Lesson 1: Welcom to the course!

Welcome!
Hi everyone and welcome to this course on deep learning with PyTorch. I'm Luis Serrano, the lead instructor for this course. I've been at Udacity for nearly three years, teaching various machine learning, deep learning, and AI topics. Before Udacity, I was a Machine Learning Engineer at Google. And before that, I received a PhD in mathematics from the University of Michigan, and a Postdoctoral Fellowship at the University of Quebec at Montreal.

Course overview
We've built this course as an introduction to deep learning. Deep learning is a field of machine learning utilizing massive neural networks, massive datasets, and accelerated computing on GPUs. Many of the advancements we've seen in AI recently are due to the power of deep learning. This revolution is impacting a wide range of industries already with applications such as personal voice assistants, medical imaging, automated vehicles, video game AI, and more.

In this course, we'll be covering the concepts behind deep learning and how to build deep learning models using PyTorch. We've included a lot of hands-on exercises so by the end of the course, you'll be defining and training your own state-of-the-art deep learning models.

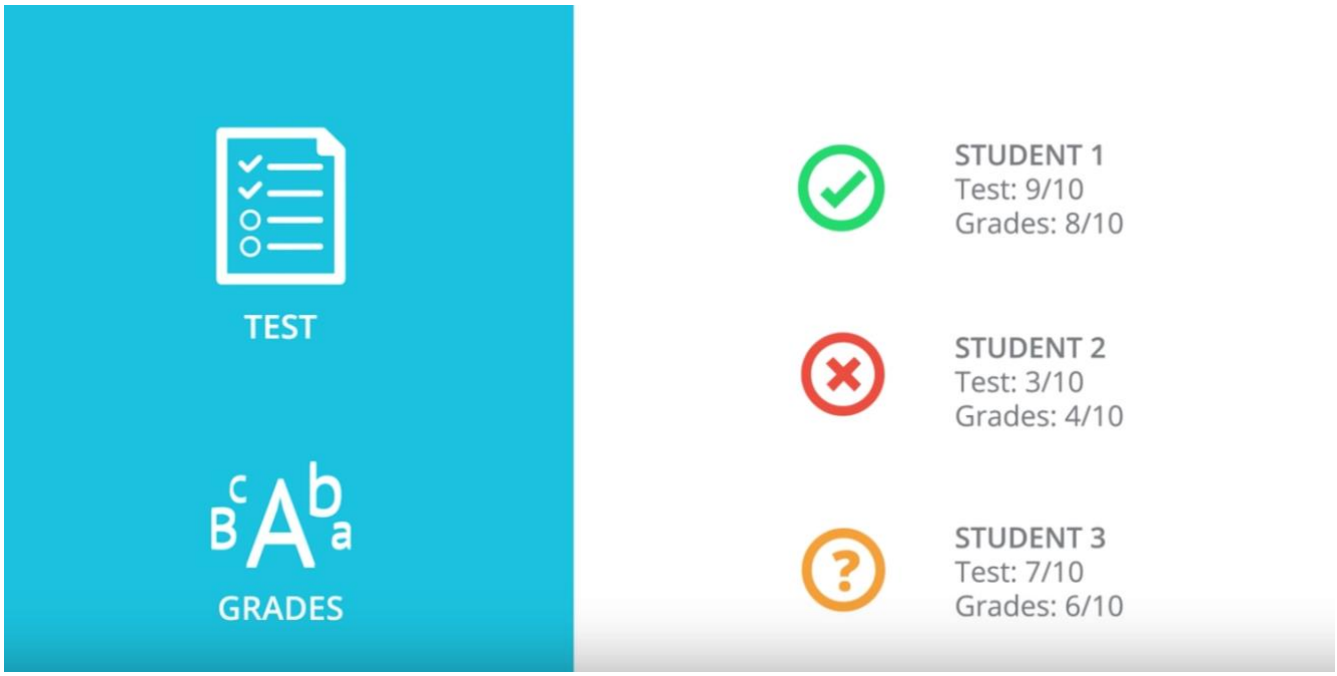
PyTorch
PyTorch (ch is chee no k, tao) is an open-source Python framework from the Facebook AI Research team used for developing deep neural networks. I like to think of PyTorch as an extension of Numpy that has some convenience classes for defining neural networks and accelerated computations using GPUs. PyTorch is designed with a Python-first philosophy, it follows Python conventions and idioms, and works perfectly alongside popular Python packages.

PyTorch Community and Facebook Developers Community
Our friends at Facebook have collaborated with us to bring you access to this free course and develop your skills in deep learning using PyTorch. Make sure you register to stay up to date on the latest PyTorch news, product updates, meetups, and programs like Developer Circles.

Lesson 2: Introduction to Neural Networks

1. So let's start with two questions, what is deep learning, and what is it used for? The answer to the second question is pretty much everywhere. Recent applications include things such as beating humans in games such as Go, or even jeopardy, detecting spam in emails, forecasting stock prices, recognizing images in a picture, and even diagnosing illnesses sometimes with more precision than doctors. And of course, one of the most celebrated applications of deep learning is in self-driving cars. And what is at the heart of deep learning? This wonderful object called neural networks. Neural networks vaguely mimic the process of how the brain operates, with neurons that fire bits of information. It sounds pretty scary, right? As a matter of fact, the first time I heard of a neural network, this is the image that came into my head, some scary robot with artificial brain. But then, I got to learn a bit more about neural networks and I realized that there are actually a lot scarier than that. This is how a neural network looks. As a matter of fact, this one here is a deep neural network. Has lots of nodes, lots of edges, lots of layers, information coming through the nodes and

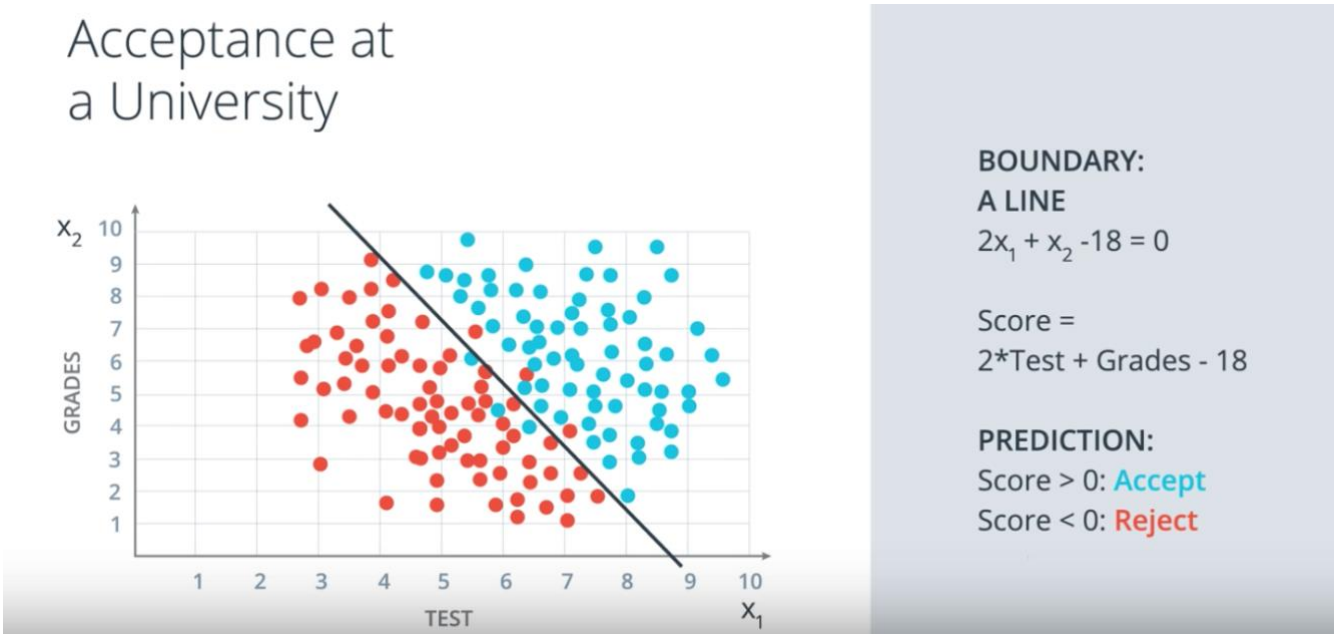
leaving, it's quite complicated. But after looking at neural networks for a while, I realized that they're actually a lot simpler than that. When I think of a neural network, this is actually the image that comes to my mind. There is a child playing in the sand, with some red and blue shells and we are the child. Can you draw a line that separates the red and the blue shells? And the child draws this line. That's it. That's what a neural network does. Given some data in the form of blue or red points, the neural network will look for the best line that separates them. And if the data is a bit more complicated like this one over here, then we'll need a more complicated algorithm. Here, a deep neural network will do the job and find a more complex boundary that separates the points. So with that image in mind, let's dive in and learn about neural networks.



2. So, let's start with one classification example. Let's say we are the admissions office at a university and our job is to accept or reject students. So, in order to evaluate students, we have two pieces of information, the results of a test and their grades in school. So, let's take a look at some sample students. We'll start with Student 1 who got 9 out of 10 in the test and 8 out of 10 in the grades. That student did quite well and got accepted. Then we have Student 2 who got 3 out of 10 in the test and 4 out of 10 in the grades, and that student got rejected. And now, we have a new Student 3 who got 7 out of 10 in the test and 6 out of 10 in the grades, and we're wondering if the student gets accepted or not.

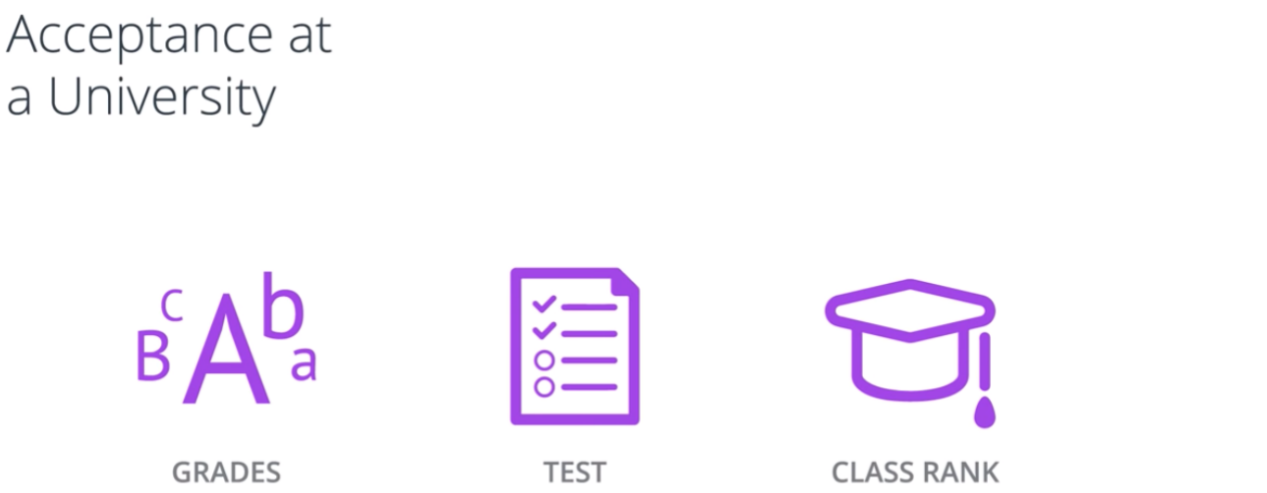
So, our first way to find this out is to plot students in a graph with the horizontal axis corresponding to the score on the test and the vertical axis corresponding to the grades, and the students would fit here. The students who got three and four gets located in the point with coordinates (3,4), and the student who got nine and eight gets located in the point with coordinates (9,8). And now we'll do what we do in most of our algorithms, which is to look at the previous data. This is how the previous data looks. These are all the previous students who got accepted or rejected. The blue points correspond to students that got accepted, and the red points to students that got rejected. So we can see in this diagram that the students would did well in the test and grades are more likely to get accepted, and the students who did poorly in both are more likely to get rejected. So let's start with a quiz. The quiz says, does the Student 3 get accepted or rejected? What do you think? Enter your answer below.

3. Correct. Well, it seems that this data can be nicely separated by a line which is this line over here, and it seems that most students over the line get accepted and most students under the line get rejected. So this line is going to be our model. The model makes a couple of mistakes since there are a few blue points that are under the line and a few red points over the line. But we're not going to care about those. I will say that it's safe to predict that if a point is over the line the student gets accepted and if it's under the line then the student gets rejected. So based on this model we'll look at the new student that we see that they are over here at the point 7:6 which is above the line. So we can assume with some confidence that the student gets accepted. So if you answered yes, that's the correct answer. And now a question arises. The question is, how do we find this line? So we can kind of eyeball it. But the computer can't. We'll dedicate the rest of the session to show you algorithms that will find this line, not only for this example, but for much more general and complicated cases.

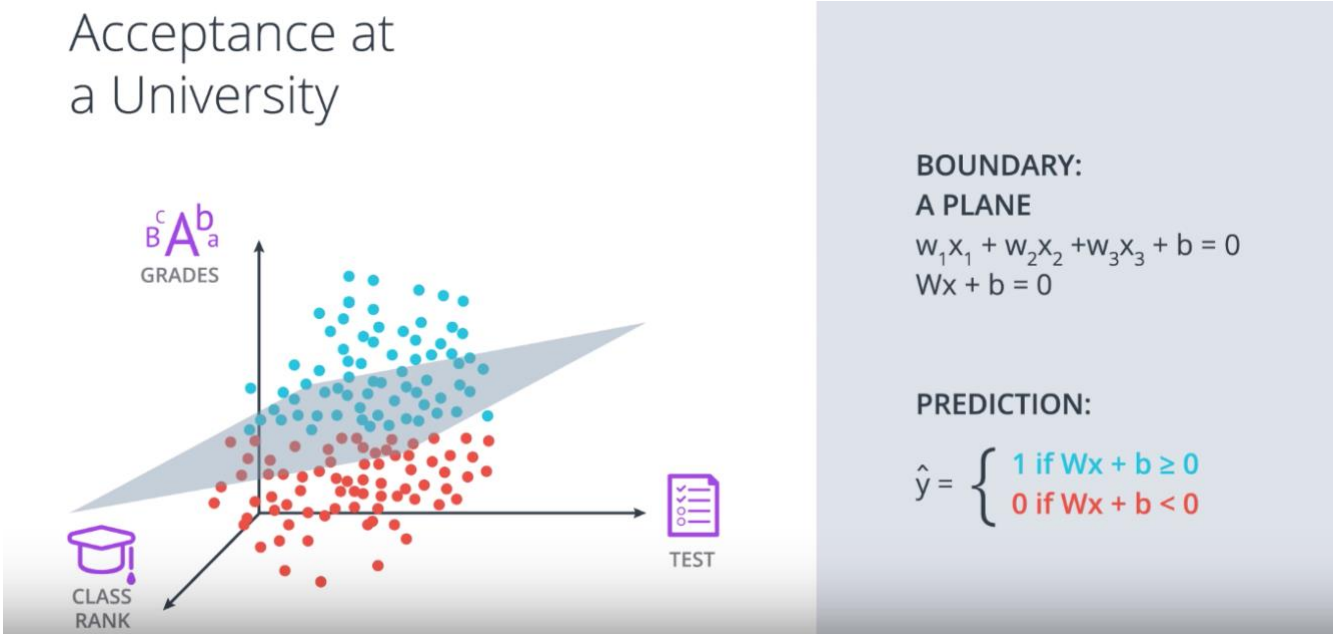


4. So, first let's add some math. We're going to label the horizontal axis corresponding to the test by the variable x1, and the vertical axis corresponding to the grades by the variable x2. So this boundary line that separates the blue and the red points is going to have a linear equation. The one drawn has equation 2x1+x2-18=0. What does this mean? This means that our method for accepting or rejecting students simply says the following: take this equation as our score, the score is 2xtest+grades-18. Now when the student comes in, we check their score. If their score is a positive number, then we accept the student and if the score is a negative number then we reject the student. This is called a prediction. We can say by convention that if the score is 0, we'll accept a student although this won't matter much at the end. And that's it. That linear equation is our model.

In the more general case, our boundary will be an equation of the following wx1+w2x2+b=0. We'll abbreviate this equation in vector notation as wx+b=0, where w is the vector w1w2 and x is the vector x1x2. And we simply take the product of the two vectors. We'll refer to x as the input, to w as the weights and b as the bias. Now, for a student coordinates x1x2, we'll denote a label as Y and the label is what we're trying to predict. So if the student gets accepted, namely the point is blue, then the label is Y+1. And if the student gets rejected, namely the point is red and then the label is Y=0. Thus, each point is in the form x1x2Y or Y is 1 for the blue points and 0 for the red points. And finally, our prediction is going to be called Y-hat and it will be what the algorithm predicts that the label will be. In this case, Y-hat is one of the algorithm predicts that the student gets accepted, which means the point lies over the line. And, Y-hat is 0 if the algorithm predicts that this didn't get rejected, which means the point is under the line. In math terms, this means that the prediction Y-hat is 1 if wx+b is greater than or equal to zero and 0 if wx+b is less than 0. So, to summarize, the points above the line have Y hat=1 and the points below the line have Y-hat=0. And, the blue points have Y=1 and the red points have Y=0. And, the goal of the algorithm is to have Y-hat resembling Y as closely as possible, which is exactly equivalent to finding the boundary line that keeps most of the blue points above it and most of the red points below it.



5. Now, you may be wondering what happens if we have more data columns so not just testing grades, but maybe something else like the ranking of the student in the class. How do we fit three columns of data? Well the only difference is that now, we won't be working in two dimensions, we'll be working in three.



So now, we have three axis: x_1 for the test, x_2 for the grades and x_3 for the class ranking. And our data will look like this, like a bunch of blue and red points flying around in 3D. On our equation won't be a line in two dimension, but a plane in three dimensions with a similar equation as before. Now, the equation would be w_1_x_1 plus w_2_x_2 plus w_3_x_3 plus b equals zero, which will separate this space into two regions. This equation can still be abbreviated by Wx plus b equals zero, except our vectors will now have three entries instead of two. And our prediction will still be y head equals one if Wx plus b is greater than or equal to zero, and zero if Wx plus b is less than zero.

Acceptance at a University

	X_1	X_2	X_3		X_n	y
	EXAM 1	EXAM 2	GRADES	...	ESSAY	PASS?
STUDENT 1	9	6	5	...	6	1(yes)
STUDENT 2	8	4	8	...	3	0(no)
...	...	...	...	...	...	
STUDENT n	6	7	2	...	8	1(yes)

n columns

n-dimensional space

$x_1, x_2, \ldots, x_n$

BOUNDARY:

n-1 dimensional hyperplane

$w_1x_1 + w_2x_2 + w_nx_n + b = 0$

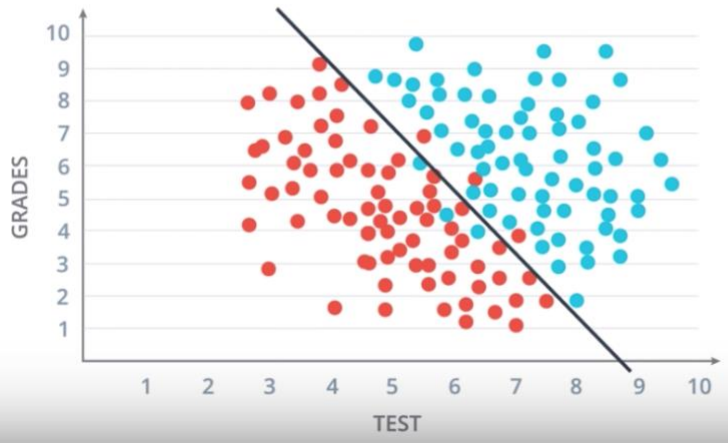
$Wx + b = 0$

PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

And what if we have many columns like say n of them? Well, it's the same thing. Now, our data just leaps in n-dimensional space. Now, I have trouble picturing things in more than three dimensions. But if we can imagine that the points are just things with n coordinates called x_1, x_2, x_3 all the way up to x_n with our labels being y, then our boundaries just an n minus one dimensional hyperplane, which is a high dimensional equivalent of a line in 2D or a plane in 3D. And the equation of this n minus one dimensional hyperplane is going to be w_1_x_1 plus w_2_x_2 plus all the way to w_n_x_n plus b equals zero, which we can still abbreviate to Wx plus b equals zero, where our vectors now have n entries. And our prediction is still the same as before. It is y head equals one if Wx plus b is greater than or equal to zero and y head equals zero if Wx plus b is less than zero.

Acceptance at a University



BOUNDARY:

A LINE

$2x_1 + x_2 - 18 = 0$

Score=

2*Test + Grades -18

PREDICTION:

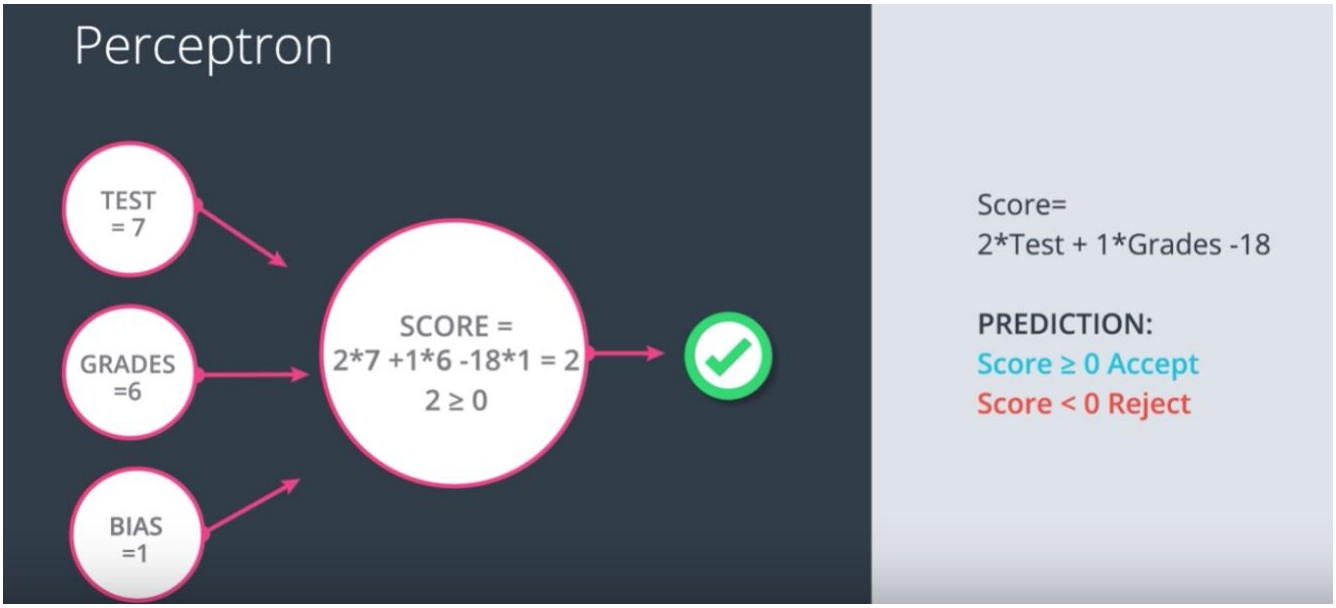
Score ≥ 0 Accept

Score < 0 Reject

6. So let's recap. We have our data which is all these students. The blue ones have been accepted and the red ones have been rejected. And we have our model which consists of the equation two times test plus grades minus 18, which gives rise to this boundary which the point where the score is zero and a prediction. The prediction says that the student gets accepted of the score is positive or zero, and rejected if the score is negative. So now we'll introduce the notion of a preceptron, which is the building block of neural networks, and it's just an encoding of our equation into a small graph. The way we've build it is the following.

Here we have our data and our boundary line and we fit it inside a node. And now we add small nodes for the inputs which, in this case, they are the test and the grades. Here we can see an example where test equals seven and grades equals six. And what the perceptron does is it blocks the points seven, six and checks if the point is in the positive or negative area. If the point is in the positive area, then it returns a yes. And if it is in the negative area, it returns and no. So let's recall that our equation is score equals two times test plus one times grade minus 18, and that our prediction consists of accepting the student if the score is positive or zero, and rejecting them if the score is negative. These weights two, one, and minus 18, are what define the linear equation, and so we'll use them as labels in the graph. The two and the one will label the edges coming from X1 and X 2 respectively, and the bias unit minus 18 will label the node.

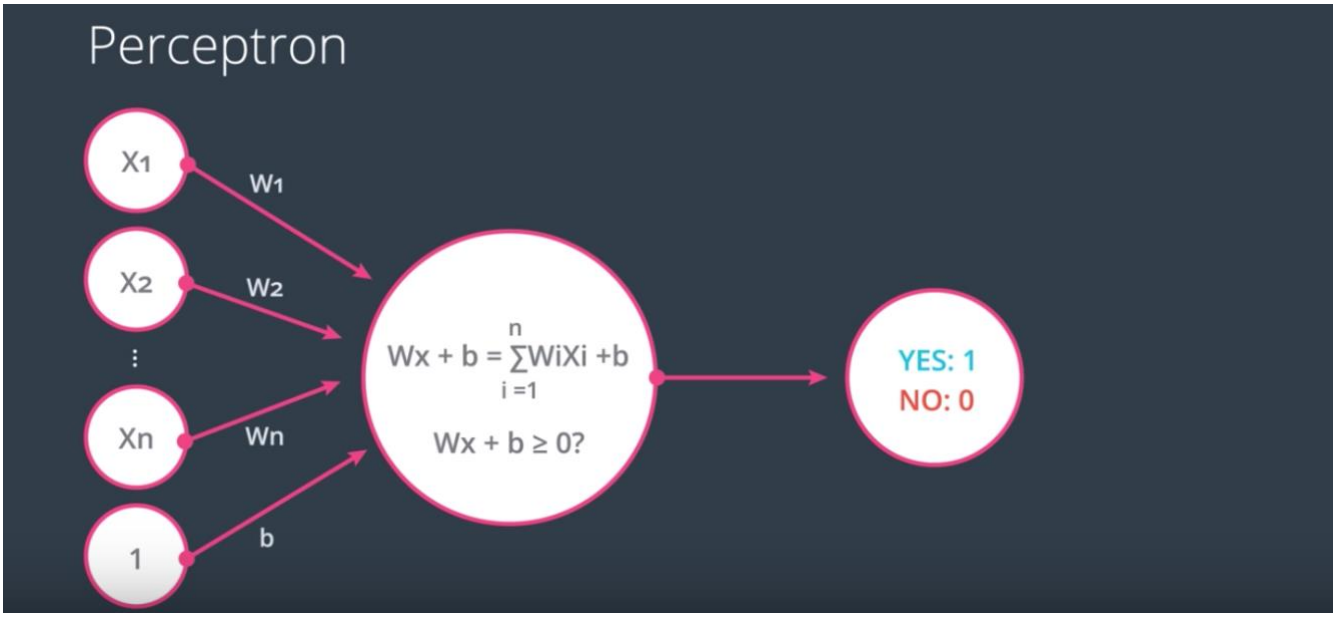
Thus, when we see a node with these labels, we can think of the linear equation they generate.



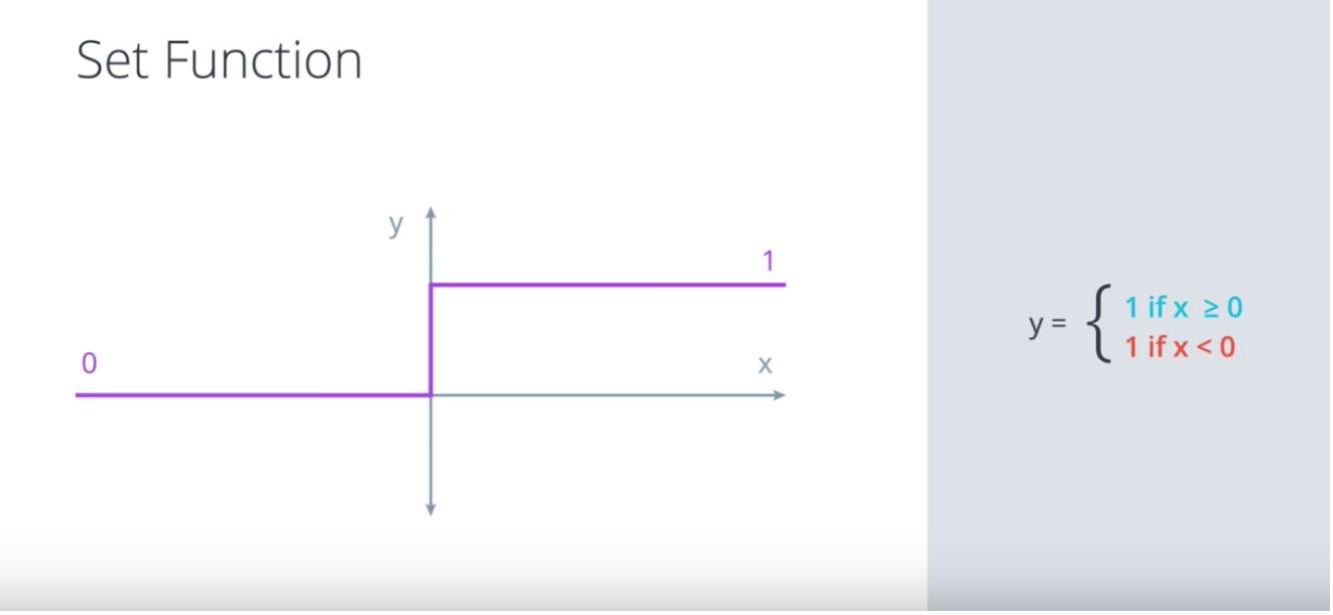
Another way to grab this node is to consider the bias as part of the input. Now since W1 gets multiplied by X1 and W2 by X2, It's natural to think that B gets multiplied by a one. So we'll have the B labeling and and edge coming from a one. Then what the node does is it multiplies the values coming from the incoming nodes by the values and the corresponding edges. Then it adds them

and finally, it checks if the result is greater that are equal to zero. If it is, then the node returns a yes or a value of one, and if it isn't then the node returns a no or a value of zero.

We'll be using both notations throughout this class although the second one will be used more often.

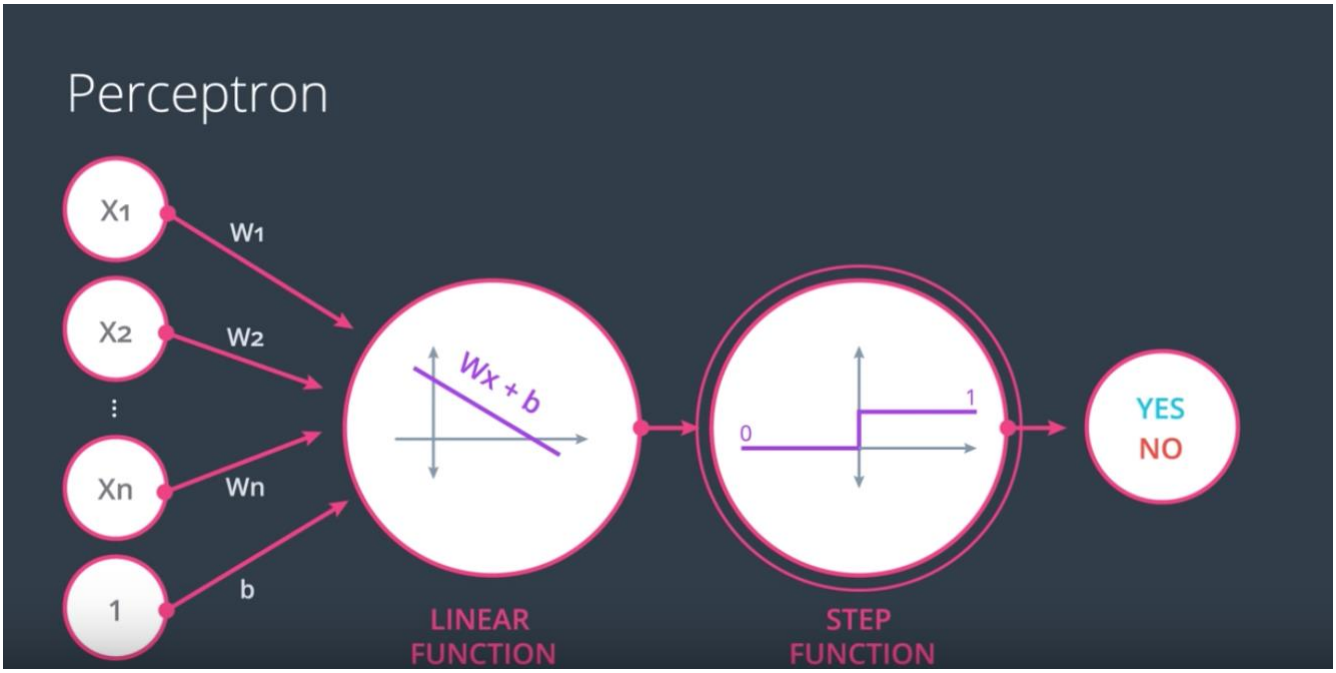


In the general case, this is how the nodes look. We will have our node over here then end inputs coming in with values X1 up to Xn and one, and edges with weights W1 up to Wn, and B corresponding to the bias unit. And then the node calculates the linear equation Wx plus B, which is a summation from I equals one to n, of WIXI plus B. This node then checks if the value is zero or bigger, and if it is, then the node returns a value of one for yes and if not, then it returns a value of zero for no.

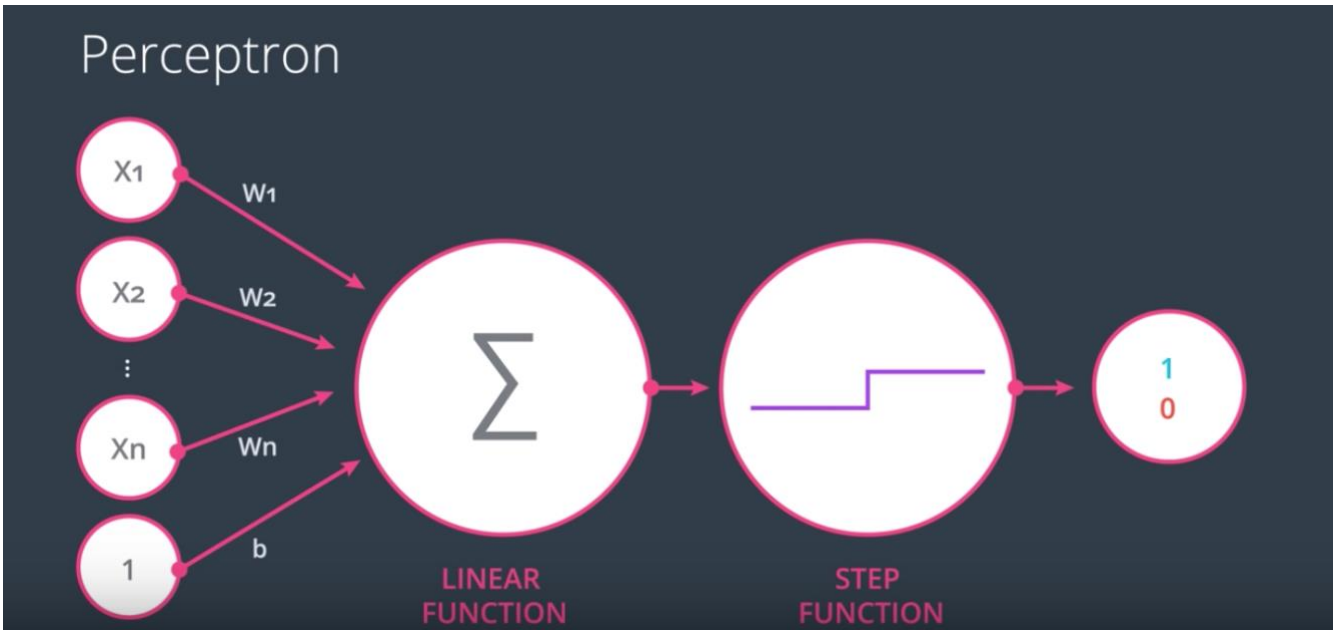


Correction: the definition of the Step function should be:
y=1 if x >= 0;
y=0 if x<0

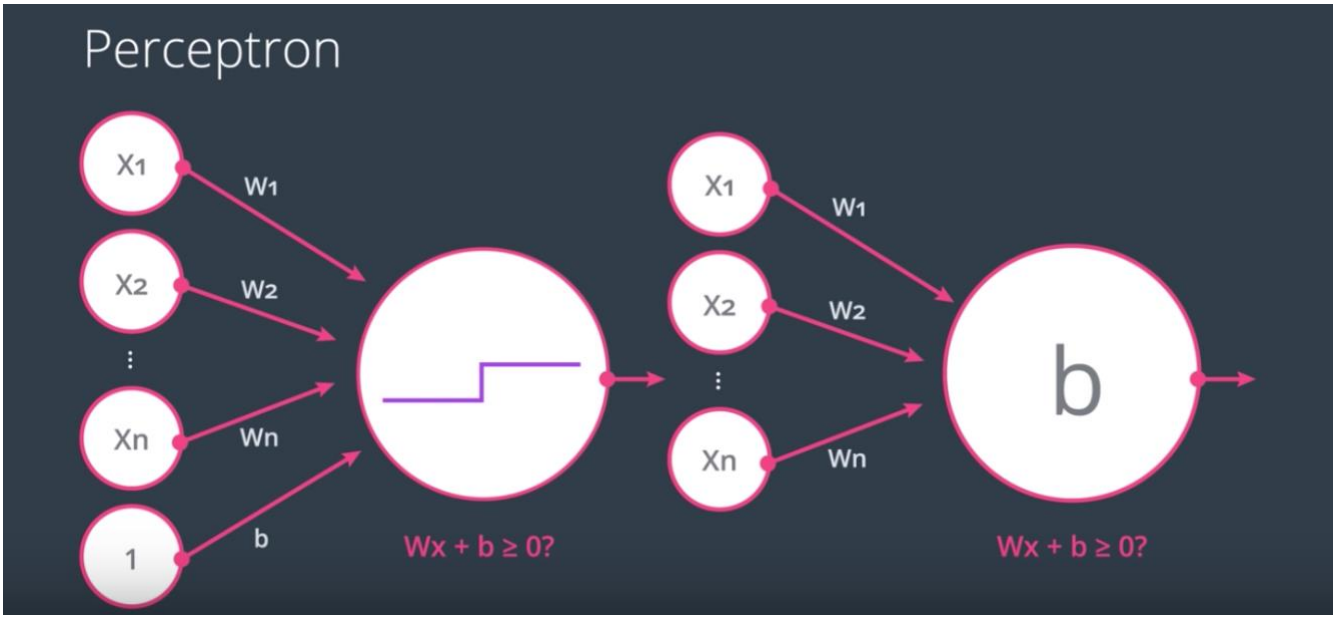
Note that we're using an implicit function, here, which is called a step function. What the step function does is it returns a one if the input is positive or zero, and a zero if the input is negative.



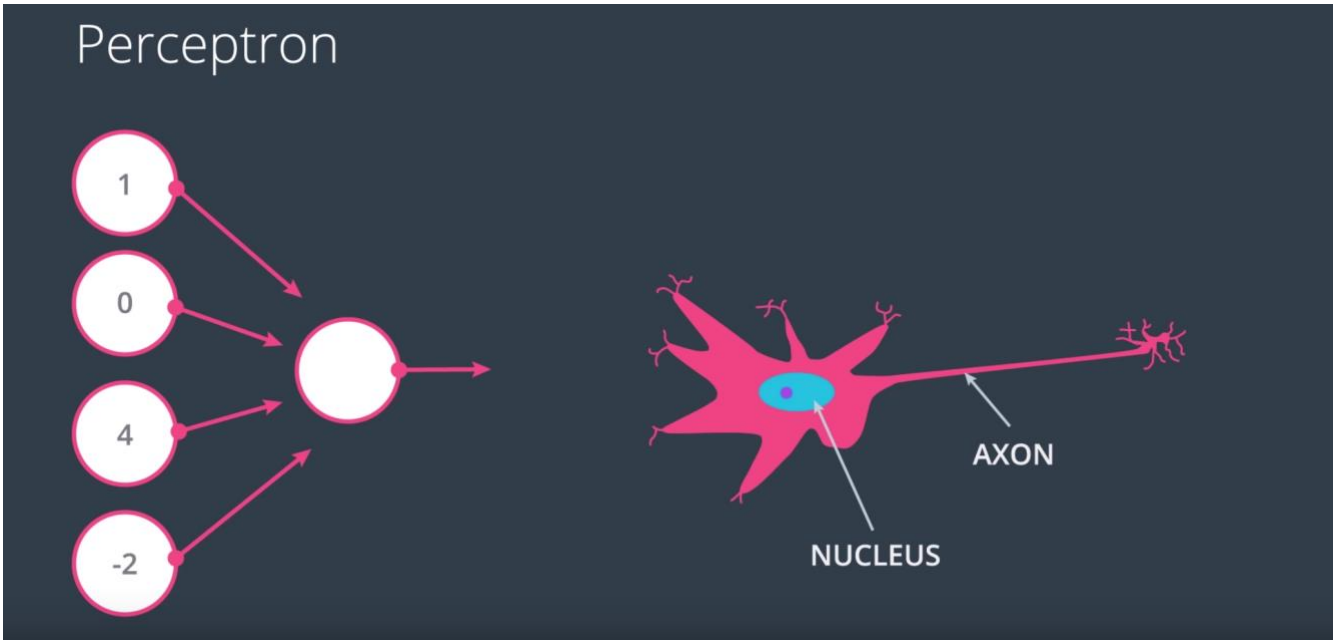
So in reality, these perceptrons can be seen as a combination of nodes, where the first node calculates a linear equation and the inputs on the weights, and the second node applies the step function to the result.



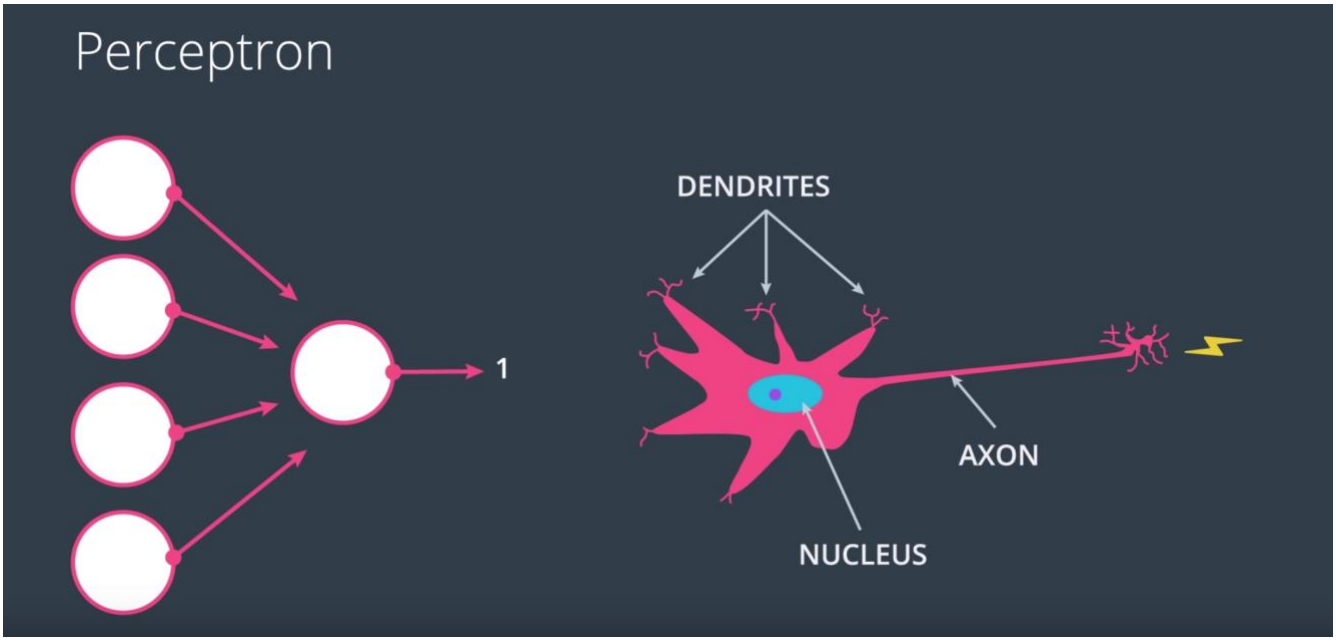
These can be graphed as follows: the summation sign represents a linear function in the first node, and the drawing represents a step function in the second node. In the future, we will use different step functions. So this is why it's useful to specify it in the node.



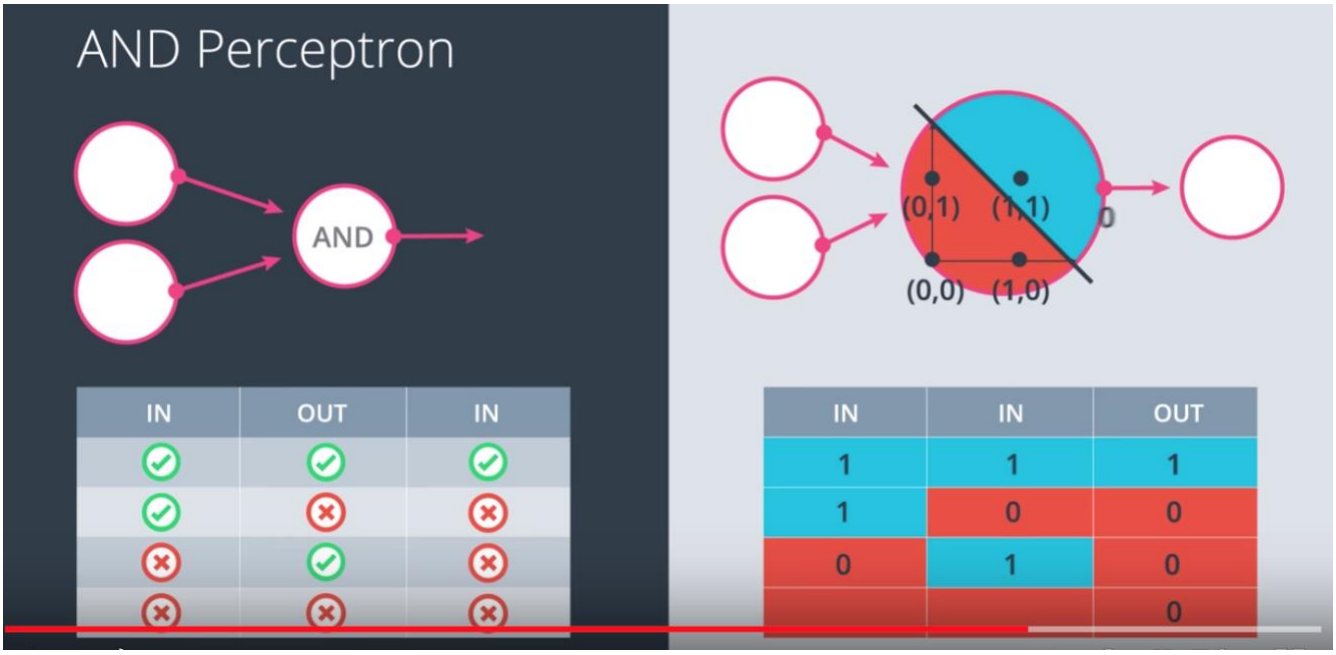
So as we've seen there are two ways to represent perceptions. The one on the left has a bias unit coming from an input node with a value of one, and the one in the right has the bias inside the node.



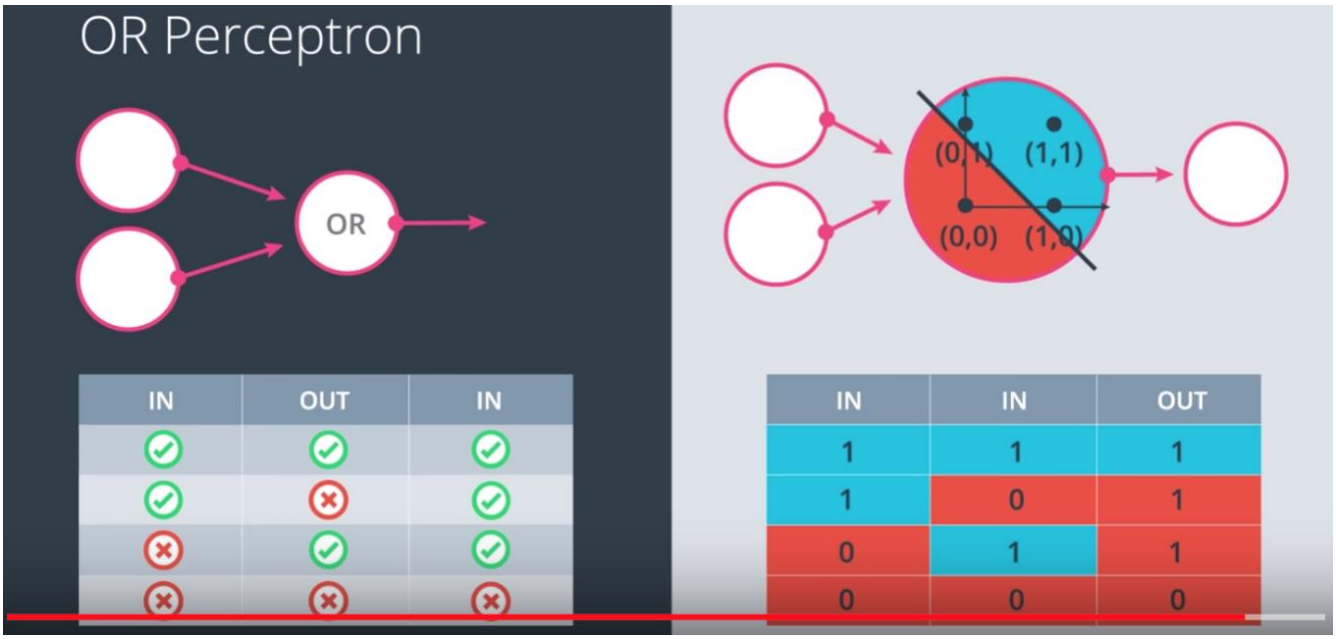
7. So you may be wondering why are these objects called neural networks. Well, the reason why they're called neural networks is because perceptions kind of look like neurons in the brain.



In the left we have a perception with four inputs. The number is one, zero, four, and minus two. And what the perception does, it calculates some equations on the input and decides to return a one or a zero. In a similar way neurons in the brain take inputs coming from the dendrites. These inputs are nervous impulses. So what the neuron does is it does something with the nervous impulses and then it decides if it outputs a nervous impulse or not through the axon. The way we'll create neural networks later in this lesson is by concatenating these perceptions so we'll be mimicking the way the brain connects neurons by taking the output from one and turning it into the input for another one.

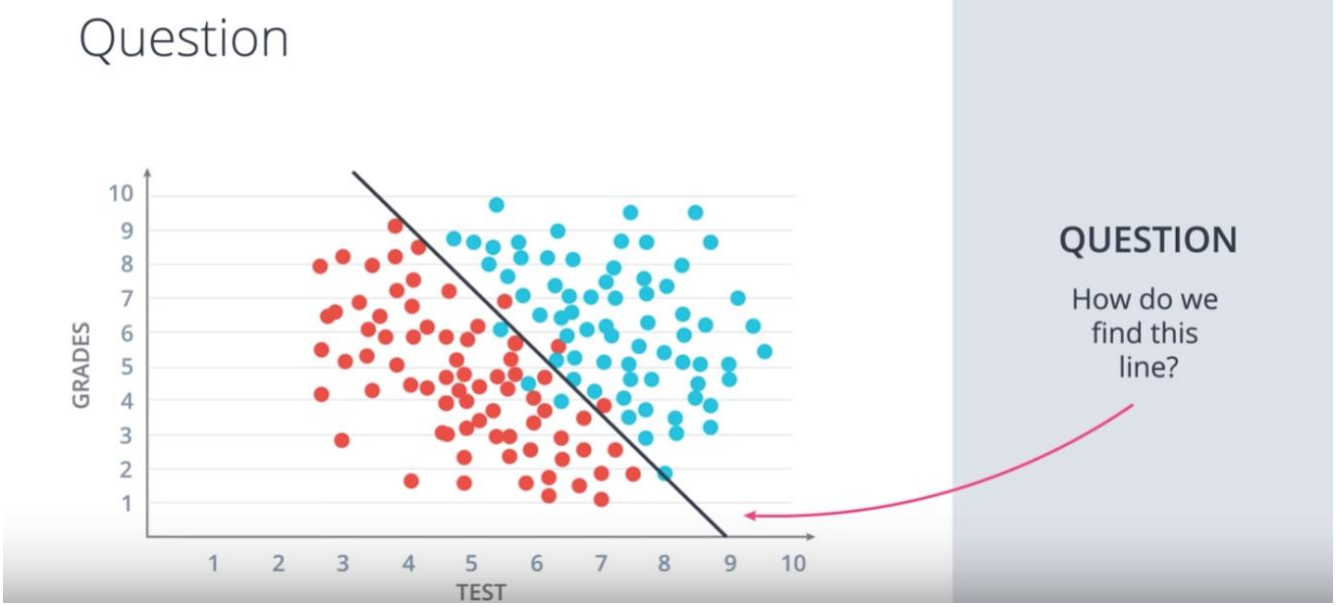


8. So, here's something very interesting about perceptrons, and it's that some logical operators can be represented as perceptrons. Here for example, we have the AND operator, and how does that work? The AND operator takes two inputs and it returns an output. The inputs can be true or false, but the output is only true if both of the inuts are true. So for instance, if the inputs are true and true, then the output is true. If the inputs are true and false, the the output is false. If the inputs are false and true, then the output is false. Finally, if the inputs are false and false, then the output is false. Now how do we turn this into a perceptron? Well the first step is to turn this table of true, false into a table of zeros and ones, where the one corresponds to true and zero corresponds to false. Now we draw this perceptron over here which works just as before. It has a line defined by weights and bias and it has a positive area, which is colored blue and a negative area which is colored red. What this perceptron is going to do is it'll plot each point. If the point falls in the positive area, then it returns a one, and if it falls in the negative area, then it returns a zero. So let's try. The one one gets plotted in the positive area. So the perceptron returns a one. The one zero gets plotted in the negative area. So the perceptron returns a zero. The zero one gets blurred in the negative area. So the perceptron returns a zero. Finally, the zero zero also gets plotted in the negative area. So the perceptron returns a zero.



Other logical operators can also be turned into perceptrons. For example, here is the OR operator which returns true if any of its two inputs is true. That gets turned into this table, which gets turned into this perceptron, which is very similar as the one before. Except the line has different weights on a different bias. What are the weights and bias for the AND and the OR perceptror? You'll have the chance to play with them in the quiz below.

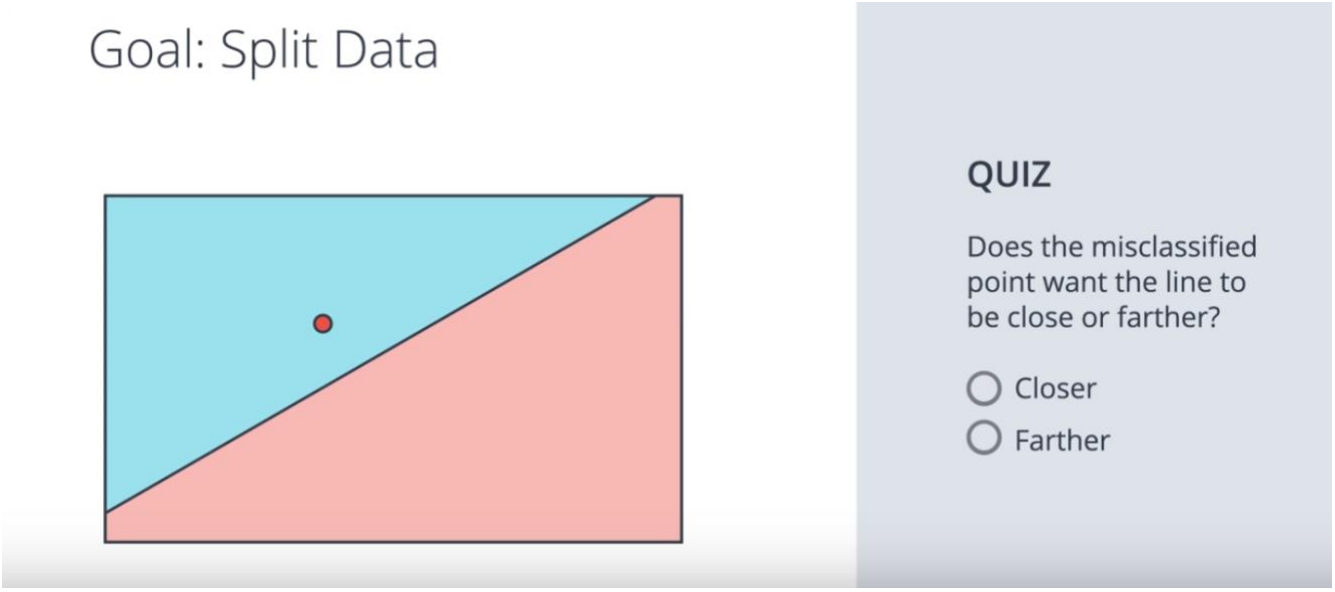
9. (It is the quiz in webpage, no video and no subtitle, and it is omitted by Tao.)



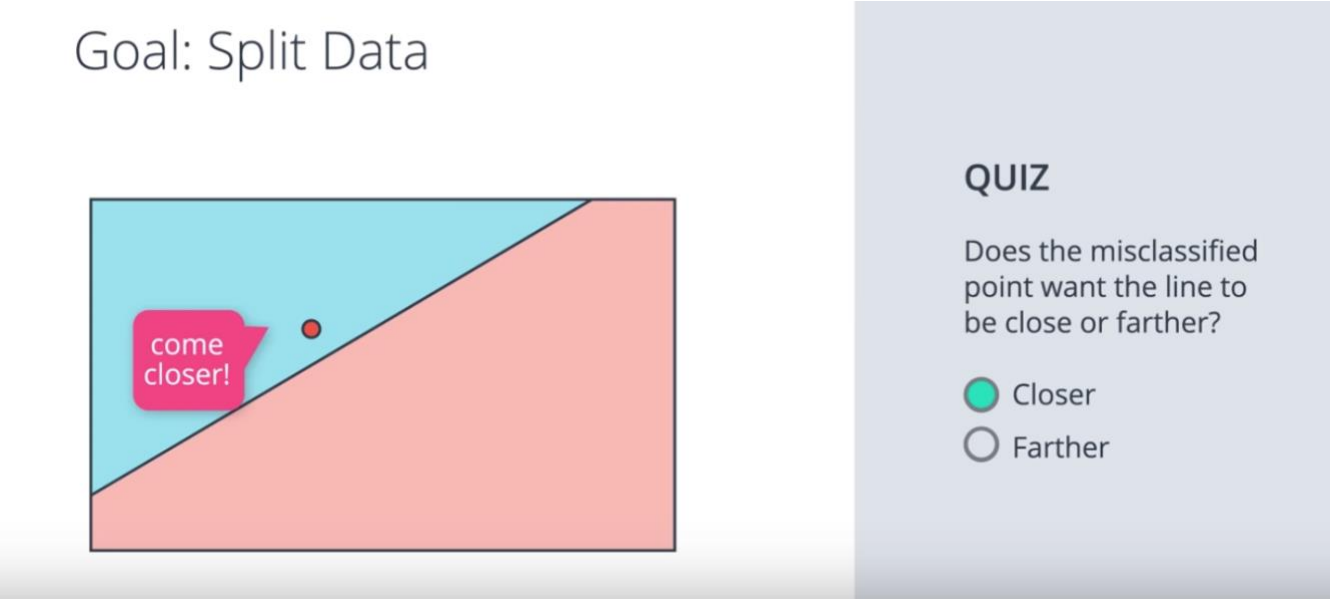
10. So we had a question we're trying to answer and the question is, how do we find this line that separates the blue points from the red points in the best possible way?

Let's answer this question by first looking at a small example with three blue points and three red points. And we're going to describe an algorithm that will find the line that splits these points properly.

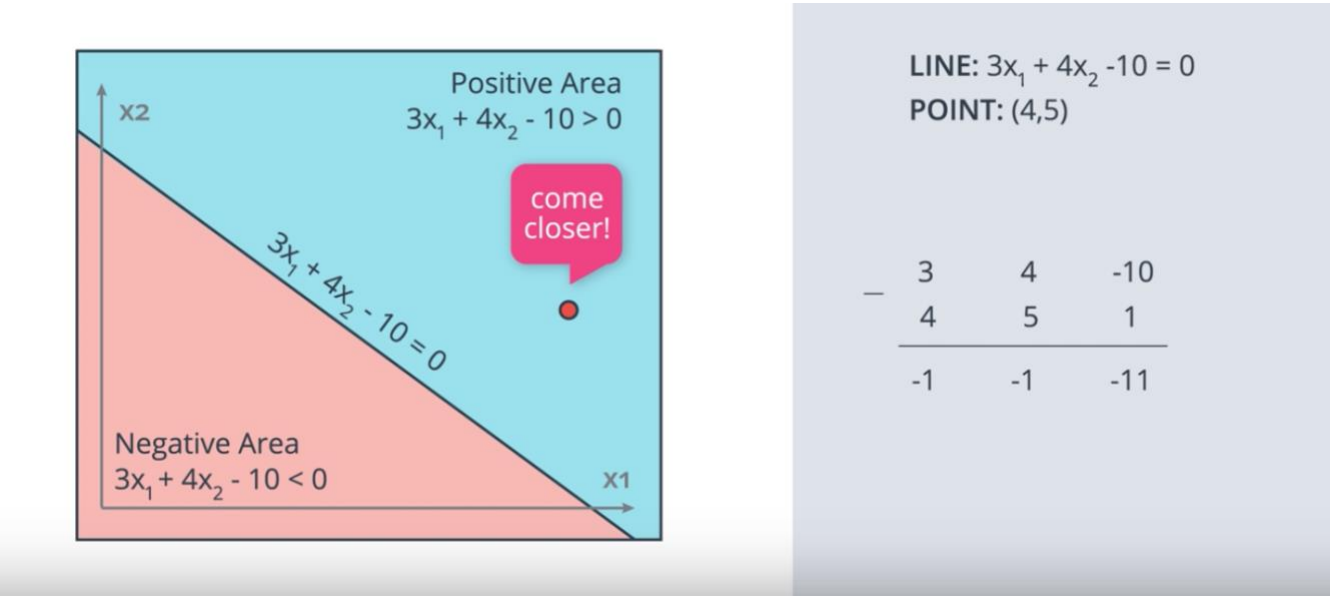
So the computer doesn't know where to start. It might as well start at a random place by picking a random linear equation. This equation will define a line and a positive and negative area given in blue and red respectively. What we're going to do is to look at how badly this line is doing and then move it around to try to get better and better. Now the question is, how do we find how badly this line is doing? So let's ask all the points. Here we have four points that are correctly classified. They are these two blue points in the blue area and these two red points in the red area. And these points are correctly classified, so they say, "I'm good." And then we have these two points that are incorrectly classified. That's this red point in the blue area and this blue point in the red area.



We want to get as much information from them so we want them to tell us something so that we can improve this line. So what is it that they can tell us? So here we have a misclassified point, this red point in the blue area. Now think about this. If you were this point, what would you tell the line to do? Would you like it to come closer to you or farther from you? That's our quiz. Will the misclassified point want the line to come closer to it or farther from it?

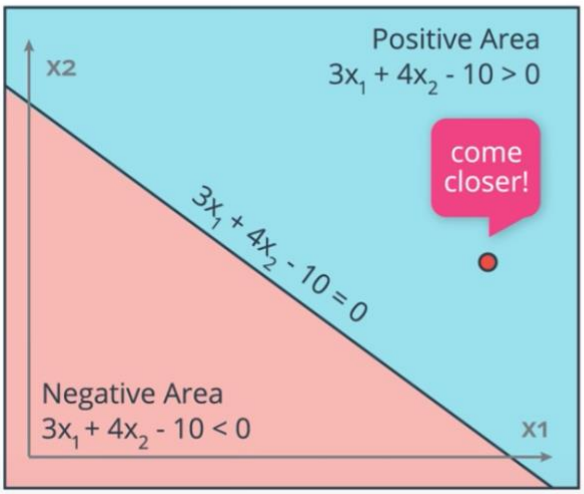


11. Well, consider this. If you're in the wrong area, you would like the line to go over you, in order to be in the right area. Thus, the points just come closer! So the line can move towards it and eventually classify it correctly.



12. Now, let me show you a trick that will make a line go closer to a point. Let's say we have our linear equation for example, 3x1 + 4x2 -10. And that linear equation gives us a line which is the

points where the equation is zero and two regions. The positive region drawn in blue where 3x1 + 4x2 - 10 is positive, and the negative region drawn in red with 3x1 + 4x2 - 10 is negative. So here we have our lonely misclassified point, the point (4, 5) which is a red point in the blue area, and the point has to come closer. So how do we get that point to come closer to the line? Well, the idea is we're going to take the four and five and use them to modify the equation of the line in order to get the line to move closer to the point. So here are parameters of the line 3, 4 and -10 and the coordinates of the point are 4 and 5, and let's also add a one here for the bias unit. So what we'll do is subtract these numbers from the parameters of the line to get 3 - 4, 4 - 5, and -10 -1. The new line will have parameters -1, -1, -11. And this line will move drastically towards the point, possibly even going over it and placing it in the correct area.

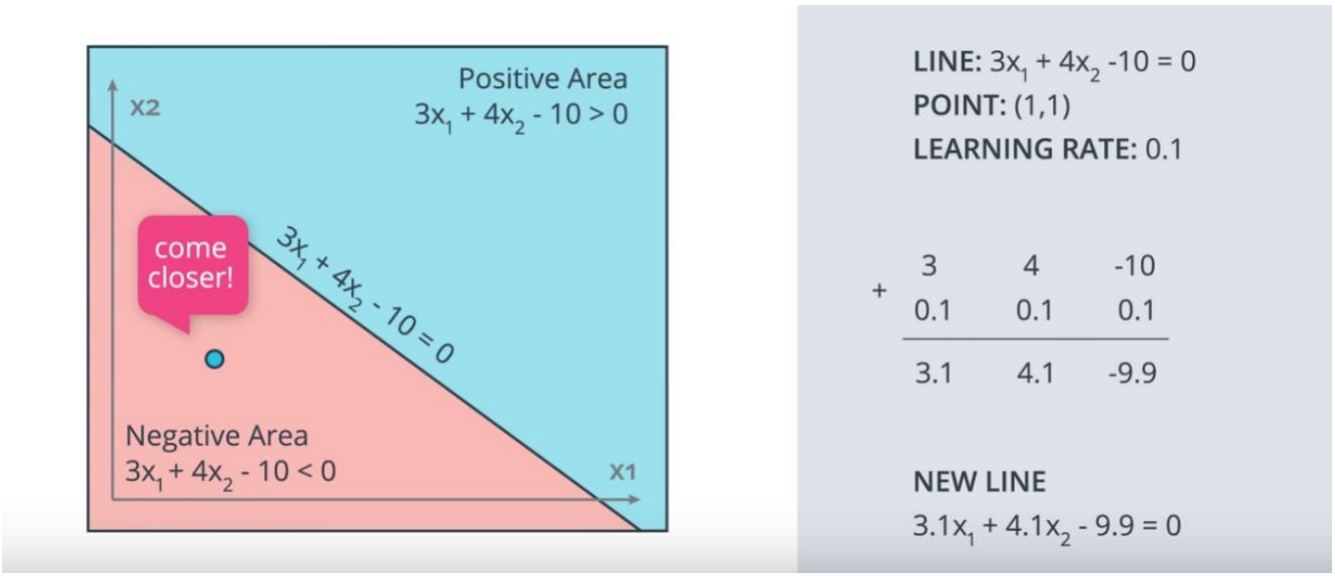


LINE: $3x_1 + 4x_2 - 10 = 0$
POINT: (4,5)
LEARNING RATE: 0.1

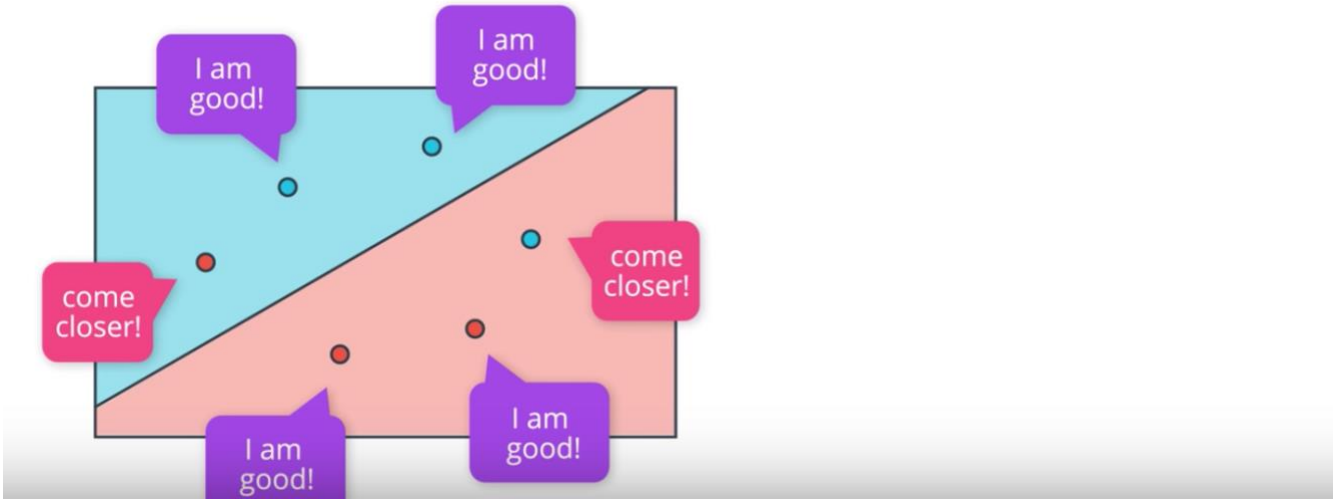
	3	4	-10
−	0.4	0.5	0.1
	2.6	3.5	-10.1

NEW LINE
$2.6x_1 + 3.5x_2 - 10.1 = 0$

Now, since we have a lot of other points, we don't want to make any drastic moves since we may accidentally misclassify all our other points. We want the line to make a small move towards that point and for this, we need to take small steps towards the point. So here's where we introduce the learning rate, the learning rate is a small number for example, 0.1 and what we'll do is instead of subtracting 4, 5 and 1 from the coordinates of the line, we'll multiply these numbers by 0.1 and then subtract them from the equation of the line. This means we'll be subtracting 0.4, 0.5, and 0.1 from the equation of the line. Obtaining a new equation of 2.6x1 + 3.5x 2 - 10.1 = 0. This new line will actually move closer to the point.

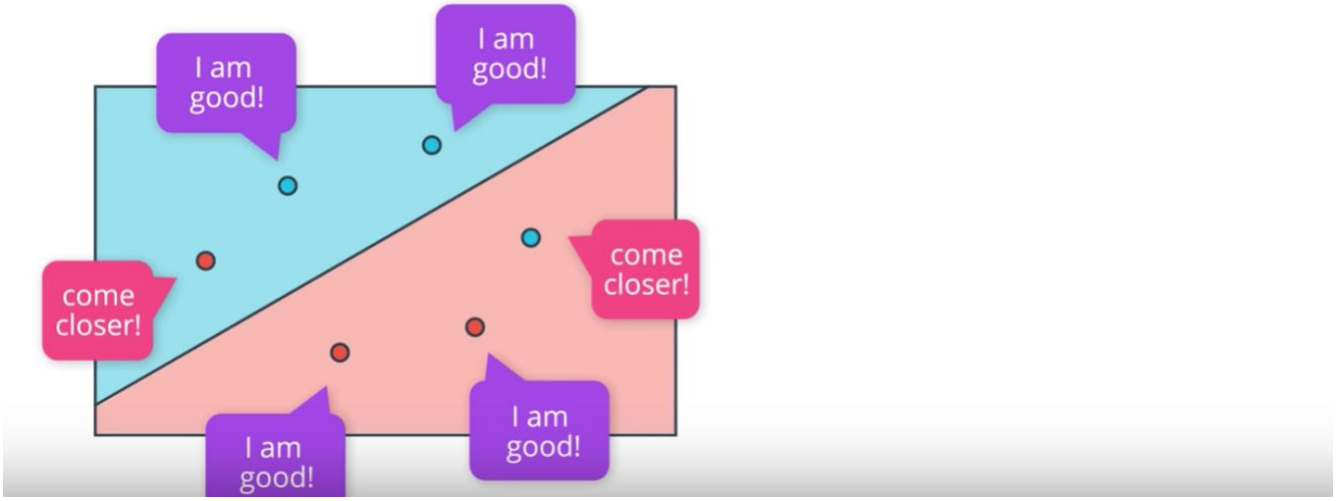


In the same way, if we have a blue point in the red area, for example, the point (1,1) is a positively labeled point in the negative area. This point is also misclassified and it says, come closer. So what do we do here is the same thing, except now instead of subtracting the coordinates to the parameters of the line, we add them. Again, we multiply by the learning rate in order to make small steps. So here we take the coordinates of the point (1,1) and put an extra one for the constant term and now, we multiply them by the learning rates 0.1. Now, we add them to the parameters of the line and we get a new line with equation 3.1x1 + 4.1x2 - 9.9. And magic, this line is closer to the point. So that's the trick we're going to use repeatedly for the Perceptron Algorithm.

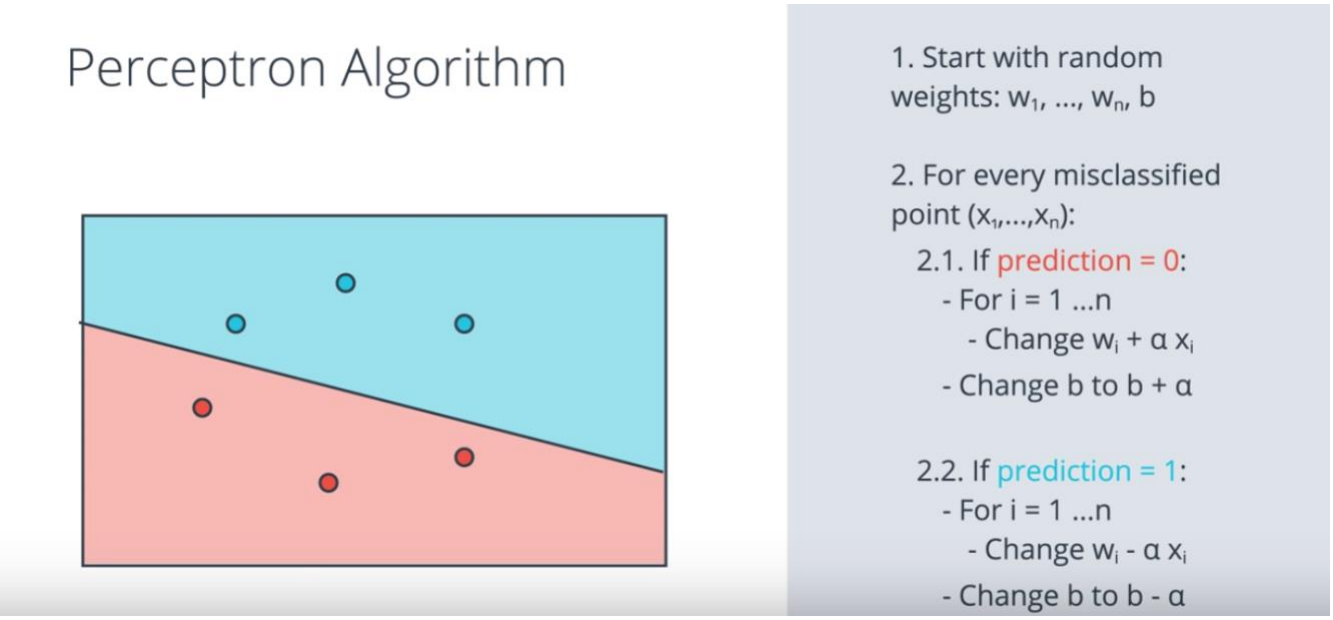


13. Now, we finally have all the tools for describing the perceptron algorithm. We start with the random equation, which will determine some line, and two regions, the positive and the negative region. Now, we'll move this line around to get a better and better fit. So, we ask all the points how they're doing. The four correctly classified points say, "I'm good." And the two incorrectly classified points say, "Come closer." So, let's listen to the point in the right, and apply the trick to

make the line closer to this point. So, here it is. Now, this point is good. Now, let's listen to the point in the left. The points says, "Come closer."



We apply the trick, and now the line goes closer to it, and it actually goes over it classifying correctly. Now, every point is correctly classified and happy.

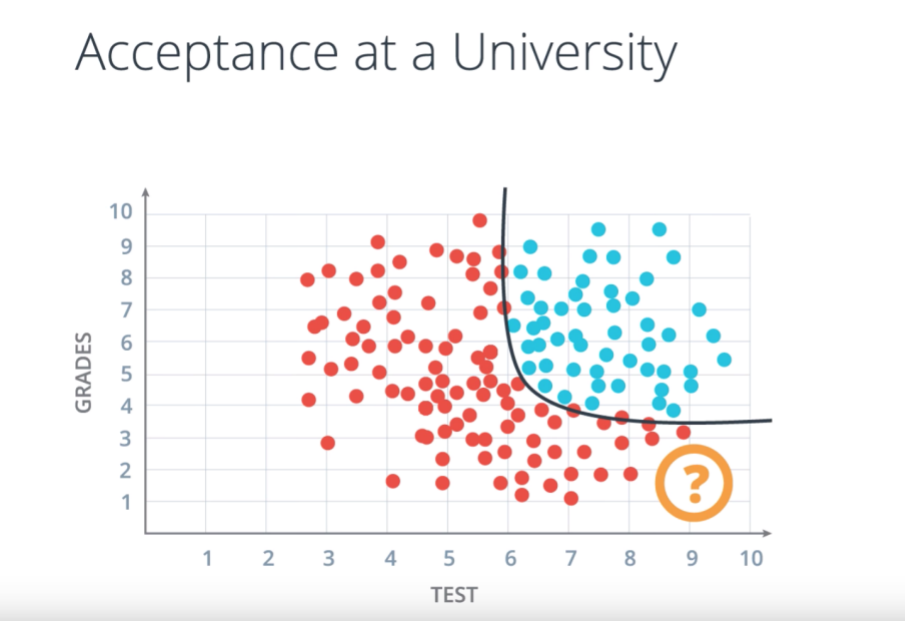


So, let's actually write the pseudocode for this perceptron algorithm. We start with random weights, w1 up to wn and b. This gives us the question wx plus b, the line, and the positive and negative areas. Now, for every misclassified point with coordinates x1 up to xn, we do the following. If the prediction was zero, which means the point is a positive point in the negative area, then we'll update the weights as follows: for i equals 1 to n, we change wi, to wi plus alpha times xi, where

alpha is the learning rate. In this case, we're using 0.1. Sometimes, we use 0.01 etc. It depends. Then we also change the bi as unit to b plus alpha. That moves the line closer to the misclassified point. Now, if the prediction was one, which means a point is a negative point in the positive area, then we'll update the weights in a similar way, except we subtract instead of adding. This means for i equals 1, change wi, to wi minus alpha xi, and change the bi as unit b to b minus alpha. And now, the line moves closer to our misclassified point. And now, we just repeat this step until we get no errors, or until we have a number of error that is small. Or simply we can just say, do the step a thousand times and stop. We'll see what are our options later in the class.

14. Okay, so let's look more carefully at this model for accepting and rejecting students. Let's say we have this student four, who got nine in the test, but only one on the grades. According to our

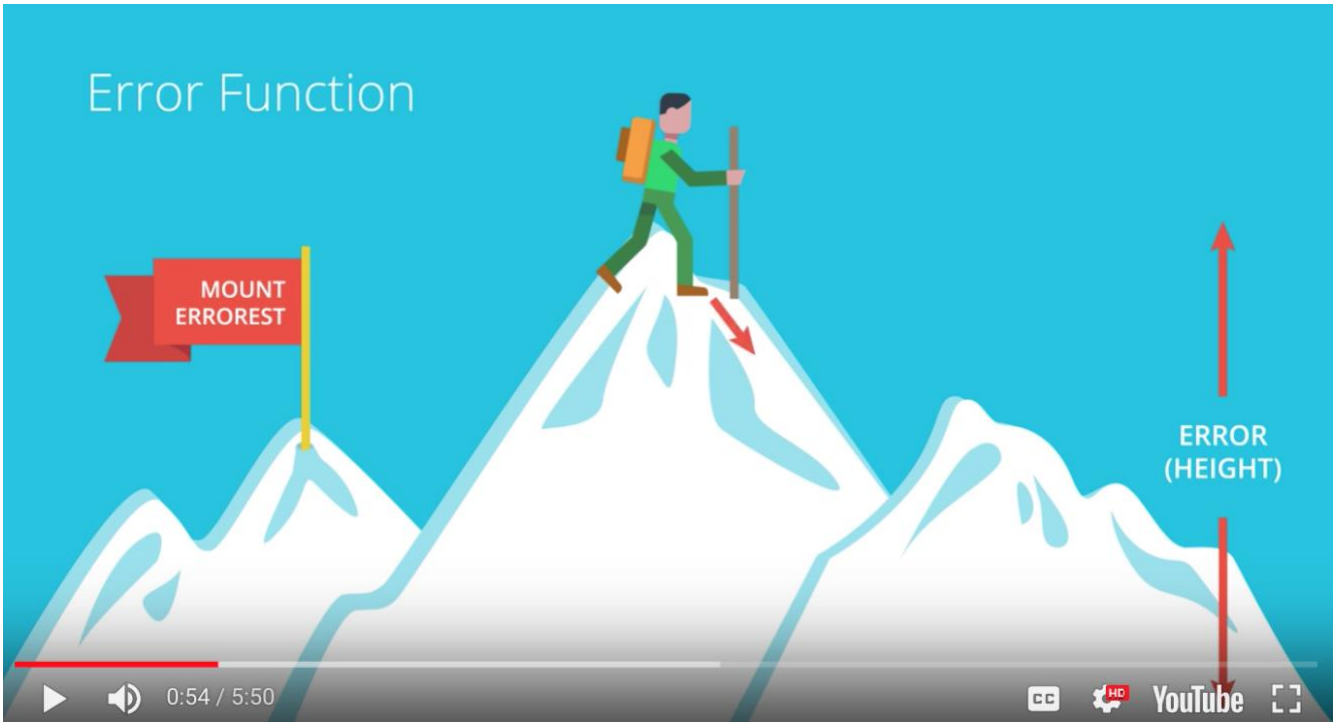
model this student gets accepted since it's placed over here in the positive region of this line. But let's say we don't want that since we'll say, "If your grades were terrible, no matter what you got on the test, you won't get accepted".



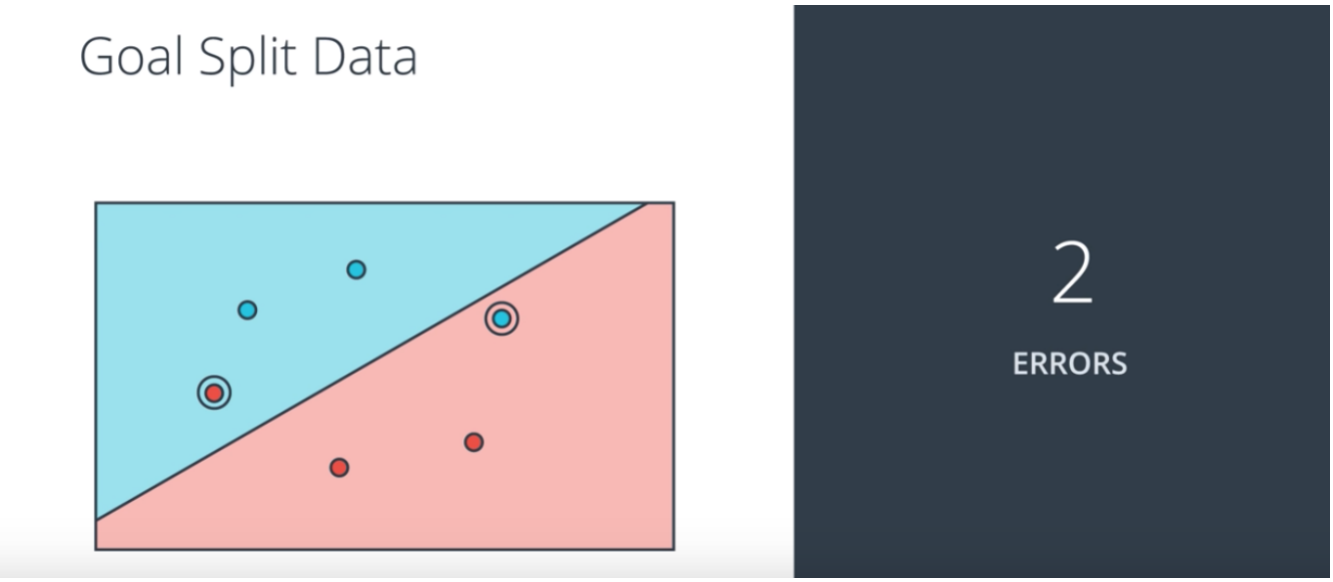
So our data should look more like this instead. This model is much more realistic but now we have a problem which is the data can no longer be separated by just a line. So what is the next thing after a line? Maybe a circle. A circle would work. Maybe two lines. That could work, too. Or maybe a curve like this. That would also work. So let's go with that. Let's go with the curve. Now, unfortunately, the perceptron algorithm won't work for us this time. We'll have to come up with something more complex and actually the solution will be, we need to redefine our perceptron algorithm for a line in a way that it'll generalize to other types of curves.

(No picture for the following paragraph 15)

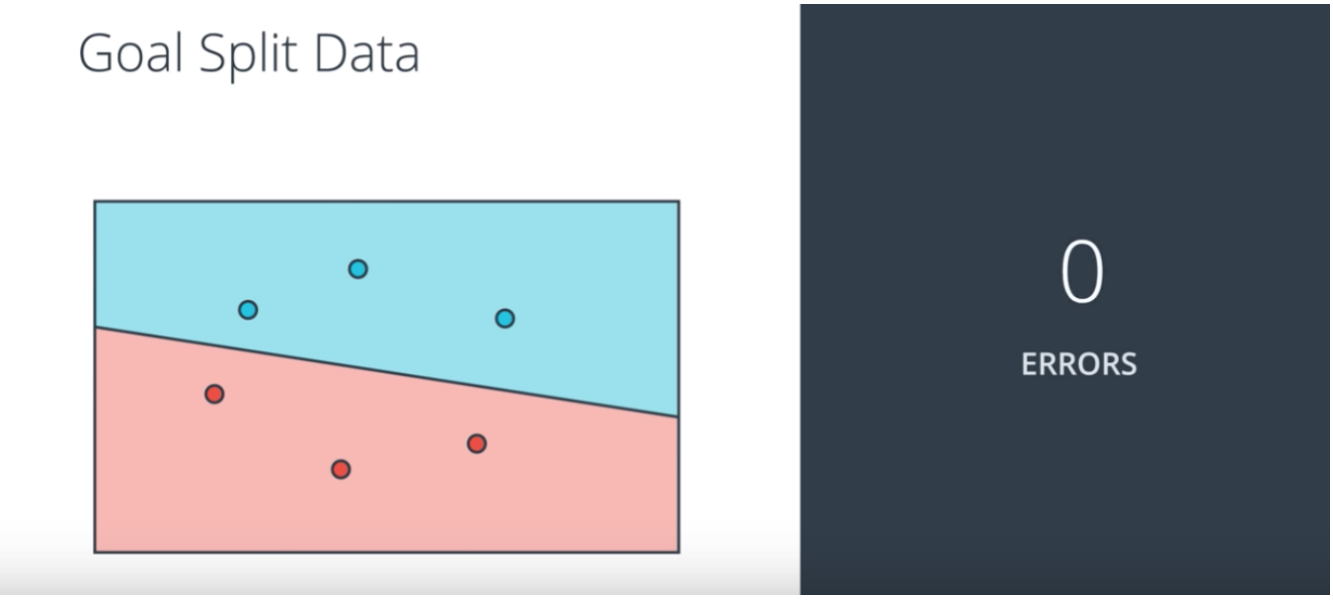
15. So the way we'll solve our problems from now on is with the help of an error function. An error function is simply something that tells us how far we are from the solution. For example, if I'm here and my goal is to get to this plant, an error function will just tell me the distance from the plant. My approach would then be to look around myself, check in which direction I can take a step to get closer to the plant, take that step and then repeat. Here the error is simply the distance from the plant.



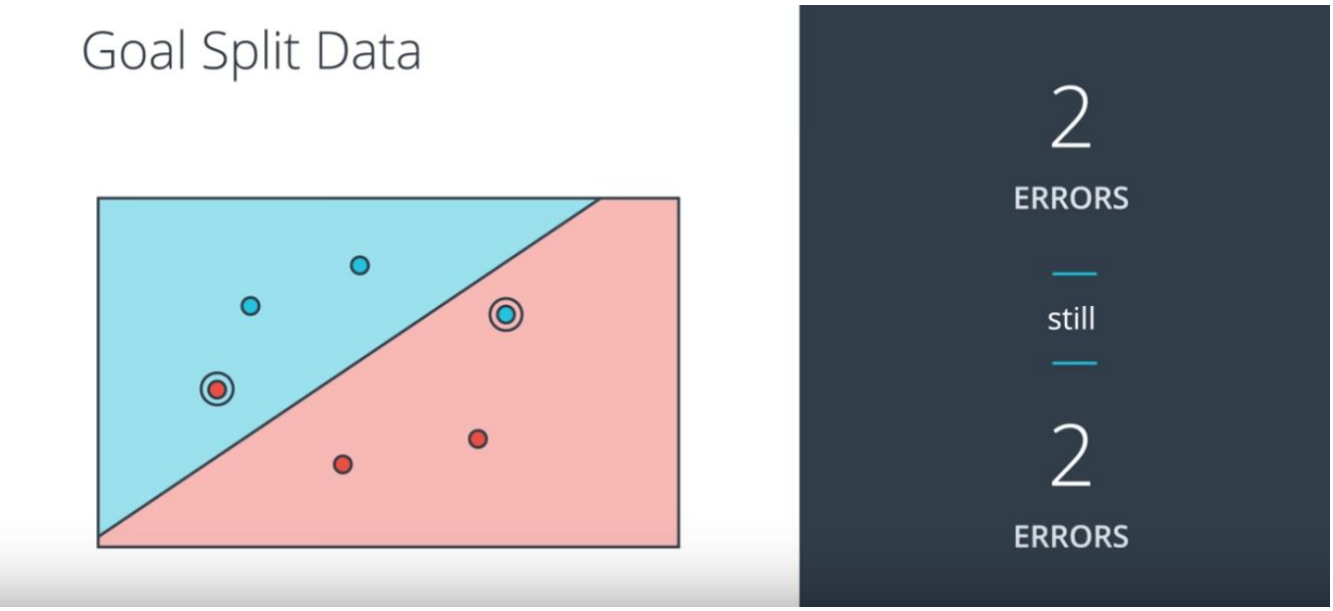
16. Here is obvious realization of the error function. We're standing on top a mountain, Mount Errorest and I want to descend but it's not that easy because it's cloudy and the mountain is very big, so we can't really see the big picture. What we'll do to go down is we'll look around us and we consider all the possible directions in which we can walk. Then we pick a direction that makes us descend the most. Let's say it's this one over here. So we take a step in that direction. Thus, we've decreased the height. Once we take the step and we start the process again and again always decreasing the height until we go all the way down the mountain, minimizing the height. In this case the key metric that we use to solve the problem is the height. We'll call the height the error. The error is what's telling us how badly we're doing at the moment and how far we are from an ideal solution. And if we constantly take steps to decrease the error then we'll eventually solve our problem, descending from Mt. Errorest. Some of you may be thinking, wait, that doesn't necessarily solve the problem. What if I get stuck in a valley, a local minimum, but that's not the bottom of the mountain. This happens a lot in machine learning and we'll see different ways to solve it later in this Nanodegree. It's also worth noting that many times a local minimum will give us a pretty good solution to a problem. This method, which we'll study in more detail later, is called gradient descent.



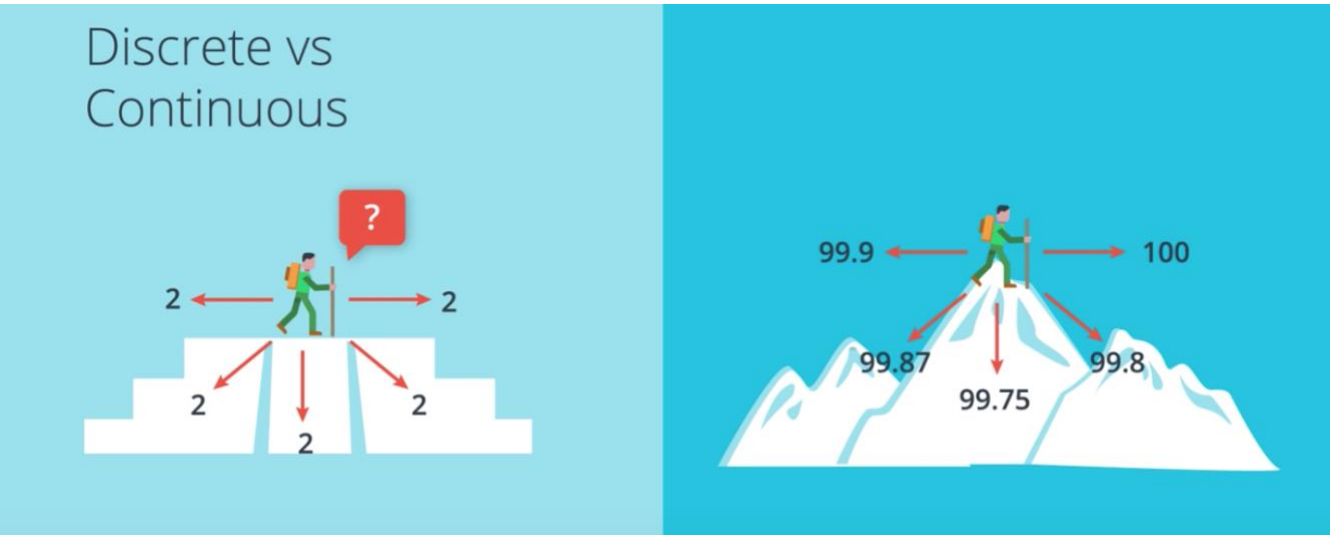
So let's try that approach to solve a problem. What would be a good error function here? What would be a good way to tell the computer how badly it's doing? Well, here's our line with our positive and negative area. And the question is how do we tell the computer how far it is from a perfect solution? Well, maybe we can count the number mistakes. There are two mistakes here. So that's our height. That's our error. So just as we did to descend from the mountain, we look around all the directions in which we can move the line in order to decrease our error.



So let's say we move in this direction. We'll decrease the number of errors to one and then if we're moving in that direction, we'll decrease the number of errors to zero.

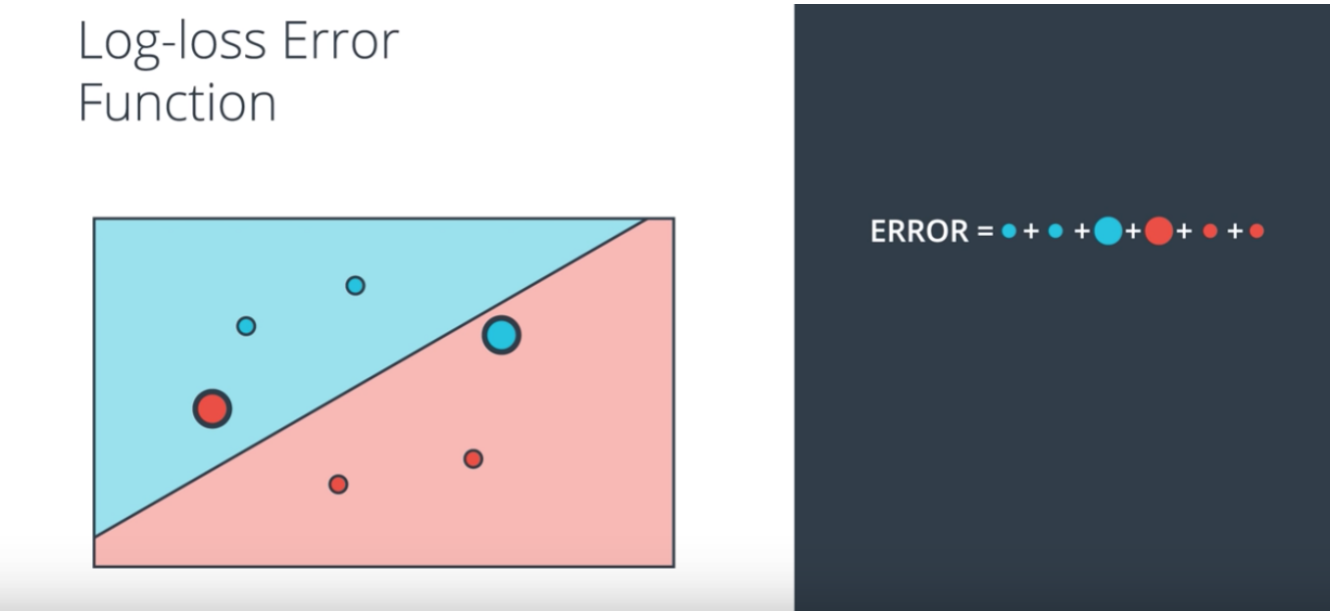


And then we're done, right? Well, almost. There's a small problem with that approach. In our algorithms we'll be taking very small steps and the reason for that is calculus, because our tiny steps will be calculated by derivatives. So what happens if we take very small steps here? We start with two errors and then move a tiny amount and we're still at two errors. Then move a tiny amount again and we're still two errors. Another tiny amount and we're still at two and again and again. So not much we can do here.

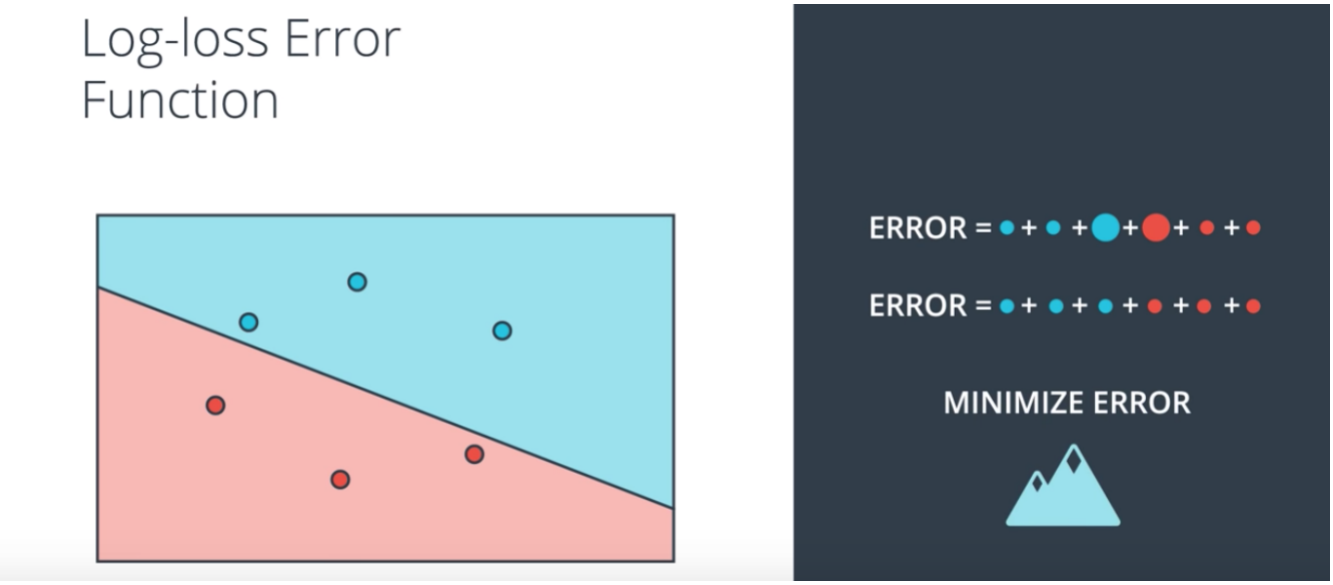


This is equivalent to using gradient descent to try to descend from an Aztec pyramid with flat steps. If we're standing here in the second floor, for the two errors and we look around ourselves, we'll always see two errors and we'll get confused and not know what to do. On the other hand in Mt. Errorest we can detect very small variations in height and we can figure out in what direction it can decrease the most. In math terms this means that in order for us to do gradient descent our error function can not be discrete, it should be continuous. Mt. Errorest is continuous since small

variations in our position will translate to small variations in the height but the Aztec pyramid does not since the high jumps from two to one and then from one to zero. As a matter of fact, our error function needs to be differentiable, but we'll see that later.



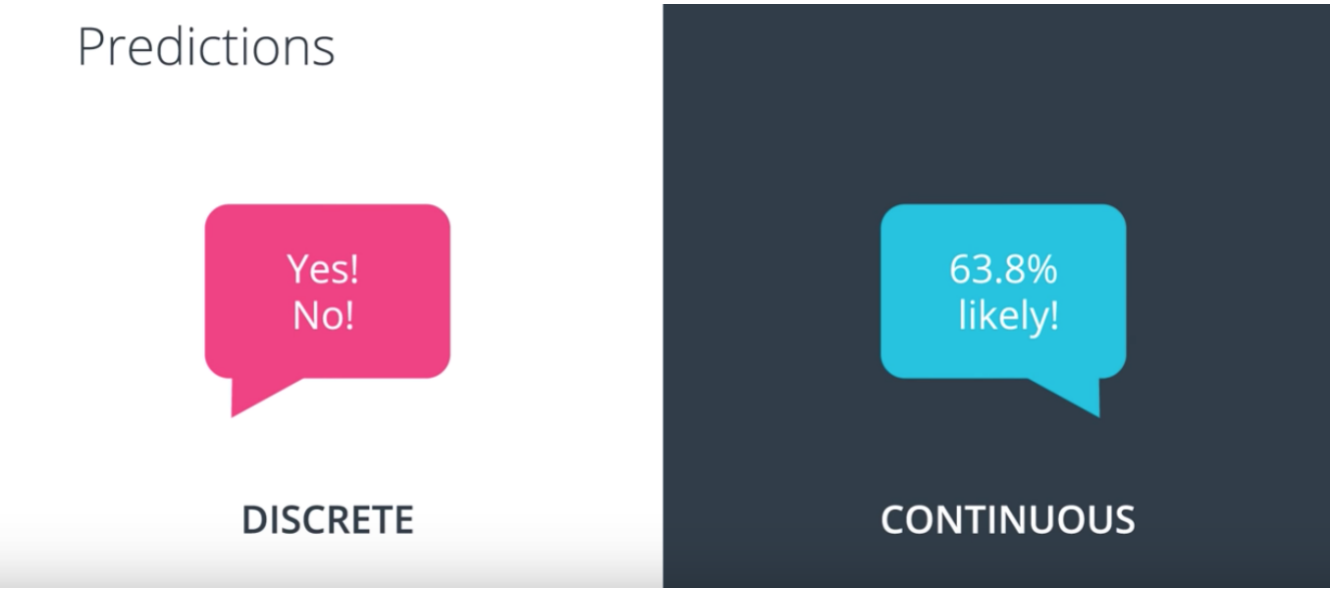
So, what we need to do here is to construct an error function that is continuous and we'll do this as follows. So here are six points with four of them correctly classified, that's two blue and two red, and two of them incorrectly classified, that is this red point at the very left and this blue point at the very right. The error function is going to assign a large penalty to the two incorrectly classified points and small penalties to the four correctly classified points. Here we are representing the size of the point as the penalty. The penalty is roughly the distance from the boundary when the point is misclassified and almost zero when the point is correctly classified. We'll learn the formula for the error later in the class.



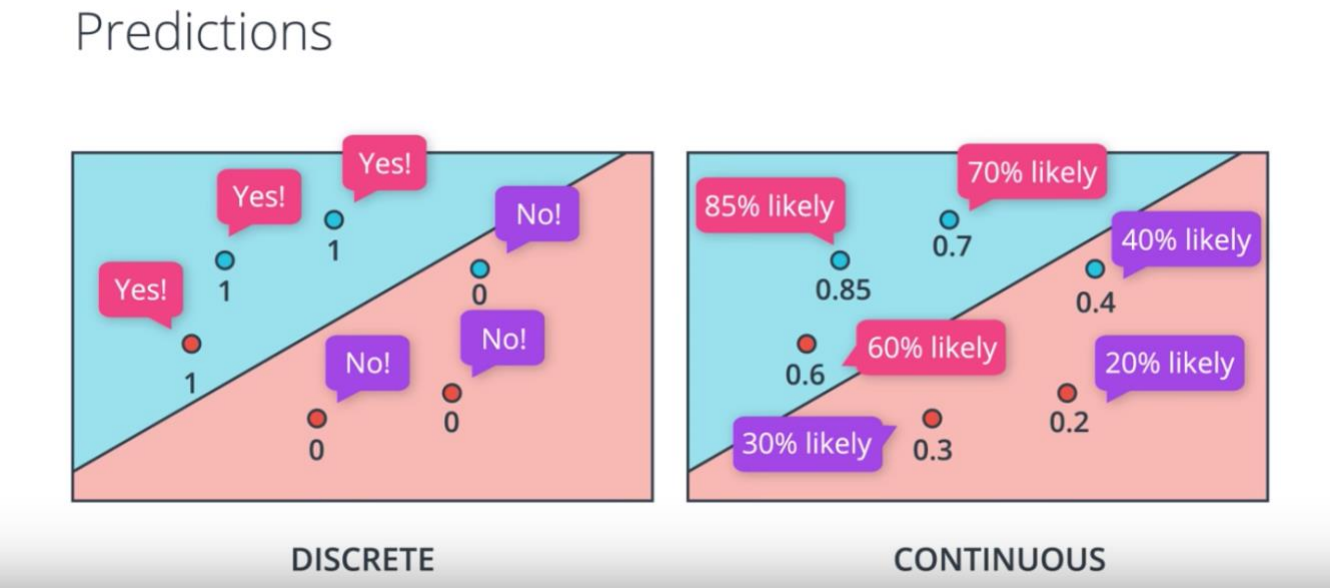
So, now we obtain the total error by adding all the errors from the corresponding points. Here we have a large number so it is two misclassified points add a large amount to the error. And the idea now is to move the line around in order to decrease these error. But now we can do it because we can make very tiny changes to the parameters of the line which will amount to very tiny changes in the error function. So, if you move the line, say, in this direction, we can see that some errors decrease, some slightly increase, but in general when we consider the sum, the sum gets smaller and we can see that because we've now correctly classified the two points that were misclassified before. So once we are able to build an error function with this property, we can now use gradient descent to solve our problem. So here's the full picture. Here we are at the summit of Mt. Errorest. We're quite high up because our error is large. As you can see the error is the height which is the sum of the blue and red areas. We explore around to see what direction brings us down the most, or equivalently, what direction can we move the line to reduce the error the most, and we take a step in that direction. So in the mountain we go down one step and in the graph we've reduced the error a bit by correctly classifying one of the points. And now we do it again. We calculate the error, we look around ourselves to see in what direction we descend the most, we take a step in that direction and that brings us down the mountain. So on the left we have reduced the height and successfully descended from the mountain and on the right we have reduced the error to its minimum possible value and successfully classified our points. Now the question is, how do we define this error function? That's what we'll do next.

(No picture for the following paragraph 17)

17. In the last section we pointed out the difference between a discrete and a continuous error function and discovered that in order for us to use gradient descent we need a continuous error function. In order to do this we also need to move from discrete predictions to continuous predictions. Let me show you what I mean by that.

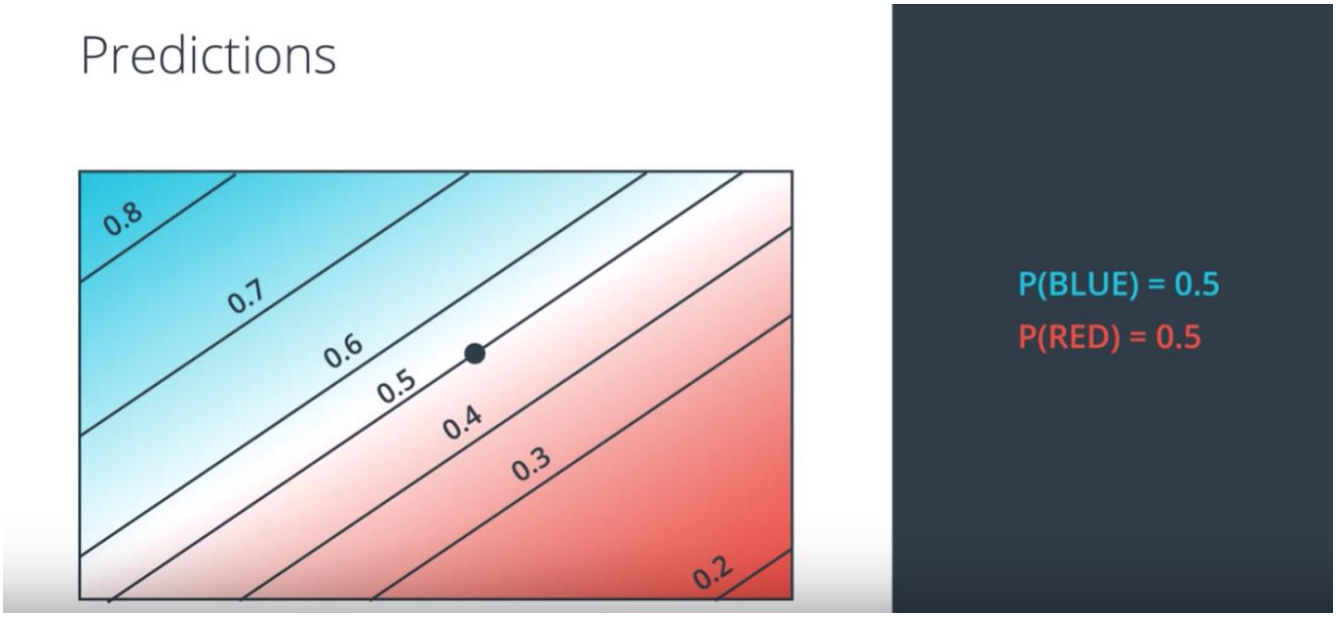


18. The prediction is basically the answer we get from the algorithm. A discreet answer will be of the form yes, no. Whereas a continued answer will be a number, normally between zero and one which we'll consider a probability.



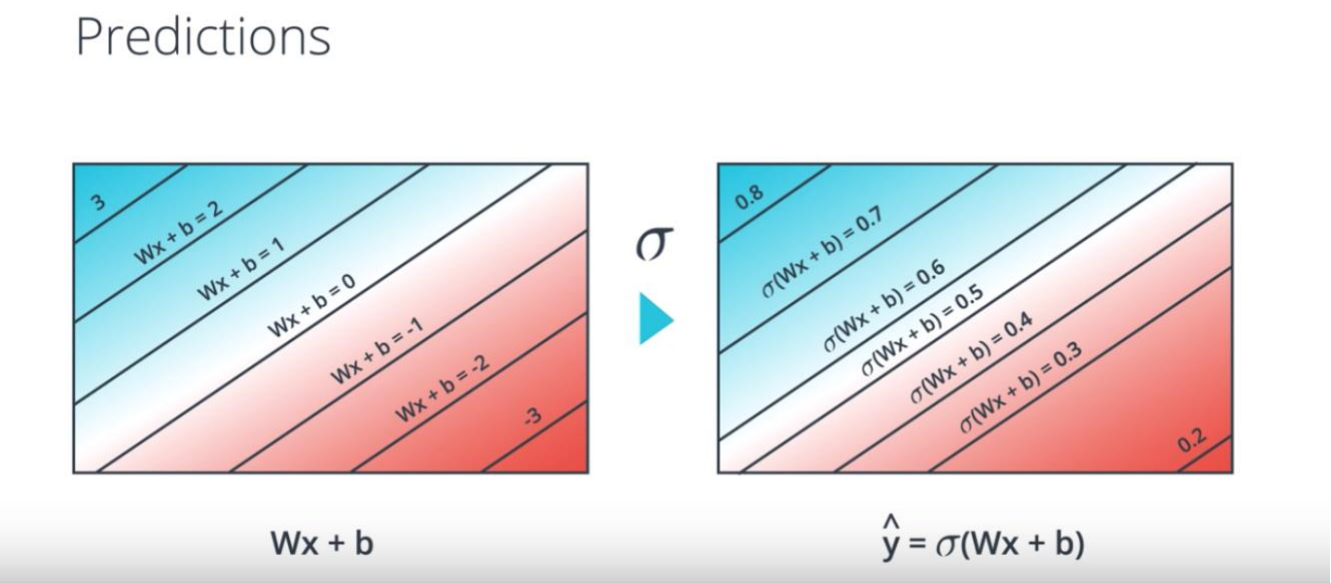
In the running example, here we have our students where blue is accepted and red is rejected. And the discrete algorithm will tell us if a student is accepted or rejected by typing a zero for rejected students and a one for accepted students. On the other hand, the farther our point is from the black line, the more drastic these probabilities are. Points that are well into the blue area get very high probabilities, such as this point with an 85% probability of being blue. And points that are well into the red region are given very low probabilities, such as this point on the bottom that is given a 20% probability of being blue. The points over the line are all given a 50% probability of being blue. As you can see the probability is a function of the distance from the line.

The way we move from discrete predictions to continuous, is to simply change your activation function from the step function in the left, to the sigmoid function on the right. The sigmoid function is simply a function which for large positive numbers will give us values very close to one. For large negative numbers will give us values very close to zero. And for numbers that are close to zero, it'll give you values that are close to point five. The formula is sigmoid effects equals $\sigma(x) = 1/(1 + \exp(-x))$

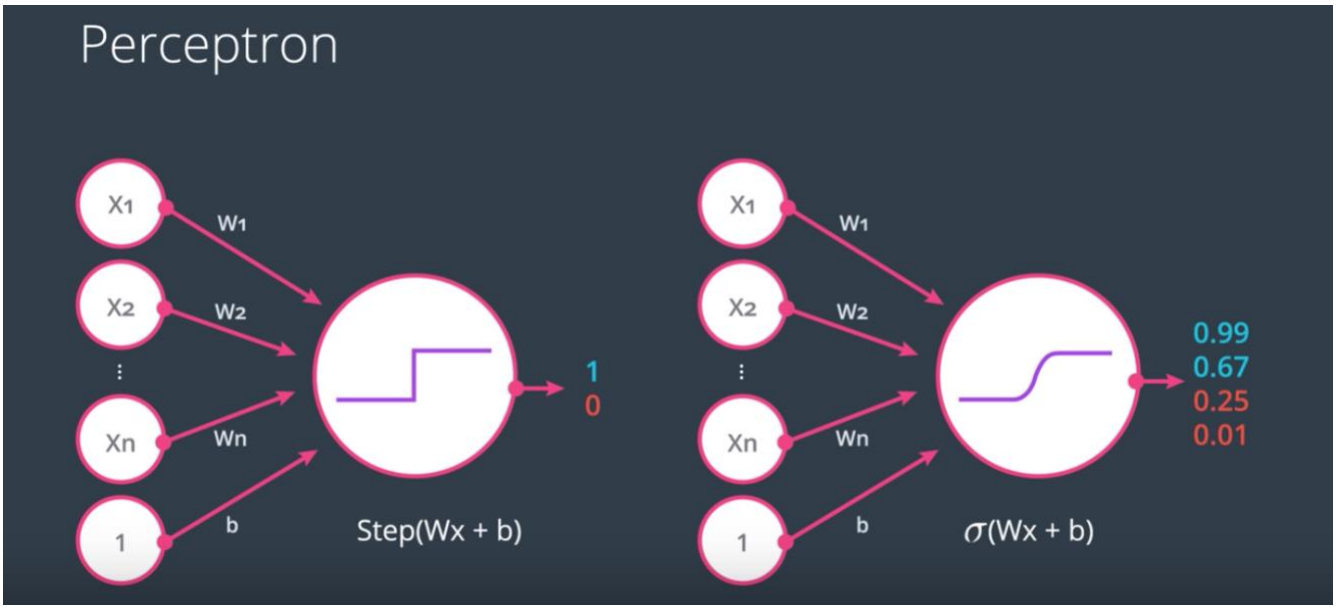


So, before our model consisted of a line with a positive region and a negative region. Now it consists of an entire probability space or for each point in the plane we are given the probability that the label of the point is one for the blue points, and zero for the red points. For example, for

this point the probability of being blue is 50% and of being red is 50%. For this point, the probabilities are 40% for being blue, and 60% for being red. For this one over here it's 30% for blue, and 70% for red. And for this point all over here is 80% for being blue and 25 percent for being red.



The way we obtain this probability space is very simple. We just combine the linear function WX + b with the sigmoid function. So in the left we have the lines that represent the points for which WX + b is zero, one, two, minus one, minus two, etc. And once we apply the sigmoid function to each of these values in the plane, we then obtain numbers from zero to one for each point. These numbers are just the probabilities of the point being blue. The probability of the point being blue is a prediction of the model Y hat to sigmoid of W x plus b. Here we can see the lines for which the prediction is point five, point six, point seven, point four, point three, et cetera. As you can see, as we get more into the blue area, σ(Wx + b) gets closer and closer to one. And as we move into the red area, σ(Wx + b) gets closer and closer to zero. When we're over the main line, W x plus b is zero, which means sigmoid of W s plus b is exactly zero point five.

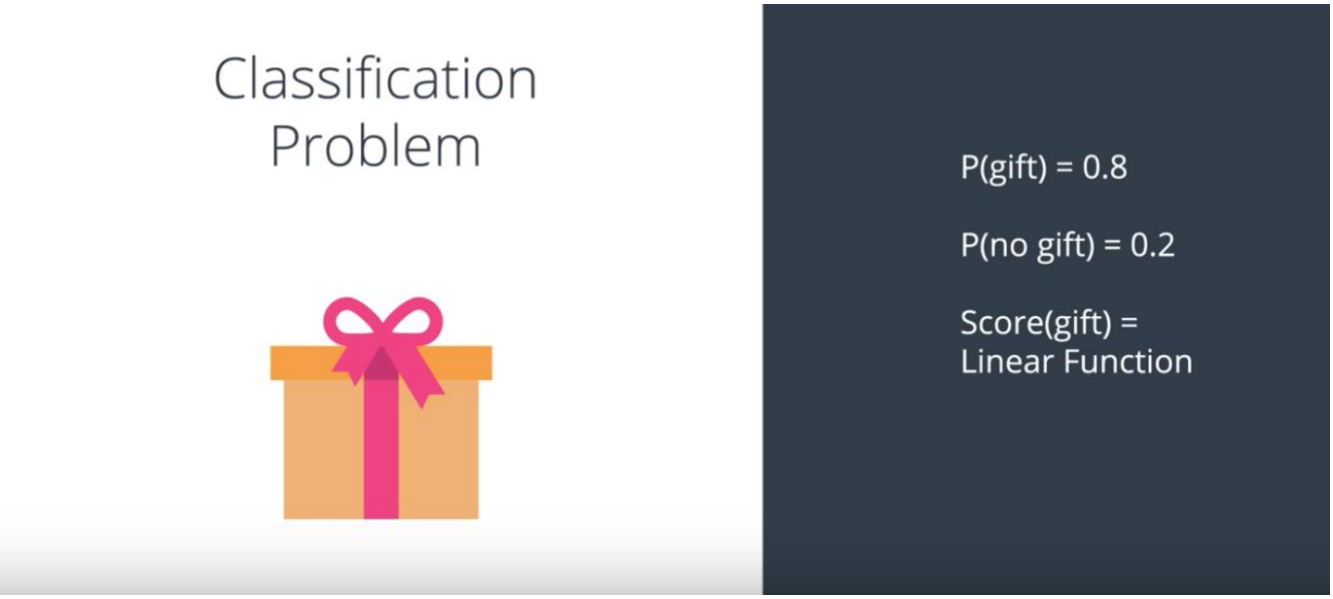


So here on the left we have our old perceptron with the activation function as a step function. And on the right we have our new perceptron, where the activation function is the sigmoid function. What our new perceptron does, it takes the inputs, multiplies them by the weights in the edges and adds the results, then applies the sigmoid function. So instead of returning one and zero like before it returns values between zero and one such as 0.99 or 0.67 etc. Before it used to say the student got accepted or not, and now it says the probability of the student got accepted is this much.

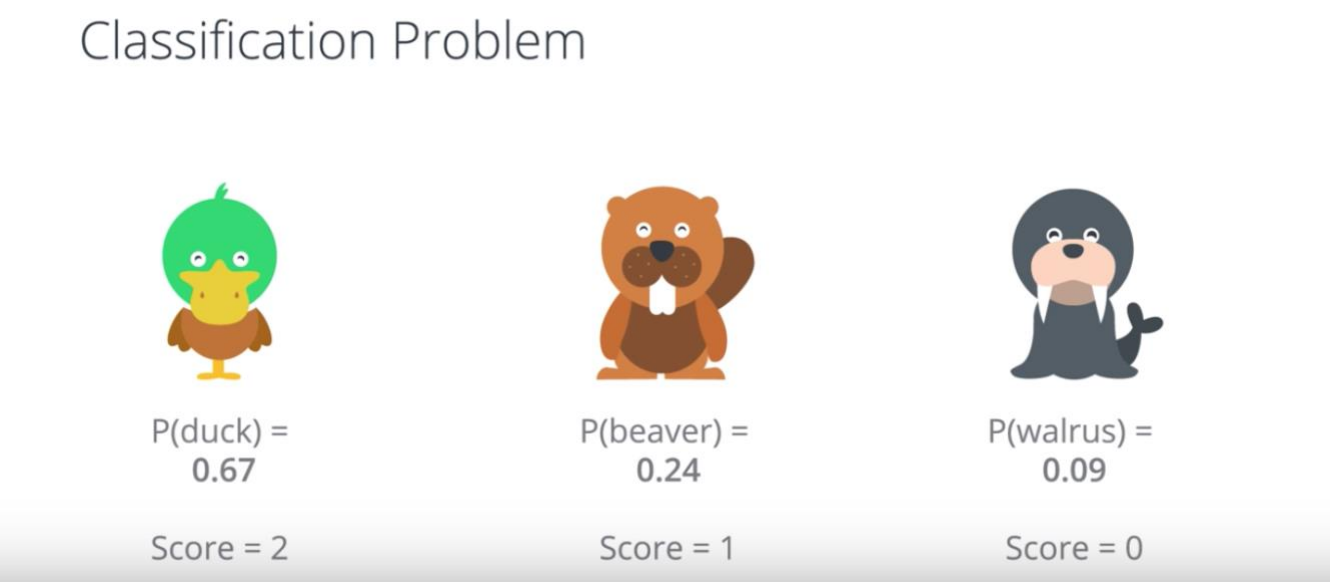
Tao: the following paragraphs 19-21are just arguing why it is reasonable to use sigmoid function.

(No picture for the following paragraph 19)

19. So far we have models that give us an answer of yes/no or the probability of a label being positive or negative. What if we have more classes? What if we want our model to tell us if something is red, blue, yellow or dog, cat, bird? In this video I'll show you what to do.

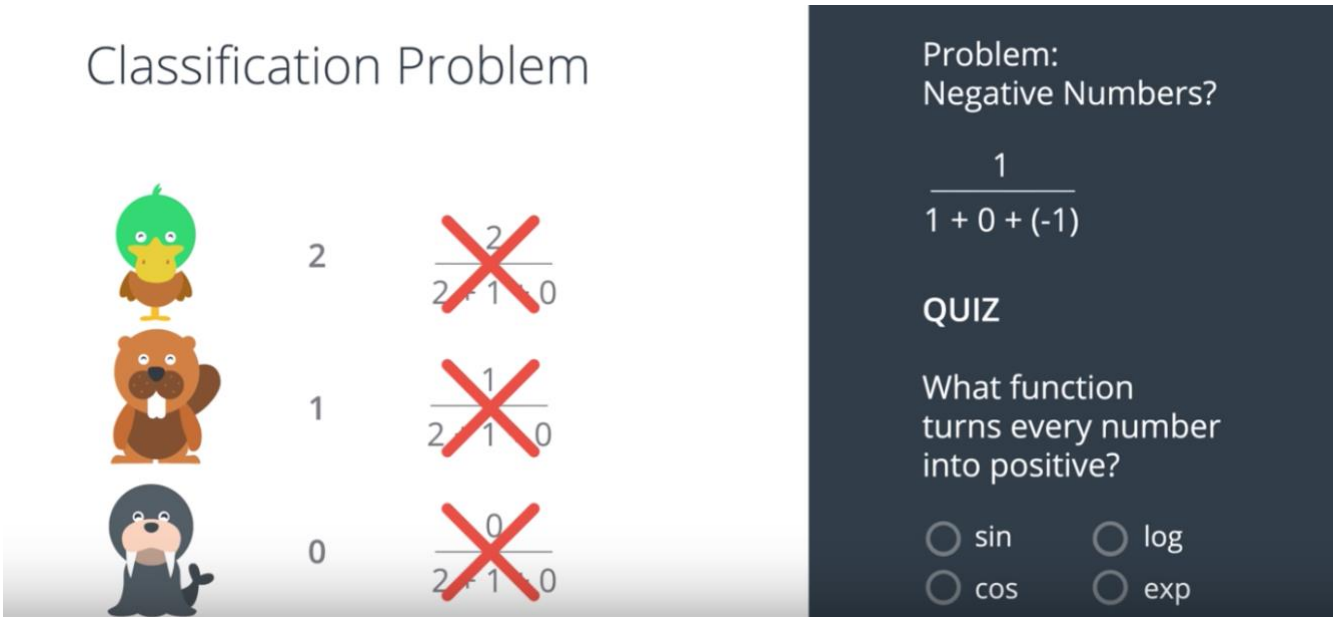


20. Let's switch to a different example for a moment. Let's say we have a model that will predict if you receive a gift or not. So, the model use predictions in the following way. It says, the probability that you get a gift is 0.8, which automatically implies that the probability that you don't receive a gift is 0.2. And what does the model do? What the model does is take some inputs. For example, is it your birthday or have it been good all year? And based on those inputs, it calculates a linear model which would be the score. Then, the probability that you get the gift or not is simply the sigmoid function applied to that score.

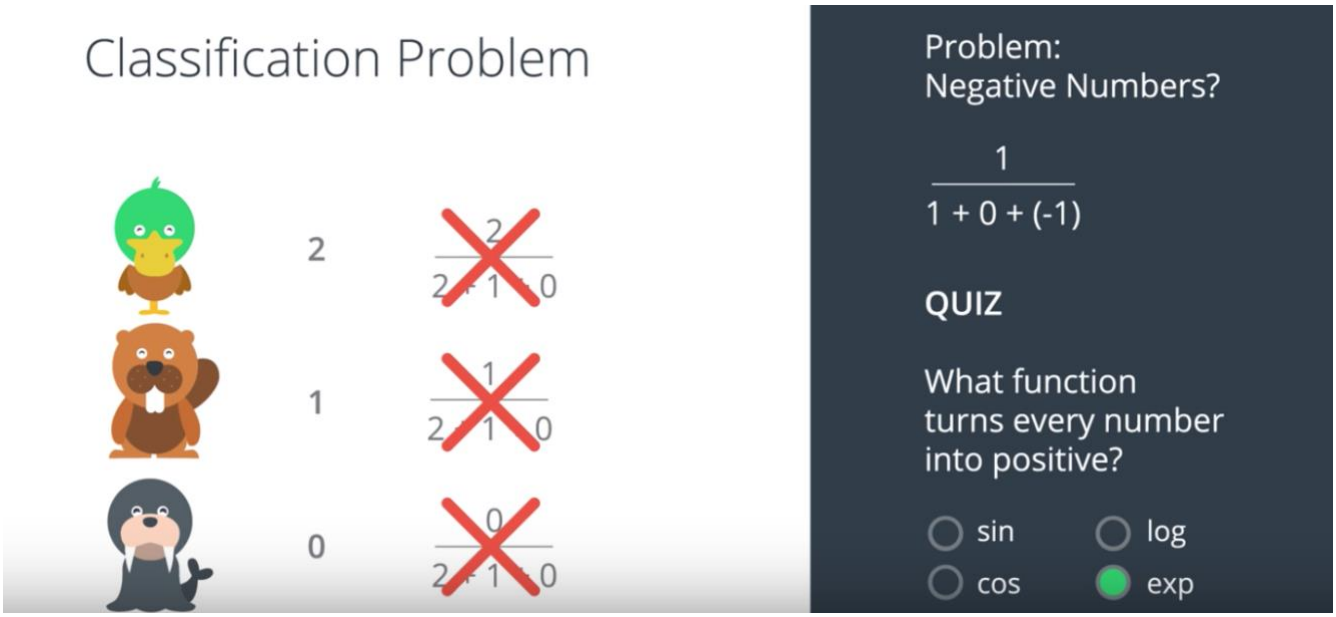


Now, what if you had more options than just getting a gift or not a gift? Let's say we have a model that just tell us what animal we just saw, and the options are a duck, a beaver and a walrus. We want a model that tells an answer along the lines of, the probability of a duck is 0.67, the probability of a beaver is 0.24, and the probability of a walrus is 0.09. Notice that the probabilities need to add to one. Let's say we have a linear model based on some inputs. The inputs could be, does it have a

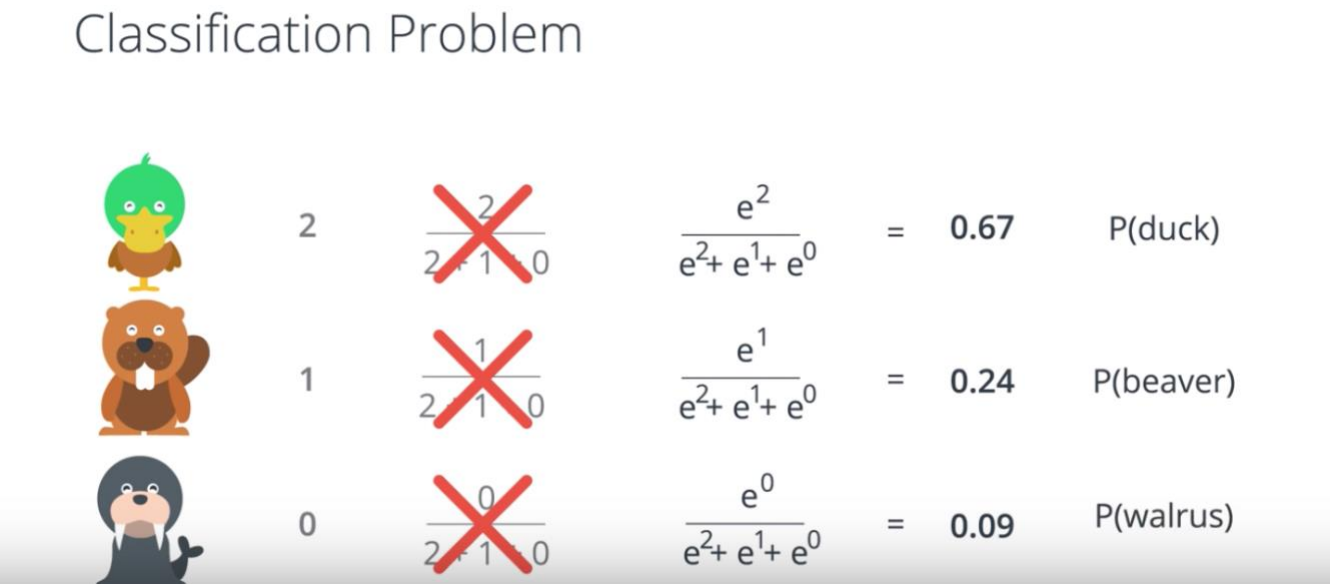
beak or not? Number of teeth. Number of feathers. Hair, no hair. Does it live in the water? Does it fly? Etc. We calculate linear function based on those inputs, and let's say we get some scores. So, the duck gets a score of two, and the beaver gets a score of one, and the walrus gets a score of zero. And now the question is, how do we turn these scores into probabilities?



The first thing we need to satisfy with probabilities is as we said, they need to add to one. So the two, the one, and the zero do not add to one. The second thing we need to satisfy is, since the duck had a higher score than the beaver and the beaver had a higher score than the walrus, then we want the probability of the duck to be higher than the probability of the beaver, and the probability of the beaver to be higher than the probability of the walrus. Here's a simple way of doing it. Let's take each score and divide it by the sum of all the scores. The two becomes two divided by two plus one plus zero, the one becomes one divided by two plus one plus zero, and the zero becomes zero divided by two plus one plus zero. This kind of works because the probabilities we obtain are two thirds for the duck, one third for the beaver, and zero for the walrus. That works but there's a little problem. Let's think about it. What could this problem be? The problem is the following. What happens if our scores are negative? This is completely plausible since the scores are linear function which could give negative values. What if we had, say, scores of 1, 0 and (-1)? Then, one of the probabilities would turn into one divided by one plus zero plus minus one which is zero, and we know very well that we cannot divide by zero. This unfortunately won't work, but the idea is good. How can we turn this idea into one that works all the time even for negative numbers? Well, it's almost like we need to turn these scores into positive scores. How do we do this? Is there a function that can help us? This is the quiz. Let's look at some options. There's sine, cosine, logarithm, and exponential. Quiz. Which one of these functions will turn every number into a positive number? Enter your answer below.

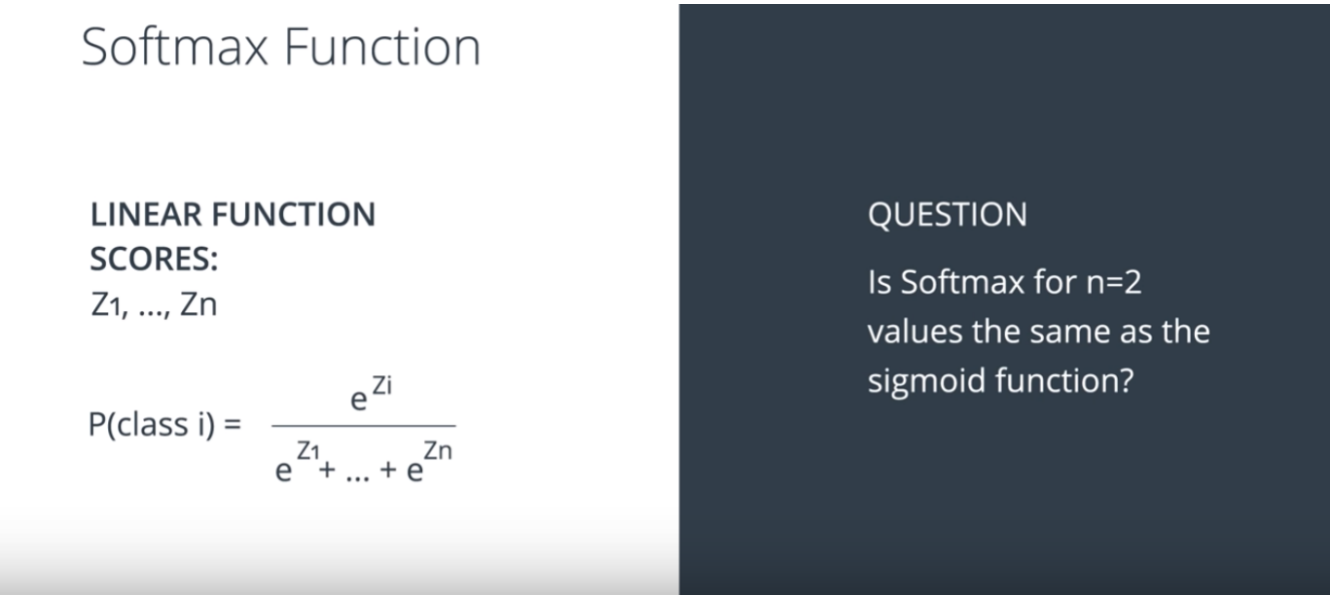


21. So, if you said exponential, you are correct. Because this is a function that returns a positive number for every input. E to the X is always a positive number.

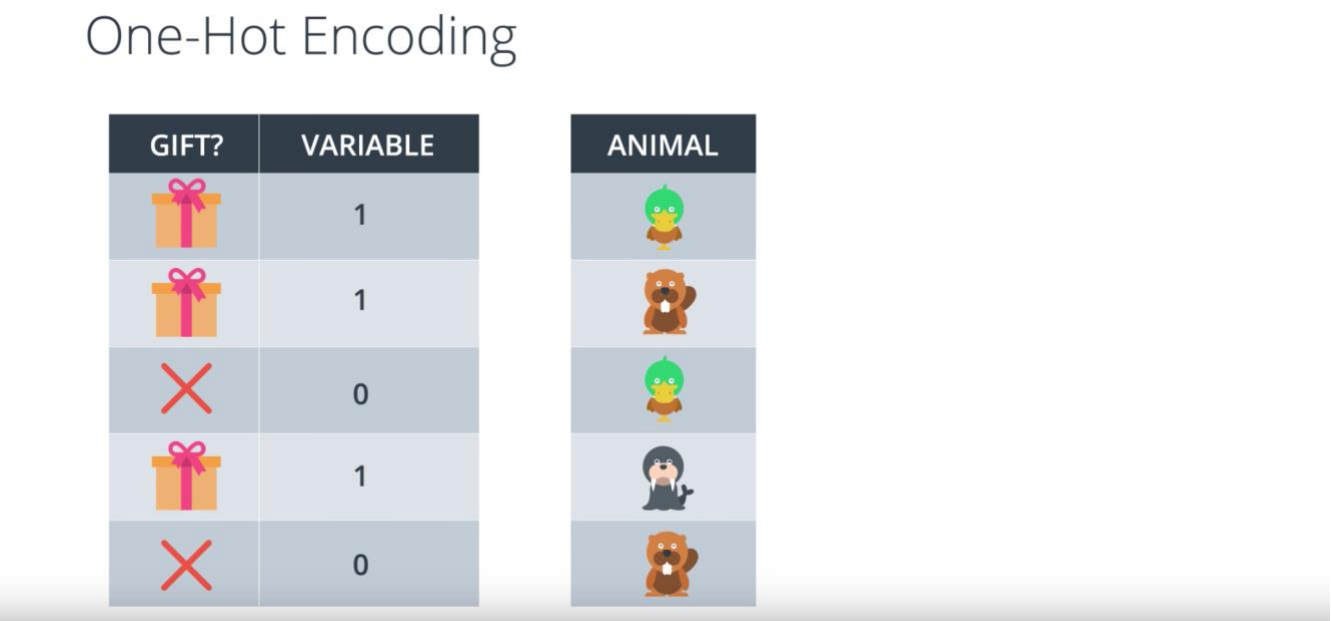


So, what we're going to do is exactly what we did before, except, applying it to the X to the scores. So, instead of 2,1, 0, we have E to the 2, E to the 1 and E to the 0. So, that 2 becomes E to the 2 divided by E to the two plus E to the 1 plus E to the 0. And, similarly for 1 and 0. So, the probabilities we obtain now are as 0.67, 0.24 and 0.09. This clearly add to 1. And, also notice that

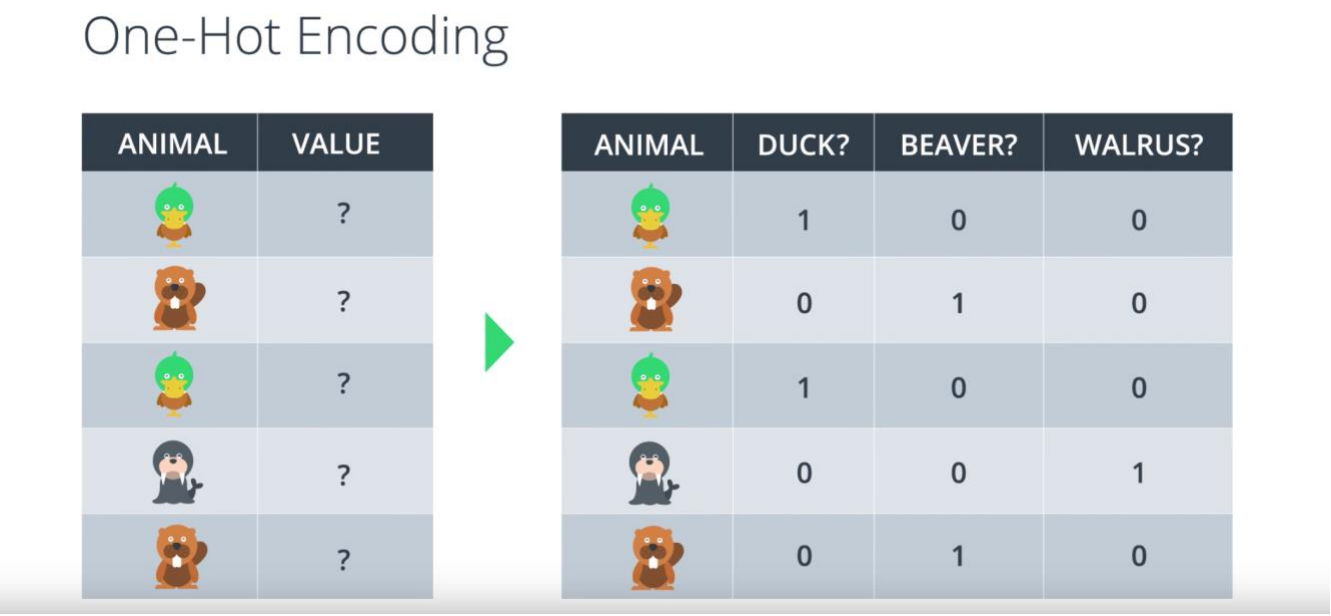
since the exponential function is increasing, then the duck has a higher probability than the beaver. And this one has a higher probability than the walrus.



This function is called the Softmax function and it's defined formally like this. Let's say we have N classes and a linear model that gives us the following scores. Z1, Z2, up to ZN. Each score for each of the classes. What we do to turn them into probabilities is to say the probability that the object is in class I is going to be E to the power of the ZI divided by the sum of E to the power of Z1 plus all the way to E to the power ZN. That's how we turn scores into probabilities. So, here's a question for you. When we had two classes, we applied the sigmoid function to the scores. Now, that we have more classes we apply the softmax function to the scores. The question is, is the softmax function for N equals to the same as the sigmoid function? I'll let you think about it. The answer is actually, yes, but it's not super trivial why. And, it's a nice thing to remember.



22. So, as we've seen so far, all our algorithms are numerical. This means we need to input numbers, such as a score in a test or the grades, but the input data will not always look like numbers. Sometimes it looks like this. Let's say the module receives as an input the fact that you got a gift or didn't get a gift. How do we turn that into numbers? Well, that's easy. If you've got a gift, we'll just say that the input variable is 1. And, if you didn't get a gift, we'll just say that the input variable is 0.

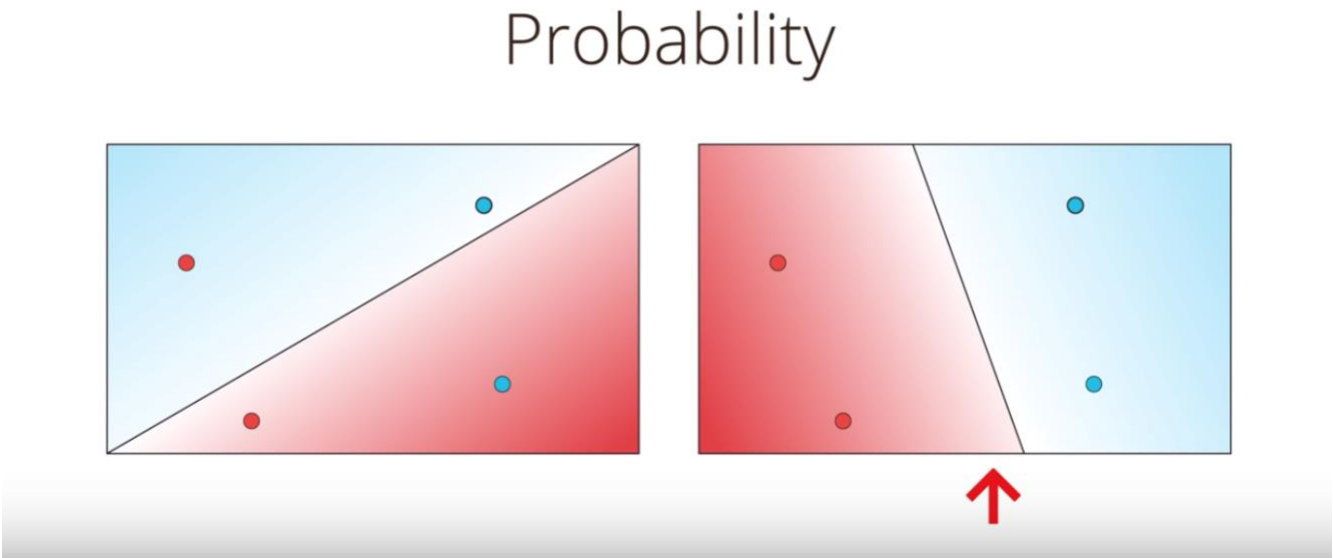


But, what if we have more classes as before or, let's say, our classes are Duck, Beaver and Walrus? What variable do we input in the algorithm? Maybe, we can input a 0 or 1 and a 2, but that would not work because it would assume dependencies between the classes that we can't have. So, this is what we do. What we do is, we come up with one variable for each of the classes. So, our table

becomes like this. That's one variable for Duck, one for Beaver and one for Walrus. And, each one has its corresponding column. Now, if the input is a duck then the variable for duck is 1 and the variables for beaver and walrus are 0. Similarly for the beaver and the walrus. We may have more columns of data but at least there are no unnecessary dependencies. This process is called The One-Hot Encoding and it will be used a lot for processing data.

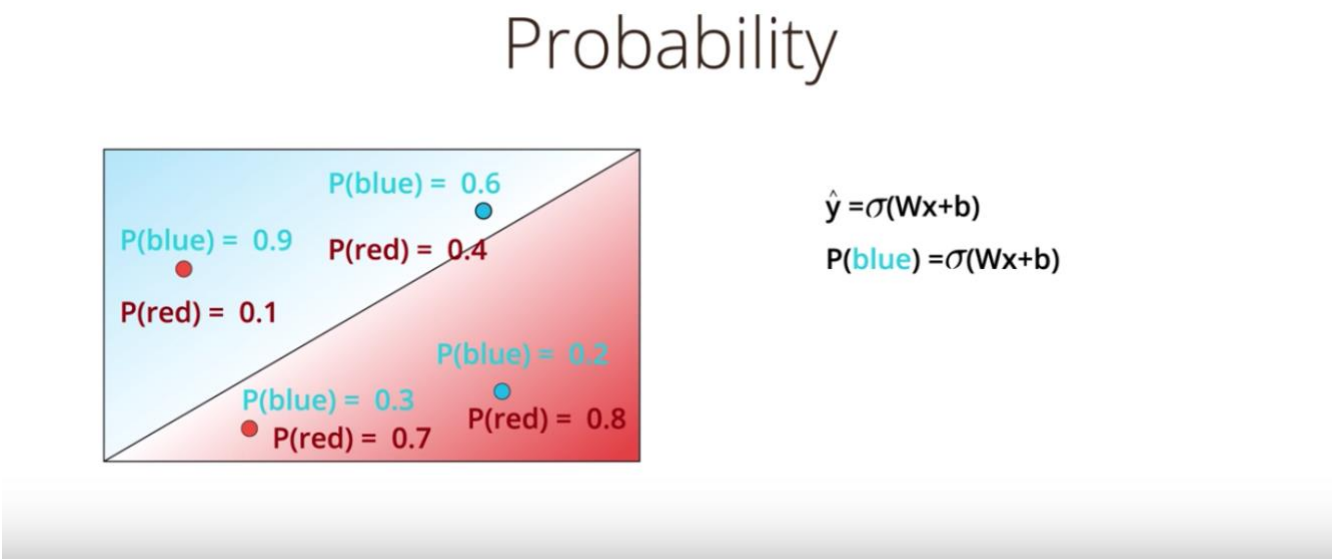
(This paragraph has no picture).

23. So we're still in our quest for an algorithm that will help us pick the best model that separates our data. Well, since we're dealing with probabilities then let's use them in our favor. Let's say I'm a student and I have two models. One that tells me that my probability of getting accepted is 80% and one that tells me the probability is 55%. Which model looks more accurate? Well, if I got accepted then I'd say the better model is probably the one that says 80%. What if I didn't get accepted? Then the more accurate model is more likely the one that says 55 percent. But I'm just one person. What if it was me and a friend? Well, the best model would more likely be the one that gives the higher probabilities to the events that happened to us, whether it's acceptance or rejection. This sounds pretty intuitive. The method is called maximum likelihood. What we do is we pick the model that gives the existing labels the highest probability. Thus, by maximizing the probability, we can pick the best possible model.

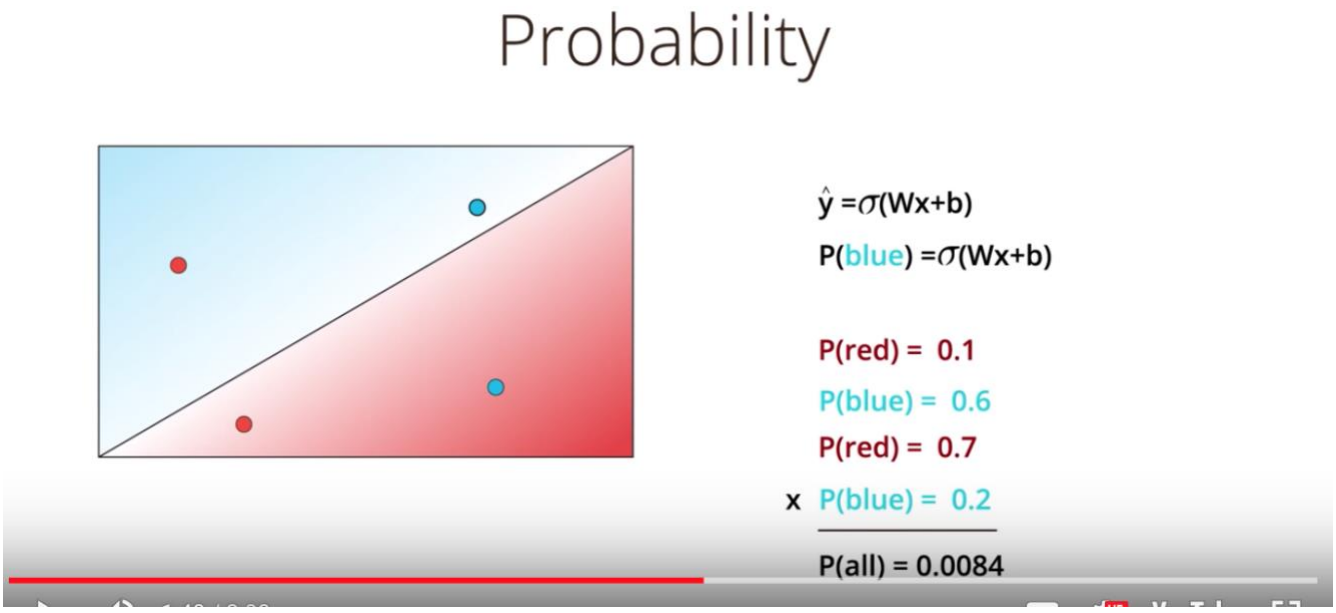


24. So let me be more specific. Let's look at the following four points: two blue and two red and two models that classify them, the one on the left and the one on the right. Quick. Which model looks better? You are correct. The model on the right is much better since it classifies the four points correctly whereas the model in the left gets two points correctly and two points incorrectly.

But let's see why the model in the right is better from the probability perspective. And by that, we'll show you that the arrangement in the right is much more likely to happen than the one in the left.

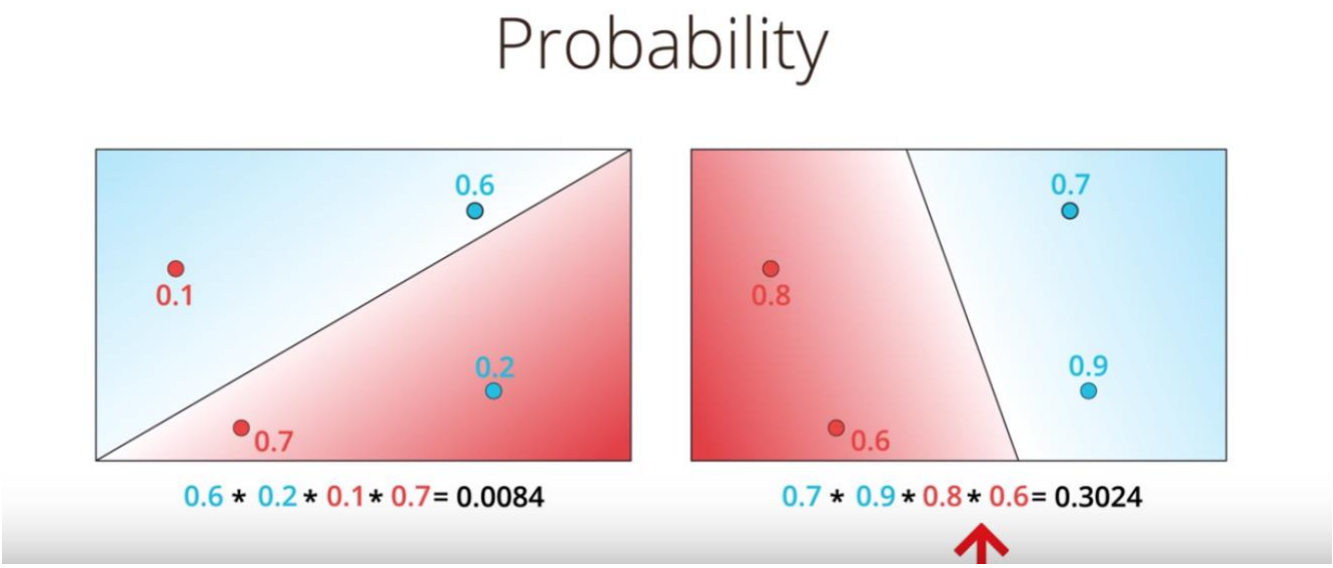


So let's recall that our prediction is $\hat{y} = \sigma(Wx+b)$ and that that is precisely the probability of a point being labeled positive which means blue. So for the points in the figure, let's say the model tells you that the probability of being blue are 0.9, 0.6, 0.3, and 0.2. Notice that the points in the blue region are much more likely to be blue and the points in the red region are much less likely to be blue. Now obviously, the probability of being red is one minus the probability of being blue. So in this case, the probability of some of the points being red are 0.1, 0.4, 0.7 and 0.8.

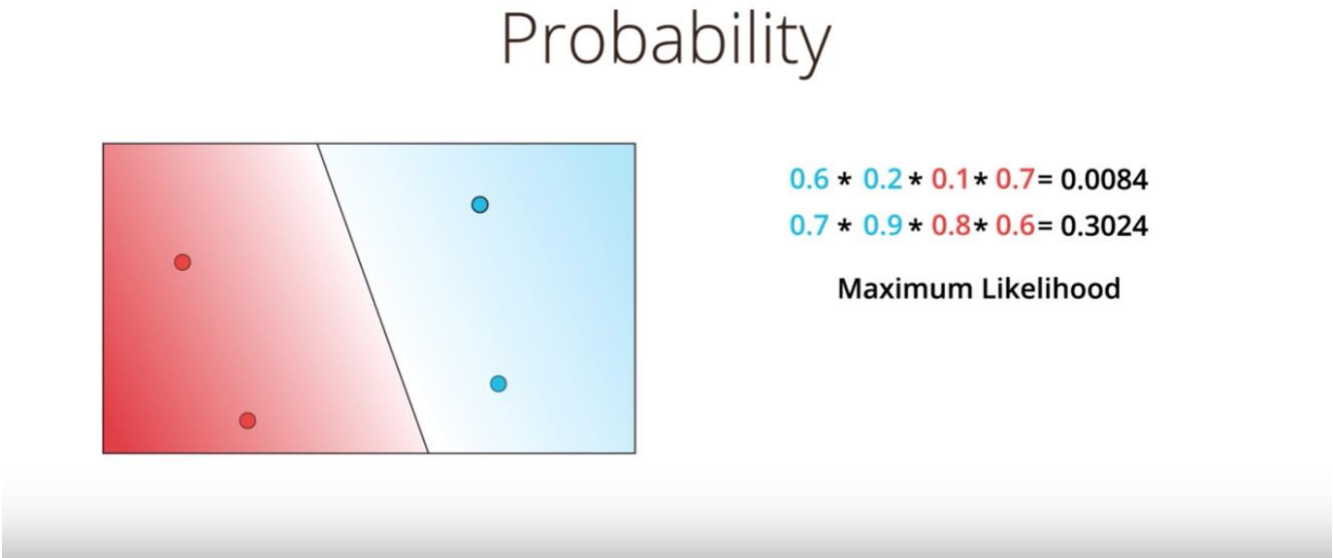


Now what we want to do is we want to calculate the probability of the four points are of the colors that they actually are. This means the probability that the two red points are red and that the two blue points are blue. Now if we assume that the colors of the points are independent events then

the probability for the whole arrangement is the product of the probabilities of the four points. This is equal to $0.1 \times 0.6 \times 0.7 \times 0.2 = 0.0084$. This is very small. It's less than 1%. What we mean by this is that if the model is given by these probability spaces, then the probability that the points are of these colors is 0.0084.

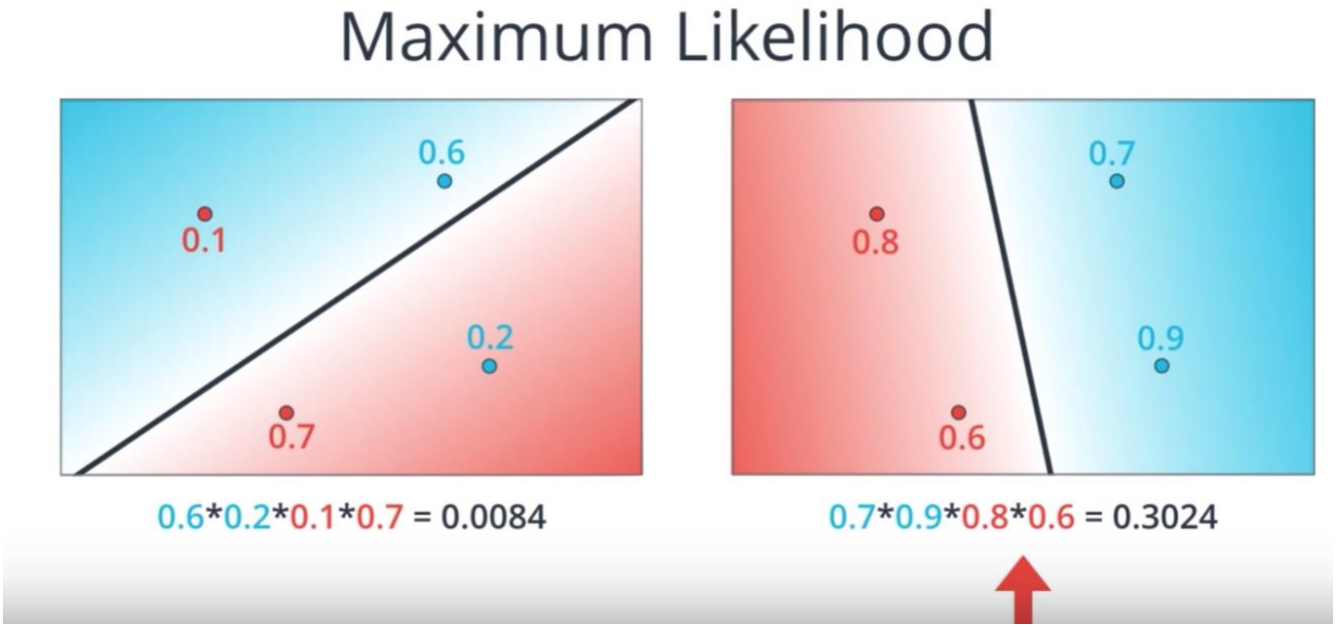


Now let's do this for both models. As we saw the model on the left tells us that the probabilities of these points being of those colors is 0.0084. If we do the same thing for the model on the right. Let's say we get that the probabilities of the two points in the right being blue are 0.7 and 0.9 and of the two points in the left being red are 0.8 and 0.6. When we multiply these we get 0.3024 which is around 30%. This is much higher than 0.0084. Thus, we confirm that the model on the right is better because it makes the arrangement of the points much more likely to have those colors.



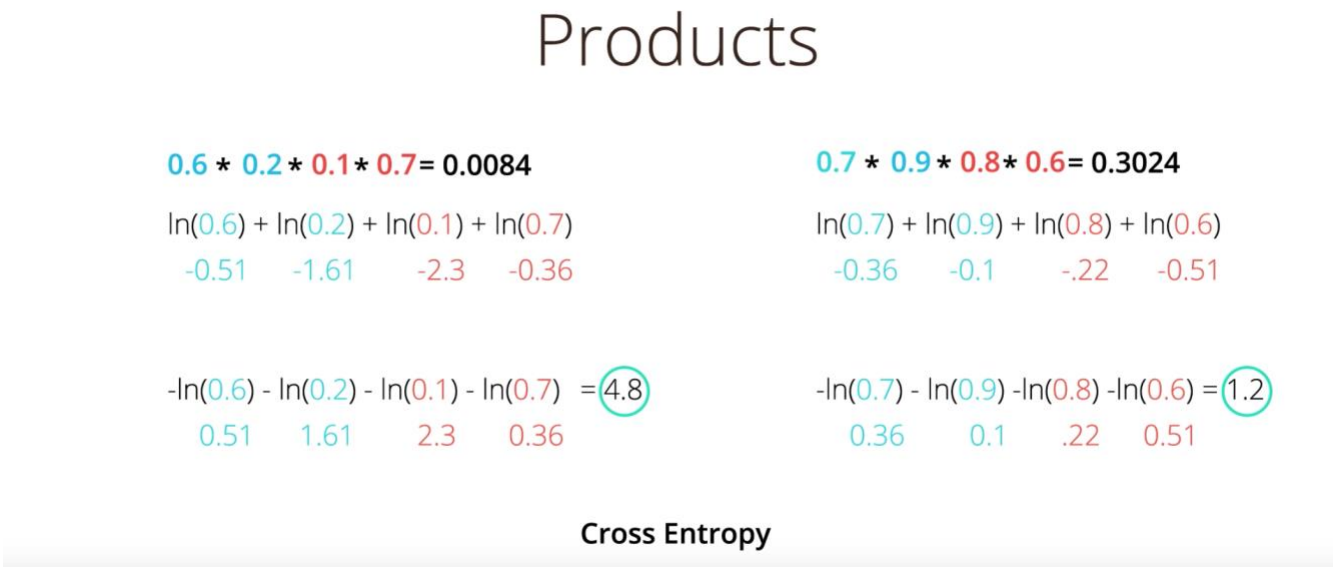
So now, what we do is the following? We start from the bad modeling, calculate the probability that the points are those colors, multiply them and we obtain the total probability is 0.0084. Now if we just had a way to maximize this probability we can increase it all the way to 0.3024. Thus, our new goal becomes precisely that, to maximize this probability. This method, as we stated before, is called maximum likelihood.

25. Well we're getting somewhere now. We've concluded that the probability is important. And that the better model will give us a better probability. Now the question is, how we maximize the probability. Also, if remember correctly we're talking about an error function and how minimizing this error function will take us to the best possible solution. Could these two things be connected? Could we obtain an error function from the probability? Could it be that maximizing the probability is equivalent to minimizing the error function? Maybe.

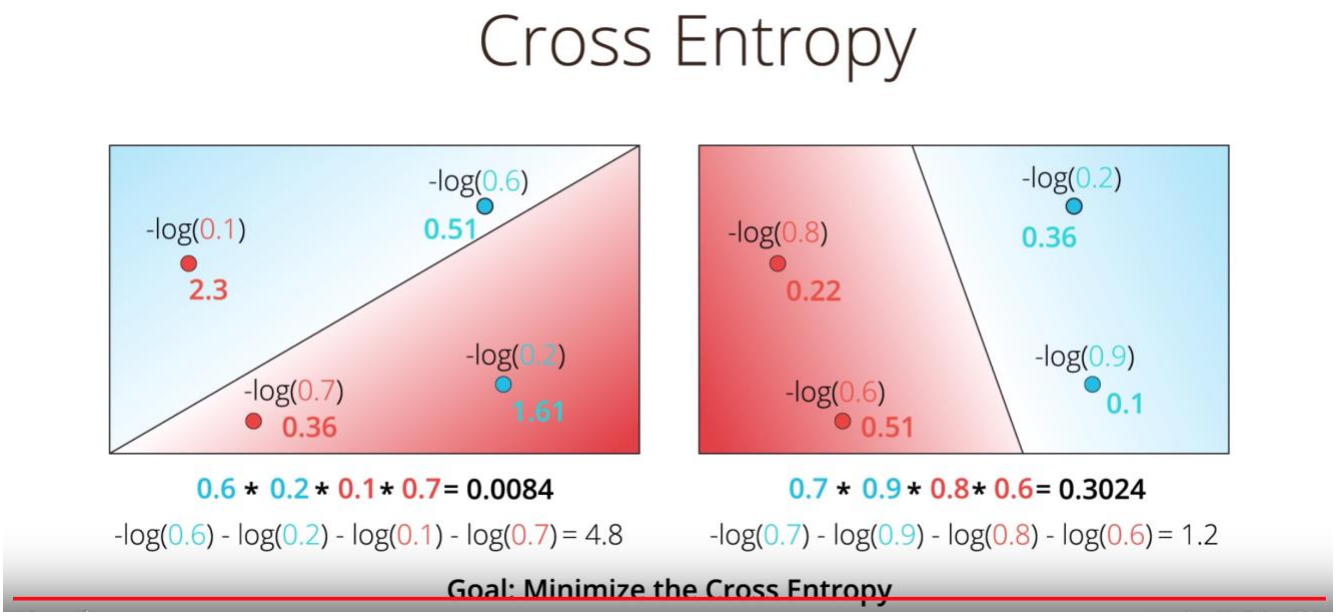


26. So a quick recap. We have two models, the bad one on the left and the good one on the right. And the way to tell they're bad or good is to calculate the probability of each point being the color it is according to the model. Multiply these probabilities in order to obtain the probability of the whole arrangement and then check that the model on the right gives us a much higher probability than the model on the left.

Now all we need to do is to maximize this probability. But probability is a product of numbers and products are hard. Maybe this product of four numbers doesn't look so scary. But what if we have thousands of datapoints? That would correspond to a product of thousands of numbers, all of them between zero and one. This product would be very tiny, something like 0.0000 something and we definitely want to stay away from those numbers. Also, if I have a product of thousands of numbers and I change one of them, the product will change drastically. In summary, we really want to stay away from products. And what's better than products? Well, let's ask our friend here. Products are bad, but sums are good. Let's do sums. So let's try to turn these products into sums. We need to find a function that will help us turn products into sums. What would this function be? It sounds like it's time for a quiz. Quiz. Which function will help us out here? Sine, cosine, logarithm or the exponential function? Enter your answer below.



27. Correct. The answer is logarithm, because logarithm has this very nice identity that says that the logarithm of the product A times B is the sum of the logarithms of A and B. So this is what we do. We take our products and we take the logarithms, so now we get a sum of the logarithms of the factors. So the ln(0.6*0.2*0.1*0.7) is equal to ln(0.6) + ln(0.2) + ln(0.1) + ln(0.7) etc. Now from now until the end of class, we'll be taking the natural logarithm which is base e instead of 10. Nothing different happens with base 10. Everything works the same as everything gets scaled by the same factor. So it's just more for convention. We can calculate those values and get minus 0.51, minus 1.61, minus 0.23 etc. Notice that they are all negative numbers and that actually makes sense. This is because the logarithm of a number between 0 and 1 is always a negative number since the logarithm of one is zero. So it actually makes sense to think of the negative of the logarithm of the probabilities and we'll get positive numbers. So that's what we'll do. We'll take the negative of the logarithm of the probabilities. That sums up negatives of logarithms of the probabilities, we'll call them cross entropy which is a very important concept in the class. If we calculate the cross entropies, we see that the bad model on left has a cross entropy 4.8 which is high. Whereas the good model on the right has a cross entropy of 1.2 which is low. This actually happens all the time. A good model will give us a low cross entropy and a bad model will give us a high cross entropy. The reason for this is simply that a good model gives us a high probability and the negative of the logarithm of a large number is a small number and vice versa.

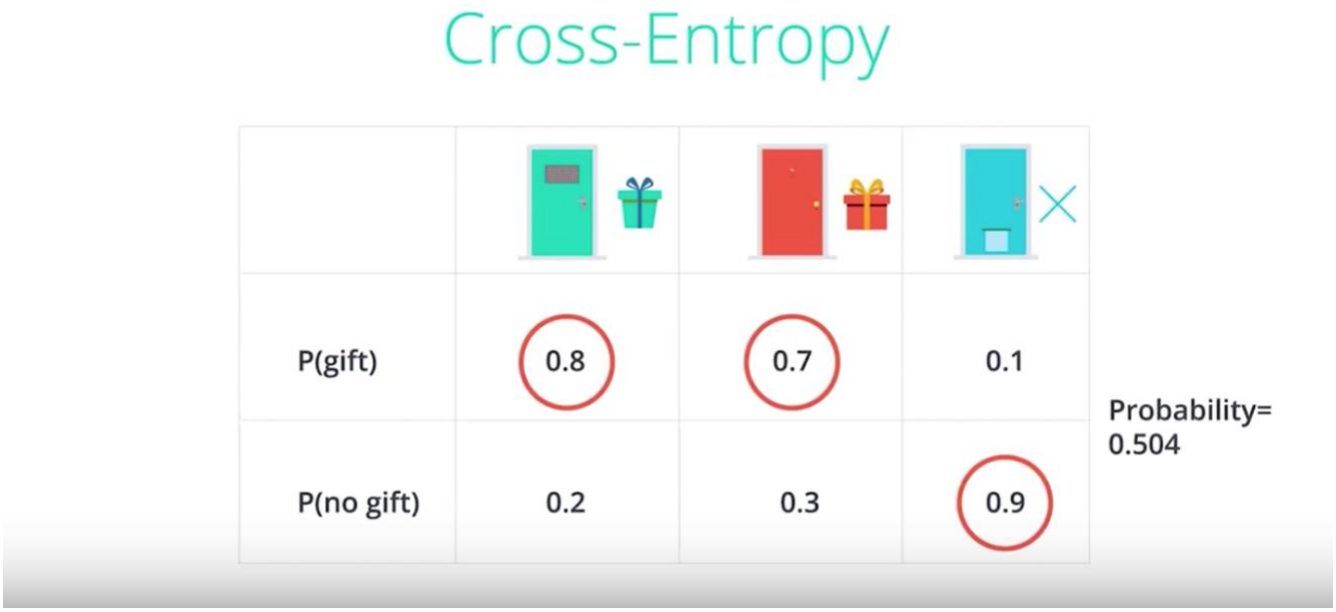


Tao: the value in the log is the model's estimation of the probability of this point being classified to its actual color.

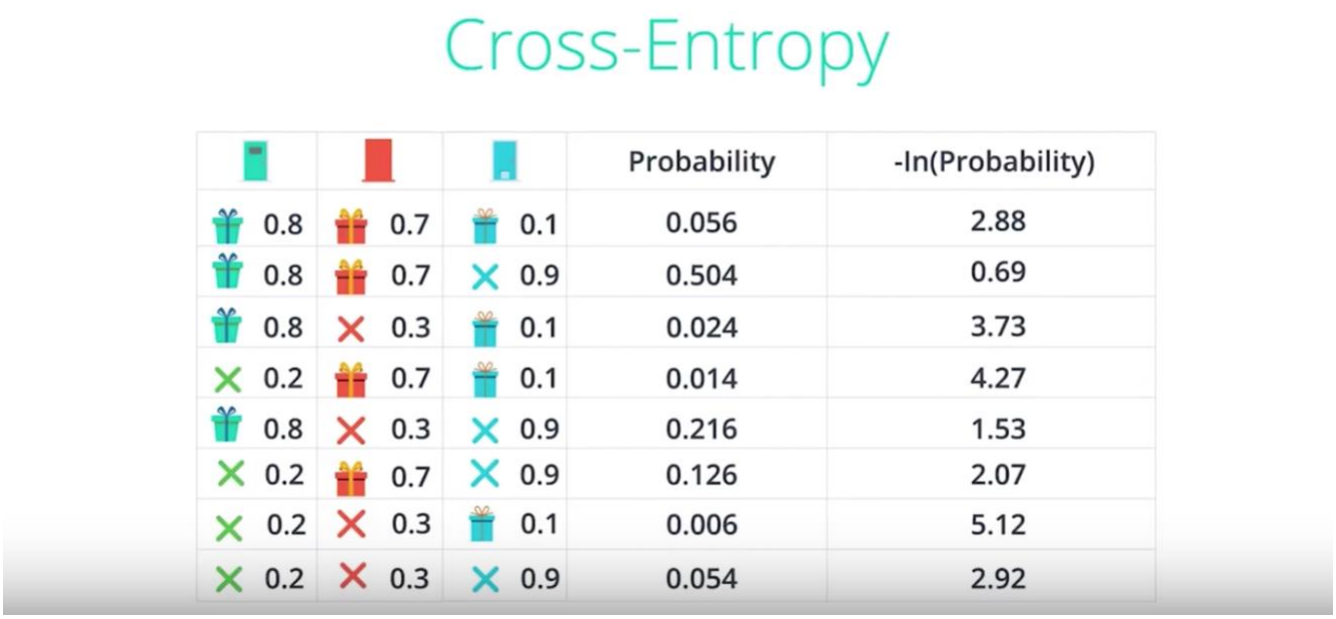
This method is actually much more powerful than we think. If we calculate the probabilities and pair the points with the corresponding logarithms, we actually get an error for each point. So again, here we have probabilities for both models and the products of them. Now, we take the negative of the logarithms which gives us sum of logarithms and if we pair each logarithm with the point where it came from, we actually get a value for each point. And if we calculate the values, we get this. Check it out. If we look carefully at the values we can see that the points that are mis-classified has like values like 2.3 for this point or 1.6 one for this point, whereas the points that are correctly classified have small values. And the reason for this is again is that a correctly classified point will have a probability that as close to 1, which when we take the negative of the logarithm, we'll get a small value. Thus we can think of the negatives of these logarithms as errors at each point. Points that are correctly classified will have small errors and points that are mis-classified will have large errors. And now we've concluded that our cross entropy will tell us if a model is good or bad. So now our goal has changed from maximizing a probability to minimizing a cross entropy in order to get from the model in left to the model in the right. And that error function that we're looking for, that was precisely the cross entropy.

(This paragraph has no picture)
28. So this cross entropy, it looks like kind of a big deal. Cross entropy really says the following. If I have a bunch of events and a bunch of probabilities (tao: these probabilities are the output of the model), how likely is it that those events happen based on the probabilities? If it's very likely, then we have a small cross entropy. If it's unlikely, then we have a large cross entropy. Let's elaborate.

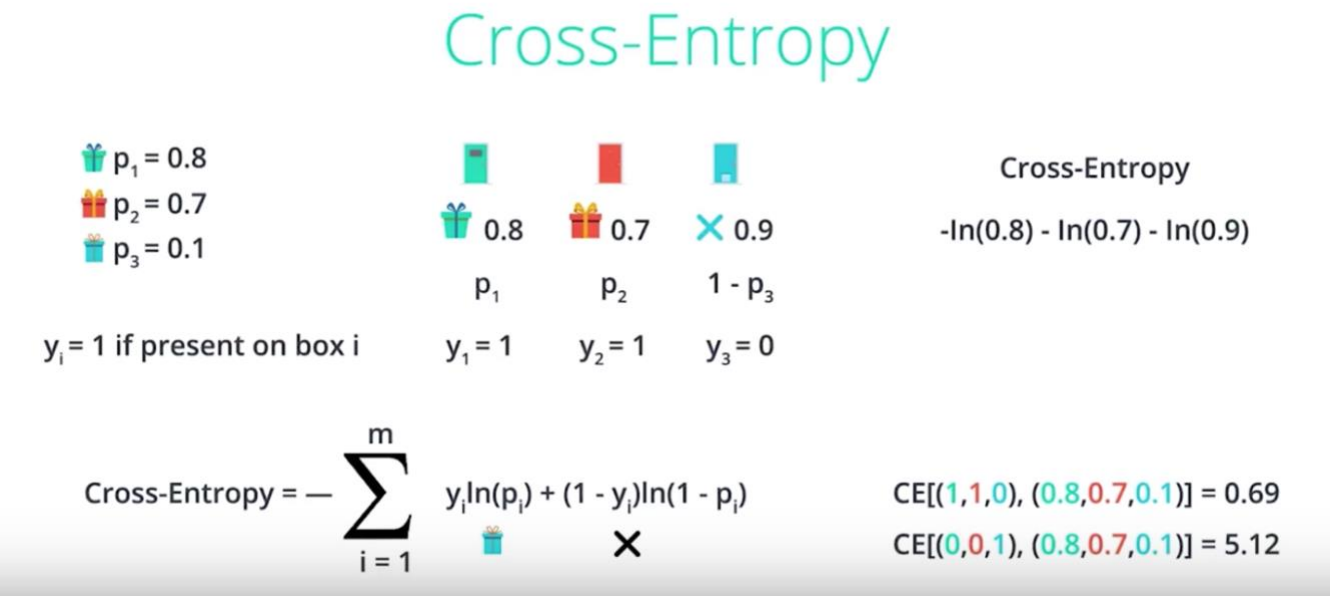
29. Let's look a bit closer into Cross-Entropy by switching to a different example. Let's say we have three doors. And no this is not the Monty Hall problem. We have the green door, the red door, and the blue door, and behind each door we could have a gift or not have a gift. And the probabilities of there being a gift behind each door is 0.8 for the first one, 0.7 for the second one, 0.1 for the third one. So for example behind the green door there is an 80 percent probability of there being a gift, and a 20 percent probability of there not being a gift.



So we can put the information in this table where the probabilities of there being a gift are given in the top row, and the probabilities of there not being a gift are given in the bottom row. So let's say we want to make a bet on the outcomes. So we want to try to figure out what is the most likely scenario here. And for that we'll assume they're independent events. In this case, the most likely scenario is just obtained by picking the largest probability in each column. So for the first door is more likely to have a gift than not have a gift. So we'll say there's a gift behind the first door. For the second door, it's also more likely that there's a gift. So we'll say there's a gift behind the second door. And for the third door it's much more likely that there's no gift, so we'll say there's no gift behind the third door. And as the events are independent, the probability for this whole arrangement is the product of the three probabilities which is 0.8, times 0.7, times 0.9, which ends up being 0.504, which is roughly 50 percent.

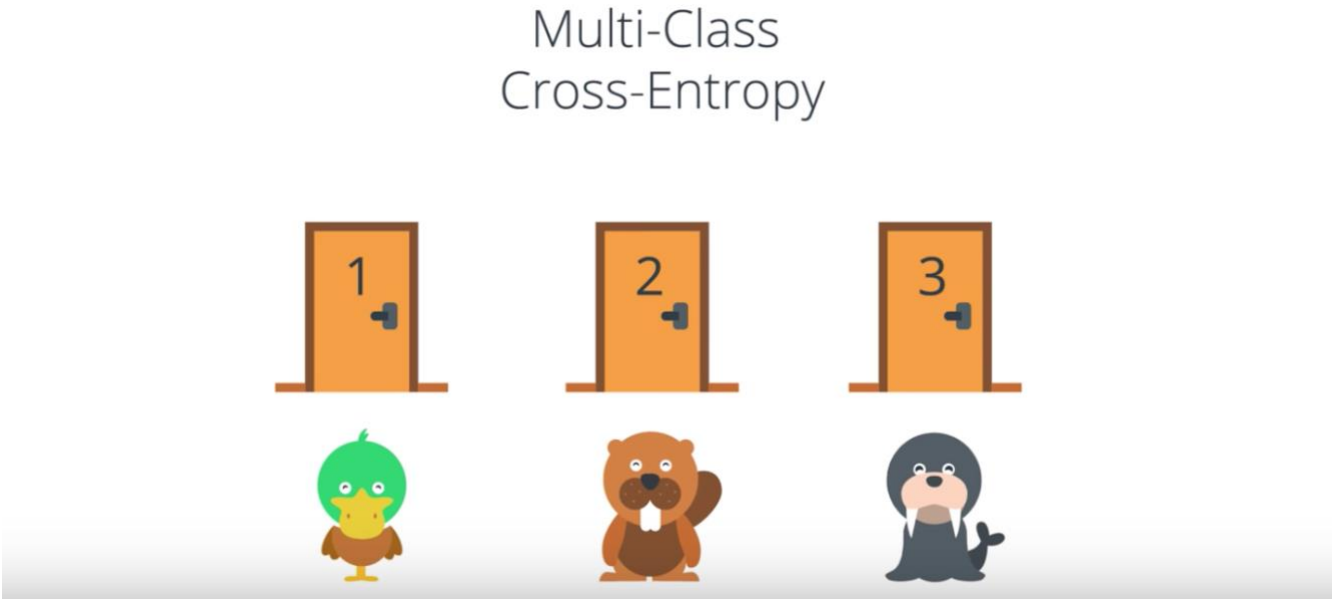


So let's look at all the possible scenarios in the table. Here's a table with all the possible scenarios for each door and there are eight scenarios since each door gives us two possibilities each, and there are three doors. So we do as before to obtain the probability of each arrangement by multiplying the three independent probabilities to get these numbers. You can check that these numbers add to one. And from last video we learned that the negative of the logarithm of the probabilities across entropy. So let's go ahead and calculate the cross-entropy. And notice that the events with high probability have low cross-entropy and the events with low probability have high cross-entropy. For example, the second row which has probability of 0.504 gives a small cross-entropy of 0.69, and the second to last row which is very very unlikely has a probability of 0.006 gives a cross entropy a 5.12.

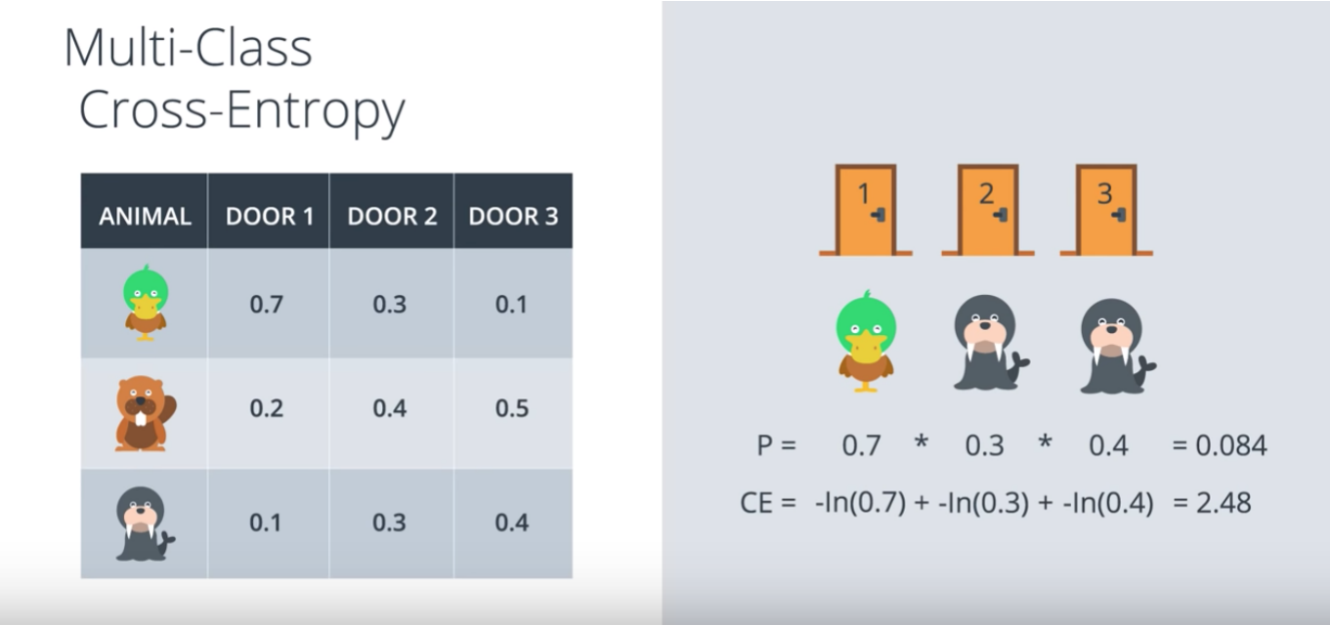


Tao: “CE” in the above picture means cross-entropy.

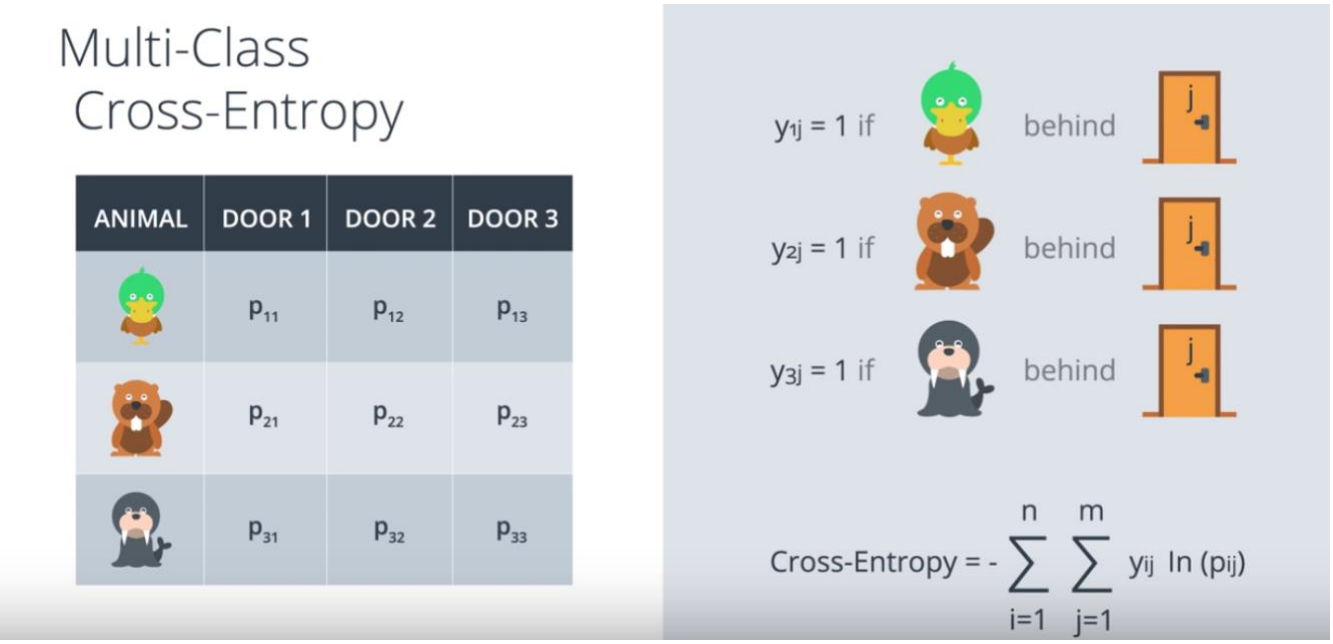
So let's actually calculate a formula for the cross-entropy. Here we have our three doors, and our sample scenario said that there is a gift behind the first and second doors, and no gift behind the third door. Recall that the probabilities of these events happening are 0.8 for a gift behind the first door, 0.7 for a gift behind the second door, and 0.9 for no gift behind the third door. So when we calculate the cross-entropy, we get the negative of the logarithm of the product, which is a sum of the negatives of the logarithms of the factors, which is negative logarithm of 0.8 minus logarithm of 0.7 minus logarithm 0.9. And in order to drive the formula we'll have some variables. So let's call P1 the probability that there's a gift behind the first door, P2 the probability there's a gift behind the second door, and P3 the probability there's a gift behind the third door. So this 0.8 here is P1, this 0.7 here is P2, and this 0.9 here is one minus P3. So it's a probability of there not being a gift is one minus the probability of there being a gift. Let's have another variable called Yi, which will be one of there's a present behind the ith door, and zero there's no present. So Yi is technically the number of presents behind the ith door. In this case Y1 equals one, Y2 equals one, and Y3 equals zero. So we can put all this together and derive a formula for the cross-entropy and it's this sum. Now let's look at the formula inside the summation. Noted that if there is a present behind the ith door, then Yi equals one. So the first term is logarithm of the Pi. And the second term is zero. Likewise, if there is no present behind the ith door, then Yi is zero. So this first term is zero. And this term is precisely logarithm of one minus Pi. Therefore, this formula really encompasses the sums of the negative of logarithms which is precisely the cross-entropy. So the cross-entropy really tells us when two vectors are similar or different. For example, if you calculate the cross entropy of the pair one one zero, and 0.8, 0.7, 0.1, we get 0.69. And that is low because (1, 1, 0)is a similar vector to (0.8, 0.7, 0.1). Which means that the arrangement of gifts given by the first set of numbers is likely to happen based on the probabilities given by the second set of numbers. But on the other hand if we calculate the cross-entropy of the pairs(0, 0, 1), and (0.8, 0.7, 0.1), that is 5.12 which is very high. This is because the arrangement of gifts being given by the first set of numbers is very unlikely to happen from the probabilities given by the second set of numbers.



30. Now that was when we had two classes namely receiving a gift or not receiving a gift. What happens if we have more classes? Let's take a look. So we have a similar problem. We still have three doors. And this problem is still not the Monty Hall problem. Behind each door there can be an animal, and the animal can be of three types. It can be a duck, it can be a beaver, or it can be a walrus. So let's look at this table of probabilities. According to the first column on the table, behind the first door, the probability of finding a duck is 0.7, the probability of finding a beaver is 0.2, and the probability of finding a walrus is 0.1. Notice that the numbers in each column need to add to one because there is some animal behind door one. The numbers in the rows do not need to add to one as you can see. It could easly be that we have a duck behind every door and that's okay.



So let's look at a sample scenario. Let's say we have our three doors, and behind the first door, there's a duck, behind the second door there's a walrus, and behind the third door there's also a walrus. Recall that the probabilities are again by the table. So a duck behind the first door is 0.7 likely, a walrus behind the second door is 0.3 likely, and a walrus behind the third door is 0.4 likely. So the probability of obtaining this three animals is the product of the probabilities of the three events since they are independent events, which in this case it's 0.084. And as we learn, that cross entropy here is given by the sums of the negatives of the logarithms of the probabilities. So the first one is negative logarithm of 0.7. The second one is negative logarithm of 0.3. And the third one is negative logarithm of 0.4. The Cross entropy's and the sum of these three which is actually 2.48.



But we want a formula, so let's put some variables here. So P11 is the probability of finding a duck behind door one. P12 is the probability of finding a duck behind door two etc. And let's have the indicator variables Y1j D1 if there's a duck behind door J. Y2j B1 if there's a beaver behind door J, and Y3j B1 if there's a walrus behind door J. And these variables are zero otherwise. And so, the formula for the cross entropy is simply the negative of the summation from i_ equals_ one to n, up to summation from y_ equals_ j to m of Yij_ times_ the logarithm of Pij. In this case, m is a number of classes. This formula works because Yij being zero one, makes sure that we're only adding the logarithms of the probabilities of the events that actually have occurred. And voila, this is the formula for the cross entropy in more classes. Now I'm going to leave this equestion. Given that we have a formula for cross entropy for two classes and one for m classes. These formulas look different but are they the same for m_ equals_ two? Obviously the answer is yes, but it's a cool exercise to actually write them down and convince yourself that they are actually the same.