PROFESSOR: Good morning, everybody. All right, I ended up the last lecture talking about how to calculate the absolute goodness of fit using something called the coefficient of determination. It's usually spelled as R-squared, or R2. And the formula was quite simple.

We measure the goodness of the fit as R-squared equals 1 minus the estimated error divided by the measured variance. As I observed, R-squared always lies between 0 and 1. If R-squared equals 1, that means that the model that we constructed, the predicted values if you will, explains all of the variability in the data so that any change in the data is explained perfectly by the model. We don't usually get 1. In fact, if I ever got 1, I would think somebody cheated me.

R-squared equals 0, or conversely, it means there's no linear relationship at all between the values predicted by the model and the actual data. That is to say, the model is totally worthless. So we have code here, the top of the screen, showing how easy it is to compute R-squared. And for those of you who have a little trouble interpreting the formula, because maybe you're not quite sure what EE and MV mean, this will give you a very straightforward way to understand it.

So now, we can run it. We can get some answers. So if we look at it, you'll remember last time, we looked at two different fits. We looked at a quadratic fit and a linear fit for the trajectory of an arrow fired from my bow. And we can now compare the two.

And not surprisingly, given what we know about the physics of projectiles, we see it is exactly what we'd expect, that the linear fit has an R-quared of 0.0177, showing that, in fact, it explains almost none of the data. Whereas the quadratic fit has a

really astonishingly good R-squared of 0.98, saying that almost all of the changes in the values of the variables, that is to say the way the y value changes with respect to the x value, is explained by the model. i.e., we have a really good model of the physical situation. Very comforting.

Essentially, it's telling us at less than 2% of the variation is explained by the linear model, 98% by the quadratic model. Presumably the other 2% is experimental error. Well, now that we know that we have a really good model of the data, we can ask the question, why do we care?

We have the data itself. What's the point of building a model of the data? And that, of course, is what we're getting when we run polyfit to get this curve. The whole purpose of creating a model, or an important purpose of creating a model, is to be able to answer questions about the actual physical situation.

So one of the questions one might ask, for example, about firing an arrow is, how fast is it going? That's kind of a useful thing to know if you're worried about whether it will penetrate a target and kill somebody on the other side, for example. We can't answer that question directly looking at the data points. You look at the data. Well, I don't know.

But we can use the model to answer the question. And that's an exercise I want to go through now to show you the interplay between models, and theory, and computation, and how we can use the three to answer relevant questions about data. No, I do not want to check for new software, thank you. In fact, let's make sure it won't do that anymore.

So let's look at the PowerPoint. So here, we'll see how I'm using a little bit of theory, not very much, to be able to understand how to use the model to compute the speed of the arrow. So what we see is we know by our model, and by the good fit, that the trajectory is given by y equals ax-squared plus bx plus c. We know that.

We also know from looking at this equation that the highest point, which I'll call yPeak, of the arrow must occur at xMid, the middle of the x-axis. So if we look at a

parabola, and it doesn't matter what the parabola is, we always know that the vertical peak is halfway along the x-axis. The math tells us that from the equation.

So we can say yPeak is x times xMid squared plus b times xMid plus c. So now, we have a model that we can tell how high the arrow can get. The next question I'll ask is if I fired the arrow from here, and it hits the target here-- I've exaggerated by it this way. It's nowhere near this steep. How long does it take to get from here to here?

We don't have anything about time in our data. Yet, I claim we have enough information to go from the distance here and the distance here to how long it's going to take the arrow to get from here to the target. Why do I know that? What determines how long it's going to take to get from here to here? It's going to be how long it takes it to fall that far. It's going to be gravity.

Because we know that gravity, at least on this planet, is a constant or close enough to it. Unless maybe the arrow were going a million miles. And it's going to be gravity that tells me how long it takes to get from here to here. And when it gets to the bottom, it's going to be here,

So again, I can use some very simple math and say that the time will be the square root of 2 times the yPeak divided by the gravitational constant. Because I know that however long it takes to get from this height to this height is going to be the same time it takes to get from this point to this point. And that will therefore let me compute the average speed from here to here. And once I know that, I'm done.

Now again, this is assuming no drag and things like that. The thing that we always have to understand about a model is no model is actually ever correct. On the other hand, many models are very useful and they're close enough to correct. So I left out things like gravity, wind shear and stuff like that. But in fact, the answer we get here will turn out to be very close to correct.

We can now go back and look at some code. Get rid of this. And so now, you'll see this on the handout. I'm going to just write a little bit of code that just goes through

the math I just showed you to compute the average x velocity. Got a print statement here I use to debug it. And I'm going to return it. And then, we'll just be able to run it and see what we get.

Well, all right, that we looked at before. I forgot to close the previous figure. So now, I'm sure this is a problem you've all seen. And we'll fix it the way we always fix things, just start over.

I'll bet you guys have also seen this happen. What this is suggesting, as we've seen before, is that the process, the old process, still exists. Not a good thing. Again, I'm sure you've all seen these. Let's make sure we don't have anything running here that looks like IDLE. We don't. Just takes it a little time. There it is, all right.

All right, now we'll go back. All of this happened because I forgot to close the figure and executed pyLab.show twice, which we know can lead to bad things. So let's get rid of this. Now, we'll run it.

And now, we have our figure just using the quadratic fit. And we see that the speed is 136.25 feet per second. Do I believe 136.25? Not really. I know it's the ballpark. I confused precision with accuracy here by giving you it to two decimal places. I can compute it as precisely as I want. But that doesn't mean it's actually accurate. Probably, I should have just said it's about 135 or something like that. But it's pretty good. And for those of who don't know how to do this arithmetic in your head like me, this is about 93 miles per hour. And for comparison, the speed of sound, instead of 136 feet per second, is 1,100 feet per second. So it's traveling pretty fast.

Well, what's the point of this? I don't really care if you know how fast an arrow travels. I don't expect you'll ever need to compute that. But I wanted to show you this as an example of a pattern that we use a lot. So what we did is we started with an experiment. You didn't see this, but I actually stood in my backyard and shot a bunch of arrows and measured them, got real data out of that. And this gave me some data about the behavior of a physical system.

That's what I get for wearing a tie. Maybe it'll be quieter if I put it in my shirt.

Actually, it looks silly. Excuse me. I hope none of you will mind if I take my tie off? It seems to be making noises in the microphone. Maybe we should write a computation. All right, so ends my experiment with trying to look dignified. Not something I'm good at.

OK, had an experiment. That gave us some data. We then use computation to both find and very importantly evaluate a model. It's no good just to find the model. You need to do some evaluation to convince yourself that it's a good model of the actual physical system. And then, finally, we use some theory and analysis and computation to derive the consequence of the model.

And then, since we believe the accuracy of the model, we assume this consequence was also a true fact about the physical system we started with. This is a pattern that we see over and over again these days in all branches of science and engineering. And it's just the kind of thing that you should get used to doing. It is what you will do if you go onto a career in science or engineering.

OK, that's all I want to say now about the topic of data and experiments and analysis. We will return to this topic of interpretation of data later in the semester near the end when we start talking about machine learning and clustering. But for now, I want to pull back and start down a new track that will, I'm sure you'll be pleased to hear, dovetail nicely with the next few problem sets that you're going to have to work on.

What I want to talk about is the topic of optimization. Not so much optimization in the sense of how do you make a program fast, though we will talk a little about that, but what people refer to as optimization problems. How do we write programs to find optimal solutions to problems that occur in real life?

Every optimization problem we'll look at is going to have two parts. There's going to be (1) an objective function that will either be maximized or minimized. So for example, I might want to find the minimal air fare between Boston and Istanbul. Or more likely, the minimum bus fare between Boston and New York. So there's an objective function. Sometimes, you find the least. Sometimes, you find the most.

Maybe I want to maximize my income.

And (2) a set of constraints that have to be satisfied. So maybe I want to find the minimum transportation, minimum cost transportation between Boston and New York subject to the constraint that it not take more than eight hours or some such thing. So the objective function that you're minimizing or maximizing, and some set of constraints that must be obeyed.

A vast number of problems of practical importance can be formulated this way. Once we've formulated in this systematic way, we can then think about how to attack them with a computation that will help us solve the problem. You guys do this all the time.

I heard a talk yesterday by Jeremy Wertheimer, an MIT graduate, who founded a company called ITA. If you ever use Kayak, for example, or many of these systems to find an airline fare, they use some of Jeremy's code and algorithms to solve these various optimization problems like this. If you've ever used Google or Bing, they solve optimization problems to decide what pages to show you. They're all over the place.

There are a lot of classic optimization problems that people have worked on for decades. What we often do when confronted with a new problem, and it's something you'll get some experience on in problem sets, is take a seemingly new problem and map it onto a classic problem, and then use one of the classic solutions. So we'll go through this section of the course. And we'll look at a number of classic optimization problems. And then, you can think about how you would map other problems onto those.

This is the process known as problem reduction, where we take a problem and map it onto an existing problem that we already know how to solve. I'm not going to go through a list of classic optimization problems right now. But we'll see a bunch of them as we go forward.

Now, an important thing to think about when we think about optimization problems is

how long, how hard, they are to solve. So far, we have looked at problems that, for the most part, have pretty fast solutions, often sub-linear, binary search, sometimes linear, and at worst case, low-order polynomials.

Optimization problems, as we'll see, are typically much worse than that. In fact, what we'll see is there is often no computationally efficient way to solve them. And so we end up dealing with approximate solutions to them, or what people might call best effort solutions. And we see that as an increasing trend in tackling problems. All right, enough of this abstract stuff. Let's look at an example.

So one of the classic optimization problems is called the knapsack problem. People know what a knapsack is? Sort of an archaic term. Today, people would use the word backpack. But in the old days, they called them knapsacks when they started looking at these things. And the problem is also discussed in the context of a burglar or various kinds of thieves.

So it's not easy being a burglar, by the way. I don't know if any of you ever tried it. You've got some of the obvious problems, like making sure the house is empty and picking locks, circumventing alarms, et cetera. But one of the really hard problems a burglar has to deal with is deciding what to steal. Because you break into the typical luxury home-- and why would you break into a poor person's house if you were a burglar-- there's usually far more to steal than you can carry away.

And so the problem is formulated in terms of the burglar having a backpack. They can put a certain amount of stuff in it. And they have to maximize the value of what they steal subject to the constraint of how much weight they can actually carry. So it's a classic optimization problem. And people have worked for years at how to solve it, not so much because they want to be burglars. But as you'll see, these kinds of optimization problems are actually quite common. So let's look at an example.

You break into the house. And among other things, you have a choice of what to steal. You have a rather strange looking clock, some artwork, a book, a Velvet Elvis in case you lean in that direction, all sorts of things. And for some reason, the owner

was nice enough to leave you information about how much everything cost and how much it weighed. So you find this piece of paper. And now, you're trying to decide what to steal based upon this in a way to maximize your value.

How do we go about doing it? Oh, I should show you, by the way. There's a picture of a typical knapsack. All right, it's almost Easter, after all. Well, the simplest solution is probably a greedy algorithm. And we'll talk a lot about greedy algorithms because they are very popular and often the right way to tackle a hard problem.

So the notion of a greedy algorithm is it's iterative. And at each step, you pick the locally optimal solution. So you make the best choice, put that item in the knapsack. Ask if you have room, if you're out of weight. If not, you make the best choice of the remaining ones. Ask the same question. You do that until you can't fit anything else in.

Now of course, to do that, that assumes that we know at each stage what we mean by locally optimal. And of course, we have choices here. We're trying to figure out, in some sense, what greedy algorithm, what approach to being greedy, will give us the best result.

So one could, for example, say, all right. At each step, I'll choose the most valuable item and put that in my knapsack. And I'll do that till I run out of valuable items. Or, you could, at each step, say, well, what I'm really going to choose is the one that weights the least. That will give me the most items. And maybe that will give me the most total value when I'm done. Or maybe, at each step, you could say, well, let me choose the one that has the best value to weight ratio and put that in. And maybe that will give me the best solution.

As we will see, in this case, none of those is guaranteed to give you the best solution all the time. In fact, as we'll see, none of them is guaranteed to be better than any of the others all the time. And that's one of the issues with greedy algorithms.

I should point out, by the way, that this version of the knapsack problem that we're

talking about is typically called the 0/1 knapsack problem. And that's because we either have to take the entire item or none of the item. We're not allowed to cut the Velvet Elvis in half and take half of it.

This is in contrast to the continuous knapsack problem. If you imagine you break into the house and you see a barrel of gold dust, and a barrel of silver dust, and a barrel of raisins, what you would do is you would fill your knapsack with as much gold as you could carry, or until you ran out of gold. And then, you would fill it with as much silver as you could carry. And then, if there's any room left, you'd put in the raisins.

For the continuous knapsack problem, a greedy algorithm provides an optimal solution. Unfortunately, most of the problems we actually encounter in life, as we'll see, are 0/1 knapsack problems. You either take something or you don't. And that's more complicated.

All right, let's look at some code. So I'm going to formulate it. I'm first going to start by putting in a class, just so the rest of my code is simpler. This is something we've been talking about, that increasingly people want to start by putting in some useful data abstractions. So I've got a class item where I can put in the item. I can get its name. I get its value. I can get its weight. And I can print it. Kind of a boring class, but useful to have.

Then, I'm going to use this class to build items. And in this case, I'm going to build the items based upon what we just looked at, the table that-- I think it's in your hand out. And it's also on this slide. Later, if we want, we can have a randomized program to build up a much bigger choice of items. But here, we'll just try the clock, the painting, the radio, the vase, the book, and the computer.

Now comes the interesting part. I've written a function, greedy, that takes three arguments-- the set of items that I have to choose from, makes sense, the maximum weight the burglar can carry. And there's something called key function, which is defining essentially what I mean by locally optimal.

Then, it's quite simple. I'm going to sort the items using the key function. Remember, sort has this optional argument that says, what's the ordering? So maybe I'll order it by value. Maybe I'll order it by density. Maybe I'll order it by weight. I'm going to reverse it, because I want the most valuable first, not the least valuable, for example.

And then, I'm going to just take the first thing on my list until I run out of weight, and then I'm done. And I'll return the result and the total value. To make life simple, I'm going to define some functions. These are the functions that I can use for the ordering. Value, which is just return the value of the item. The inverse of the weight, because I'm thinking, as a greedy algorithm, I'll take the lightest, not the heaviest. And since I'm reversing it, I want to do the inverse. And the density, which is just the value divided by the weight. OK, make sense to everybody? You with me? Speak now, or not.

And then, we'll test it. So again, kind of a theme of this part of the course. As we write these more complex programs, we tend to have to worry about our test harnesses. So I've got a function that tests the greedy algorithm, and then another function that tests all three greedy approaches-- the one algorithm with different functions-- and looks at what our results are. So let's run it.

See what we get. Oh, you know what I did? Just the same thing I did last time. But this time, I'm going to be smarter. We're going to get rid of this figure and comment out the code that generated it. And now, we'll test the greedy algorithms.

So we see the items we had to choose from, which I printed using the string function and items. And if I use greedy by value to fill a knapsack of size 20, we see that I end up getting just the computer if I do greedy by value. This is for the nerd burglar.

If I use weight, I get a different-- I get more things, not surprisingly -- but lower value. And if I use density, I also get four things, but four different things, and I get a higher value. So I see that I can run these greedy algorithms. I can get an answer. But it's not always the same answer.

As I said earlier, greedy by density happens to work best here. But you shouldn't assume that will always be the case. I'm sure you can all imagine a different assignment of weights and values that would make greedy by density give you a bad answer.

All right, before we talk about how good these answers are-- and we will come back to that as, in particular, suppose I want the best answer-- I want to stop for a minute and talk about the algorithmic efficiency of the greedy algorithm. So let's go back and look at the code. And this is why people use greedy algorithms. Actually, there are two reasons. One reason is that they're easy to program, and that's always a good thing. And the other is that they are typically highly efficient.

So what's the efficiency of this? How would we think about the efficiency of this greedy algorithm? What are we looking at here?

Well, the first thing we have to ask is, what's the first thing it does? It sorts the list, right? So one thing that governs the efficiency might be the amount of time it takes to sort the list of items. Well, we know how long that takes. Or we can speculate at least. And let's assume it does something like merge sort. So what's that term going to be? Order what? Len of items times what? Log n, right?

So maybe that's going to tell us the complexity, but maybe not. The next thing we have to do is look at the while loop and see how many times are we going through the while loop. What's the worst case? Somebody? I know I didn't bring any candy today, but you could answer the question anyway. Be a sport. Do it for free. Yeah?

AUDIENCE: The length of the items.

PROFESSOR: Well, we know this one is bigger. So it looks like that's the complexity, right? So we can say, all right, pretty good. Slightly worse than linear in the length of the items, but not bad at all. And that's a big attraction of greedy algorithms. They are typically order length of items, or order length of items times the log of the length. So greedy algorithms are usually very close to linear. And that's why we really like them.

Why we don't like them is it may be that the accumulation of a sequence of locally

optimal solutions does not yield a globally optimal solution. So now, let's ask the question, suppose that's not good enough. I have a very demanding thief. Or maybe the thief works for a very demanding person and needs to choose the absolute optimal set.

Let's think first about how we formulate that carefully. And then, what the complexity of solving it would be. And then, algorithms that might be useful.

Again, the important step here, I think, is not the solution to the problem, but the process used to formulate the problem. Often, it is the case that once one has done a careful formulation of a problem, it becomes obvious how to solve it, at least in a brute force way. So now, let's look at a formalization of the 0/1 knapsack problem. And it's a kind of formalization we'll use for a lot of problems.

So step one, we'll represent each item by a pair. Because in fact, in deciding whether or not to take an item, we don't care what its name. We don't care if it's a clock, or a radio, or whatever. What matters is what's its value and what's its weight. We'll write W as the maximum weight that the thief can carry, or that can fit in the knapsack. So far, so good. Nothing complicated there.

Now comes the interesting step. We're going to represent the set of available items as a vector. We'll call it I. And then we'll have another vector, V, which indicates whether or not each item in I has been taken. So V is a vector. And if V_i is equal to 1, that implies I_i-- big I sub little i-- has been taken, is in the knapsack. Conversely, if V_i is 0, it means I_i is not in the knapsack.

So having formulated the situation thusly, we can now go back to our notion of an optimization problem as an objective function and a set of constraints to carefully state the problem. So for the objective function, we want to maximize the sum of V_i times I_i dot value, where i ranges over the length of the vectors.

So that's the trick of the 0/1. If I don't take it, it's 0. So it's 0 times the value. If I take it, it's 1 times the value. So this is going to give me the sum of the values of the items I've taken. And then, I have this subject to the constraint. And again, we'll do a

summation. And it will look very similar. V_i times I_i, but this time, dot weight is less than or equal to W.

Straightforward, but a useful kind of skill. And people do spend a lot of time on doing that. If you've ever used MATLAB, you know it wants everything to be a vector. And that's often because a lot of these problems can be nicely formulated in this kind of way. All right, now let's return to the question of complexity.

What happens if we implement this in the most straightforward way? What would the most straightforward implementation look like? Well, we could enumerate all possibilities and then choose the best that meets the constraint. So this would be the obvious brute force solution to the optimization problem. Look at all possible solutions. Choose the best one.

I think you can see immediately that this is guaranteed to give you the optimal solution. Actually, an optimal solution. Maybe there's more than one best. But in that case, you can just choose whichever one you like or whichever comes first, for example. The question is, how long will this take to run?

Well, we can think about that by asking the question, how big will this set be? How many possibilities are there? Well, we can think about that in a pretty straightforward way because if we look at our formulation, we can ask ourselves, how many possible vectors are there? How many vector V's could there be which shows which items were taken and which weren't? And what's the answer to that?

Well, if we have n items, how long will V be? Length n, right? 0, 1 for each. If we have a vector of 0's and 1''s of length n, how many different values can that vector take on? We asked this question before. What's the answer? Somebody shout it out. I've got a vector of length n. Every value in the vector is either a 0 or a 1. So maybe it looks something like this. How many possible combinations of (0, 1)'s are there?

AUDIENCE: 2 to the n?

PROFESSOR: 2 to the n. Because essentially, this is a binary number, exactly. And so if I had an

n-bit binary number, I can represent 2 to the n different values. And so we see that we have 2 to the n possible combinations to look at if we use a brute force solution.

How bad is this? Well, if the number of items is small, it's not so bad. And you'll see that, in fact, I can run this on the example we've looked. 2 to the 5 is not a huge number. Suppose I have a different number.

Suppose I have 50 items to choose from. Not a big problem. I heard yesterday that the number of different airfares between two cities in the US is order of 500 -- 500 different airfares between, say, Boston and Chicago. So looking at the best there might be 2 to the 500, kind of a bigger number. Let's look at 2 to the 50.

Let's say there were 50 items to choose from in this question. And let's say for the sake of argument, it takes a microsecond, one millionth of a second, to generate a solution. How long will it take to solve this problem in a brute force way for 50 items? Who thinks you can do it in under four seconds? How about under four minutes? Wow, skeptics. Four hours? That's a lot of computation.

Four hours, you're starting to get some people. Four days? All right. Well, how about four years? Still longer, just under four decades? Looking at one choice every microsecond, it takes you roughly 36 years to evaluate all these possibilities. Certainly for people of my age, that's not a practical solution to have to wait 36 years for an answer. So we have to find something better. And we'll be talking about that later.