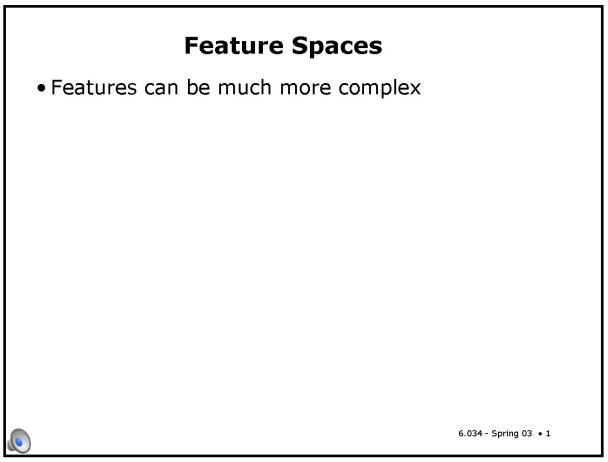
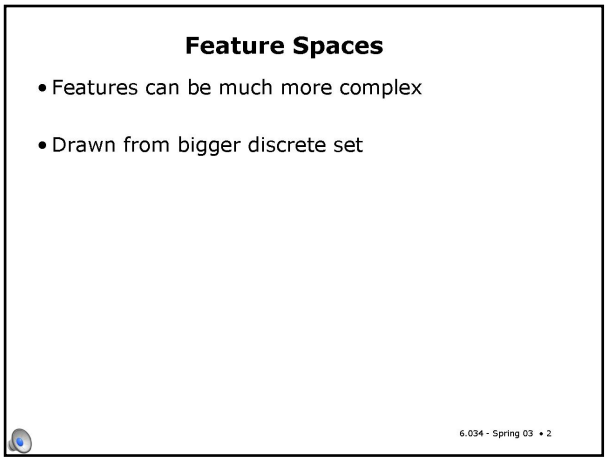
6.034 Notes: Section 3.1

Slide 3.1.1
So far, we've only talked about binary features. But real problems are typically characterized by much more complex features.



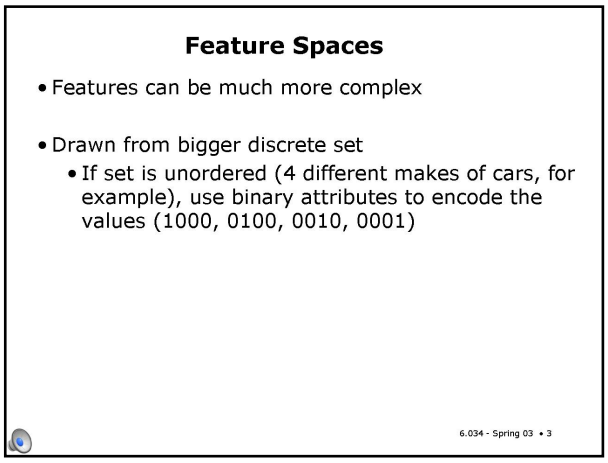


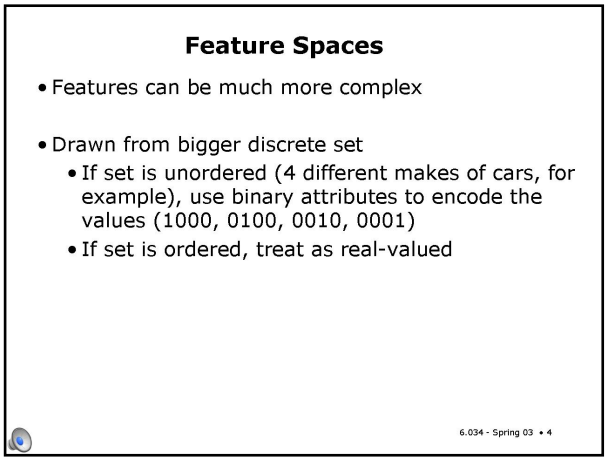
Slide 3.1.2
Some features can take on values in a discrete set that has more than two elements. Examples might be the make of a car, or the age of a person.

Slide 3.1.3
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.

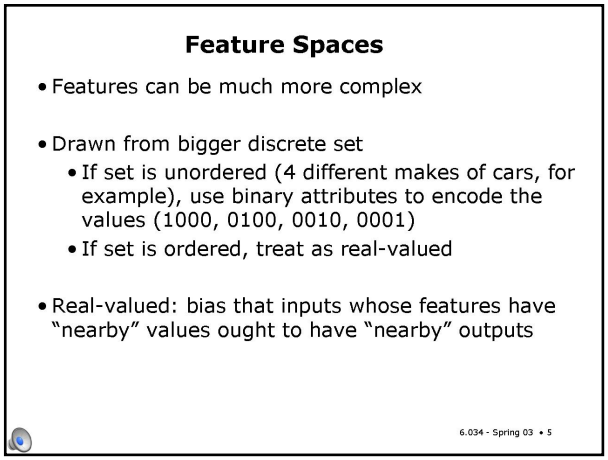
Instead, we typically make it easier on our algorithms by encoding such sets in unary, with one bit per element in the set. Then, for each value, we turn on one bit and set the rest to zero. So, we could encode a four-item set as 1000, 0100, 0010, 0001.

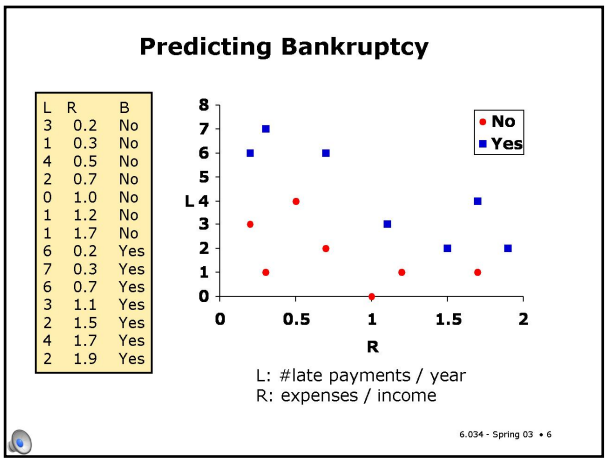




Slide 3.1.4
On the other hand, when the set has a natural order, like someone's age, or the number of bedrooms in a house, it can usually be treated as if it were a real-valued attribute using methods we're about to explore.

Slide 3.1.5
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.





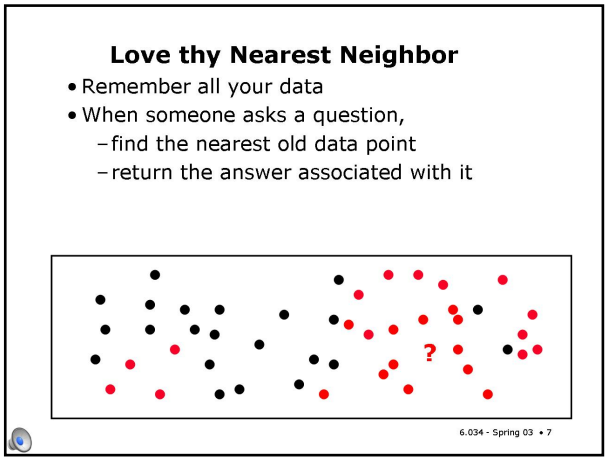
Slide 3.1.6
We'll use the example of predicting whether someone is going to go bankrupt. It only has two features, to make it easy to visualize.

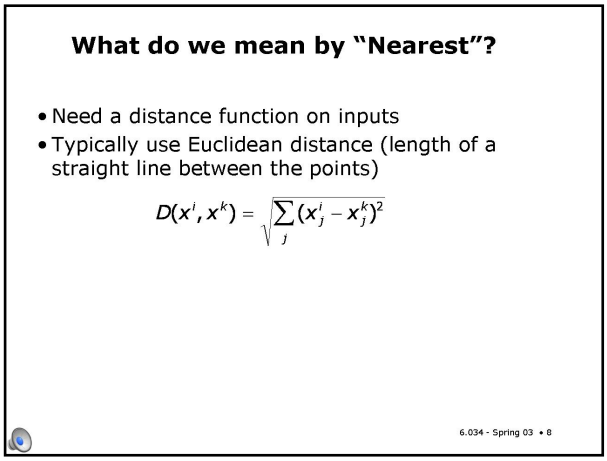
One feature, L, is the number of late payments they have made on their credit card this year. This is a discrete value that we're treating as a real.

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.

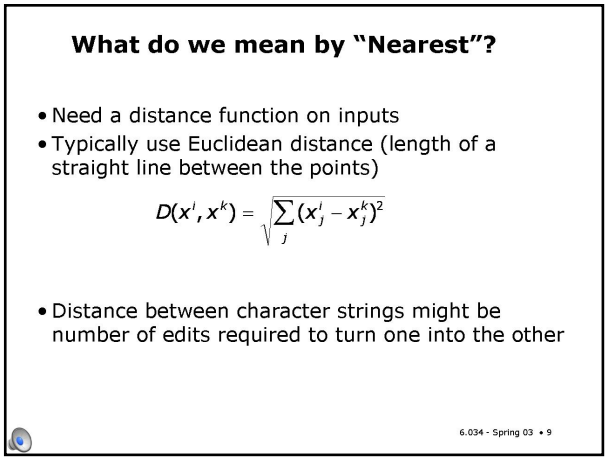
Slide 3.1.7
We took a brief look at the nearest neighbor algorithm in the first segment on learning. The idea is that you remember all the data points you've ever seen and, when you're given a query point, you find the old point that's nearest to the query point and predict its y value as your output.

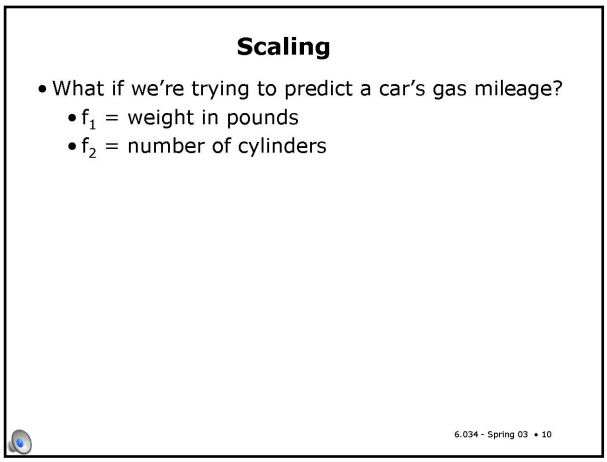




Slide 3.1.8
In order to say what point is nearest, we have to define what we mean by "near". Typically, we use Euclidean distance between two points, which is just the square root of the sum of the squared differences between corresponding feature values.

Slide 3.1.9
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.

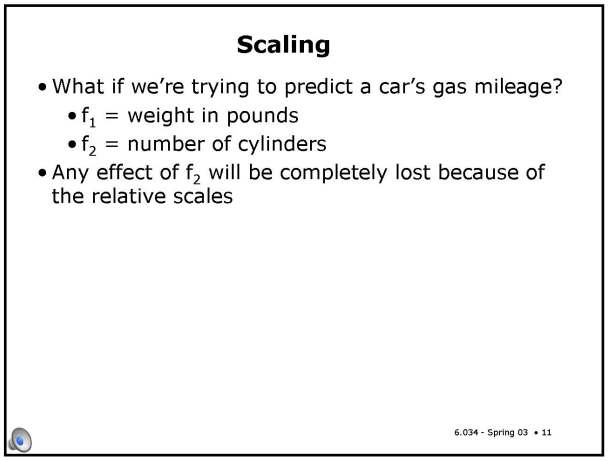


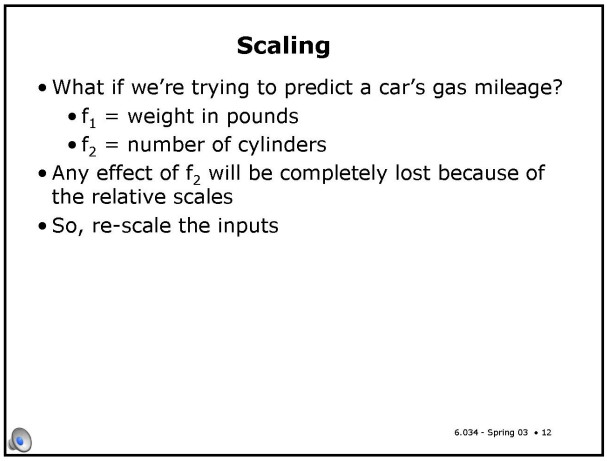


Slide 3.1.10
The naive Euclidean distance isn't always appropriate, though.

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 3.1.11
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.

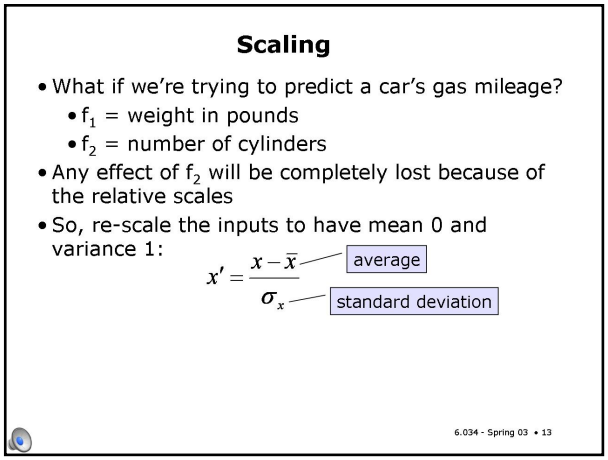


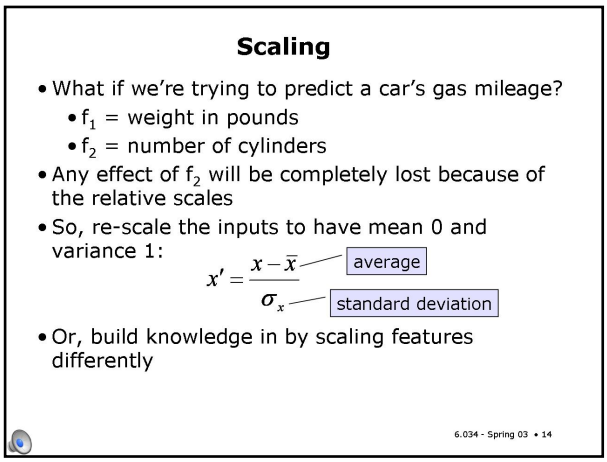


Slide 3.1.12
One standard method for addressing this problem is to re-scale the features.

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.

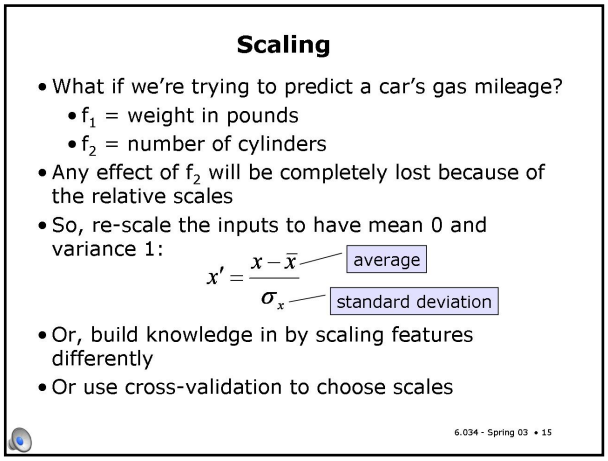
Slide 3.1.13
A somewhat more robust method (in case you have a crazy measurement, perhaps due to a noise in a sensor, that would make the range huge) is to scale the inputs to have 0 mean and standard deviation 1. If you haven't seen this before, it means to compute the average value of the feature, 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.

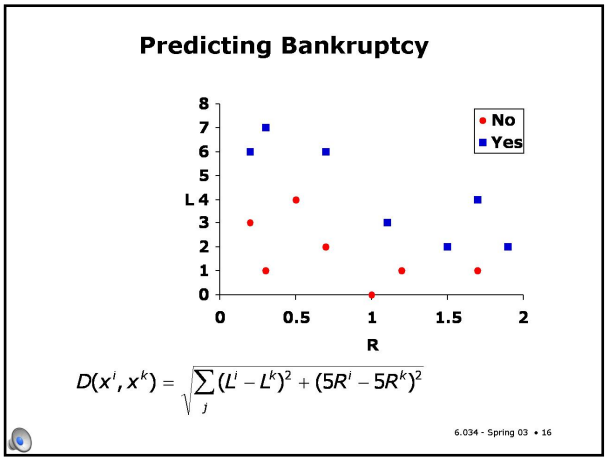




Slide 3.1.14
Of course, you may not want to have all your features on equal footing. It may be that you happen to know, based on the nature of the domain, that some features are more important than others. In such cases, you might want to multiply them by a weight that will increase their influence in the distance calculation.

Slide 3.1.15
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.

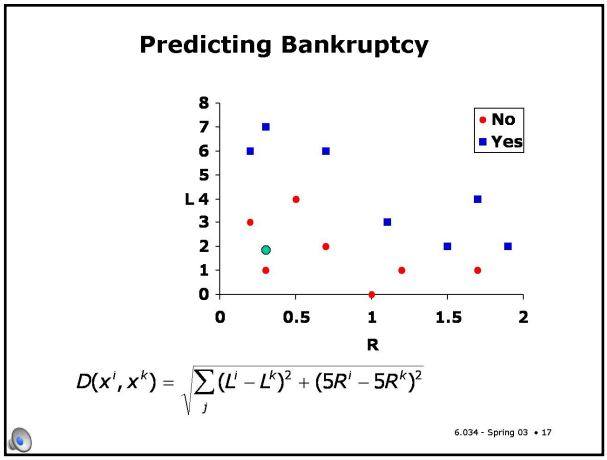


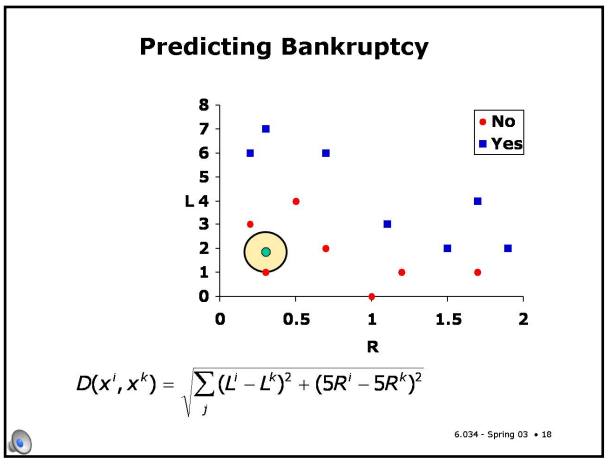


Slide 3.1.16
Okay. Let's see how nearest neighbor works on our bankruptcy example. Let's say we've thought about the domain and decided that the R feature (ratio between expenses and income) needs to be scaled up by 5 in order to be appropriately balanced against the L feature (number of late payments).

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.

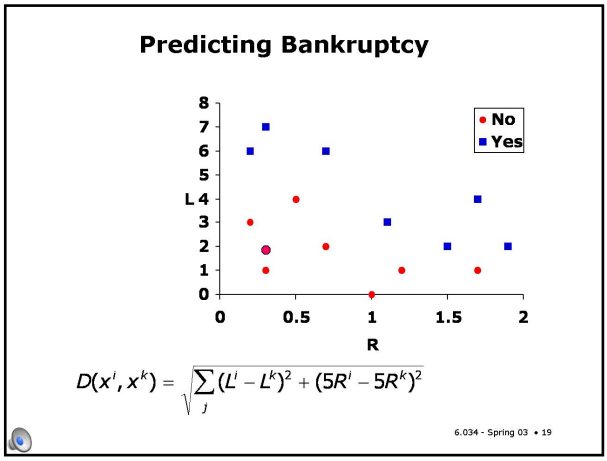
Slide 3.1.17
Now, let's say we have a new person with R equal 0.3 and L equal to 2. What y value should we predict?

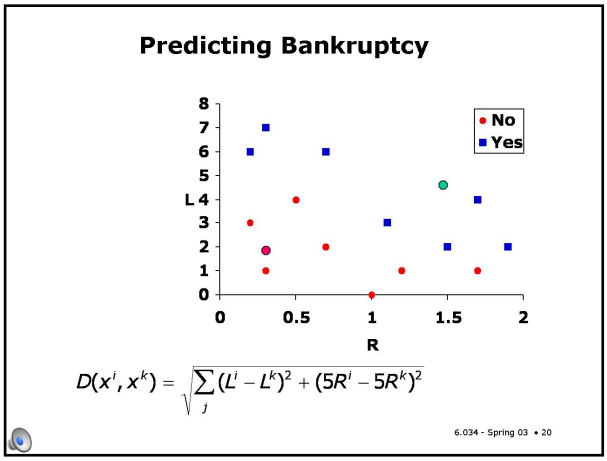




Slide 3.1.18
We look for the nearest point, which is the red point at the edge of the yellow circle. The fact that there are no old points in the circle means that this red point is indeed the nearest neighbor of our query point.

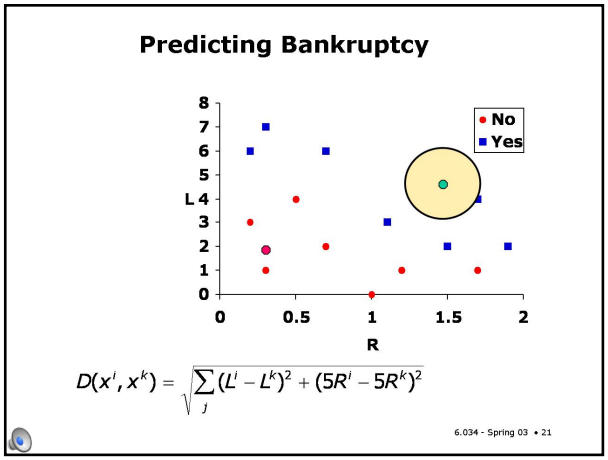
Slide 3.1.19
And so our answer would be "no".

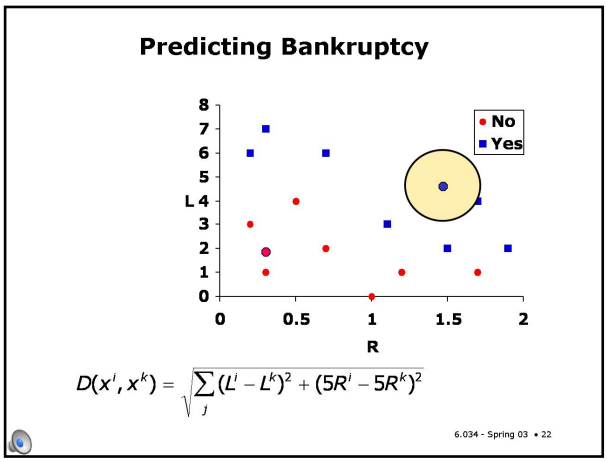




Slide 3.1.20
Similarly, for another query point,

Slide 3.1.21
we find the nearest neighbor, which has output "yes"

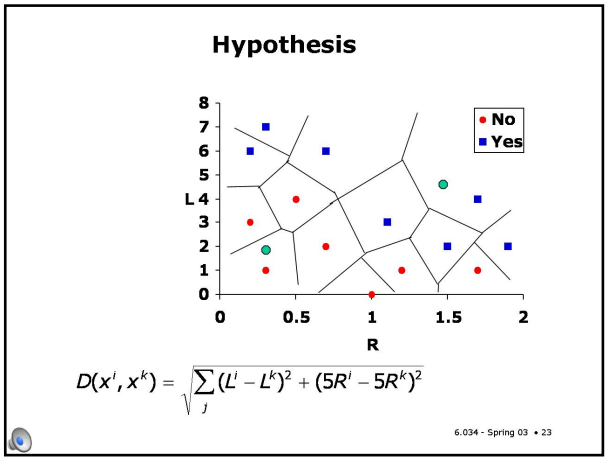


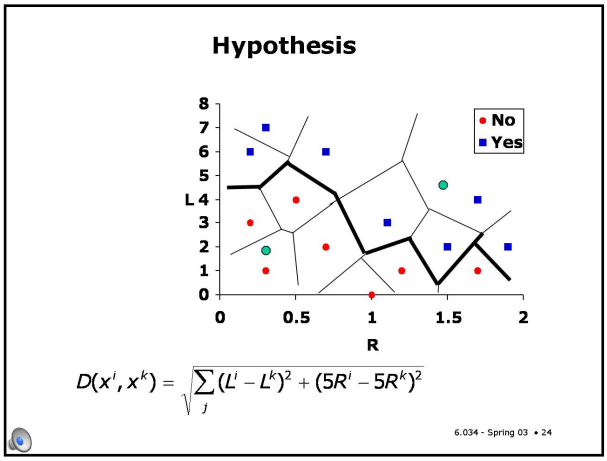


Slide 3.1.22
and generate "yes" as our prediction.

Slide 3.1.23
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.

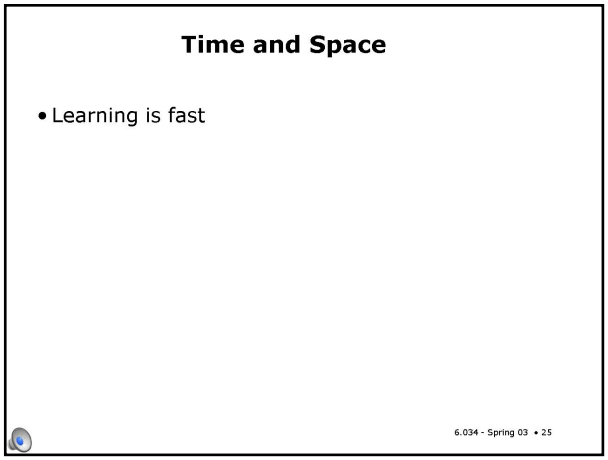


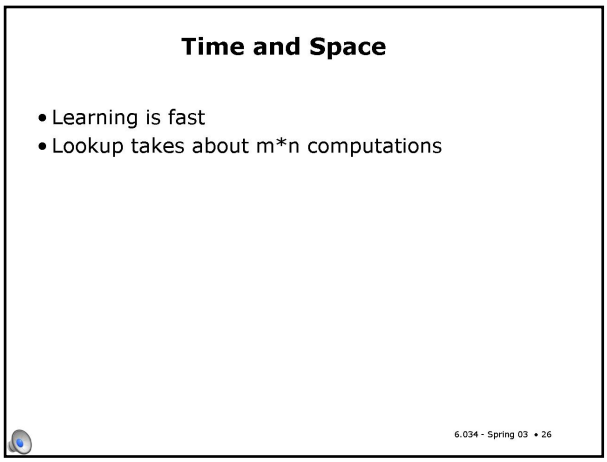


Slide 3.1.24
Now, we can think of our hypothesis as being represented by the edges in the Voronoi partition that separate a region associated with a positive point from a region associated with a negative one. In our example, that generates this bold boundary.

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

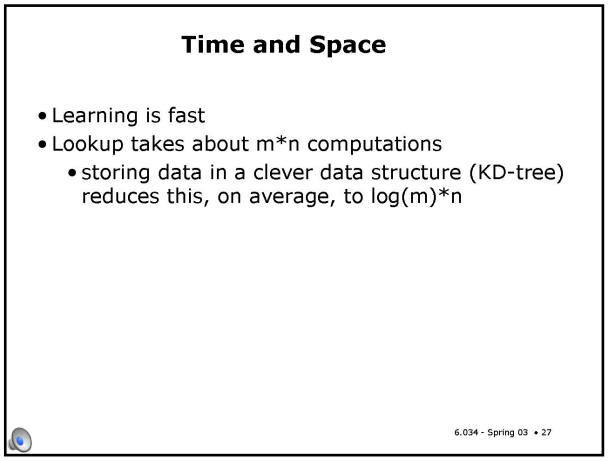
Slide 3.1.25
It's useful to spend a little bit of time thinking about how complex this algorithm is. Learning is very fast. All you have to do is remember all the data you've seen!

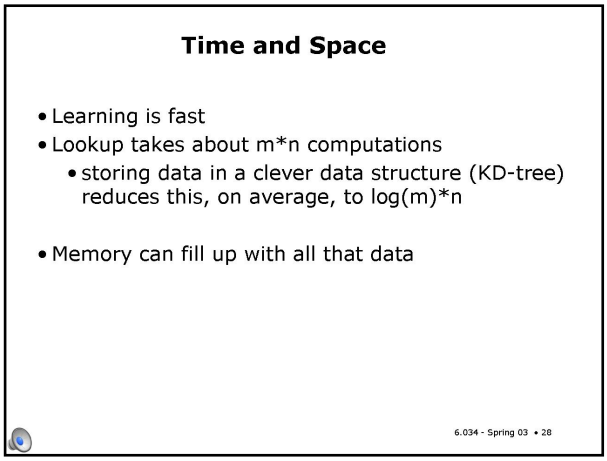




Slide 3.1.26
What takes longer is answering a query. Naively, you have to, for each point in your training set (and there are m of them) compute the distance to the query point (which takes about n computations, since there are n features to compare). So, overall, this takes about m * n time.

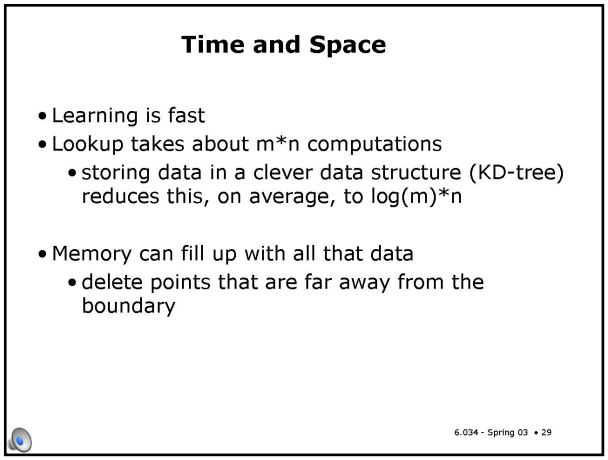
Slide 3.1.27
It's possible to organize your data into a clever data structure (one such structure is called a K-D tree). It will allow you to find the nearest neighbor to a query point in time that's, on average, proportional to the log of m, which is a huge savings.

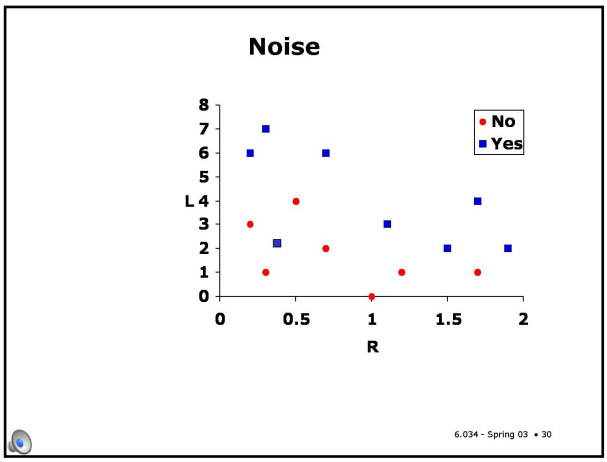




Slide 3.1.28
Another issue is memory. If you gather data over time, you might worry about your memory filling up, since you have to remember it all.

Slide 3.1.29
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.

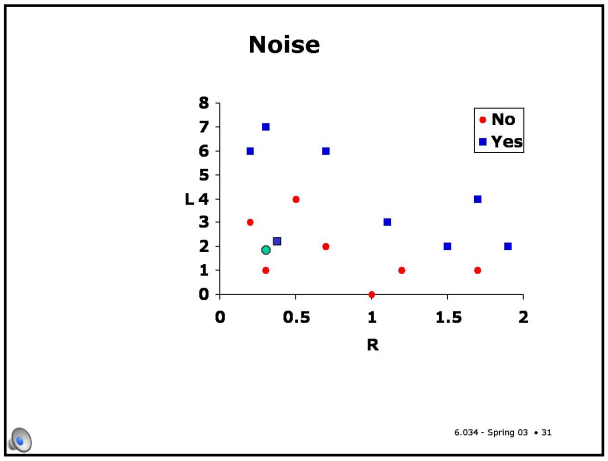


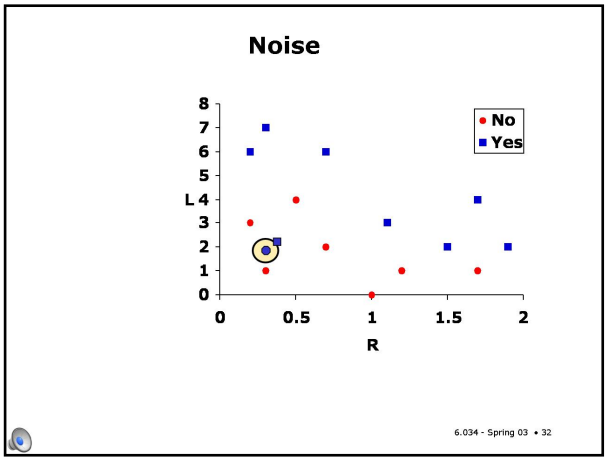


Slide 3.1.30
In our example so far, there has not been much (apparent) noise; the boundary between positives and negatives is clean and simple. Let's now consider the case where there's a blue point down among the reds. Someone with an apparently healthy financial record goes bankrupt.

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.

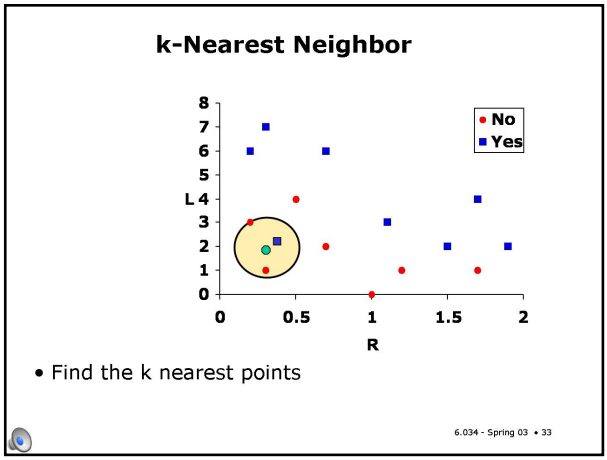
Slide 3.1.31
So, what happens in nearest neighbor if we get a query point next to this point?

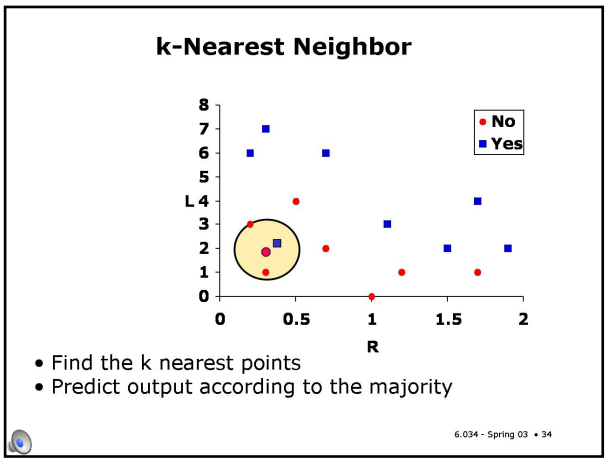




Slide 3.1.32
We find the nearest neighbor, which is a "yes" point, and predict the answer "yes". This outcome is consistent with the first view; that is, that this blue point represents some important property of the problem.

Slide 3.1.33
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.

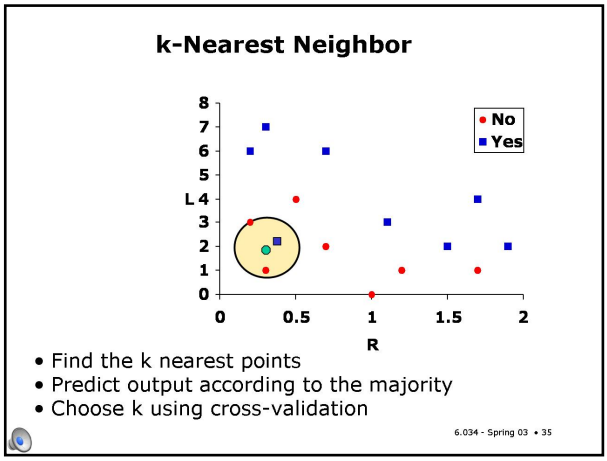


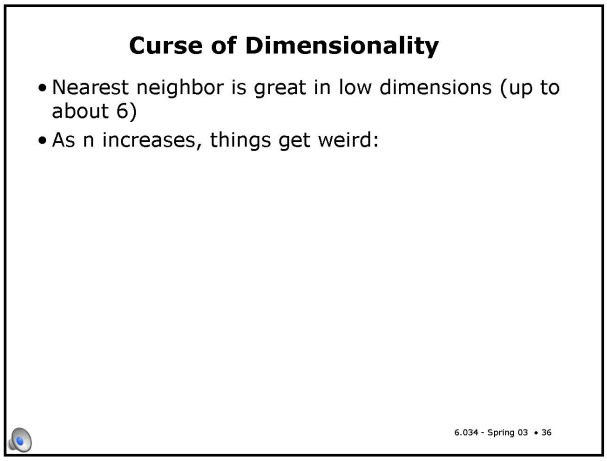


Slide 3.1.34
In this case, we've chosen k to be 3. The three closest points consist of two "no"s and a "yes", so our answer would be "no".

Slide 3.1.35
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.



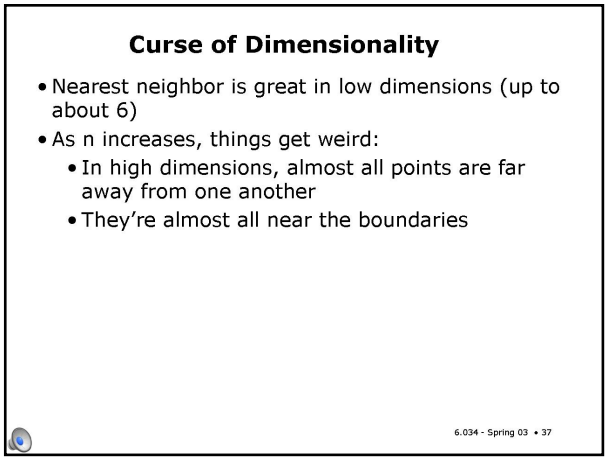


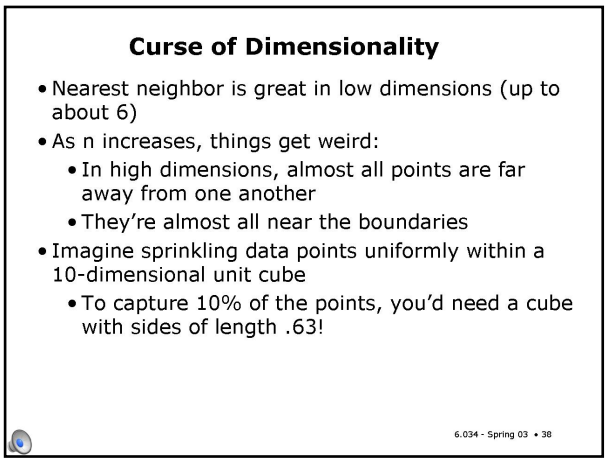
Slide 3.1.36
Nearest neighbor works very well (and is often the method of choice) for problems in relatively low-dimensional real-valued spaces.

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 3.1.37
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.

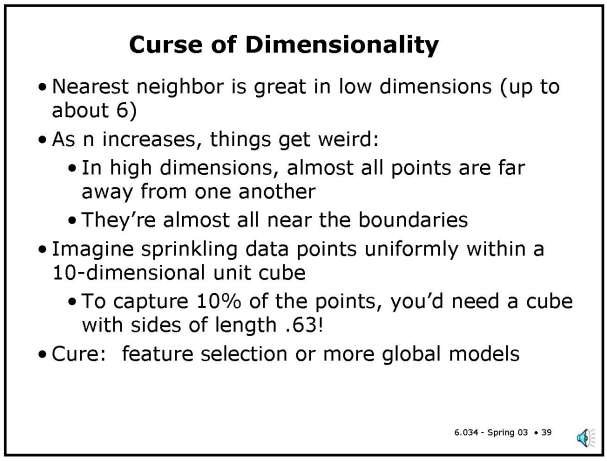


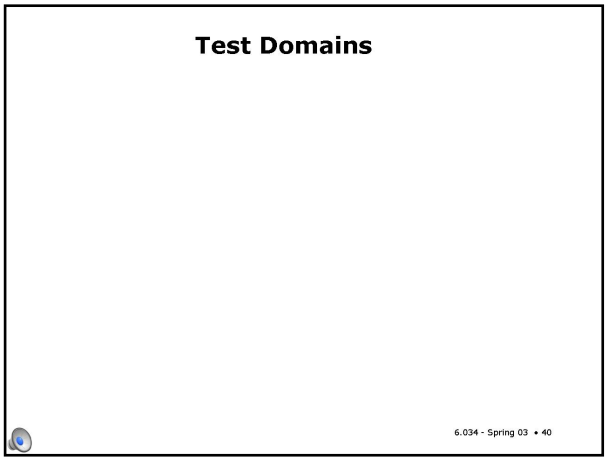


Slide 3.1.38
Imagine sprinkling data points uniformly within a 10-dimensional unit cube (cube whose sides are of length 1).

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

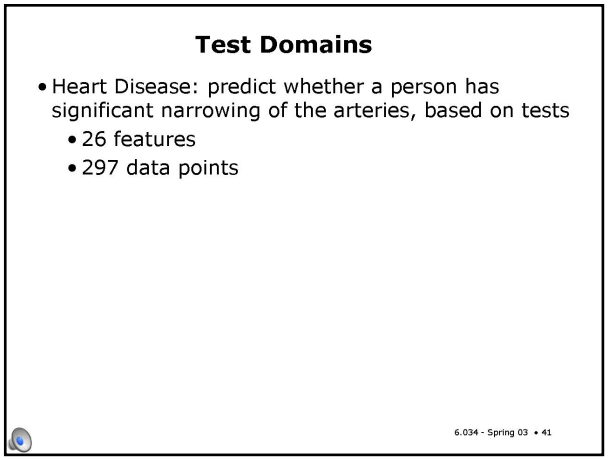
Slide 3.1.39
All this means that the notions of nearness providing a good generalization principle, which are very effective in low-dimensional spaces, become fairly ineffective in high-dimensional spaces. There are two ways to handle this problem. One is to do "feature selection", and try to reduce 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.

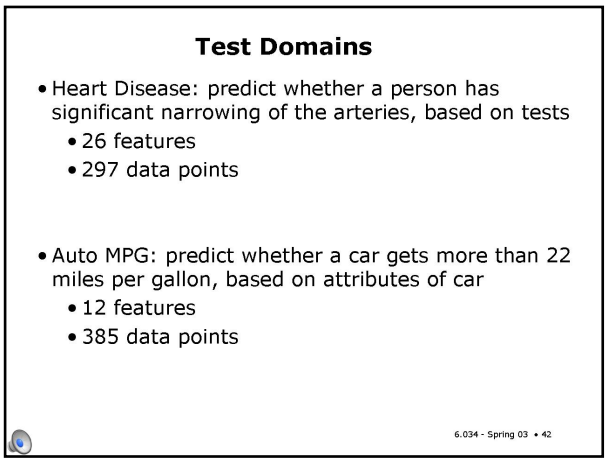




Slide 3.1.40
We'll look at how nearest neighbor performs on two different test domains.

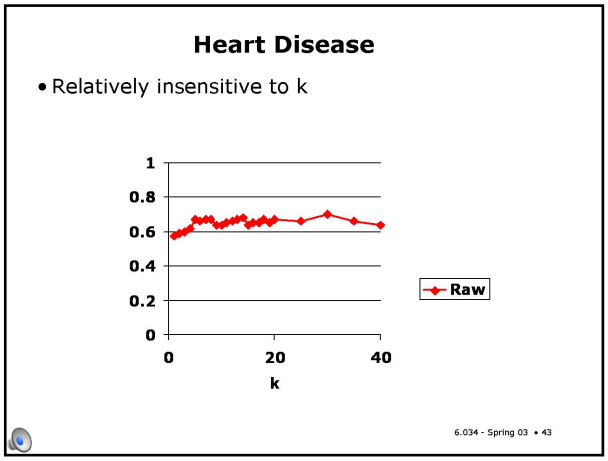
Slide 3.1.41
The first domain is predicting whether a person has heart disease, represented by a significant narrowing of the arteries, based on the results of a variety of tests. This domains has 297 different data points, each of which is characterized by 26 features. A lot of these features are actually 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.

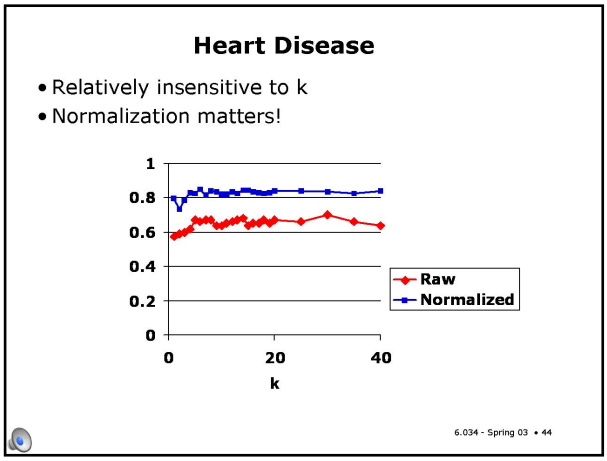




Slide 3.1.42
In the second domain, we're trying to predict whether a car gets more than 22 miles-per-gallon fuel efficiency. We have 385 data points, characterized by 12 features. Again, a number of the features are binary.

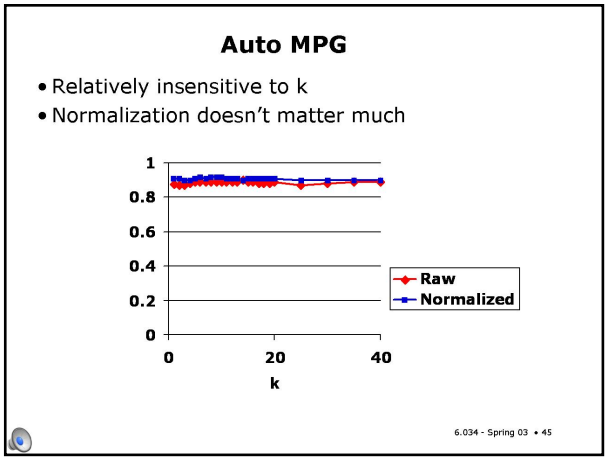
Slide 3.1.43
Here's a graph of the cross-validation accuracy of nearest neighbor on the heart disease data, shown as a function of k. Looking at the data, we can see that the performance is relatively insensitive to the choice of k, though it seems like maybe it's useful to have k be greater than about 5.

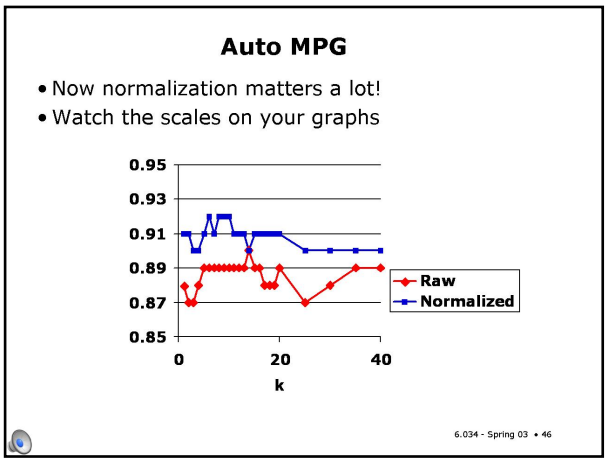




Slide 3.1.44
The red curve is the performance of nearest neighbor using the features directly as they are measured, without any scaling. We then normalized all of the features to have mean 0 and standard deviation 1, and re-ran the algorithm. You can see here that it makes a noticable increase in performance.

Slide 3.1.45
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.

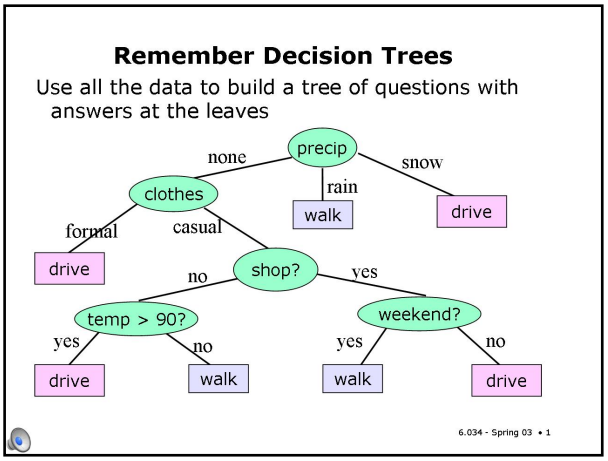


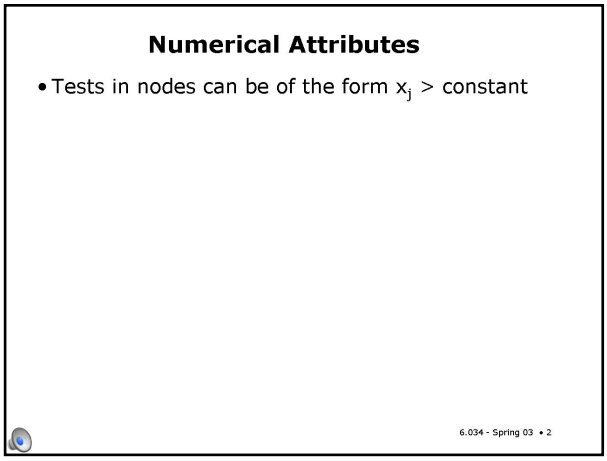


Slide 3.1.46
Watch out for tricky graphing! It's always possible to make your algorithm look much better than the other leading brand (as long as it's a little bit better), by changing the scale on your graphs. The previous graph had a scale of 0 to 1. This graph has a scale of 0.85 to 0.95. Now the 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 3.2

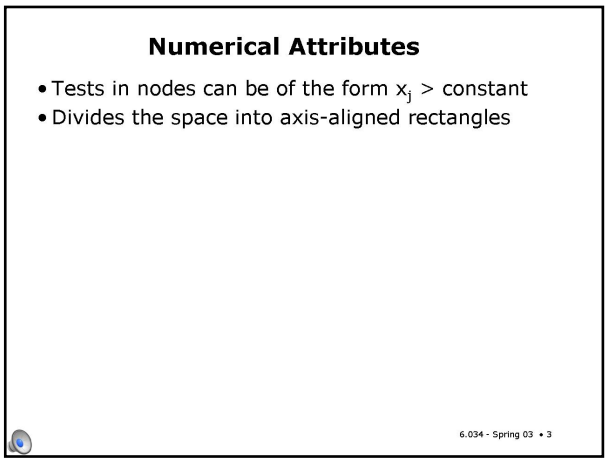
Slide 3.2.1
Now, let's go back to decision trees, and see if we can apply them to problems where the inputs are numeric.

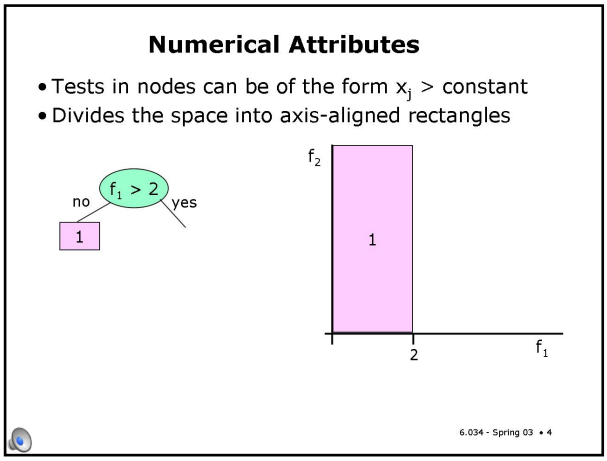




Slide 3.2.2
When we have features with numeric values, we have to expand our hypothesis space to include different tests on the leaves. We will allow tests on the leaves of a decision tree to be comparisons of the form x_j > c, where c is a constant.

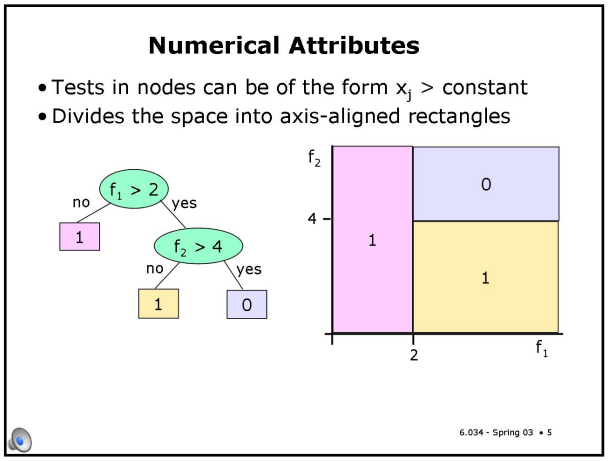
Slide 3.2.3
This class of splits allows us to divide our feature-space into a set of exhaustive and mutually exclusive hyper-rectangles (that is, rectangles of potentially high dimension), with one rectangle for each leaf of the tree. So, each rectangle will have an output value (1 or 0) associated with it. The set of rectangles and their output values constitutes our hypothesis.

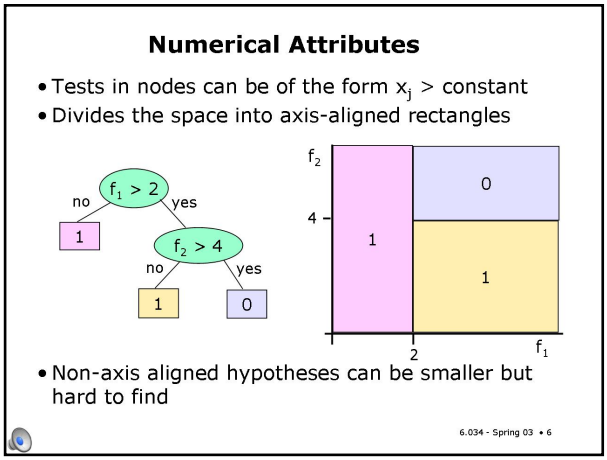




Slide 3.2.4
So, in this example, at the top level, we split the space into two parts, according to whether feature 1 has a value greater than 2. If not, then the output is 1.

Slide 3.2.5
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.

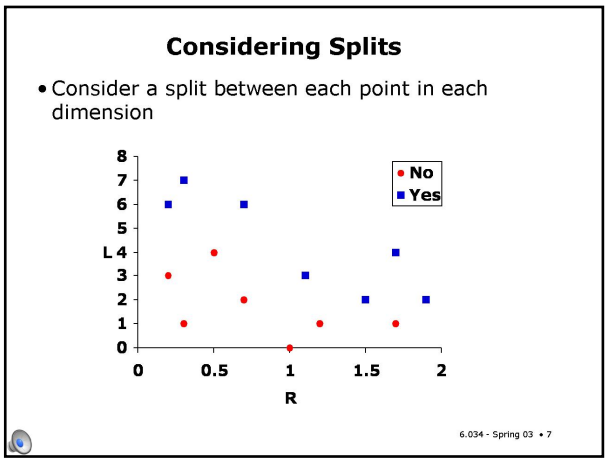


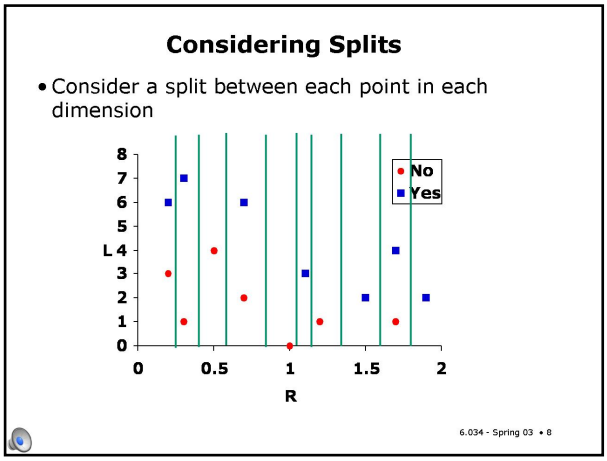


Slide 3.2.6
This class of hypotheses is fairly rich, but it can be hard to express some concepts.

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.

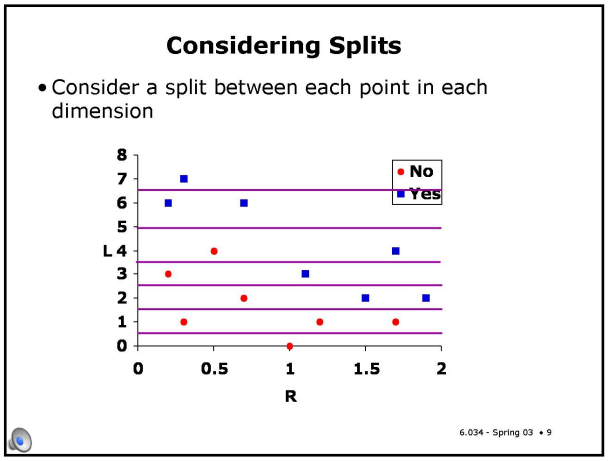
Slide 3.2.7
The only thing we really need to do differently in our algorithm is to consider splitting between each data point in each dimension.

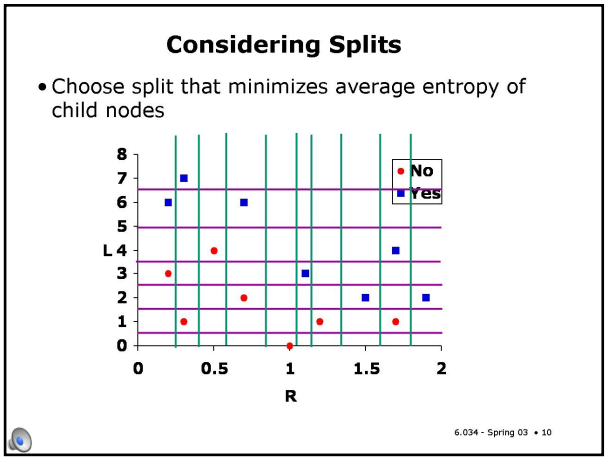




Slide 3.2.8
So, in our bankruptcy domain, we'd consider 9 different splits in the R dimension (in general, you'd expect to consider m - 1 splits, if you have m data points; but in our data set we have some examples with equal R values).

Slide 3.2.9
And there are another 6 possible splits in the L dimension (because L is an integer, really, there are lots of duplicate L values).

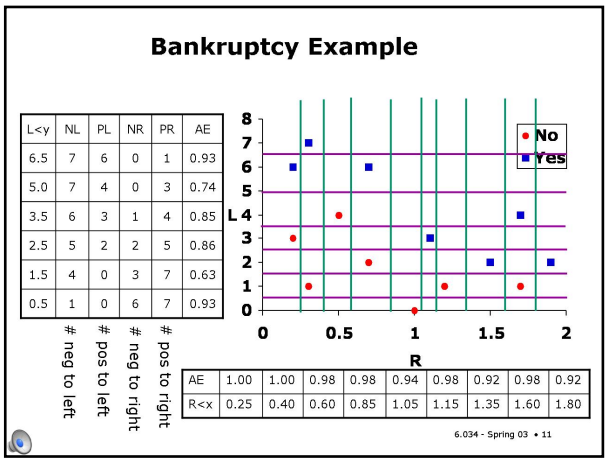


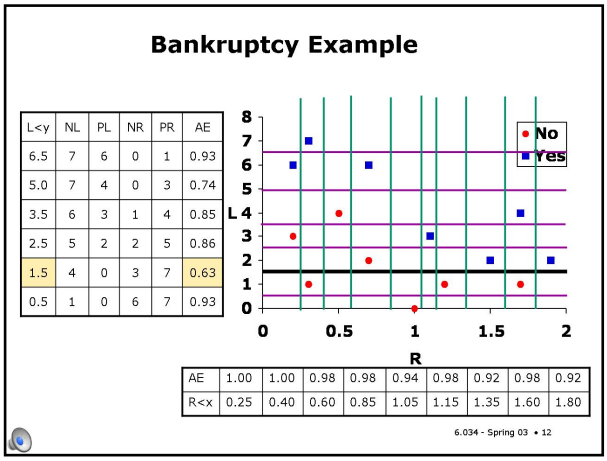


Slide 3.2.10
All together, this is a lot of possible splits! As before, when building a tree, we'll choose the split that minimizes the average entropy of the resulting child nodes.

Slide 3.2.11
Let's see what actually happens with this algorithm in our bankruptcy domain.

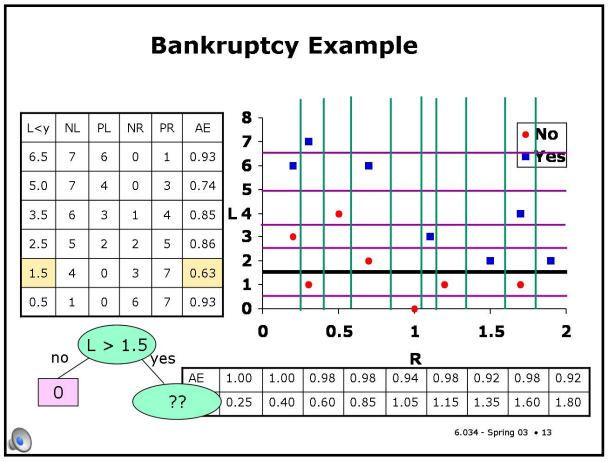
We consider all the possible splits in each dimension, and compute their average entropies.

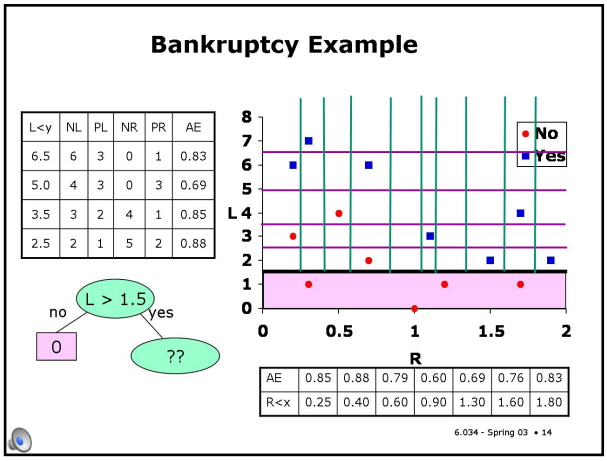




Slide 3.2.12
Splitting in the L dimension at 1.5 will do the best job of reducing entropy, so we pick that split.

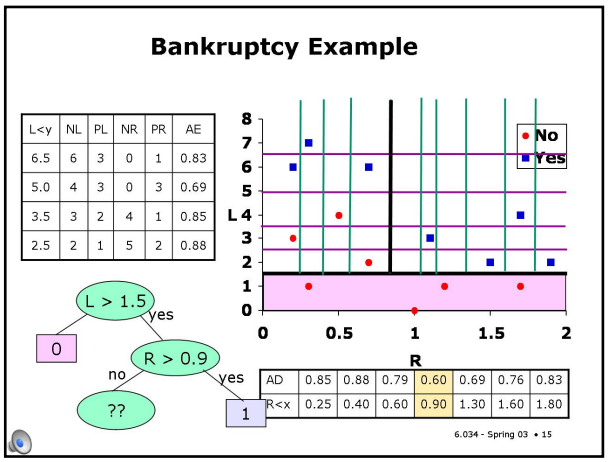
Slide 3.2.13
And we see that, conveniently, all the points with L not greater than 1.5 are of class 0, so we can make a leaf there.

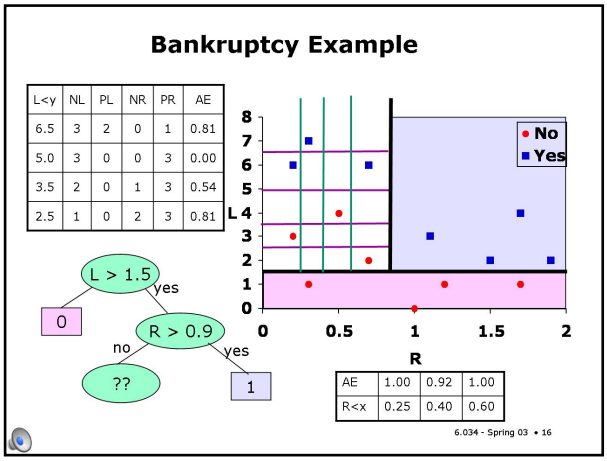




Slide 3.2.14
Now, we consider all the splits of the remaining part of the space. Note that we have to recalculate all the average entropies again, because the points that fall into the leaf node are taken out of consideration.

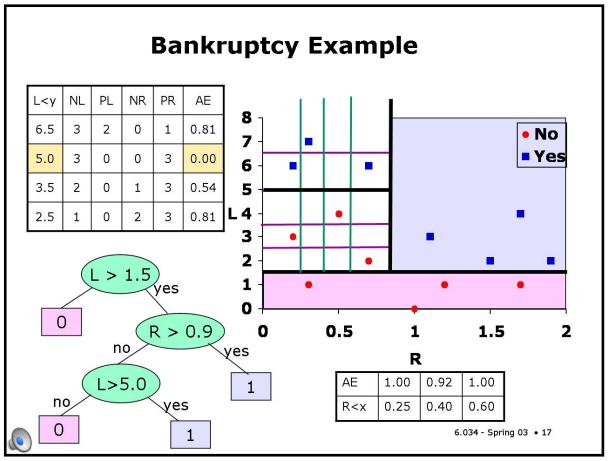
Slide 3.2.15
Now the best split is at R > 0.9. And we see that all the points for which that's true are positive, so we can make another leaf.

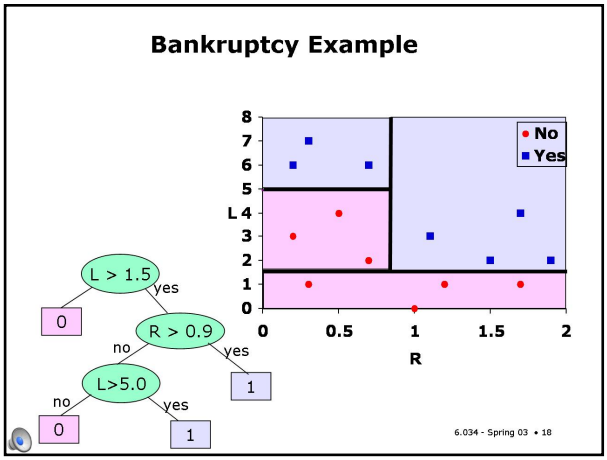




Slide 3.2.16
Again we consider all possible splits of the points that fall down the other branch of the tree.

Slide 3.2.17
And we find that splitting on L > 5.0 gives us two homogenous leaves.



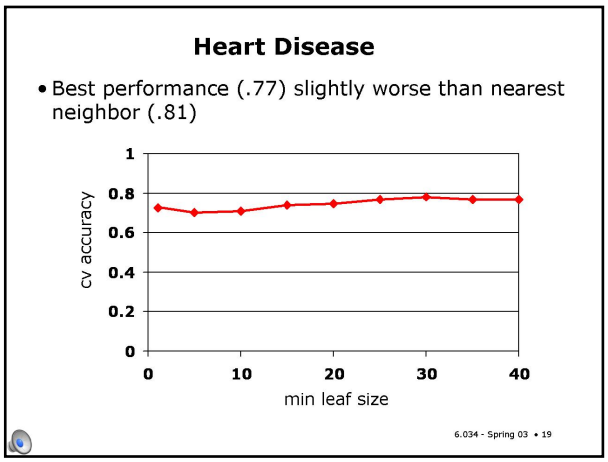


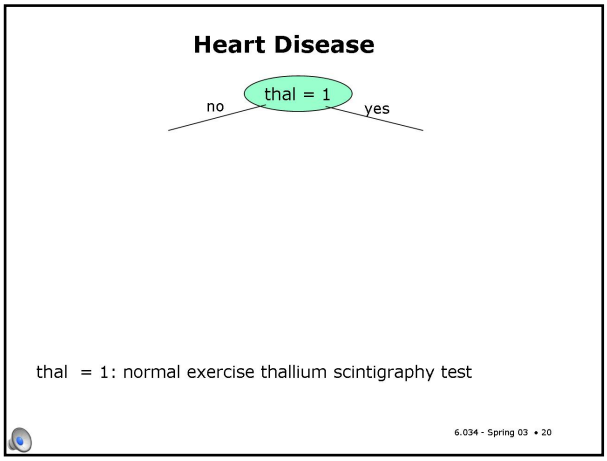
Slide 3.2.18
So, we finish with this tree, which happens to have zero error on our data set.

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 3.2.19
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.

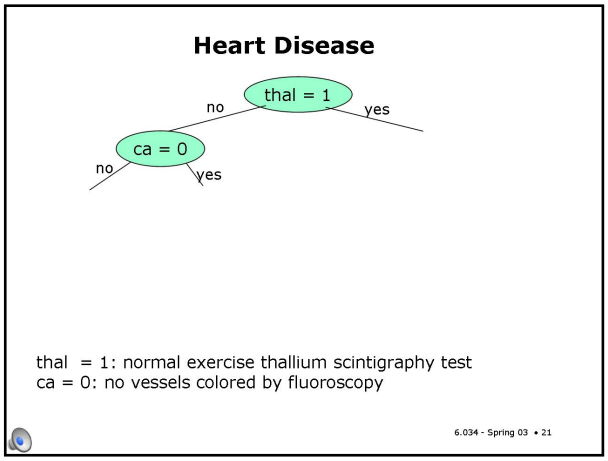


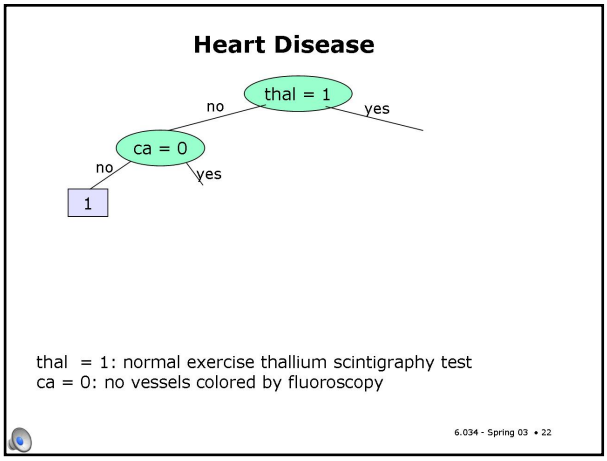


Slide 3.2.20
But performance isn't everything. One of the nice things about the decision tree algorithm is that we can interpret the hypothesis we get out. Here is an example decision tree resulting from the learning algorithm.

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.

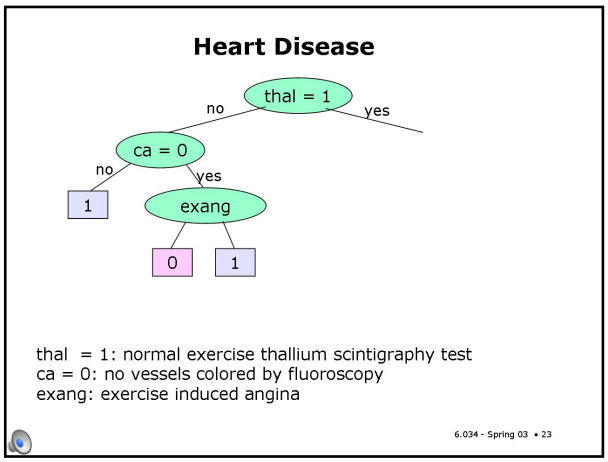
Slide 3.2.21
If thal is not normal, then we look at the results of the "ca" test. This test has as results numbers 0 through 3, indicating how many blood vessels were shown to be blocked in a different test. We chose to code this feature with 4 binary attributes.

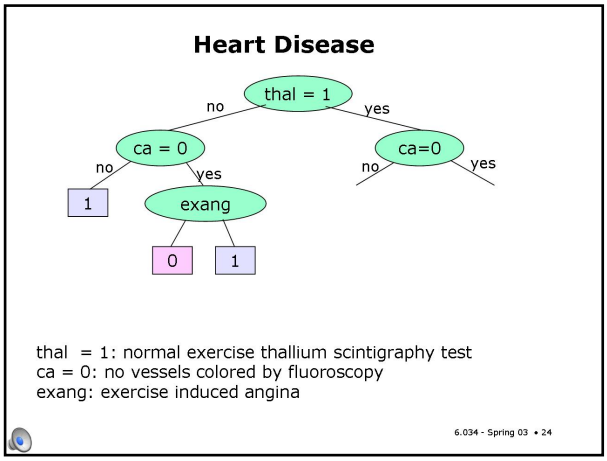




Slide 3.2.22
So "ca = 0" is false if 1 or more blood vessels appeared to be blocked. If that's the case, we assert that the patient has heart disease.

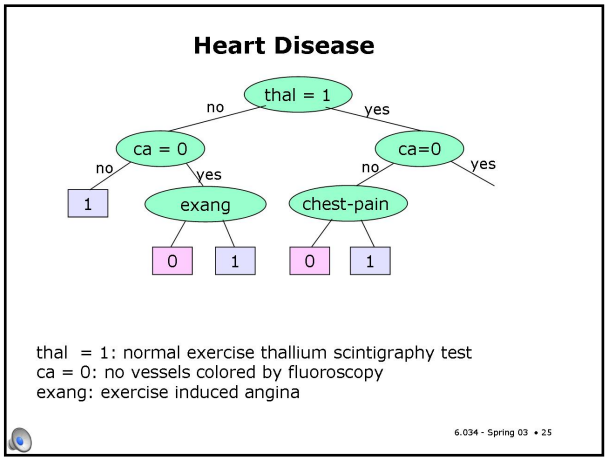
Slide 3.2.23
Now, if no blood vessels appeared to be blocked, we ask whether the patient is having exercise-induced angina (chest pain) or not. If not, we say they don't have heart disease; if so, we say they do.

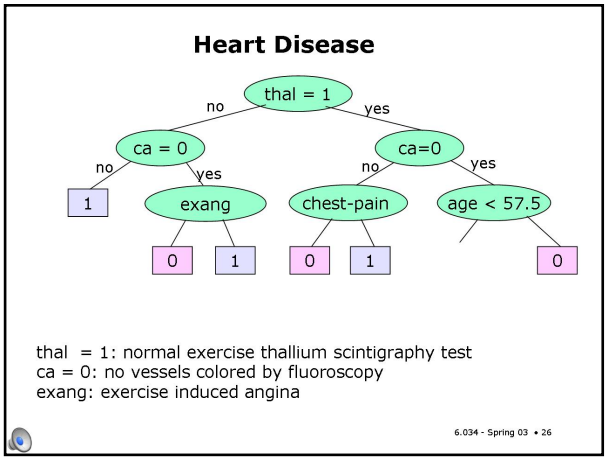




Slide 3.2.24
Now, over on the other side of the tree, where the first test was normal, we also look at the results of the ca test.

Slide 3.2.25
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.

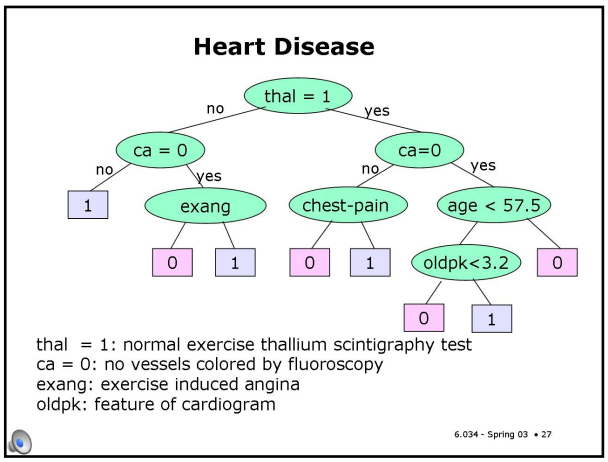


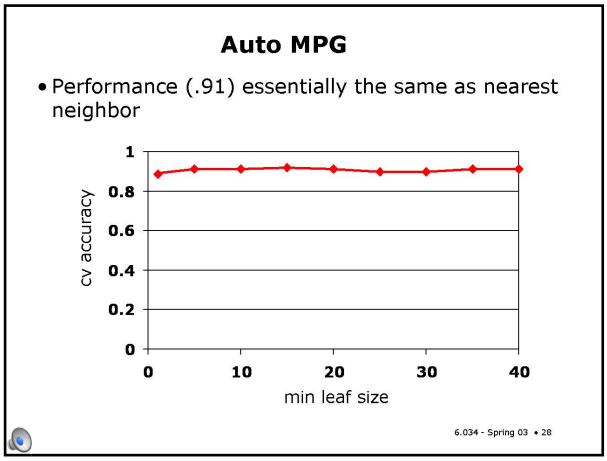


Slide 3.2.26
If no blood vessels appear to be blocked, we consider the person's age. If they're less than 57.5, then we declare them to be heart-disease free. Whew!

Slide 3.2.27
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.

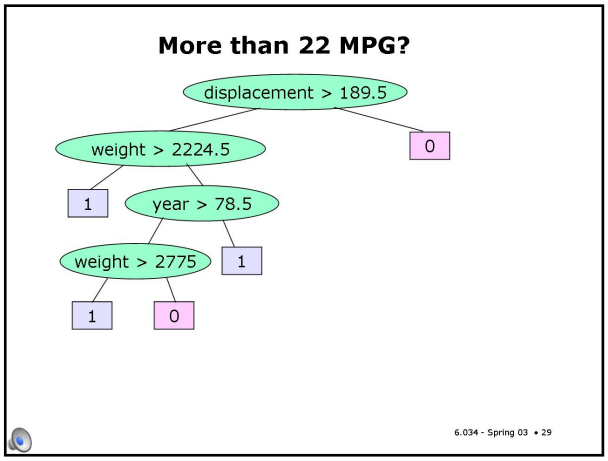




Slide 3.2.28
We also ran the decision-tree algorithm on the Auto MPG data. We got essentially the same performance as nearest neighbor, and a strong insensitivity to leaf size.

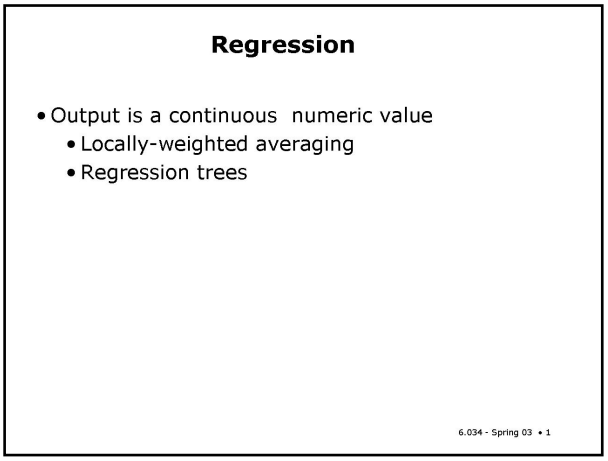
Slide 3.2.29
Here's a sample resulting decision tree. It seems pretty reasonable. If the engine is big, then we're unlikely to have good gas mileage. Otherwise, if the weight is low, then we probably do have good gas mileage. For a low-displacement, heavy car, we consider the model-year. If it's newer 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.

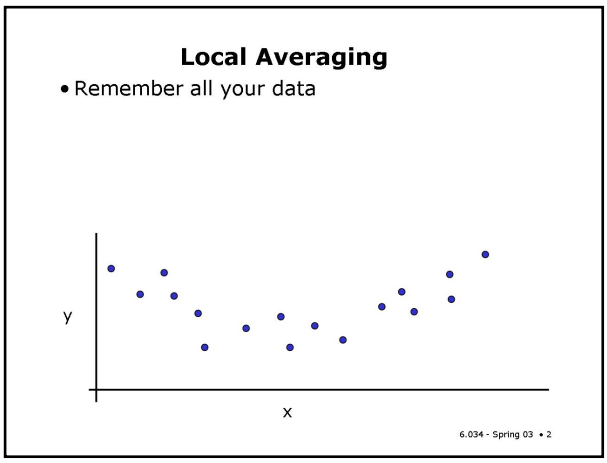
It's also possible to apply naive bayes to problems with numeric attributes, but it's hard to justify without recourse to probability, so we'll skip it. %To do: %- add a slide showing how one non-isothetic split would do the job, %but it requires a lot of rectangles.



6.034 Notes: Section 3.3

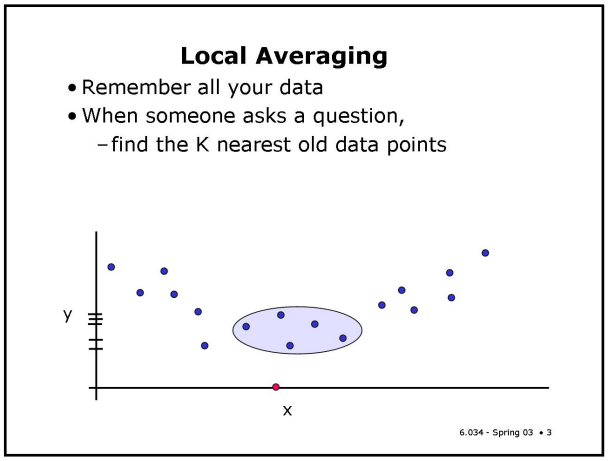
Slide 3.3.1
So far, we've spent all of our time looking at classification problems, in which the y values are either 0 or 1. Now we'll briefly consider the case where the y's are numeric values. We'll see how to extend nearest neighbor and decision trees to solve regression problems.

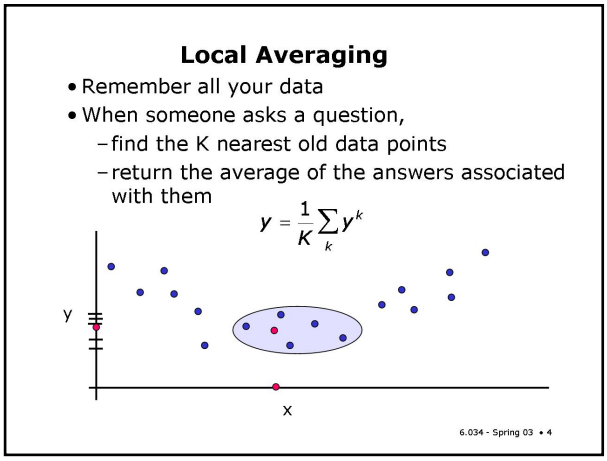




Slide 3.3.2
The simplest method for doing regression is based on nearest neighbor. As in nearest neighbor, you remember all your data.

Slide 3.3.3
When you get a new query point x, you find the k nearest points.

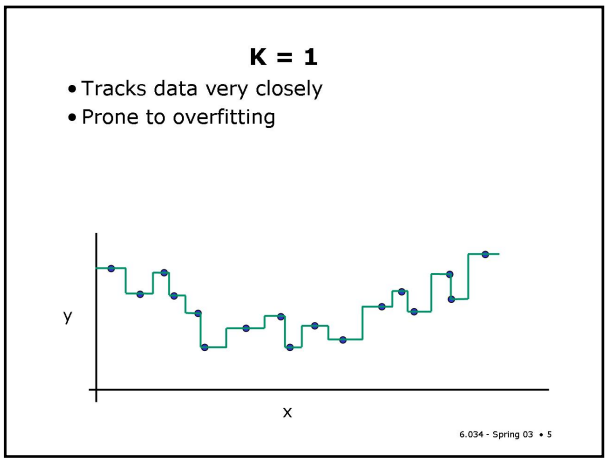


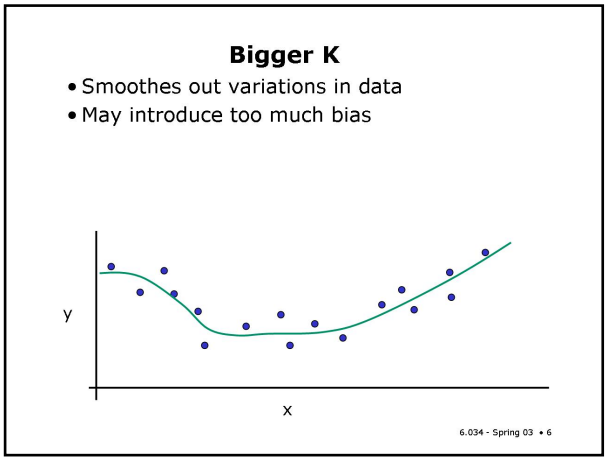


Slide 3.3.4
Then average their y values and return that as your answer.

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 3.3.5
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.





Slide 3.3.6
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 3.3.7
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 3.3.8
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 3.3.9
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"

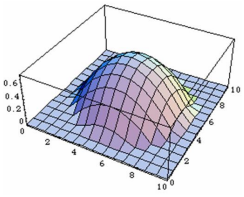
6.034 - Spring 03 • 9

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 3.3.10
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 3.3.11
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

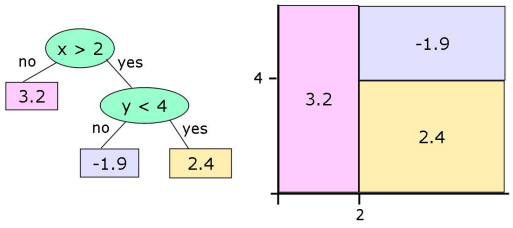
6.034 - Spring 03 • 12

Slide 3.3.12
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 3.3.13
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 3.3.14
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 3.3.15
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 3.3.16
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 3.3.17
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 :

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

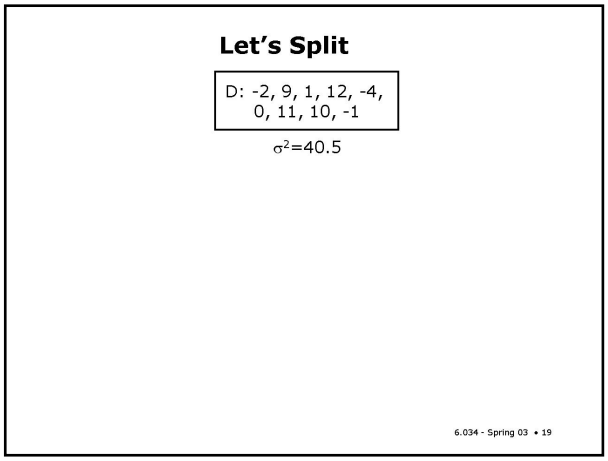
- Variance: average squared difference between z's and the mean:

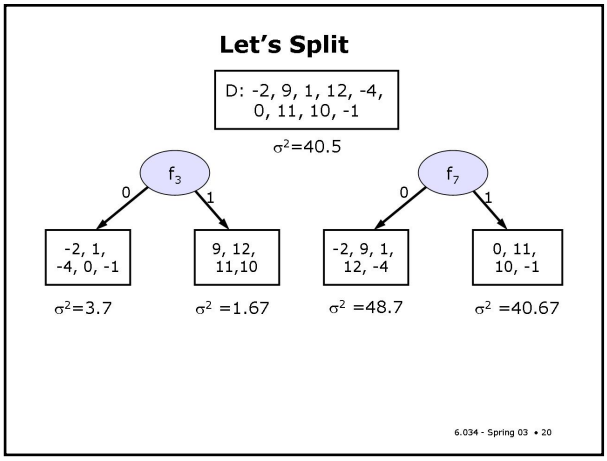
$$\sigma^2 = \frac{1}{m-1}\sum_{k=1}^{m}(z_k - \mu)^2$$

6.034 - Spring 03 • 18

Slide 3.3.18
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.

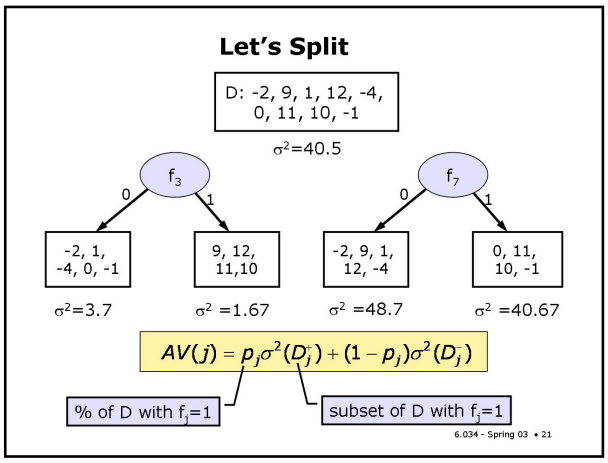
Slide 3.3.19
We're going to use the average variance of the children to evaluate the quality of splitting on a particular feature. Here we have a data set, for which I've just indicated the y values. It currently has a variance of 40.5.

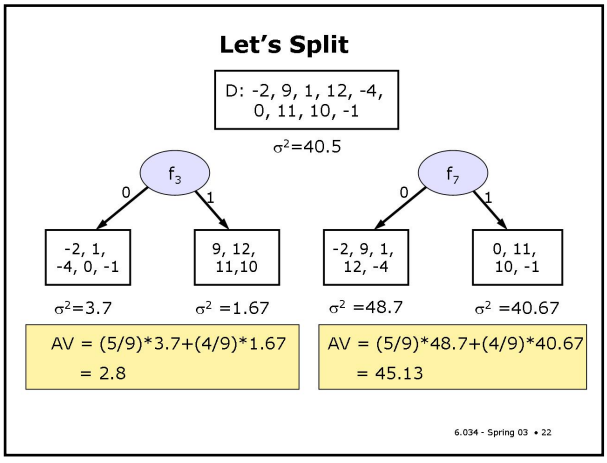




Slide 3.3.20
We're considering two splits. One gives us variances of 3.7 and 1.67; the other gives us variances of 48.7 and 40.67.

Slide 3.3.21
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.

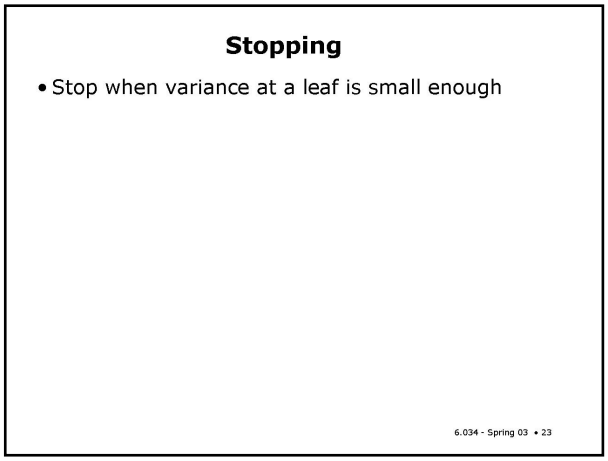


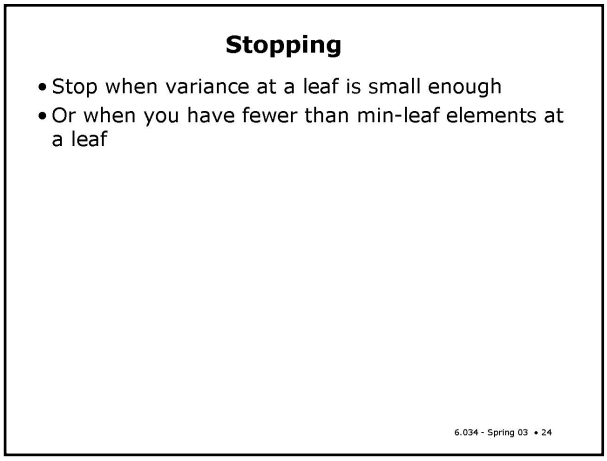


Slide 3.3.22
Doing so, we can see that the average variance of splitting on feature 3 is **much** lower than of splitting on f7, and so we'd choose to split on f3.

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

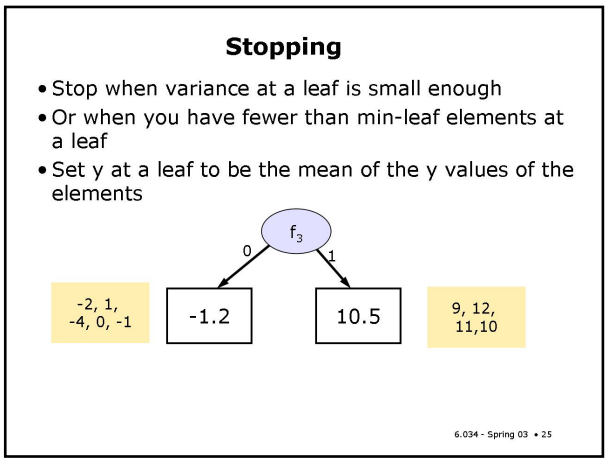
Slide 3.3.23
We can stop growing the tree based on criteria that are similar to those we used in the binary case. One reasonable criterion is to stop when the variance at a leaf is lower than some threshold.





Slide 3.3.24
Or we can use our old min-leaf-size criterion.

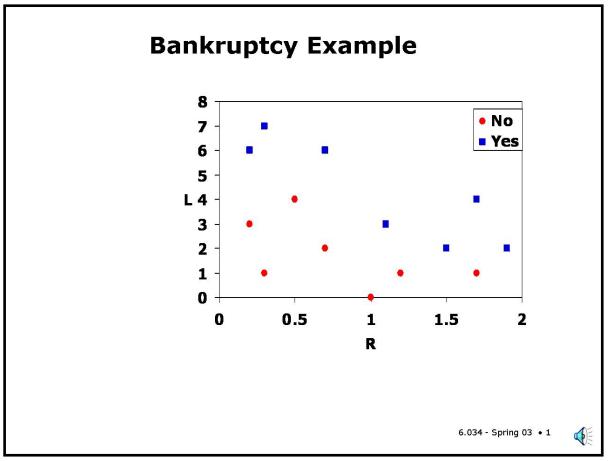
Slide 3.3.25
Once we do decide to stop, we assign each leaf the average of the values of the points in it.

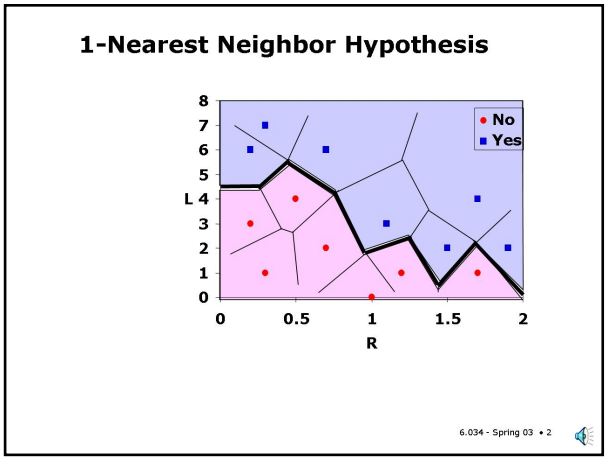


6.034 Notes: Section 3.4

Slide 3.4.1
We have been using this simulated bankruptcy data set to illustrate the different learning algorithms that operate on continuous data. Recall that R is supposed to be the ratio of earnings to expenses while L is supposed to be the number of late payments on credit cards over the past year. We will continue using it in this section where we look at a new hypothesis class, **linear separators**.

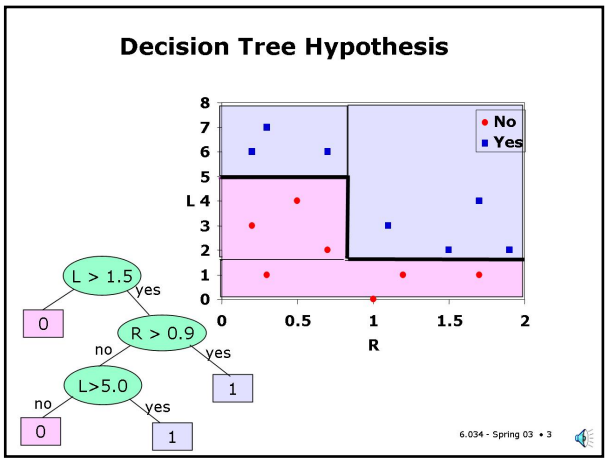
One key observation is that each hypothesis class leads to a distinctive way of defining the **decision boundary** between the two classes. The decision boundary is where the class prediction changes from one class to another. Let's look at this in more detail.

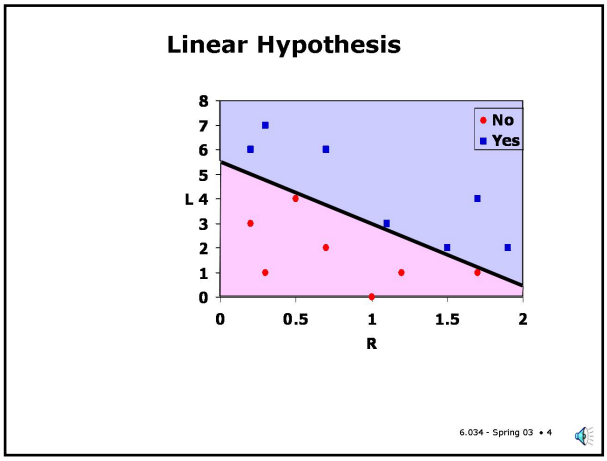




Slide 3.4.2
We mentioned that a hypothesis for the 1-nearest neighbor algorithm can be understood in terms of a Voronoi partition of the feature space. The cells illustrated in this figure represent the feature space points that are closest to one of the training points. Any query in that cell will have that training point as its nearest neighbor and the prediction will be the class of that training point. The decision boundary will be the boundary between cells defined by points of different classes, as illustrated by the bold line shown here.

Slide 3.4.3
Similarly, a decision tree also defines a decision boundary in the feature space. Note that although both 1-NN and decision trees agree on all the training points, they disagree on the precise decision boundary and so will classify some query points differently. This is the essential difference between different learning algorithms.

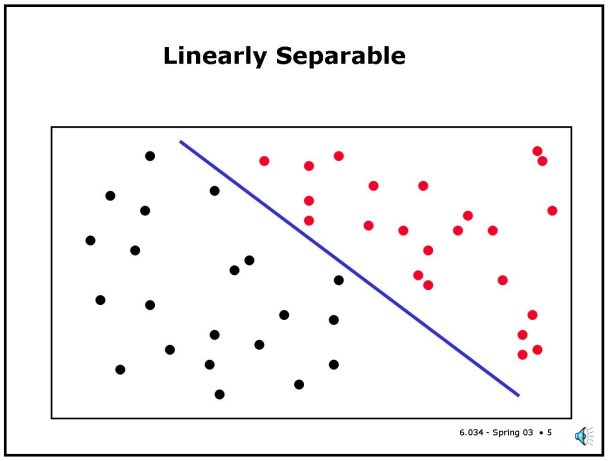


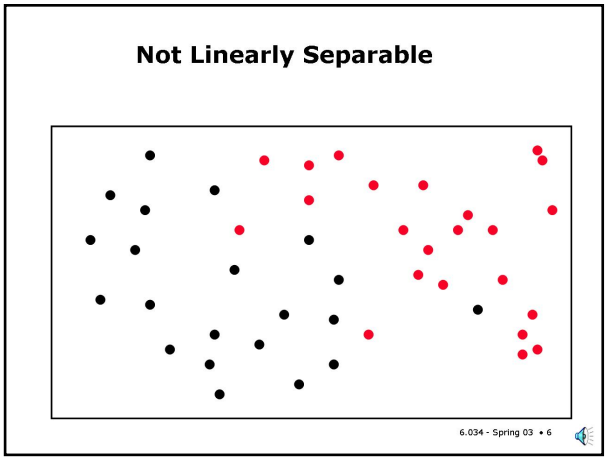


Slide 3.4.4
In this section we will be exploring **linear separators** which are characterized by a single linear decision boundary in the space. The bankruptcy data can be successfully separated in that manner. But, notice that in contrast to 1-NN and decision trees, there is no guarantee that a single linear separator will successfully classify any set of training data. The linear separator is a very simple hypothesis class, not nearly as powerful as either 1-NN or decision trees. However, as simple as this class is, in general, there will be many possible linear separators to choose from.

Also, note that, once again, this decision boundary disagrees with that drawn by the previous algorithms. So, there will be some data sets where a linear separator is ideally suited to the data. For example, it turns out that if the data points are generated by two Gaussian distributions with different means but the same standard deviation, then the linear separator is optimal.

Slide 3.4.5
A data set that can be successfully split by a linear separator is called, not surprisingly, **linearly separable**.

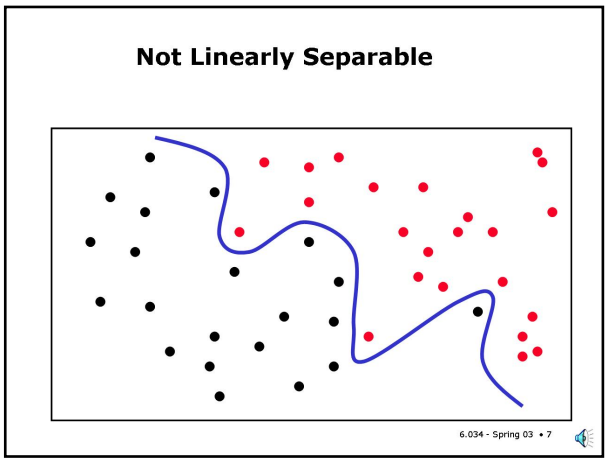


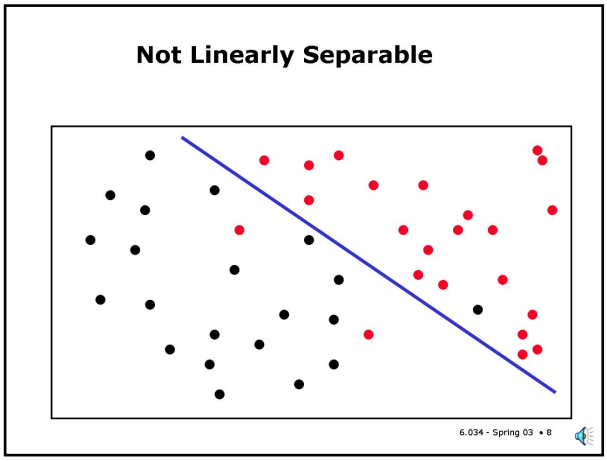


Slide 3.4.6
As we've mentioned, not all data sets are linearly separable. Here's one for example. Another classic non-linearly-separable data set is our old nemesis XOR.

It turns out, although it's not obvious, that the higher the dimensionality of the feature space, the more likely that a linear separator exists. This will turn out to be important later on, so let's just file that fact away.

Slide 3.4.7
When faced with a non-linearly-separable data set, we have two options. One is to use a more complex hypothesis class, such as shown here.



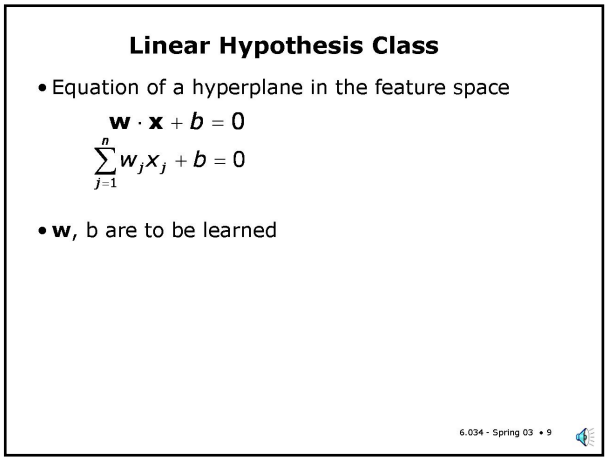


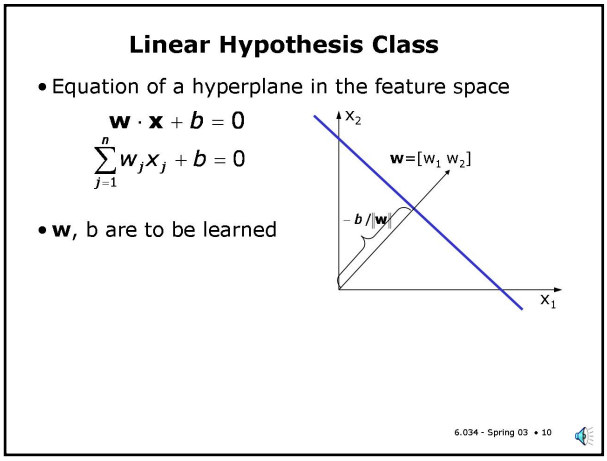
Slide 3.4.8
Or, keep the simple linear separator and accept some errors. This is the classic bias/variance tradeoff. Use a more complex hypothesis with greater variance or a simpler hypothesis with greater bias. Which is more appropriate depends on the underlying properties of the data, including the amount of noise. We can use our old friend cross-validation to make the choice if we don't have much understanding of the data.

Slide 3.4.9
So, let's look at the details of linear classifiers. First, we need to understand how to represent a particular hypothesis, that is, the equation of a linear separator. We will be illustrating everything in two dimensions but all the equations hold for an arbitrary number of dimensions.

The equation of a linear separator in an n-dimensional feature space is (surprise!) a linear equation which is determined by n+1 values, the components of an n-dimensional coefficient vector **w** and a scalar value *b*. These n+1 values are what will be learned from the data. The **x** will be some point in the feature space.

We will be using dot product notation for compactness and to highlight the geometric interpretation of this equation (more on this in a minute). Recall that the dot product is simply the sum of the componentwise products of the vector components, as shown here.

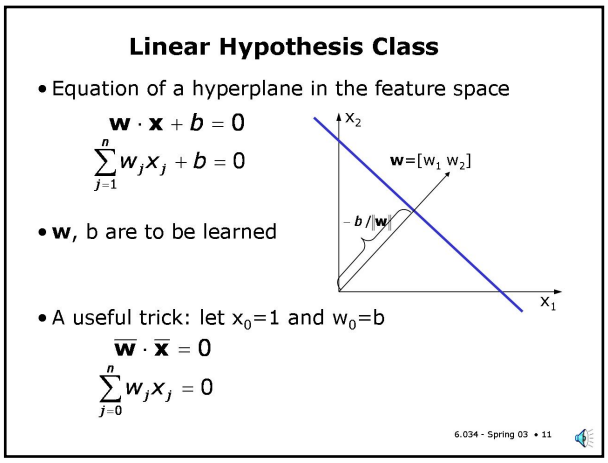


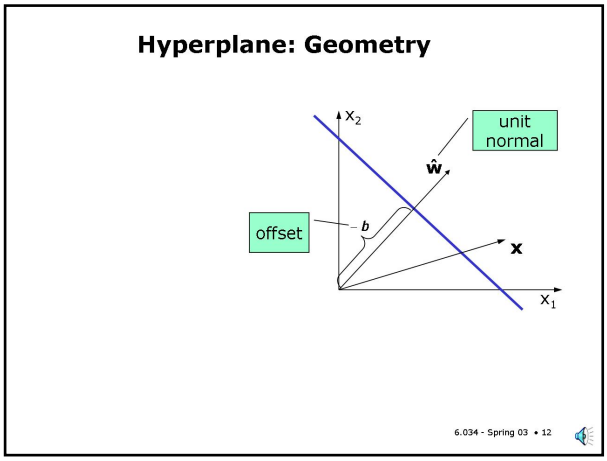


Slide 3.4.10
In two dimensions, we can see the geometric interpretation of **w** and *b*. The vector **w** is perpendicular to the linear separator; such a vector is known as the **normal** vector. Often we say "the vector normal to the surface". The scalar *b*, which we will call the **offset**, is proportional to the perpendicular distance from the origin to the linear separator. The constant of proportionality is the negative of the magnitude of the normal vector. We'll examine this in more detail soon.

By the way, the choice of the letter "w" is traditional and meant to suggest "weights", we'll see why when we look at neural nets. The choice of "b" is meant to suggest "bias" - which is the third different connotation of this word in machine learning (the bias of a hypothesis class, bias vs variance, bias of a separator). They are all fundamentally related; they all refer to a difference from a neutral value. To keep the confusion down to a dull roar, we won't call *b* a bias term but are telling you about this so you won't be surprised if you see it elsewhere.

Slide 3.4.11
Sometimes we will use the following trick to simplify the equations. We'll treat the offset as the 0^{th} component of the weight vector **w** and we'll augment the data vector **x** with a 0^{th} component that will always be equal to 1. Then we can write a linear equation as a dot product. When we do this, we will indicate it by using an overbar over the vectors.





Slide 3.4.12
First a word on terminology: the equations we will be writing apply to linear separators in *n* dimensions. In two dimensions, such a linear separator is refered to as a "line". In three dimensions, it is called a "plane". These are familiar words. What do we call it in higher dimensions? The usual terminology is **hyperplane**. I know that sounds like some type of fast aircraft, but that's the accepted name.

Let's look at the geometry of a hyperplane a bit more closely. We saw earlier that the offset *b* in the linear separator equation is proportional to the perpendicular distance from the origin to the linear separator and that the constant of proportionality is the magnitude of the **w** vector (negated). Basically, we can multiply both sides of the equation by any number without affecting the equality. So, there are an infinite set of equations all of which represent the same separator.

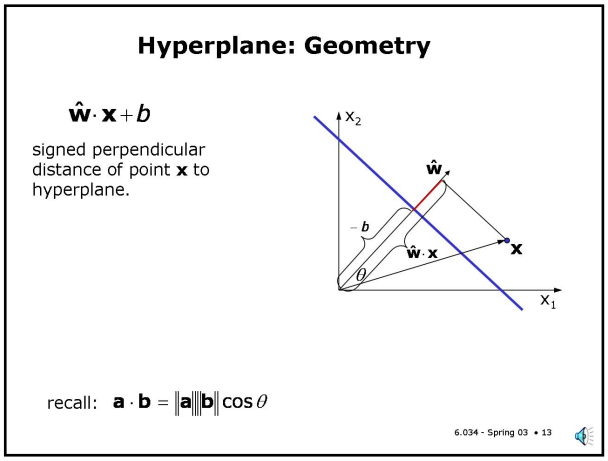
If we divide the equation through by the magnitude of **w** we end up with the situation shown in the figure. The normal vector is now unit length (denoted by the hat on the **w**) and the offset *b* is now equal to the perpendicular distance from the origin (negated).

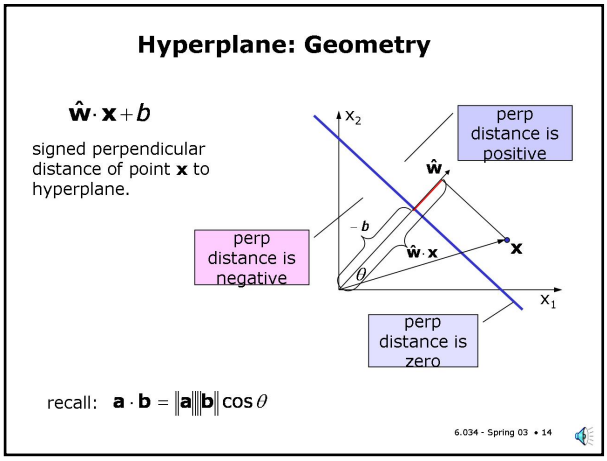
Slide 3.4.13
It's crucial to understand that the quantity **w**-hat dot **x** plus b is the perpendicular distance of point **x** to the linear separator.

If you recall, the geometric interpretation of a dot product **a . b** is that it is a number which is the magnitude of **a** times the magnitude of **b** times the cosine of the angle between the vectors. If one of the vectors, say **a**, has unit magnitude then what we have is precisely the magnitude of the projection of the **b** vector onto the direction defined by **a**. Thus **w**-hat dot **x** is the distance from **x** to the origin measured perpendicular to the hyperplane.

Looking at the right triangle defined by the **w**-hat and the **x** vector, both emanating from the origin, we see that the projection of **x** onto **w**-hat is the length of the base of the triangle, where **x** is the hypotenuse and the base angle is theta.

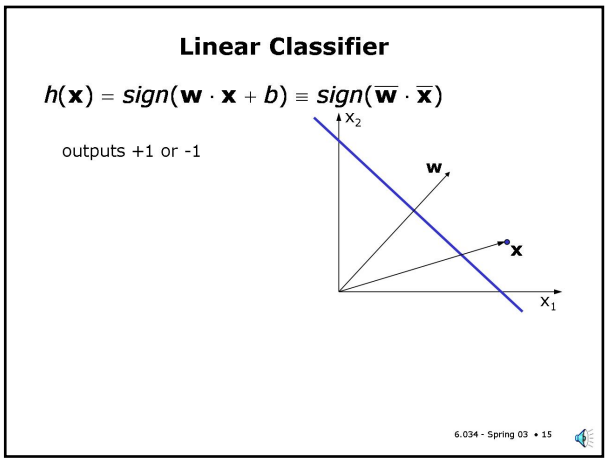
Now, if we subtract out the perpendicular distance to the origin we get the distance of **x** from the hyperplane (rather than from the origin). Note that when theta is 90 degrees (that is, **w** and **x** are perpendicular), the cosine is equal to 0 and the distance is precisely *b* as we expect.

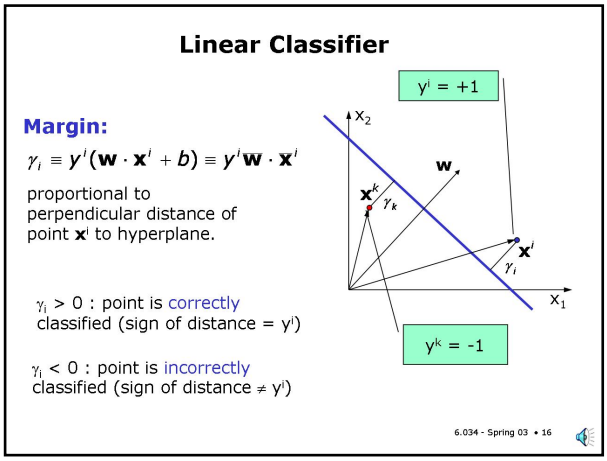




Slide 3.4.14
This distance measure from the hyperplane is **signed**. It is zero for points on the hyperplane, it is positive for points in the side of the space towards which the normal vector points, and negative for points on the other side. Notice that if you multiply the normal vector **w** and the offset *b* by -1, you get an equation for the same hyperplane but you switch which side of the hyperplane has positive distances.

Slide 3.4.15
We can now exploit the sign of this distance to define a linear classifier, one whose decision boundary is a hyperplane. Instead of using 0 and 1 as the class labels (which was an arbitrary choice anyway) we use the sign of the distance, either +1 or -1 as the labels (that is the values of the y_i).





Slide 3.4.16
A variant of the signed distance of a training point to a hyperplane is the **margin** of the point. The margin (gamma) is the product of the actual signed distance for th epoint and the desired sign of the distance, y_i. If they agree (the point is correctly classified), then the margin is positive; if they disagree (the classification is in error), then the margin is negative.

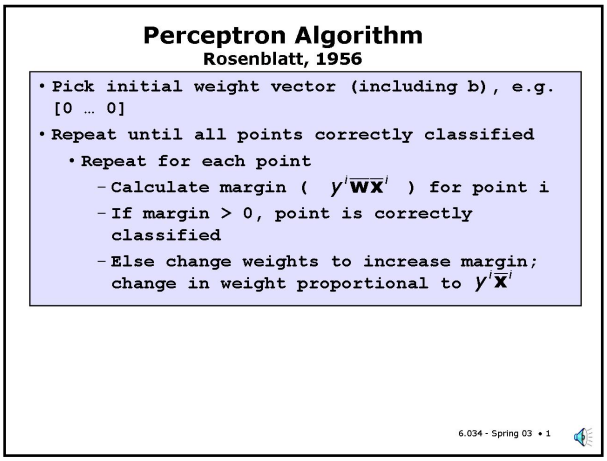
6.034 Notes: Section 3.5

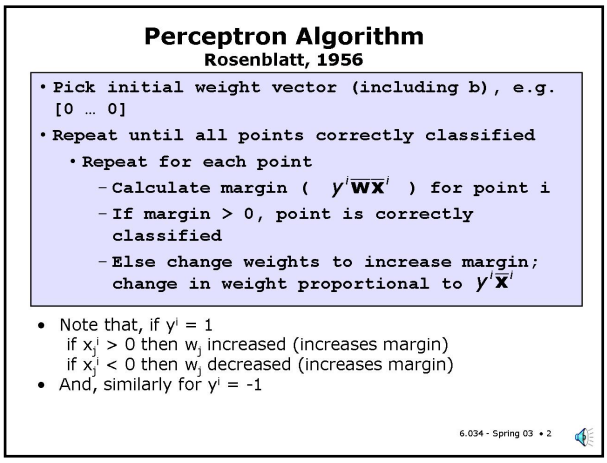
Slide 3.5.1
So far we've talked about how to represent a linear hypothesis but not how to find one. In this slide is the perceptron algorithm, developed by Rosenblatt in the mid 50's. This is not exactly the original form of the algorithm but it is equivalent and it will help us later to see it in this form.

This is a greedy, "mistake driven" algorithm not unlike the Boolean function learning algorithms we saw earlier. We will be using the extended form of the weight and data-point vectors in this algorithm. The extended weight vector is what we are trying to learn.

The first step is to start with an initial value of the weight vector, usually all zeros. Then we repeat the inner loop until all the points are correctly classified using the current weight vector. The inner loop is to consider each point. If the point's margin is positive then it is correctly classified and we do nothing. Otherwise, if it is negative or zero, we have a mistake and we want to change the weights so as to increase the margin (so that it ultimately becomes positive).

The trick is how to change the weights. It turns out that using a value proportional to $y\mathbf{x}$ is the right thing. We'll see why, formally, later. For now, let's convince ourselves that it makes sense.



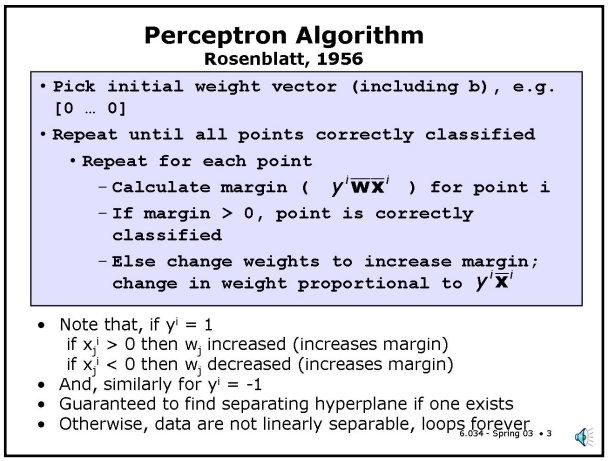


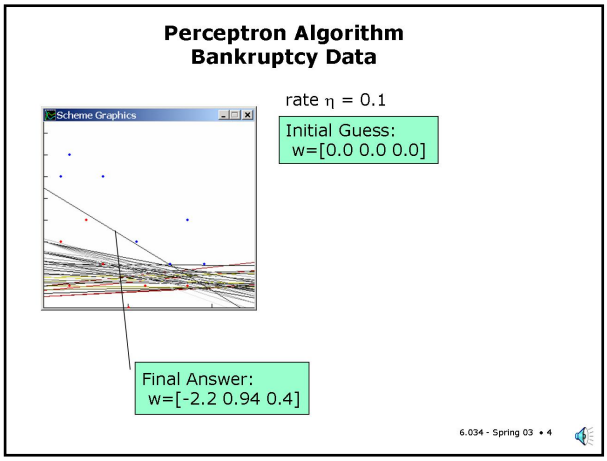
Slide 3.5.2
Consider the case in which y is positive; the negative case is analogous. If the jth component of $\mathbf{x}$ is positive then we will increase the corresponding component of $\mathbf{w}$. Note that the resulting effect on the margin is positive. If the jth component of $\mathbf{x}$ is negative then we will decrease the corresponding component of $\mathbf{w}$, and the resulting effect on the margin is also positive.

Slide 3.5.3
So, each change of $\mathbf{w}$ increases the margin on a particular point. However, the changes for the different points interfere with each other, that is, different points might change the weights in opposing directions. So, it will not be the case that one pass through the points will produce a correct weight vector. In general, we will have to go around multiple times.

The remarkable fact is that the algorithm is guaranteed to terminate with the weights for a separating hyperplane as long as the data is linearly separable. The proof of this fact is beyond our scope.

Notice that if the data is not separable, then this algorithm is an infinite loop. It turns out that it is a good idea to keep track of the best separator you've seen so far (the one that makes the fewest mistakes) and after you get tired of going around the loop, return that one. This algorithm even has a name (the **pocket** algorithm: see, it keeps the best answer in its pocket...).





Slide 3.5.4
This shows a trace of the perceptron algorithm on the bankruptcy data. Here it took 49 iterations through the data (the outer loop) for the algorithm to stop. The hypothesis at the end of each loop is shown here. Recall that the first element of the weight vector is actually the offset. So, the normal vector to the separating hyperplane is [0.94 0.4] and the offset is -2.2 (recall that is proportional to the negative perpendicular distance from origin to the line).

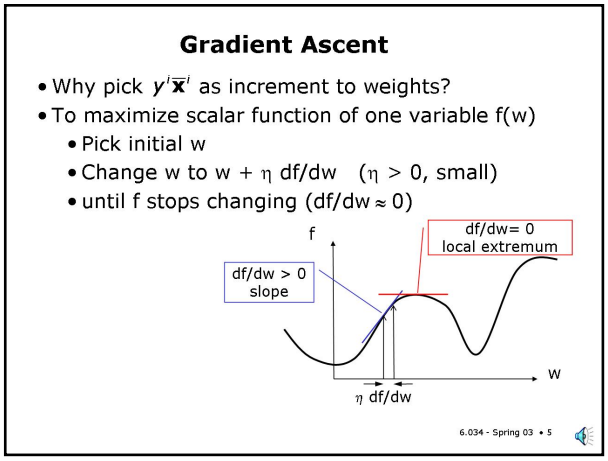
Note that the units in the horizontal and vertical directions in this graph are not equal (the tick marks along the axes indicate unit distances). We did this since the range of the data on each axis is so different.

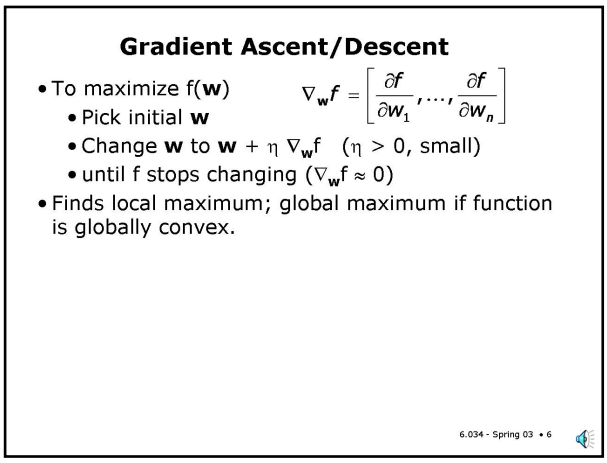
One usually picks some small "rate" constant to scale the change to **w**. It turns out that for this algorithm the value of the rate constant does not matter. We have used 0.1 in our examples, but 1 also works well.

Slide 3.5.5
Let's revisit the issue of why we picked *y***x** to increment **w** in the perceptron algorithm. It might have seemed arbitrary but it's actually an instance of a general strategy called **gradient ascent** for finding the input(s) that maximize a function's output (or gradient descent when we are minimizing).

The strategy in one input dimension is shown here. We guess an initial value of the input. We calculate the slope of the function at that input value and we take a step that is proportional to the slope. Note that the sign of the slope will tell us whether an increase of the input variable will increase or decrease the value of the output. The magnitude of the slope will tell us how fast the function is changing at that input value. The slope is basically a linear approximation of the function which is valid "near" the chosen input value. Since the approximation is only valid locally, we want to take a small step (determined by the rate constant eta) and repeat.

We want to stop when the output change is zero (or very small). This should correspond to a point where the slope is zero, which should be a local extremum of the function. This strategy will not guarantee finding the global maximal value, only a local one.

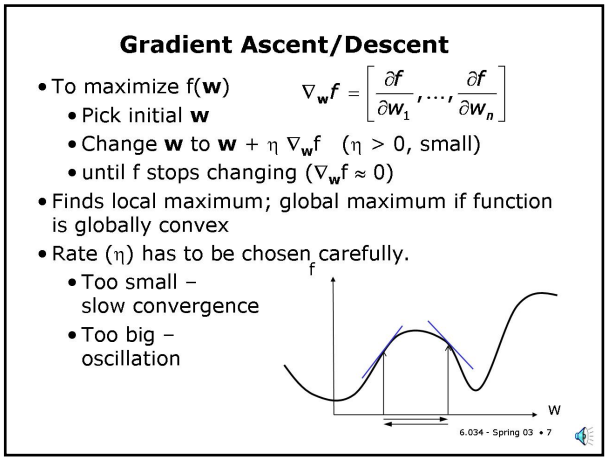


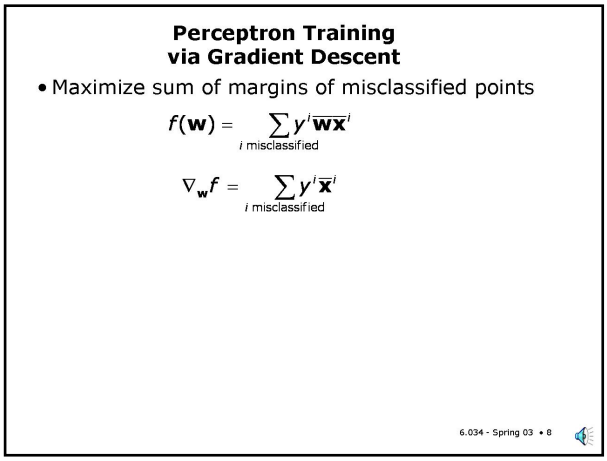


Slide 3.5.6
The generalization of this strategy to multiple input variables is based on the generalization of the notion of slope, which is the **gradient** of the function. The gradient is the vector of first (partial) derivatives of the function with respect to each of the input variables. The gradient vector points in the direction of steepest increase of the function output. So, we take a small step in that direction, recompute the gradient and repeat until the output stops changing. Once again, this will only find us a local maximum of the function, in general. However, if the function is globally convex, then it will find the global optimum.

Slide 3.5.7
In general, the choice of the rate constant (eta), which determines the step size, is fairly critical. Unfortunately, no single value is appropriate for all functions. If one chooses a very conservative small rate, it can take a long time to find a minimum, if one takes too big steps there is no guarantee that the algorithm will even converge to a minimum; it can oscillate as shown in the figure here where the sign of the slope changes and causes a back-and-forth search.

In more sophisticated search algorithms one does a search along the specified direction looking for a value of the step size that guarantees an increase in the function value.



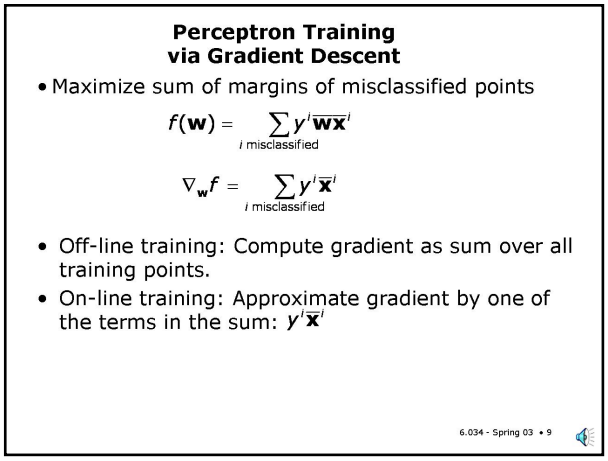


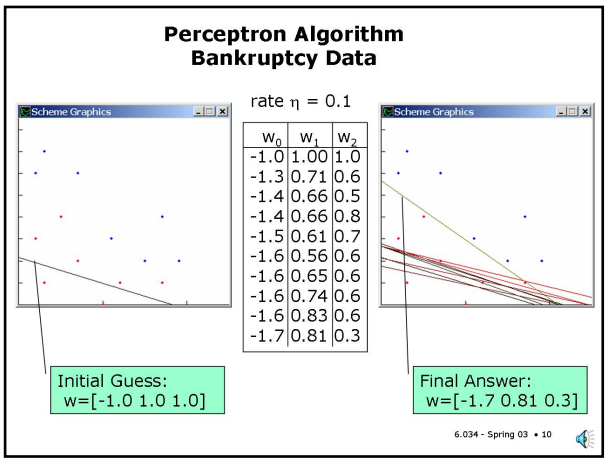
Slide 3.5.8
Now we can see that our choice of increment in the perceptron algorithm is related to the gradient of the sum of the margins for the misclassified points.

Slide 3.5.9
If we actually want to maximize this sum via gradient descent we should sum all the corrections for every misclassified point using a single **w** vector and then apply that correction to get a new weight vector. We can then repeat the process until convergence. This is normally called an **off-line** algorithm in that it assumes access to all the input points.

What we actually did was a bit different, we modified **w** based on each point as we went through the inner loop. This is called an **on-line** algorithm because, in principle, if the points were arriving over a communication link, we would make our update to the weights based on each arrival and we could discard the points after using them, counting on more arriving later.

Another way of thinking about the relationship of these algorithms is that the on-line version is using a (randomized) approximation to the gradient at each point. It is randomized in the sense that rather than taking a step based on the true gradient, we take a step based on an estimate of the gradient based on a randomly drawn example point. In fact, the on-line version is sometimes called "stochastic (randomized) gradient ascent" for this reason. In some cases, this randomness is good because it can get us out of shallow local minima.

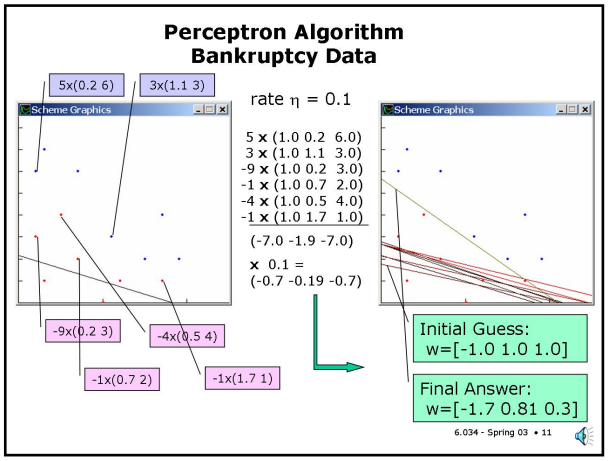


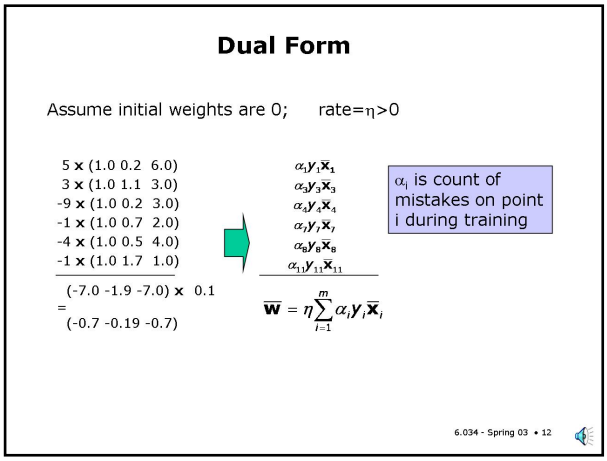


Slide 3.5.10
Here's another look at the perceptron algorithm on the bankruptcy data with a different initial starting guess of the weights. You can see the different separator hypotheses that it goes through. Note that it converges to a different set of weights from our previous example. However, recall that one can scale these weights and get the same separator. In fact these numbers are approximately 0.8 of the ones we got before, but only approximately; this is a slightly different separator.

The perceptron algorithm can be described as a gradient ascent algorithm, but its error criterion is slightly unusual in that there are many separators that all have zero error.

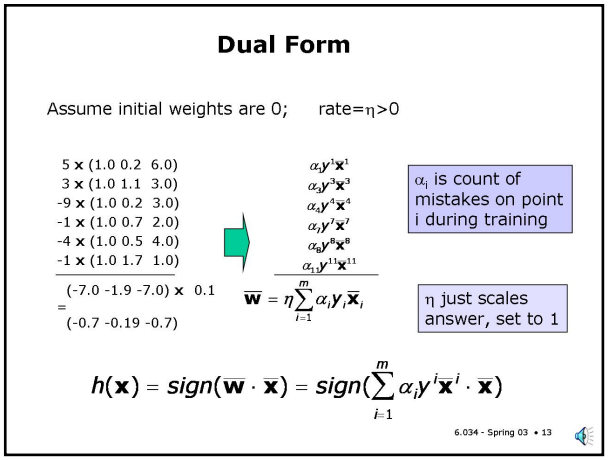
Slide 3.5.11
Recall that the perceptron algorithm starts with an initial guess for the weights and then adds in scaled versions of the misclassified training points to get the final weights. In this particular set of 10 iterations, the points indicated on the left are misclassified some number of times each. For example, the leftmost negative point is misclassified in each iteration except the last one. If we sum up the coordinates of each of these points, scaled by how many times each is misclassified and by the rate constant we get the total change in the weight vector.

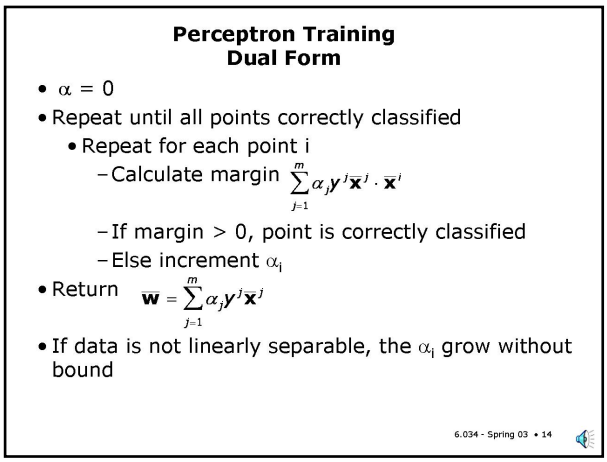




Slide 3.5.12
This analysis leads us to a somewhat different view of the perceptron algorithm, usually called the **dual form** of the algorithm. Call the count of how many times point *i* is misclassified, $alpha_i$. Then, assuming the weight vector is initalized to 0s, we can write the final weight vector in terms of these counts and the input data (as well as the rate constant).

Slide 3.5.13
Since the rate constant does not change the separator we can simply assume that it is 1 and ignore it. Now, we can substitute this form of the weights in the classifier and we get the classifier at the bottom of the slide, which has the interesting property that the data points only appear in dot-products with other data points. This will turn out to be extremely important later; file this one away.





Slide 3.5.14
We can now restate the perceptron algorithm in this interesting way. The separator is described as a weighted sum of the input points, with $alpha_i$ the weight for point *i*. Initially, set all of the alphas to zero, so the separator has all zero's as coefficients.

Then, for each point, compute its margin with respect to the current separator. If the margin is positive, the point is classified correctly, so do nothing. If the margin is negative, add that point into the weights of the separator. We can do that simply by incrementing the associated alpha.

Finally, when all of the points are classified correctly, we return the weighted sum of the inputs as the coefficients fo the separator. Note that if the data is not linearly separable, then the algorithm will loop forever, the alphas growing without bound.

You should convince yourself that this dual form is equivalent to the original. Once again, you may be wondering...so what? I'll say again; file this away. It has surprising consequences.