PROFESSOR: Hello, everybody. Before we start the material, a couple of announcements. As usual, there's some reading assignments, and you might be surprised to see something from Chapter 5 suddenly popping up. But this is my relentless attempt to introduce more Python. We'll see one new concept later today, list comprehension. Today we're going to look at classification. And you remember last, on Monday, we looked at unsupervised learning. Today we're looking at supervised learning. It can usually be divided into two categories. Regression, where you try and predict some real number associated with the feature vector, and this is something we've already done really, back when we looked at curve fitting, linear regression in particular. It was exactly building a model that, given some features, would predict a point. In this case, it was pretty simple. It was given x predict y. You can imagine generalizing that to multi dimensions.

Today I'm going to talk about classification, which is very common, in many ways more common than regression for-- in the machine learning world. And here the goal is to predict a discrete value, often called a label, associated with some feature vector. So this is the sort of thing where you try and, for example, predict whether a person will have an adverse reaction to a drug. You're not looking for a real number, you're looking for will they get sick, will they not get sick. Maybe you're trying to predict the grade in a course A, B, C, D, and other grades we won't mention. Again, those are labels, so it doesn't have to be a binary label but it's a finite number of labels.

So here's an example to start with. We won't linger on it too long. This is basically something you saw in an earlier lecture, where we had a bunch of animals and a bunch of properties, and a label identifying whether or not they were a reptile. So we start by building a distance matrix. How far apart they are, an in fact, in this case, I'm not using the representation you just saw. I'm going to use the binary representation, As Professor Grimson showed you, and for the reasons he showed you. If you're interested, I didn't produce this table by hand, I wrote some Python code to produce it, not only to compute the distances, but more delicately to produce the actual table. And you'll probably find it instructive at some point to at least remember that

that code is there, in case you need to ever produce a table for some paper.

In general, you probably noticed I spent relatively little time going over the actual vast amounts of codes we've been posting. That doesn't mean you shouldn't look at it. In part, a lot of it's there because I'm hoping at some point in the future it will be handy for you to have a model on how to do something. All right. So we have all these distances. And we can tell how far apart one animal is from another. Now how do we use those to classify animals? And the simplest approach to classification, and it's actually one that's used a fair amount in practice is called nearest neighbor. So the learning part is trivial. We don't actually learn anything other than we just remember.

So we remember the training data. And when we want to predict the label of a new example, we find the nearest example in the training data, and just choose the label associated with that example. So here I've just drawing a cloud of red dots and black dots. I have a fuschia colored X. And if I want to classify X as black or red, I'd say well its nearest neighbor is red. So we'll call X red. Doesn't get much simpler than that. All right. Let's try and do it now for our animals. I've blocked out this lower right hand corner, because I want to classify these three animals that are in gray. So my training data, very small, are these animals. And these are my test set here.

So let's first try and classify the zebra. We look at the zebra's nearest neighbor. Well it's either a guppy or a dart frog. Well, let's just choose one. Let's choose the guppy. And if we look at the guppy, it's not a reptile, so we say the zebra is not a reptile. So got one right. Look at the python, choose its nearest neighbor, say it's a cobra. The label associated with cobra is reptile, so we win again on the python. Alligator, it's nearest neighbor is clearly a chicken. And so we classify the alligator as not a reptile. Oh, dear. Clearly the wrong answer. All right. What might have gone wrong?

Well, the problem with K nearest neighbors, we can illustrate it by looking at this example. So one of the things people do with classifiers these days is handwriting recognition. So I just copied from a website a bunch of numbers, then I wrote the number 40 in my own inimitable handwriting. So if we go and we look for, say, the nearest neighbor of four-- or sorry, of whatever that digit is. It is, I believe, this one. And sure enough that's the row of fours. We're OK on this. Now if we want to classify my zero, the actual nearest neighbor, in terms of the bitmaps if you will, turns out to be this guy. A very poorly written nine. I didn't make up this nine, it was it was already there.

And the problem we see here when we use nearest neighbor is if something is noisy, if you have one noisy piece of data, in this case, it's rather ugly looking version of nine, you can get the wrong answer because you match it. And indeed, in this case, you would get the wrong answer. What is usually done to avoid that is something called K nearest neighbors. And the basic idea here is that we don't just take the nearest neighbors, we take some number of nearest neighbors, usually an odd number, and we just let them vote. So now if we want to classify this fuchsia X, and we said K equal to three, we say well these are it's three nearest neighbors. One is red, two are black, so we're going to call X black is our better guess.

And maybe that actually is a better guess, because it looks like this red point here is really an outlier, and we don't want to let the outliers dominate our classification. And this is why people almost always use K nearest neighbors rather than just nearest neighbor. Now if we look at this, and we use K nearest neighbors, those are the three nearest to the first numeral, and they are all fours. And if we look at the K nearest neighbors for the second numeral, we still have this nine but now we have two zeros. And so we vote and we decide it's a zero. Is it infallible? No. But it's typically much more reliable than just nearest neighbors, hence used much more often.

And that was our problem, by the way, with the alligator. The nearest neighbor was the chicken, but if we went back and looked at it-- maybe we should go do that. And we take the alligator's three nearest neighbors, it would be the chicken, a cobra, and the rattlesnake-- or the boa, we don't care, and we would end up correctly classifying it now as a reptile. Yes?

AUDIENCE: Is there like a limit to how many [INAUDIBLE]?

PROFESSOR: The question is is there a limit to how many nearest neighbors you'd want? Absolutely. Most obviously, there's no point in setting K equal to-- whoops. Ooh, on the rebound-- to the size of the training set. So one of the problems with K nearest neighbors is efficiency. If you're trying to define K nearest neighbors and K is bigger, it takes longer. So we worry about how big K should be. And if we make it too big-- and this is a crucial thing-- we end up getting dominated by the size of the class.

So let's look at this picture we had before. It happens to be more red dots than black dots. If I make K 10 or 15, I'm going to classify a lot of things as red, just because red is so much more prevalent than black. And so when you have an imbalance, which you usually do, you have to be very careful about K. Does that make sense?

AUDIENCE: [INAUDIBLE] choose K?

PROFESSOR: So how do you choose K? Remember back on Monday when we talked about choosing K for K means clustering? We typically do a very similar kind of thing. We take our training data and we split it into two parts. So we have training and testing, but now we just take the training, and we split that into training and testing multiple times. And we experiment with different K's, and we see which K's gives us the best result on the training data. And then that becomes our K. And that's a very common method. It's called cross-validation, and it's-- for almost all of machine learning, the algorithms have parameters in this case, it's just one parameter, K.

And the way we typically choose the parameter values is by searching through the space using this cross-validation in the training data. Does that makes sense to everybody? Great question. And there was someone else had a question, but maybe it was the same. Do you still have a question?

AUDIENCE: Well, just that you were using like K nearest and you get, like if my K is three and I get three different clusters for the K [INAUDIBLE]

PROFESSOR: Three different clusters?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, right. So if K is 3, and I had red, black, and purple and I get one of each, then what do I do? And then I'm kind of stuck. So you need to typically choose K in such a way that when you vote you get a winner. Nice. So if there's two, any odd number will do. If it's three, well then you need another number so that there's some-- so there's always a majority. Right? You want to make sure that there is a winner. Also a good question. Let's see if I get this to you directly. I'm much better at throwing overhand, I guess. Wow. Finally got applause for something.

All right, advantages and disadvantages KNN? The learning is really fast, right? I just remember everything. No math is required. Didn't have to show you any theory. Was obviously an idea. It's easy to explain the method to somebody, and the results. Why did I label it black? Because that's who it was closest to. The disadvantages is it's memory intensive. If I've got a million examples, I have to store them all. And the predictions can take a long time. If I have an example and I want to find its K nearest neighbors, I'm doing a lot of comparisons. Right?

If I have a million tank training points I have to compare my example to all a million. So I have no real pre-processing overhead. But each time I need to do a classification, it takes a long time. Now there are better algorithms and brute force that give you approximate K nearest neighbors. But on the whole, it's still not fast. And we're not getting any information about what process might have generated the data. We don't have a model of the data in the way we say when we did our linear regression for curve fitting, we had a model for the data that sort of described the pattern. We don't get that out of k nearest neighbors.

I'm going to show you a different approach where we do get that. And I'm going to do it on a more interesting example than reptiles. I apologize to those of you who are reptologists. So you probably all heard of the Titanic. There was a movie about it, I'm told. It was one of the great sea disasters of all time, a so-called unsinkable ship-- they had advertised it as unsinkable-- hit an iceberg and went down. Of the 1,300 passengers, 812 died. The crew did way worse. So at least it looks as if the curve was actually pretty heroic. They had a higher death rate.

So we're going to use machine learning to see if we can predict which passengers survived. There's an online database I'm using. It doesn't have all 1,200 passengers, but it has information about 1,046 of them. Some of them they couldn't get the information. Says what cabin class they were in first, second, or third, how old they were, and their gender. Also has their name and their home address and things, which I'm not using. We want to use these features to see if we can predict which passengers were going to survive the disaster.

Well, the first question is something that Professor Grimson alluded to is, is it OK, just to look at accuracy? How are we going to evaluate our machine learning? And it's not. If we just predict died for everybody, well then we'll be 62% accurate for the passengers and 76% accurate for the crew members. Usually machine learning, if you're 76% you say that's not bad. Well, here I can get that just by predicting died. So whenever you have a class imbalance that much more of one than the other, accuracy isn't a particularly meaningful measure.

I discovered this early on in my work and medical area. There are a lot of diseases that rarely occur, they occur in say 0.1% of the population. And I can build a great model for predicting it by just saying, no, you don't have it, which will be 0.999% accurate, but totally useless. Unfortunately, you do see people doing that sort of thing in the literature. You saw these in an earlier lecture, just to remind you, we're going to be looking at other metrics.

Sensitivity, think of that as how good is it at identifying the positive cases. In this case, positive is going to be dead. How specific is it, and the positive predictive value. If we say somebody died, what's the probability is that they really did? And then there's the negative predictive value. If we say they didn't die, what's the probability they didn't die? So these are four very common metrics. There is something called an F score that combines them, but I'm not going to be showing you that today.

I will mention that in the literature, people often use the word recall to mean sensitivity or sensitivity I mean recall, and specificity and precision are used pretty much interchangeably. So you might see various combinations of these words. Typically, people talk about recall n precision or sensitivity and specificity. Does that makes sense, why we want to look at the measures other than accuracy? We will look at accuracy, too, and how they all tell us kind of different things, and how you might choose a different balance.

For example, if I'm running a screening test, say for breast cancer, a mammogram, and trying to find the people who should get on for a more extensive examination, what do I want to emphasize here? Which of these is likely to be the most important? Or what would you care about most? Well, maybe I want sensitivity. Since I'm going to send this person on for future tests, I really don't want to miss somebody who has cancer, and so I might think sensitivity is more important than specificity in that particular case.

On the other hand, if I'm deciding who is so sick I should do open heart surgery on them, maybe I want to be pretty specific. Because the risk of the surgery itself are very high. I don't want to do it on people who don't need it. So we end up having to choose a balance between these things, depending upon our application. The other thing I want to talk about before actually building a classifier is how we test our classifier, because this is very important.

I'm going to talk about two different methods, leave one out class of testing and repeated random subsampling. For leave one out, it's typically used when you have a small number of examples, so you want as much training data as possible as you build your model. So you take all of your n examples, remove one of them, train on n minus 1, test on the 1. Then you put that 1 back and remove another 1. Train on n minus 1, test on 1. And you do this for each element of the data, and then you average your results.

Repeated random subsampling is done when you have a larger set of data, and there you might say split your data 80/20. Take 80% of the data to train on, test it on 20. So this is very similar to what I talked about earlier, and answered the question about how to choose K. I haven't seen the future examples, but in order to believe in my model and say my parameter settings, I do this repeated random subsampling or leave one out, either one. There's the code for leave one out. Absolutely nothing interesting about it, so I'm not going to waste your time looking at it.

Repeated random subsampling is a little more interesting. What I've done here is I first sample-- this one is just to splitted 80/20. It's not doing anything repeated, and I start by sampling 20% of the indices, not the samples. And I want to do that at random. I don't want to say get consecutive ones. So we do that, and then once I've got the indices, I just go through and assign each example, to either test or training, and then return the two sets.

But if I just sort of sampled one, then I'd have to do a more complicated thing to subtract it from the other. This is just efficiency. And then here's the-- sorry about the yellow there-- the random splits. Obviously, I was searching for results when I did my screen capture. I'm just going to for range and number of splits, I'm going to split it 80/20. It takes a parameter method, and that's interesting, and we'll see the ramifications of that later. That's going to be the machine learning method.

We're going to compare KNN to another method called logistic regression. I didn't want to have to do this code twice, so I made the method itself a parameter. We'll see that introduces a slight complication, but we'll get to it when we get to it. So I split it, I apply whatever that method is the training the test set, I get the results, true positive false positive, true negative false negatives. And then I call this thing get stats, but I'm dividing it by the number of splits, so that will give me the average number of true positives, the average number of false positives, etc. And then I'm just going to return the average.

Get stats actually just prints a bunch of statistics for us. Any questions about the two methods, leave one out versus repeated random sampling? Let's try it for KNN on the Titanic. So I'm not going to show you the code for K nearest classify. It's in the code we uploaded. It takes four arguments the training set, the test set, the label that we're trying to classify. Are we looking for the people who died? Or the people who didn't die? Are we looking for reptiles or not reptiles? Or if case there were six labels, which one are we trying to detect?

And K as in how many nearest neighbors? And then it returns the true positives, the false positives, the true negatives, and the false negatives. Then you'll recall we'd already looked at lambda in a different context. The issue here is K nearest classify takes four arguments, yet if we go back here, for example, to random splits, what we're seeing is I'm calling the method with only two arguments. Because after all, if I'm not doing K nearest neighbors, maybe I don't need to pass in K. I'm sure I don't.

Different methods will take different numbers of parameters, and yet I want to use the same function here method. So the trick I use to get around that-- and this is a very common programming trick-- in math. It's called currying, after the mathematician Curry, not the Indian dish. I'm creating a function a new function called KNN. This will be a function of two arguments, the training set and the test set, and it will be K nearest classifier with training set and test set as variables, and two constants, survived-- so I'm going to predict who survived-- and 3, the K.

I've been able to turn a function of four arguments, K nearest classify, into a function of two arguments KNN by using lambda abstraction. This is something that people do fairly frequently, because it lets you build much more general programs when you don't have to worry about the number of arguments. So it's a good trick to keeping your bag of tricks. Again, it's a trick we've used before. Then I've just chosen 10 for the number of splits, and we'll try it, and we'll try it for both methods of testing. Any questions before I run this code?

So here it is. We'll run it. Well, I should learn how to spell finished, shouldn't I? But that's OK. Here we have the results, and they're-- well, what can we say about them? They're not much different to start with, so it doesn't appear that our testing methodology had much of a difference on how well the KNN worked, and that's actually kind of comforting. The accurate-- none of the evaluation criteria are radically different, so that's kind of good. We hoped that was true.

The other thing to notice is that we're actually doing considerably better than just always predicting, say, didn't survive. We're doing better than a random prediction. Let's go back now to the Power Point. Here are the results. We don't need to study them anymore. Better than 62% accuracy, but not much difference between the experiments. So that's one method. Now let's look at a different method, and this is probably the most common method used in machine learning.

It's called logistic regression. It's, in some ways, if you look at it, similar to a linear regression, but different in some important ways. Linear regression, you will I'm sure recall, is designed to predict a real number. Now what we want here is a probability, so the probability of some event. We know that the dependent variable can only take on a finite set of values, so we want to predict survived or didn't survive. It's no good to say we predict this person half survived, you know survived, but is brain dead or something. I don't know. That's not what we're trying to do.

The problem with just using regular linear regression is a lot of time you get nonsense predictions. Now you can claim, OK 0.5 is there, and it means has a half probability of dying, not that half died. But in fact, if you look at what goes on, you could get more than one or less than 0 out of linear regression, and that's nonsense when we're talking about probabilities. So we need a different method, and that's logistic regression. What logistic regression does is it finds what are called the weights for each feature.

You may recall I complained when Professor [? Grimson ?] used the word weights to mean something somewhat different. We take each feature, for example the gender, the cabin class, the age, and compute for that weight that we're going to use in making predictions. So think of the weights as corresponding to the coefficients we get when we do a linear regression. So we have now a coefficient associated with each variable. We're going to take those coefficients, add them up, multiply them by something, and make a prediction.

A positive weight implies-- and I'll come back to this later-- it almost implies that the variable is positively correlated with the outcome. So we would, for example, say the have scales is positively correlated with being a reptile. A negative weight implies that the variable is negatively correlated with the outcome, so number of legs might have a negative weight. The more legs an animal has, the less likely it is to be a reptile. It's not absolute, it's just a correlation.

The absolute magnitude is related to the strength of the correlation, so if it's being positive it means it's a really strong indicator. If it's big negative, it's a really strong negative indicator. And then we use an optimization process to compute these weights from the training data. It's a little bit complex. It's key is the way it uses the log function, hence the name logistic, but I'm not going to make you look at it. But I will show you how to use it.

You start by importing something called sklearn.linear_model. Sklearn is a Python library, and in that is a class called logistic regression. It's the name of a class, and here are three methods of that class. Fit, which takes a sequence of feature vectors and a sequence of labels and returns an object of type logistic regression. So this is the place where the optimization is done. Now all the examples I'm going to show you, these two sequences will be-- well all right. So think of this as the sequence of feature vectors, one per passenger, and the labels associated with those. So this and this have to be the same length.

That produces an object of this type, and then I can ask for the coefficients, which will return the weight of each variable, each feature. And then I can make a prediction, given a feature vector returned the probabilities of different labels. Let's look at it as an example. So first let's build the model. To build the model, we'll take the examples, the training data, and I just said whether we're going to print something. You'll notice from this slide I've elighted the printed stuff. We'll come back in a later slide and look at what's in there. But for now I want to focus on actually building the model.

I need to create two vectors, two lists in this case, the feature vectors and the labels. For e in examples, featurevectors.append(e.getfeatures e.getfeatures e.getlabel. Couldn't be much simpler than that. Then, just because it wouldn't fit on a line on my slide, I've created this identifier called logistic regression, which is sklearn.linearmodel.logisticregression. So this is the thing I imported, and this is a class, and now I'll get a model by first creating an instance of the class, logistic regression. Here I'm getting an instance, and then I'll call dot fit with that instance, passing it feature vecs and labels.

I now have built a logistic regression model, which is simply a set of weights for each of the variables. This makes sense? Now we're going to apply the model, and I think this is the last piece of Python I'm going to introduce this semester, in case you're tired of learning about Python. And this is at least list comprehension. This is how I'm going to build my set of test feature vectors. So before we go and look at the code, let's look at how list comprehension works.

In its simplest form, says some expression for some identifier in some list, L. It creates a new list by evaluating this expression Len (L) times with the ID in the expression replaced by each element of the list L. So let's look at a simple example. Here I'm saying L equals x times x for x in range 10. What's that going to do? It's going to, essentially, create a list. Think of it as a list, or at least a sequence of values, a range type actually in Python 3-- of values 0 to 9.

It will then create a list of length 10, where the first element is going to be 0 times 0. The second element 1 times 1, etc. OK? So it's a simple way for me to create a list that looks like that. I can be fancier and say for x times L equals x times x for x in range 10, and I add and if. If x mod 2 is equal to 0. Now instead of returning all-- building a list using each value in range 10, it will use only those values that satisfy that test. We can go look at what happens when we run that code.

You can see the first list is 1 times 1, 2 times 2, et cetera, and the second list is much shorter, because I'm only squaring even numbers. Well, you can see that list comprehension gives us a convenient compact way to do certain kinds of things. Like lambda expressions, they're easy to misuse. I hate reading code where I have list comprehensions that go over multiple lines on my screen, for example. So I use it quite a lot for small things like this. If it's very large, I find another way to do it.

Now we can move forward. In applying the model, I first build my testing feature of x, my e.getfeatures for e in test set, so that will give me the features associated with each element in the test set. I could obviously have written a for loop to do the same thing, but this was just a little cooler. Then we get model.predict for each of these. Model.predict_proba is nice in that I don't have to predict it for one example at a time. I can pass it as set of examples, and what I get back is a list of predictions, so that's just convenient.

And then setting these to 0, and for I in range len of probs, here a probability of 0.5. What's that's saying is what I get out of logistic regression is a probability of something having a label. I then have to build a classifier, give a threshold. And here what I've said, if the probability of it being true is over a 0.5, call it true. So if the probability of survival is over 0.5, call it survived. If it's below, call it not survived. We'll later see that, again, setting that probability is itself an interesting thing, but the default in most systems is half, for obvious reasons.

I get my probabilities for each feature vector, and then for I in ranged lens of probabilities, I'm just testing whether the predicted label is the same as the actual label, and updating true positives, false positives, true negatives, and false negatives accordingly. So far, so good? All right, let's put it all together. I'm defining something called LR, for logistic regression. It takes the training data, the test data, the probability, it builds a model, and then it gets the results by calling apply model with the label survived and whatever this prob was.

Again, we'll do it for both leave one out and random splits, and again for 10 random splits. You'll notice it actually runs-- maybe you won't notice, but it does run faster than KNN. One of the nice things about logistic regression is building the model takes a while, but once you've got the model, applying it to a large number of variables-- feature vectors is fast. It's independent of the number of training examples, because we've got our weights. So solving the optimization problem, getting the weights, depends upon the number of training examples. Once we've got the weights, it's just evaluating a polynomial. It's very fast, so that's a nice advantage.

If we look at those-- and we should probably compare them to our earlier KNN results, so KNN on the left, logistic regression on the right. And I guess if I look at it, it looks like logistic regression did a little bit better. That's not guaranteed, but it often does outperform because it's more subtle in what it does, in being able to assign different weights to different variables. It's a little bit better. That's probably a good thing, but there's another reason that's really important that people prefer logistic regression, is it provides insights about the variables.

We can look at the feature weights. This code does that, so remember we looked at build model and I left out the printing? Well here I'm leaving out everything except the printing. Same function, but leaving out everything except the printing. We can do model underbar classes, so model.classes underbar gives you the classes. In this case, the classes are survived, didn't survive. I forget what I called it. We'll see. So I can see what the classes it's using are, and then for I in range len model dot cof underbar, these are giving the weights of each variable. The coefficients, I can print what they are.

So let's run that and see what we get. We get a syntax error because I turned a comment into a line of code. Our model classes are died and survived, and for label survived-- what I've done, by the way, in the representation is I represented the cabin class as a binary variable. It's either 0 or 1, because it doesn't make sense to treat them as if they were really numbers because we don't know, for example, the difference between first and second is the same as the difference between second and third. If we treated the class, we just said cabin class and used an integer, implicitly the learning algorithm is going to assume that the difference between 1 and 2 is the same as between 2 and 3.

If you, for example, look at the prices of these cabins, you'll see that that's not true. The difference in an airplane between economy plus and economy is way smaller than between economy plus him first. Same thing on the Titanic. But what we see here is that for the label survived, pretty good sized positive weight for being in first class cabin. Moderate for being in the second, and if you're in the third class well, tough luck. So what we see here is that rich people did better than the poor people. Shocking.

If We look at age, we'll see it's negatively correlated. What does this mean? It's not a huge weight, but it basically says that if you're older, the bigger your age, the less likely you are to have survived the disaster. And finally, it says it's really bad to be a male, that the men-- being a male was very negatively correlated with surviving. We see a nice thing here is we get these labels, which we can make sense of. One more slide and then I'm done.

These values are slightly different, because different randomization, different example, but the main point I want to say is you have to be a little bit wary of reading too much into these weights. Because not in this example, but other examples-- well, also in these features are often correlated, and if they're correlated, you run-- actually it's 3:56. I'm going to explain the problem with this on Monday when I have time to do it properly. So I'll see you then.