PROFESSOR: Good morning. Oh, it's so nice to get a response. Thank you. I appreciate it.

i have a confession to make. I stopped at my usual candy store this morning and they didn't have any. So I am bereft of anything other than crummy little Tootsie Rolls for today. But I promise I'll have a new supply by Thursday.

We ended up the last lecture looking at pseudocode for k-means clustering and talking a little bit about the whole idea and what it's doing. So I want to start today by moving from the pseudocode to some real code. This was on a previous handout but it's also on today's handout. So let's look at it. Not so surprisingly I've chosen to call it k-means. And you'll notice that it's got some arguments.

The point to be clustered, k, and that's an interesting question. Unlike hierarchical clustering, where we could run it and get what's called a dendrogram and stop at any level and see what we liked, k-means involves knowing in the very beginning how many clusters we want.

We'll talk a little bit about how we could choose k.

A cutoff. What the cutoff is doing, you may recall that in the pseudocode k-means was iterative and we keep re-clustering until the change is small enough that we feel it's stable. That is to say, the new clusters are not that much different from the old clusters. The cutoff is the definition of what we mean by small enough. We'll see how that gets used.

The type of point to be clustered.

The maximum number of iterations. There's no guarantee that things will converge. As we'll see, they usually converge very quickly in a small number of iterations. But

it's prudent to have something like this just in case things go awry.

And to print. Just my usual trick of being able to print some debugging information if I need it, but not getting buried in output if I don't.

All right. Let's look at the code. And it very much follows the outline of the pseudocode we started with last time. We're going to start by choosing k initial centroids at random. So I'm just going to go and take all the points I have. And I'm assuming, by the way, I should have written this down probably, that I have at least k points. Otherwise it doesn't make much sense. If you have 10 points, you're not going to find 100 clusters. So I'll take k random centroids, and those will be my initial centroids. There are more sophisticated ways of choosing centroids, as discussed in the problem set, but most of the time people just choose them at random, because at least if you do it repetitively it guarantees against some sort of systematic error.

Whoa. What happened? I see. Come back. Thank you.

All right. Then I'm going to say that the clusters I have initially are empty. And then I'm going to create a bunch of singleton clusters, one for each centroid. So all of this is just the initialization, getting things going. I haven't had any iterations yet. And the biggest change so far I'm just setting arbitrarily to the cutoff.

All right. And now I'm going to iterate until the change is smaller than the cutoff while biggest change is at least the cutoff. And just in case numIters is less than the maximum, I'm going to create a list containing k empty list. So these are the new clusters. And then I'm going to go through for i in range k. I'm going to append the empty cluster. These are going to be the new ones. And then for p and all the points I'm going to find the centroid in the existing clustering that's closest to p. That's what's going on here.

Once I've found that, I'm going to add p to the correct cluster, go and do it for the next point. Then when I'm done, I'm going to compare the new clustering to the old clustering and get the biggest change. And then go back and do it again.

All right? People, understand that basic structure and even some of the details of the code. It's not very complicated. But if you haven't seen it before, it can be a little bit tricky.

When I'm done I'm going to just get some statistics here about the clusters, going to keep track of the number of iterations and the maximum diameter of a cluster, so the cluster in which things are least tightly grouped. And this will give me an indication of how good a clustering I have.

OK? Does that make sense to everybody? Any questions about the k-means code?

Well, before we use it, let's look at how we use it. I've written this function testOne that uses it. Some arbitrary values for k in the cutoff. Number of trials is kind of boring here. I've only said one is the default and I've set print steps to false.

The thing I want you to notice here, because I'm choosing the initial clustering at random, I can get different results each time I run this. Because of that, I might want to run it many times and choose the quote, "best clustering." What metric am I using for best clustering? It's a minmax metric. I'm choosing the minimum of the maximum diameters. So I'm finding the worst cluster and trying to make that as good as I can make it.

You could look at the average cluster. This is like the linkage distances we talked about before.

That's the normal kind of thing. It's like when we did Monte Carlo simulations or random walks, flipping coins. You do a lot of trials and then you can either average over the trials, which wouldn't make sense for the clustering, or select the trial that has some property you like. This is the way people usually use k-means. Typically they may do 100 trials and choose the best, the one that gives them the best clustering.

Let's look at this, and let's try it for a couple of examples here. Let's start it up. And we'll just run test one on our old mammal teeth database. We get some clustering.

Now we'll run it again. We get a clustering. I don't know, is it the same clustering? Kind of looks like it is. No reason to suspect it would be. We run it again. Well you know, this is very unfortunate. It's supposed to give different answers here because it often does. I think they're the same answers, though. Aren't they? Yes? Anyone see a difference? No, they're the same.

How unlucky can you be? Every time I ran it at my desk it came up the first two times with different things. But take my word for it, and we'll see that with other examples, it could come out with different answers.

Let's try it with some printing on. We get some things here. Let's try it. What have we got out of this one? All right. Oh, well. Sometimes you get lucky and sometimes you get unlucky with randomness.

All right. So, why did we start with k-means? Not because we needed it for the mammals' teeth. The hierarchical worked fine, but because it was too slow when we tried to look at something big like the counties.

So now let's move on and talk about clustering the counties. We'll use exactly the k-means code. It's one of the reasons we're allowed to pass in the point type as an argument. But the interesting thing will be what we do for the counties themselves. This gets a little complicated.

In particular, what I've added to the counties is this notion of a filter. The reason I've done this is, as we've seen before, the choice of features can make a big difference in what clustering you get. I didn't want to do a lot of typing as we do in these examples, so what I did is I created a bunch of filters. For example, no wealth, which says, all right, we're not going to look at home value. We're giving that a weight of 0. We're giving income a weight of 0, we're giving poverty level a rate of 0. But we're giving the population a weight of 1, et cetera.

OK. What we see here is each filter supplies the weight, in this case either 0 or 1, to a feature. This will allow me as we go forward to run some experiments with different features. All features, everything has a weight of 1. I made a mistake

though. That should have been a 1. Then I have filter names, which are just a dictionary. And that'll make it easy for me to run various kinds of tests with different filters.

Then I've got init, which takes as its arguments the things you would expect, plus the filter name. So it takes the original attributes, the normalized attributes. And you will recall that, why do we need to normalize attributes? If we don't, we have something like population, which could number in the millions, and we're comparing it to percent female, which we know cannot be more than a 100. So the small values become totally dominated by the big absolute values and when we run any clustering it ends up only looking at population or number of farm acres, or something that's big. Has a big dynamic range. Manhattan has no farm acres. Some county in Iowa has a lot. Maybe they're identical in every other respect. Unlikely, but who knows? Except I guess there's no baseball teams in Iowa. But at any rate, we always scale or we try and normalize so that we don't get fooled by that.

Then I go through and, if I haven't already, this is a class variable attribute filter, which is initially set to none. Not an instance variable, but a class variable. And what we see here is, if that class variable is still none, this will mean it's the first time we've generated a point of type county, then what we're going to do is set up the filter to only look at the attributes we care about. So only the attributes which have a value of 1.

And then I'm going to override distance from class point to look at the features we care about.

OK. Does this basic structure and idea make sense to people? It should. I hope it does, because the current problem set requires you to understand it in which you all will be doing some experiments.

So now I want to do some experiments with it. I'm not going to spend too much time, even though it would be fun, because I don't want to deprive you of the fun of doing your problem sets.

So let's look at an example. I've got test, which is pretty much like testOne. Runs k-means number of times and chooses the best. And we can start. Well, let's start by running some examples ourselves. So I'm going to start by clustering on education level only. I'm going to get 20 clusters, 20 chosen just so it wouldn't take too long to run, and we'll filter on education. And we'll see what we get.

Well, I should have probably done more than one cluster just to make it work. But we've got it and just for fun I'm keeping track of what cluster Middlesex County, the county in which MIT shows up. So we can see that it's similar to a bunch of other counties. And it happens to have an average income of $28,665, or at least it did then. And if we look, we should also see-- no, let me go back.

I foolishly didn't uncomment pyLab.show. So we better go back and do that. Well, we're just going to nuke it and run it again because it's easy and I wanted to run it with a couple of trials anyway.

So, we'll first do the clustering. We get cluster 0. Now we're getting a second one. It's going to choose whichever was the tightest. And we'll see that that's what it looks like. So we've now clustered the counties based on education level, no other features. And we see that it's got some interesting properties. There is a small number of counties, clusters, out here near the right side with high income. And, in fact, we'll see that we are fortunate to be in that cluster. One of the clusters that contains wealthy counties. And you could look at it and see whether you recognize any of the other counties that hang out with Middlesex. Things like Marin County, San Francisco County. Not surprisingly. Remember, we're clustering by education and these might be counties where you would expect the level of education to be comparable to the level of education in Middlesex.

All right. Let me get rid of that for now. Sure. I ran it. I didn't want you to have to sit through it, but I ran it on a much bigger sample size. So here's what I got when I ran it asking for 100 clusters. And I think it was 5 trials. And you'll notice that this case, actually, we have a much smaller cluster containing Middlesex. Not surprising, because I've done 100 rather than 20. And it should be pretty tight since I chose the

best-- you can see we have a distribution here.

Now, remember that the name of the game here is we're trying to see whether we can infer something interesting by clustering. Unsupervised learning. So one of the questions we should ask is, how different is what we're getting here from if we chose something at random? Now, remember we did not cluster on things based on income. I happened to plot income here just because I was curious as to how this clustering related to income. Suppose we had just chosen at random and split the counties at random into 100 different clusters, What would you have expected this kind of graph to look like? Do we have something that is different, obviously different, from what we might have gotten if we'd just done a random division into 100 different clusters? Think about it. What would you get?

AUDIENCE: A bell curve?

PROFESSOR: Pardon?

AUDIENCE: We'd get a bell curve.

PROFESSOR: Well, a bell curve is a good guess because bell curves occur a lot in nature. And as I said, I apologize for the rather miserable quality of the rewards. It's a good guess but I think it's the wrong guess. What would you expect? Would you expect the different clusters-- yeah, go ahead.

AUDIENCE: You probably might expect them all to average at a certain point for a very sharp bell curve?

PROFESSOR: A very sharp bell curve was one comment. Well, someone else want to try it? That's kind of close. I thought you were on the right track in the beginning.

Well, take a different example. Let's take students. If I were to select a 100 MIT students at random and compute their GPA, would you it to be radically different from the GPA of all of MIT? The average GPA of all MIT students? Probably not, right? So if I take 100 counties and put them into a cluster, the average income of that cluster is probably pretty close to the average income in the country. So you'd

actually expect it to be kind of flat, right? That each of the randomly chosen clusters would have the same income, more or less.

Well, that's clearly not what we have here. So we can clearly infer from the fact that this is not flat that there is some interesting correlation between level of income and education. And for those of us who earn our living in education, we're glad to see it's positive, actually. Not negative.

As another experiment, just for fun, I clustered by gender only. So this looked only at the female/male ratio in the counties. And here you'll see Middlesex is with-- remember we had about 3,000 counties to start with. So the fact that there were so few in the cluster on education was interesting, right? Here we have more. And we get a very different-looking picture. Which says, perhaps, that the female/male ratio is not unrelated to income, but it's a rather different relation than we get from education. This is what would be called a bi-modal distribution. A lot here and a lot here and not much in the middle.

But again the dynamic range is much smaller. But we do have some counties where the income is pretty miserable.

All right. We could play a lot more with this but I'm not going to. I do want to, before we leave it, because we're about to leave machine learning, reiterate a few of the major points that I wanted to make sure were the take home messages.

So, we talked about supervised learning much less than we talked about unsupervised. Interestingly, because unsupervised learning is probably used more often in the sciences than supervised. And when we did supervised learning, we started with a training set that had labels. Each point had a label. And then we tried to infer the relationships between the features of the points and the associated labels. Between the features and the labels.

We then looked at unsupervised learning. The issue here was, our training set was all unlabeled data. And what we try and infer is relationships among points. So, rather than trying to understand how the features relate to the labels, we're just

trying to understand how the points, or actually, the features related to the points, relate to one another.

Both of these, as I said earlier, are similar to what we saw when we did regression where we tried to fit curves to data. You need to be careful and wary of over-fitting just as you did with regression. In particular, if the training data is small, a small set of training data, you may learn things that are true of the training data that are not true of the data on which you will subsequently run the algorithm to test it. So you need to be wary of that.

Another important lesson is that features matter. Which features matter? It matters whether they're normalized. And in some cases you can even weight them if you want to make some features more important than the others.

Features need to be relevant to the kind of knowledge that you hope to acquire. For example, when I was trying to look at the eating habits of mammals, I chose features based upon teeth, not features based upon how much hair they had or their color of the lengths of the tails. I chose something that I had domain knowledge which would suggest that it was probably relevant to the problem at hand. Question at hand. And then we discovered it was. Just as here, I said, well, maybe education has something to do with income. We ran it and we discovered, thank goodness, that it does.

OK. So I probably told you ten times that features matter. If not, I should have because they do. And it's probably the most important thing to get right in doing machine learning.

Now, our foray into machine learning is part of a much larger unit. In fact, the largest unit of the course really, is about how to use computation to make sense of the kind of information one encounters in the world. A big part of this is finding useful ways to abstract from the situation you're initially confronted with to create a model about which one can reason. We saw that when we did curve fitting. We would abstract from the points to a curve to get a model. And we see that with machine learning, that we abstract from every detail about a county to say, the

education level, to give us a model of the counties that might be useful.

I now want to talk about another kind of way to build models that's as popular a way as there is. Probably the most common kinds of models. Those models are graph theoretic. There's a whole rich theory about graphs and graph theory that are used to understand these models.

Suppose, for example, you had a list of all the airline flights between every city in the United States and what each flight cost. Suppose also, counterfactual supposition, that for all cities A, B and C, the cost of flying from A to C by way of B was the cost of A to B and the cost from B to C. We happen to know that's not true, but we can pretend it is. So what are some of the questions you might ask if I gave you all that data? And in fact, there's a company called ITA Software in Cambridge, recently acquired by Google for, I think, $700 million, that is built upon answering these kinds of questions about these kinds of graphs.

So you could ask, for example, what's the shortest number of hops between two cities? If I want to fly from here to Juneau, Alaska, what's the fewest number of stops? I could ask, what's the least expensive-- different question-- flight from here to Juneau. I could ask what's the least expensive way involving no more than two stops, just in case I don't want to stop too many places. I could say I have ten cities. What's the least expensive way to visit each of them on my vacation?

All of these problems are nicely formalized as graphs. A graph is a set of nodes. Think of those as objects. Nodes are also often called vertices or a vertex for one of them. Those nodes are connected by a set of edges, often called arcs. If the edges are uni-directional, the equivalent of a one-way street, it's called a digraph, or directed graph. Graphs are typically used in situations in which there are interesting relationships among the parts.

The first documented use of this kind of a graph was in 1735 when the Swiss mathematician Leonard Euler used what we now call graph theory to formulate and solve the Konigberg's Bridges problem. So this is a map of Konigsberg, which was then the capital of East Prussia, a part of what's today Germany. And it was built at

the intersection of two rivers and contained a lot of islands. The islands were connected to each other and to the mainland by seven bridges.

For some bizarre reason which history does not record and I cannot even imagine, the residents of this city were obsessed with the question of whether it was possible to take a walk through the city that involved crossing each bridge exactly once. Could you somehow take a walk and go over each bridge exactly once? I don't know why they cared. They seemed to care. They debated it, they walked around, they did things.

It probably would be unfair for me to ask you to look at this map and answer the question. But it's kind of complicated.

Euler's great insight was that you didn't have to actually look at the level of detail represented by this map to answer the question. You could vastly simplify it. And what he said is, well, let's represent each land mass by a point, and each bridge as a line. So, in fact, his map of Konigsberg looked like that. Considerably simpler. This is a graph. We have some vertices and some edges. He said, well, we can just look at this problem and now ask the question.

Once he reformulated the problem this way it became a lot simpler to think about and he reasoned as follows. If a walk were to traverse each bridge exactly once, it must be the case that each node, except for the first and the last, in the walk must have an even number of edges. So if you were to go to an island and leave the island and traverse each bridge to the island unless there were an even number, you couldn't traverse each one exactly once. If there were only one bridge, once you got to the island you were stuck. If there were two bridges you could get there and leave. But if there were three bridges you could get there, leave, get there and you're stuck again.

He then looked at it and said, well, none of these nodes have an even number of edges. Therefore you can't do it. End of story. Stop arguing.

Kind of a nice piece of logic. And then Euler later went on to generalize this theorem

to cover a lot of other situations. But what was important was not the fact that he solved this problem but that he thought about the notion of taking a map and formulating it as a graph. This was the first example of that and since then everything has worked that way.

So if you take this kind of idea and now you extend it to digraphs, you can deal with one-way bridges or one-way streets.

Or suppose you want to look at our airline problem, you can extend it to include weights. For example, the number of miles between two cities or the amount of toll you'd have to pay on some road. So, for example, once you've done this you can easily represent the entire US highway system or any roadmap by a weighted directed graph.

Or, used more often, probably, the World Wide Web is today typically modeled as a directed graph where there's an edge from page A to page B, if there's a link on page A to page B.

And then maybe if you want to care, ask the question how often do people go from A to B, a very important question, say, to somebody like Google, who wants to know how often people click on a link to get to another place so they can charge for those clicks, you use a weighted graph, which says how often does someone go from here to there.

And so a company like Google maintains a model of what happens and uses a weighted, directed graph to essentially represent the Web, the clicks and everything else, and can do all sorts of analysis on traffic patterns and things like that.

There are also many less obvious uses of graphs. Biologists use graphs to measure things ranging from the way proteins interact with each other to, more obviously, gene expression networks are clearly graphs.

Physicists use graphs to model phase transitions with typically the weight of the edge representing the amount of energy needed to go from one phase to another. Those are again weighted directed graphs. The direction is, can you get from this

phase to that phase? And the weight is how much energy does it require?

Epidemiologists use graphs to model diseases, et cetera. We'll see an example of that in a bit.

All right. Let's look at some code now to implement graphs. This is also in your handout and I'm going to comment this out just so we don't run it by accident.

As you might expect, I'm going to use classes to implement graphs. I start with a class node which is at this point pretty simple. It's just got a name.

Now, you might say, well, why did I even bother introducing a class here? Why don't I just use strings? Well, because I was kind of wary that sometime later I might want to associate more properties with nodes. So you could imagine if I'm using a graph to model the World Wide Web, I don't want more than the URL for a page. I might want to have all the words on the page, or who knows what else about it. So I just said, for safety, let's start with a simple class, but let's make it a class so that any code I write can be reused if at some later date I decide nodes are going to be more complicated than just strings. Good programming practice.

An edge is only a little bit more complicated. It's got a source, a destination, and a weight. So you can see that I'm using the most general form of an edge so that I will be able use edges not only for graphs and digraphs, but also weighted directed graphs by having all the potential properties I might need and then some simple things to fetch things.

The next cluster in the hierarchy is a digraph. So it's got an init, of course, I can add nodes to it. I'm not going to allow myself to add the same node more than once. I can add edges. And I'm going to check to make sure that I'm only connecting nodes that are in the graph.

Then I've got children of, which gives me all the descendants of a node, and has node in string.

And then interestingly enough, maybe surprising to some of you, I've made graph a

sub-class of digraph. Maybe that seems a little odd. After all, when I started I started talking about digraphs. About graphs, and then said, and we can add this feature. But now I'm going the other way around. Why is that? Why do you think that's the right way to structure a hierarchy? What's the relation of graphs to digraphs?

Digraphs are more general than graphs. A graph is a specialization of a digraph. Just like a county is a specialization of a point. So, typically as you design these class hierarchies, the more specialized something is, the further it has to be down. More like a subclass. Does that make sense? I can't really turn this on its head. I can specialize a digraph to get a graph. I can't specialize a graph to get a digraph. And that's why this hierarchy is organized the way it is.

What else is there are interesting to say about this? A key question, probably the most important question in designing and implementation of graphs, is the choice of data structures to represent the digraph in this case.

There are two possibilities that people typically use. They can use an adjacency matrix. So if you have N nodes, you have an N by N matrix where each entry gives, in the case of a digraph, a weighted digraph, the weight-connecting nodes on that edge. Or in the case of a graph it can be just true or false. So this is, can I get from A to B?

Is that going to be sufficient? Suppose you have a graph that looks like this. And we'll call that Boston and this New York. And I want to model the roads. Well, I might have a road that looks like this and a road that looks like this from Boston to New York. As in, I might have more than one road. So I have to be careful when I use an adjacency matrix representation, to realize that each element of the matrix could itself be somewhat more complicated in the case that there are multiple edges connecting two nodes. And in fact, in many graphs we will see there are multiple edges connecting the same two nodes. It would be surprising if there weren't.

All right. Now, the other common representation is an adjacency list. In an adjacency list, for every node I list all of the edges emanating from that node.

Which of these is better? Well, neither. It depends upon your application. An adjacency matrix is often the best choice when the connections are dense. Everything is connected to everything else. But is very wasteful if the connections are sparse. If there are no roads connecting most of your cities or no airplane flights connecting most of your cities, then you don't want to have a matrix where most of the entries are empty.

Just to make sure that people follow the difference, which am I using in my implementation here? Am I using an adjacency matrix or an adjacency list? I heard somebody say an adjacency list and because my candy supply is so meager they didn't even bother raising in their hand so I know who said it. But yes, it is an adjacency list. And we can see that by looking what happens when we add an edge. I'm associating it with the source node. So from each node, and we can see that when we look at the children of-- here, I just return the edges of that node. And that's the list of all the places you can get to from that node.

So it's very simple, but it's very useful. Next lecture we'll look-- Yeah? Thank you. Question. I love questions.

AUDIENCE: Going back to the digraph, what makes the graph more specialized?

PROFESSOR: What makes--

AUDIENCE: The graph more specialized?

PROFESSOR: Good question. The question is, what makes the graph more specialized? What we'll see here when we look at graph, it's not a very efficient implementation, but every time you add an edge, I add an edge in the reverse direction. Because if you can get from node A to node B you can get from node B to node A. So I've removed the possibility that you have, say, one-way streets in the graph. And therefore it's a specialization. There are things I can not do with graphs that I can do with digraphs, but anything I can do with a graph I can do with the digraph. It's more general. That make sense? That's a great question and I'm glad you asked it.

All right. Thursday we're going to look at a bunch of classic problems that can be

solved using graphs, and I think that should be fun.