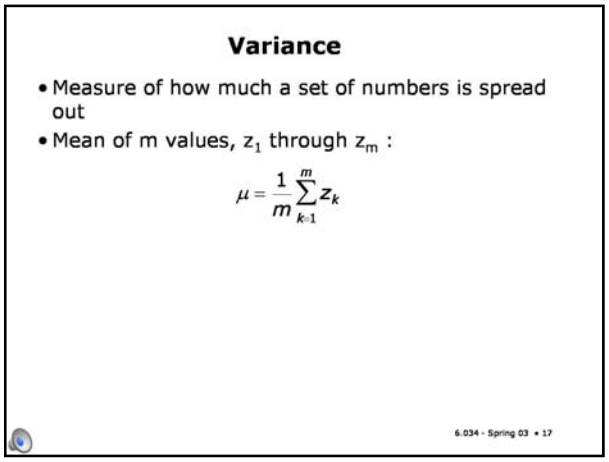
Variance

- Measure of how much a set of numbers is spread out

6.034 - Spring 03 • 16

Slide 10.3.16
Lucky for us, the statistics folks have a good measure of how spread out a set of numbers is (and, therefore, how different the individuals are from the average); it's called the variance of a set.

Slide 10.3.17
First we need to know the mean, which is traditionally called mu. It's just the average of the values. That is, the sum of the values divided by how many there are (which we call m, here).

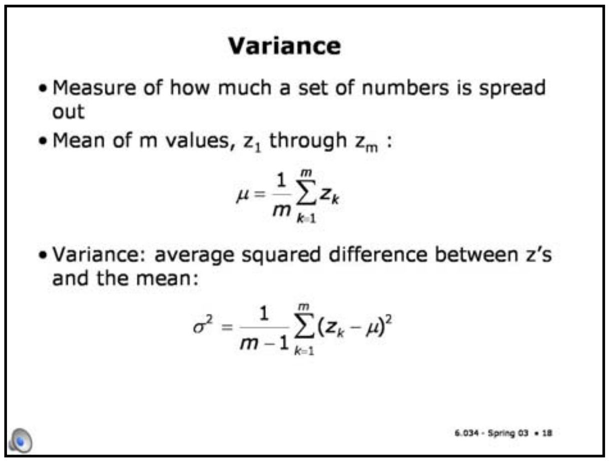


Variance

- Measure of how much a set of numbers is spread out
- Mean of m values, z_1 through z_m :

$$\mu = \frac{1}{m}\sum_{k=1}^{m} z_k$$

6.034 - Spring 03 • 17



Variance

- Measure of how much a set of numbers is spread out
- Mean of m values, z_1 through z_m :

$$\mu = \frac{1}{m}\sum_{k=1}^{m} z_k$$

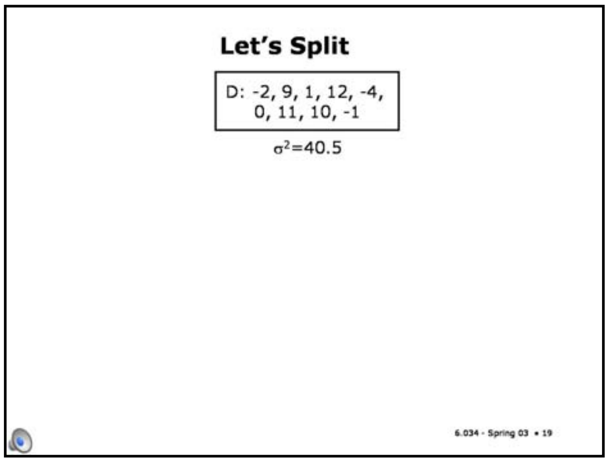
- Variance: average squared difference between z's and the mean:

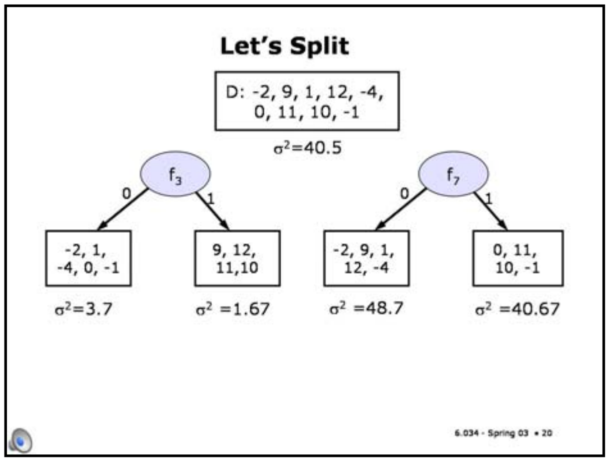
$$\sigma^2 = \frac{1}{m-1}\sum_{k=1}^{m}(z_k - \mu)^2$$

6.034 - Spring 03 • 18

Slide 10.3.18
Then the variance is essentially the average of the squared distance between the individual values and the mean. If it's the average, then you might wonder why we're dividing by m-1 instead of m. I could tell you, but then I'd have to shoot you. Let's just say that dividing by m-1 makes it an unbiased estimator, which is a good thing.

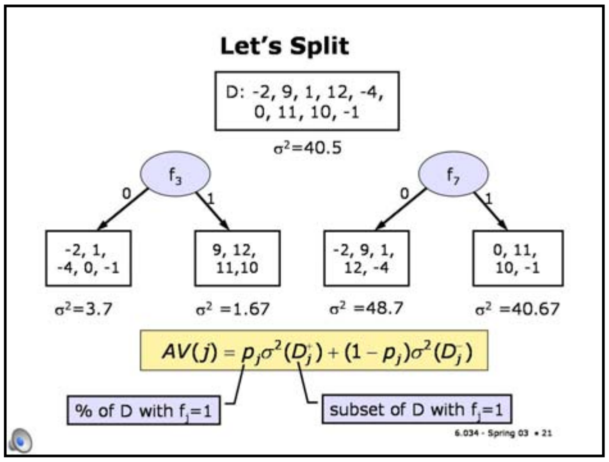
Slide 10.3.19
We're going to use the average variance of the children to evaluate the quality of splitting on a particular feature. Here we have a data set, for which I've just indicated the y values. It currently has a variance of 40.5.

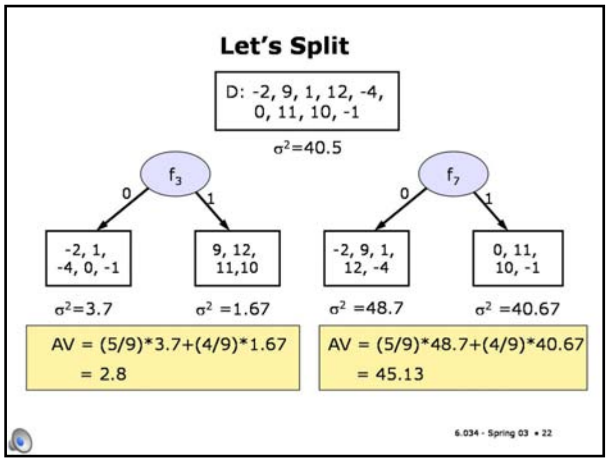




Slide 10.3.20
We're considering two splits. One gives us variances of 3.7 and 1.67; the other gives us variances of 48.7 and 40.67.

Slide 10.3.21
Just as we did in the binary case, we can compute a weighted average variance, depending on the relative sizes of the two sides of the split.

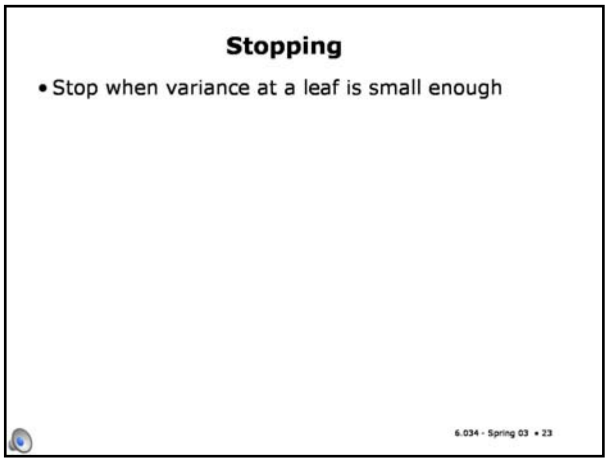




Slide 10.3.22
Doing so, we can see that the average variance of splitting on feature 3 is **much** lower than of splitting on f7, and so we'd choose to split on f3.

Just looking at the data in the leaves, f3 seems to have done a much better job of dividing the values into similar groups.

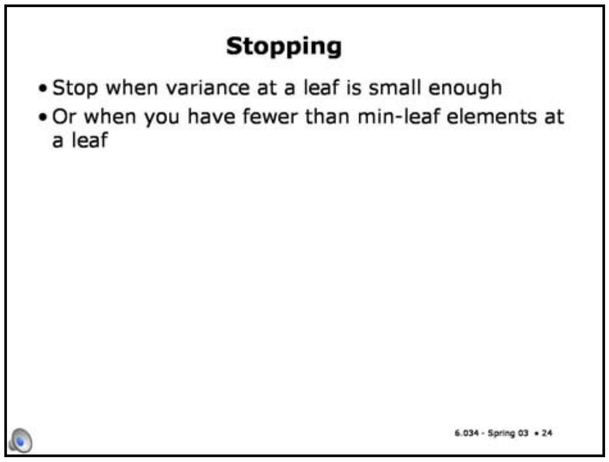
Slide 10.3.23
We can stop growing the tree based on criteria that are similar to those we used in the binary case. One reasonable criterion is to stop when the variance at a leaf is lower than some threshold.



Stopping

- Stop when variance at a leaf is small enough

6.034 - Spring 03 • 23



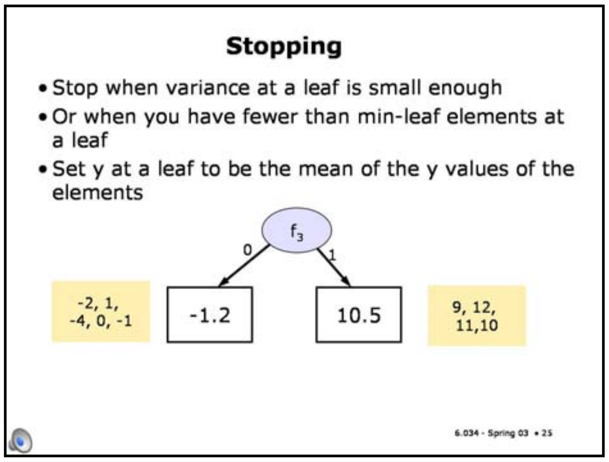
Stopping

- Stop when variance at a leaf is small enough
- Or when you have fewer than min-leaf elements at a leaf

6.034 - Spring 03 • 24

Slide 10.3.24
Or we can use our old min-leaf-size criterion.

Slide 10.3.25
Once we do decide to stop, we assign each leaf the average of the values of the points in it.



Stopping

- Stop when variance at a leaf is small enough
- Or when you have fewer than min-leaf elements at a leaf
- Set y at a leaf to be the mean of the y values of the elements

6.034 - Spring 03 • 25