PROFESSOR: Last Tuesday, we ended up the lecture talking about knapsack problems. We talked about the continuous knapsack problem and the fact that you could solve that optimally with a greedy algorithm. And we looked at the 0-1 knapsack problem and discussed the fact that while we could write greedy algorithms that would solve the problem quickly, we have to be careful what we mean by "solve," and that while those algorithms would choose a set of items that we could indeed carry away, there was no guarantee that it would choose the optimal items, that is to say, one that would meet the objective function of maximizing the value.

We looked after that at a brute force algorithm on the board only for finding an optimal solution, a guaranteed optimal solution, but observed the fact that on even a moderately sized set of items, it might take a decade or so to run. Decided that wasn't very good.

Nevertheless, I want to start today looking at some code that implements a brute force algorithm, not because I expect anyone to actually run this on a real example, but because a bit later in the term, we'll see how we could modify this to something that would be practical. And there's some things to learn by looking at it.

So let's look at some code here. I don't expect you to understand in real time all the details of this code. It's more I want you to understand the basic idea behind it and then the result we get.

So you'll remember that we looked at the complexity by saying well, really, it's like binary numbers. So the first helper subroutine I'm going to use is something that generates binary numbers. So it takes an n, some natural number, and the number of digits and returns a binary string of that length, representing the decimal number n.

Why am I giving it this number of digits? Because I need to zero pad it. If I want to have a vector that represents whether or not I take items, if I take only one item, say the first one, I don't want just a binary spring with one digit in it.

Because I need all those zeros to indicate that I'm not taking the other items. And so the second argument tells me, in effect, how many zeros I'm going to need. And there's nothing mysterious about the way it does it.

OK. The next helper function generates the power set of the items. What is a power set? If you take a set, you can then ask the question what are all the subsets of the set?

What's the smallest subset of a set? It's the empty set. No items.

What's the largest subset of a set? All of the items. And then we have everything in between, the set that contains the first item, the set that contains the second item, et cetera, the set that contains the first and the second, the first and the third.

There are a lot of them. And of course, how many is a lot? Well, 2 to the n is a lot.

But now we're going to generate every possible subset of items. And we're going to do this simply using the decimal to binary function to tell us whether or not we keep each one, so we can enumerate them. We can generate them all. And now we have the set of all possible items one might take, irrespective of whether they obey the constraint of not weighing too much.

The next function is the one that does the work. This is the interesting one. Choose best. It takes a power set, the constraint, and two functions. One, getValue-- it tells me the value of an item. And the other getWeight-- it tells me the weight of an item. Then it just goes through.

And it enumerates all possibilities and eventually chooses-- I won't say the best set, because it might not be unique. There might be more than one optimal answer. But it finds at least one optimal answer. And then it returns that.

Again it's a very straightforward implementation of the brute force algorithm I sketched on the board. And then we can run it with testBest, which is going to build the items using the function we looked at last time. It's then going to get the power set of the items. It's going to call chooseBest and then print the result. So let's see what happens if we run it.

We get an error. Oh dear. I hadn't expected that. And it says test-- oh testBest is not defined. All right. Let's try that again. Sure looks like it's defined to me. There it is. OK.

And you may recall that this is a better answer than anything that was generated by the greedy algorithm on Tuesday. You may not recall it, but believe me it is. It happened to have found a better solution. And not surprisingly, that's because I contrived the example to make sure that would happen.

Why does it work better in the sense-- or why does it find a better answer? Why might it find a better answer? Well, because the greedy algorithm chose something that was locally optimal at each step. But there was no guarantee that a sequence of locally optimal decisions would reach a global optimum.

What this algorithm does is it finds a global optimum by looking at all solutions. And that's something we'll see again and again, as we go forward, that there's always a temptation to do things one step at a time, finding local optimum-- optima-- because it's fast. It's easy. But there's no guarantee it will work well.

Now the problem, of course, with finding the global optimum is, as we discussed, it is prohibitively expensive. Now you could ask is it prohibitively expensive because I chose a stupid algorithm, the brute force algorithm? Well, it is a stupid algorithm. But in fact, this is a problem that is what we would call inherently exponential.

We've looked at this concept before. That in addition to talking about the complexity of an algorithm, we can talk about the complexity of a problem in which we ask the question how fast can the absolute best solution, fastest solution to this problem, be? And here you can construct a mathematical proof that says the problem is inherently exponential. No matter what we do, we're not going to be able to find something that's guaranteed to find the optimal, that is faster than exponential.

Well, now let's be careful of that statement. What that means is the worst case is inherently exponential. As we will see in a couple of weeks-- it'll take us a while to get there-- there are actually algorithms that people use to solve these inherently exponential problems and solve them fast enough to be useful.

So, for example, when you go to look at airline fares on Kayak to try and find the best fare from A to B, it is an inherently exponential problem, but you get an answer pretty quickly. And that's because there are techniques you can use. Now, in fact, one of the reasons you get it is they don't guarantee that you actually get an optimal solution. But there are techniques that guarantee to give you an optimal solution that almost all the time will run quickly. And we'll look at one of those a bit later in the term.

Before we do that, however, I want to leave for a while the whole question of complexity behind and look at another class of optimization problems. We'll look at several different kinds of optimization problems as the term goes forward. The kind I want to look at today is probably what I would say is the most exciting branch of computer science today. And of course I might have a bias. And that's machine learning.

It's a word you'll hear a lot about. And it's a technique that many of you will apply. You might not write your own codes. But I guarantee you were either be the beneficiary or the victim of machine learning almost every time you log on to the web these days.

I should probably start by defining what machine learning is. But that's hard to do. I really don't know how to do it. Superficially, you could say that machine learning deals with the question of how to build programs that learn. However, I think in a very real sense every program we write learns something. If I implement Newton's method, it's learning what the roots of the polynomial is. Certainly when we looked at curve fitting-- fitting curves to data-- we were learning a model of the data. That's what that regression is.

Wikipedia says-- and of course, it must be true if Wikipedia says it-- that machine learning is a scientific discipline that is concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data. I'm not sure how helpful this definition is. But it was the best I could find. And it doesn't really matter.

But it sort of gets at the issue that a major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data. This whole process is something called inductive inference. The basic idea is one observes-- actually one doesn't. The program observes examples that represent an incomplete information about some statistical phenomena and then tries to generate a model, just like with curve fitting, that summarizes some statistical properties of that data and can be used to predict the future, for example, give you information about unseen data.

There are roughly speaking two distinctive approaches to machine learning called supervised learning and unsupervised learning. Let's first talk about supervised learning. It's a little easier to appreciate how it might work.

In supervised learning, we associate a label with each example in a training set. So think of that as an answer to a query about an example. If the label is discrete, we typically call it a classification problem.

So we would try and classify, for example, a transaction on a credit card as belonging to the owner of that credit card or not belonging to the owner, as i.e., with some probability, a stolen credit card. So it's discrete. It belongs to the owner. It doesn't belong to the owner.

If the labels are real valued, we think of it as a regression problem. And so indeed, when we did the curve fitting, we were doing machine learning. And we were handling a regression problem.

Based on the examples from the training set, the goal is to build a program that can predict the answer for other cases before they were explicitly observed. So we're trying to generalize from the statistical properties of a training set to be able to make predictions about things we haven't seen.

Let's look at an example. So here, I've got red and blue circles. And I'm trying to learn what makes a circle red or what's the difference between red and blue, other than the color? Think of my information as the (x,y) values and the label as the color, red or blue. So I've labeled each one. And now I'm trying to learn something.

Well, it's kind of tricky. What are the questions I need to answer to think about this? And then we'll look at how we might do it. So, a first question I need to ask is are the labels accurate? And in fact, in a lot of real world examples, in most real world examples, there's no guarantee that the labels are accurate. So you have to assume that well, maybe some of the labels are wrong. How do we deal with that?

Perhaps the most fundamental question is the past representative of the future? We've seen many examples where people have learned things, for example, to predict the price of housing. And it turns out you hit some singularity which means the past is not a very good predictor of the future. And even if all of your learning is good, you get the wrong answer. So you sort of always have to ask that question.

Do you have enough data to generalize? And by this, I mean enough training data. If your training set is very small, you shouldn't have a lot of confidence in what you learn.

A big issue here is feature extraction. As we'll see when we look at real examples, the world is a pretty complex place. And we need to decide what features we're going to use. If I were to ask 25% of you to come up in the front of the room and then try and separate you based upon some feature-- if I were to say, all right, I'm going to separate the good students from the bad students, but the only features I have available are the clothes you're wearing, it might not work so well.

And very importantly, how tight should the fit be? So now let's go back to our example here. We can look at two different ways we might generalize from this data. And indeed, when we're looking at classification problems in supervised learning, what we're typically doing is trying to find some way of dividing our training data.

In this case, I've given you a two-dimensional projection. As we'll see, it's not always two-dimensional. It's not usually two-dimensional. So I might choose this rather eccentric shape and say that's great. And why is that great? It's great because it minimizes training error.

So if we look at it as an optimization problem, we might say that our objective function is how many points are correctly classified in the training data as red or blue. And this triangular shape has no training error. Every point is perfectly classified in the training data. If I choose this linear separator instead, I have some training error. This red point is misclassified in the training.

Does that mean that the triangle is better than the line? Not necessarily, right? Because my goal is to predict future points. And maybe that's mislabeled or an experimental error. Maybe it's accurately labeled but an outlier, very unusual. And this will not generalize well.

This is analogous to what we talked about as overfitting when we looked at curve fitting. And that's-- a very big problem in machine learning is if you overfit to your training data, it might not generalize well and might give you bogus answers going forward.

OK. So that's a very quick look at supervised learning. We'll come back to that. I now want to talk about unsupervised learning. The big difference here is we have training data, but we don't have labels. So I just give you a bunch of points. It's as if we looked at this picture, and I didn't tell you which were the red points and which were the blue points. They were just all points.

So what can I learn? What typically you're learning in unsupervised learning, is you're learning about regularities of the data. So if we looked at this and think away the red and the blue, we might well say well, at least, if I look at this, there is some structure to this data.

And maybe what I should do is divide it this way. It gives me kind of a nice clean separation. But maybe I should divide it this way. Or maybe I should put a circle around each of these four groupings.

Complicated, what to do. But what we see is there is clearly some structure here. And the idea of unsupervised learning is to discover that structure.

Far and away, the dominant form of unsupervised learning is clustering. And that's what I was just talking about, is finding the cluster in this data. So we'll move forward here. There it is, with everything the same color. But here I've labeled the x- and y-axes as height and weight.

What does clustering mean? It's the process of organizing the objects or the points into groups whose members are similar in some way. A key issue is what do we mean by similar? What's the metric we want to use?

And we can see that here. If I tell you that, really, I want to cluster people by height-- say, people are similar if they're the same height-- then it's pretty clear how I should divide this, right, what my clusters should be. My clusters should probably be this group of shorter people and this group of taller people.

If I tell you I'm interested in weight, then probably I want a cluster it with the divisor here between the heavier people and the lighter people. Or if I say well, I'm interested in some combination of those two, then maybe I'll get four clusters as I discussed before.

Clustering algorithms are used all over the place. For example, in marketing, they're used to find groups of customers with similar behavior. Walmart is famous for using that clustering to find that. They did a clustering to determine when people bought the same thing. And then they would rearrange their shelves to encourage people to buy things.

And sort of the most famous example they discovered was there was a strong correlation between people between people who bought beer and people who bought diapers. And so there was a period where if you walked in a Walmart store, you would find the beer and the diapers next to each other. And I leave it to you to speculate on why that was true. It just happened to be true in Walmart.

Amazon uses clustering to find people who like similar books. So every time you buy a book on Amazon, they're running a clustering algorithm to find out who looks like you. Said, oh, this person looks just like you. So if they buy a book, maybe you'll get an email suggesting you buy that book or the next time you log into Amazon.

Or when you look at a book, they tell you here are some similar books. And then they've done a clustering to group books as similar based on buying habits. Netflix uses that to recommend movies, et cetera.

Biologists spend a lot of time these days doing clustering. They classify plants or animals based on their features. We'll shortly see an example of that, as in right after Patriot's Day.

But they also use it a lot in genetics. So clustering is used to try and find genes that look like or groups of genes.

Insurance companies use that to decide how much to charge you for your automobile insurance. They cluster drivers based upon-- and use that to predict who's going to have an accident.

Document classification on the web is used all the time. It's used a lot in medicine. Just used all over the place.

So what is it exactly? Well the nice thing is we can define it very straightforwardly as an optimization problem. And so we can ask what properties does a good clustering have? Well, it should have low intra-cluster dissimilarity.

So in a good clustering, all of the points in the same cluster should be similar, by whatever metric you're using for similarity. As we'll see, there are a lot of choices there. But that's not enough. We'd also like to have high inter-cluster dissimilarity.

So we'd like the points within a cluster to be a lot like each other. But if points are in different clusters, we'd like them to be quite different from each other. That tells us that we have a good cluster.

All right, let's look at it. How might we model dissimilarity? Well, using a concept we've already seen-- variance. So we can talk about the variance of some cluster C as equal to the sum of all elements x in C, of the mean of C minus x squared. Or maybe we can take the square root of it, if we want.

But it's exactly the idea we've seen before, right? Then we say what's the average value of the cluster? And then we look at how far is each point from the average. We sum them. And that tells us how much variance we have within the cluster. Make sense?

So that's variance. So we can use that to talk about how similar or dissimilar the elements in the cluster are. We can use the same idea to compare points in separate clusters and compute various different ways-- and we'll look at different ways-- to look at the distance between clusters.

So combining these two things, we could get, say, a metric we'll call badness-- not a technical word. And now I'll ask the question is the optimization problem that we're solving in clustering finding a set of clusters-- capital C-- such that badness of that set of clusters is minimized?

Is that a sufficient definition of the problem we're trying to solve? Find a set of clusters C, such that the badness of C is minimized. is that good enough?

AUDIENCE: No.

PROFESSOR: No, why not?

AUDIENCE: Just imagine a case where you view cluster-- if you make a single cluster, every cluster has one element in it. And the variance is 0.

PROFESSOR: Exactly. So that has a trivial solution, which is probably not the one we want, of putting each point in its own cluster. Badness-- it won't be bad. It'll be a perfect clustering in some sense. But it doesn't do us any good really.

So what do we do to fix that? What do we usually do when we formulate an optimization problem? What's missing? I've given you the objective function. What have I not giving you?

AUDIENCE: Constraints.

PROFESSOR: A constraint. So we need to add some constraint here that will prevent us from finding a trivial solution. So what kind of constraints might we look at? There are different ways of doing it. A couple of ones that is usual. Sometimes you might have as a constraint, the maximum number of clusters.

Say, all right, cluster my data, but I want at most K clusters -- 10 clusters. That would be my constraint, like the weight for the knapsack problem. Or maybe I'll want to put something on the maximum distance between clusters. So I don't want the distance between any two clusters to be more than something.

In general, solving this optimization problem is computationally prohibitive. So once again, in practice, what people typically resort to is greedy algorithms. And I want to look at two kinds of greedy algorithms, probably the two most common approaches to clustering.

One is called k-means. In k-means clustering, you say I want exactly k clusters. And find the best k clustering. We'll talk about how it does that. And again, it's not guaranteed to find the best.

And the other is hierarchical clustering. We'll come back to that shortly. Both are simple to understand and widely used in practice.

So let's first talk about how we do this. Let's first look at hierarchical clustering. So we have a set of n items to be clustered. And let's assume we have an n by n distance matrix that tells me for each pair of items how far they are from each other.

So we can look at an example. So here's an n by n distance matrix for the airline distance between some cities in the United States. The distance from Boston to Boston is 0 miles. Distance from New York is 206. The distance from Chicago to San Francisco is 2,142, et cetera. All right? So I have my n by n distance matrix there.

Now let's go through how hierarchical clustering would relate these things to each other. So we start by assigning each item to its own cluster. So if we have n items, we now have n clusters. All right? That's the trivial solution that you suggested before.

The next step is to find the most similar pair of clusters and merge them. So if we look here and we just-- we start, we'll have six clusters, one for each city. And we would merge the two most similar, which I guess in this case is New York and Boston. Hard to believe that those are the most similar cities. But at least by this distance metric they're the closest. So we would merge those two.

And then you just continue the process in principle, until all items are in one cluster. So now you have a whole hierarchy of clusters. And you can cut it off where you want.

If you want to have six clusters, you could look at where the hierarchy you have six. You can look where you have two, where you have three. Of course, you don't have to go all the way to finish it if you don't want to.

This kind of hierarchical clustering is called agglomerative. Why? Well, because we're combining things. We're agglomerating them.

So this is pretty straightforward, except for two. The complication in step (2) is we have to define what it means to find the two most similar clusters. Now it's pretty easy when the clusters each contain one element, because, well, we have our metric-- in this case, distance-- and we can just do that as I did.

But it's not so obvious what you do when they have multiple elements. And in fact, different metrics can be used to get different properties. So I want to talk about some of the metrics we use for that.

These are typically called linkage criteria. So one popular one is what's called single linkage. It's also called connectedness, or minimum method.

In this, we consider the distance between a pair of clusters to be equal to the shortest distance from any member to any other member. So we take the two points in each cluster that are closest to each other and say that's the distance between the two clusters.

People also use something called complete linkage-- It's also called diameter or maximum-- where we consider the distance between any two clusters to be the distance between the points that are furthest from each other. So in one case, essentially single linkages was looking at the best case. Complete-- in English, not French-- is looking at the worst case.

And you won't be surprised to hear that you could also look at the average case, where you take all of the distances. So you take all of the pairwise things. You add them up. You take the average. You can also take the mean, the median, if you want instead.

None of these is necessarily best. But they do give you different answers. And so I want to look at that now with our example here. So let's look at it and run it.

So the first step is independent of what linkage we're using. We get these six clusters. All right, now let's look at the second step.

Well, also pretty simple since we only have one element in each one. We're going to get that clustering. All right. Now, what about the next step? What do I get if I'm using the minimal single linkage distance? What gets merged here? Somebody?

AUDIENCE: Boston, New York and Chicago.

PROFESSOR: Boston, New York, and Chicago. And it turns out we'll get the same thing if we use other linkages in this case. Let's continue to the next step. Now we'll end up merging San Francisco and Seattle. Now we get a difference.

What does the red represent and what does the blue represent? Which linkage criteria? We're saying, we could either merge Denver with Boston, New York, and Chicago. Or we could merge Denver with San Francisco and Seattle.

Which is which? Which linkage criterion has put Denver in which cluster? Well, suppose we're using single linkage. Where are we getting it from?

AUDIENCE: Boston, New York and Chicago?

PROFESSOR: Yes. Because it's not so far from Chicago. Even though it's pretty far from Boston or New York. But if we use average linkage, we see on average, it's closer to San Francisco and Seattle than it is to the average of Boston, New York, or Chicago. So we get a different answer. And then finally, at the last step, everything gets merged together.

So you can see, in this case, without having labels, we have used a feature to produce things and, say, if we wanted to have three clusters, we would maybe stop here. And we'd say, all right, these things are one cluster. This is a cluster. And this is a cluster. And that's not a bad geographical clustering, actually, for deciding how to relate these things to each other.

This technique is used a lot. It does have some weaknesses. One weakness is it's very time consuming. It doesn't scale well. The complexity is at least order n-squared, where n is the number of points to be clustered. And in fact, in many implementations, it's worse than n-squared.

And of course, it doesn't necessarily find the optimal clustering, even giving these criteria. It might never at any level have the optimal clustering, because, again, at each step, it's making a locally optimal decision, not guaranteed to find the best solution.

I should point out that a big issue in deciding to get these clusters or getting these clusters was my choice of features. And this is something we're going to come back to in spades, because I actually think it is the most important issue in machine learning-- is if we're going to say which points are similar to each other, we need to understand our feature space.

So, for example, the feature I'm using here is distance by air. Suppose, instead, I added distance by air and distance by road and distance by train. Well, particularly given this sparsity of railroads in this country, we might get very different clustering, depending upon where the trains ran.

And suppose I throw in a totally different feature like population. Well, I might get another different clustering, depending on how I use that.

What we typically need to do in these situations, dealing with multi-dimensional data-- and most data is multi-dimensional-- is we construct something called a feature vector that incorporates multiple features. So we might have for each city-- we'll just take something like the distance. Or let's say, instead of distance, we'll compute the distance by having for each city its GPS coordinates, where it is on the globe, and its population.

And let's say that's how we define a city. And that would be our feature vector. And then we would cluster it, say, using hierarchical clustering to determine which cities are most like which other cities.

Well, it's a little bit complicated. I have to ask how do I compare feature vectors? What distance metric do I use there? Do I get confused that GPS coordinates and populations are essentially unrelated? And I wouldn't like to compare those to each other.

Lots of issues there, and that's what we're going to talk about when we come back from Patriot's Day-- is how in the real world problems, we go from the large number of features associated with objects or things in the real world to feature vectors that allow us to automatically deduce which things are quote "most similar" to which other things.