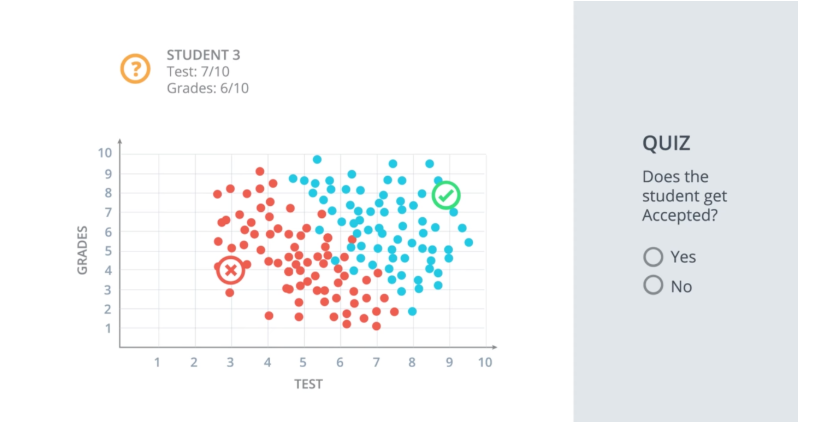
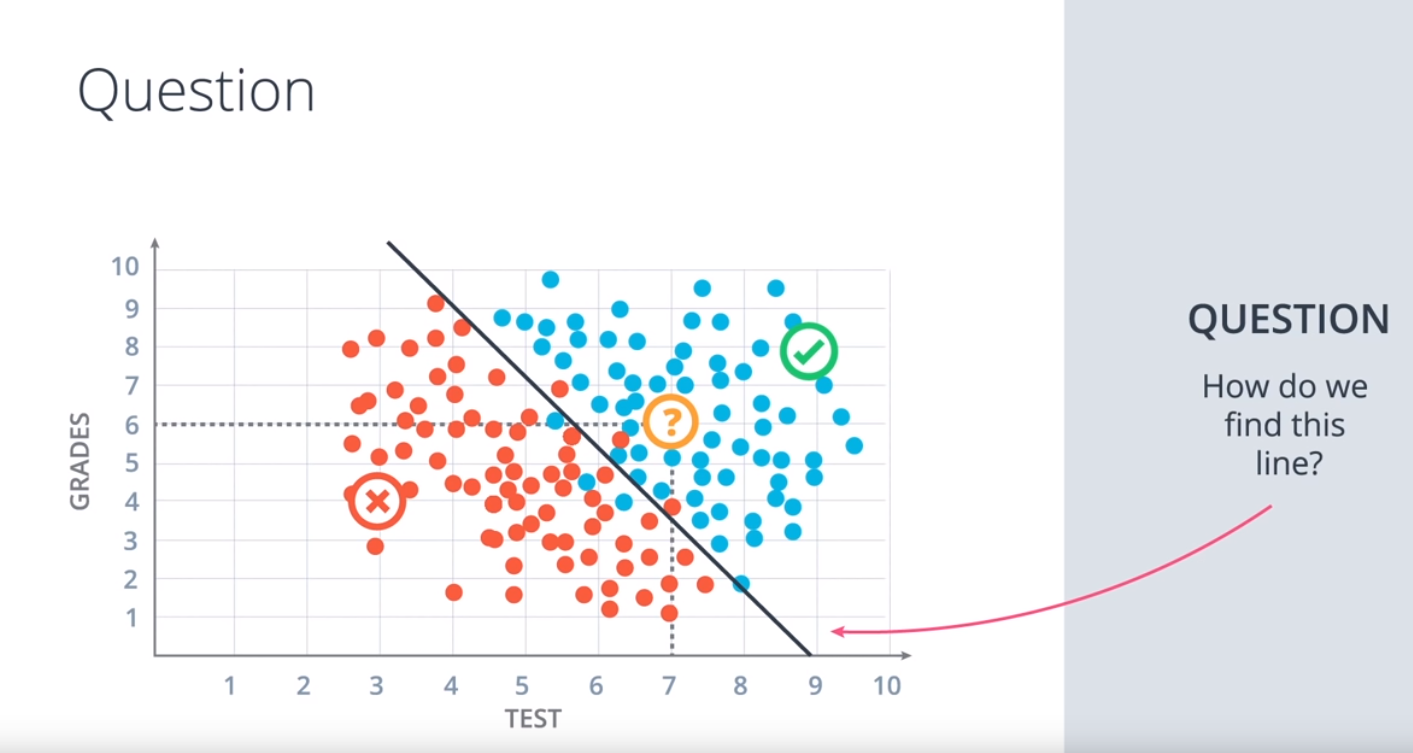
**Perceptron Algorithm – The basis for neural networks**

This algorithm is used in classification problems – problems that have a yes or a no outcome rather than a broad spectrum of outcomes.

Simple question, based off of the following graph should the student get accepted or not:



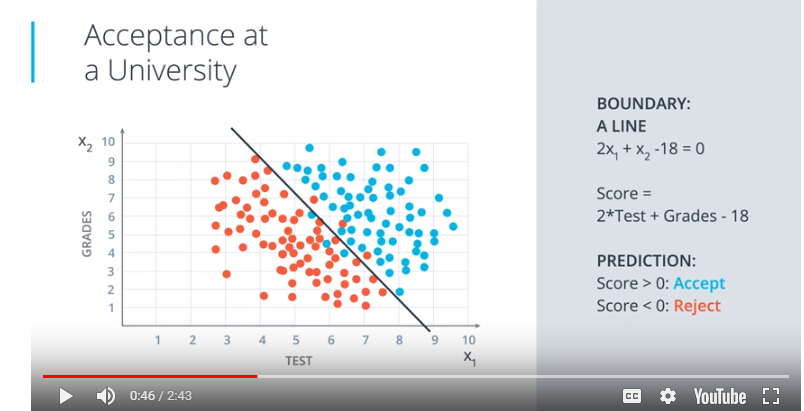
The answer is Yes they should be accepted. It looks like this data can be split by a line. Most students above this line get accepted and most students under the line get rejected.



It’s easy for us to find the line as we can eyeball it but we need algorithms for a computer to be able to find the line effectively. We will look into this now for this example and many more general and complicated cases.

**Linear Boundaries**

If we give the x and y axis the variables x1 and x2 below we can chart this line like we would in geometry. So for our above example we would get the equation of the line:

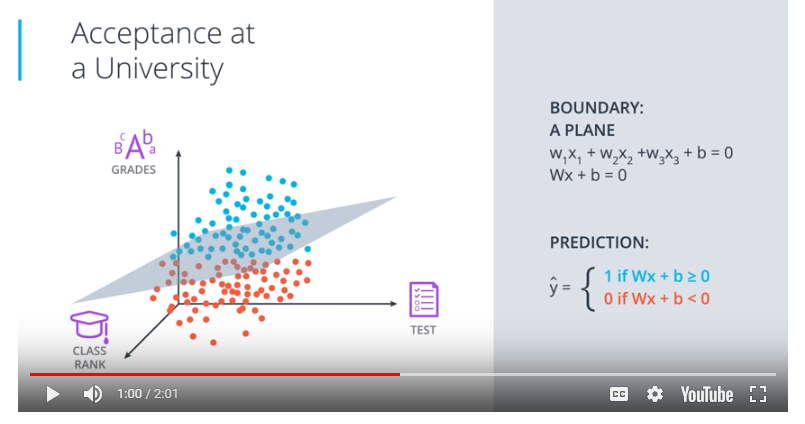


In the more general case it will be:

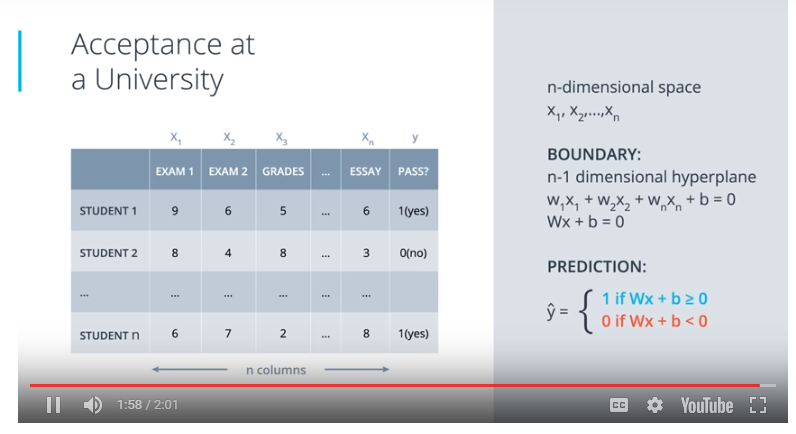


If we have more then two sets (higher dimensions) we could end up working in 3 or more axis’. A similar equation would be formed with similar parameters but of larger length:

3 dimensions:



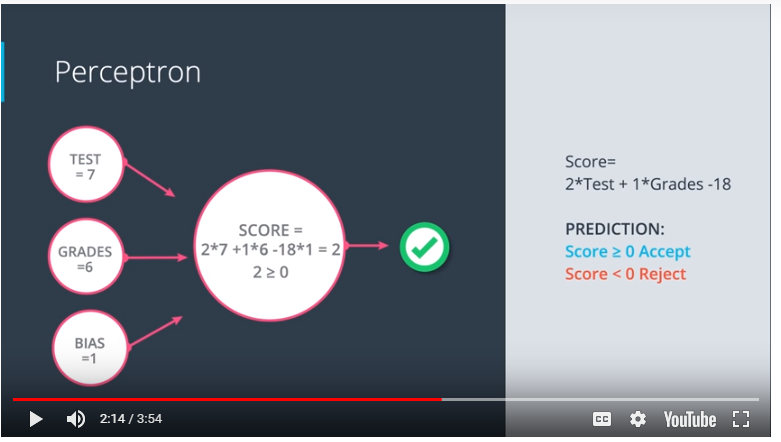
N dimensions:



**Perceptrons**

Perceptrons are the building blocks of neural networks. It’s an encoding of our equation into a small graph.

Perceptron description video just seemed to explain the above again only with nodes instead of numbers? The “node” return a yes or a no.

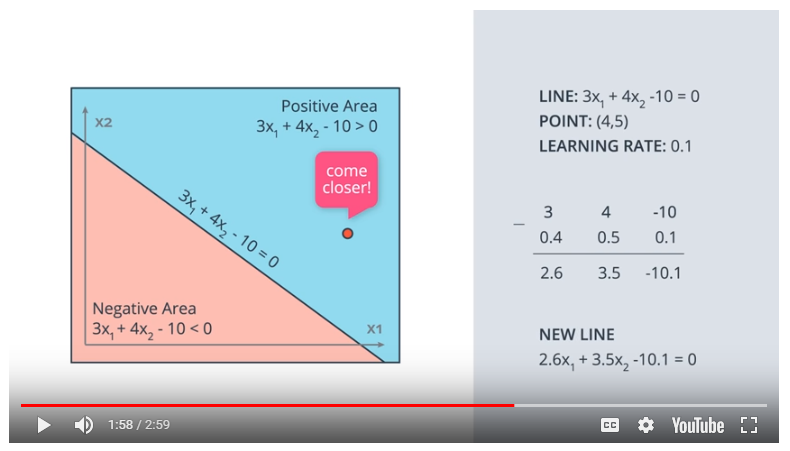


We’ll have an answer node with n inputs which then gives us either a 1 or a 0 out whether it’s true or false.

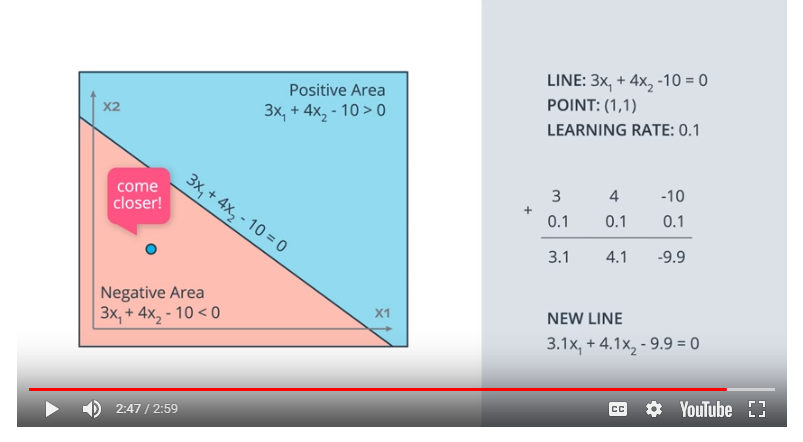
**Perceptron Trick**

In the last section you used your logic and your mathematical knowledge to create perceptrons for some of the most common logical operators. In real life, though, we can't be building these perceptrons ourselves. The idea is that we give them the result, and they build themselves. For this, here's a pretty neat trick that will help us.

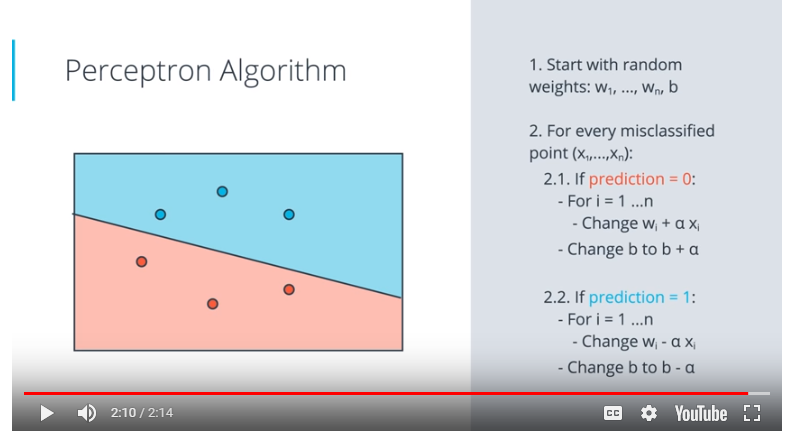
When we have misclassified points we can use the misclassified points to change the line we are using to split the data. Doing this repetitively is the Perceptron Algorithm. We will use a learning rate and the x and y coordinates to change the equation of the line like:



Or:



Perceptron algorithm pseudo code:



* So we start with random weights which gives us a random line to split the data.
* We get our points to tell us if they are misclassified or not. For every misclassified point:
* If prediction=0
  + change wi = wi + ax where a is our learning rate
  + change b to be b+ a where a is our learning rate
* If prediction=1
  + change wi = wi - ax where a is our learning rate
  + change b to be b - a where a is our learning rate
* Repeat this process until we have a line we are happy with

In Python the Perceptron step algorithm could be coded as follows:

*import numpy as np*

*# Setting the random seed, feel free to change it and see different solutions.*

*np.random.seed(42)*

*def stepFunction(t):*

*if t >= 0:*

*return 1*

*return 0*

*def prediction(X, W, b):*

*return stepFunction((np.matmul(X,W)+b)[0])*

*# TODO: Fill in the code below to implement the perceptron trick.*

*# The function should receive as inputs the data X, the labels y,*

*# the weights W (as an array), and the bias b,*

*# update the weights and bias W, b, according to the perceptron algorithm,*

*# and return W and b.*

*def perceptronStep(X, y, W, b, learn\_rate = 0.001):*

*for i in range(len(X)):*

*y\_hat = prediction(X[i],W,b)*

*if y[i]-y\_hat == 1:*

*W[0] += X[i][0]\*learn\_rate*

*W[1] += X[i][1]\*learn\_rate*

*b += learn\_rate*

*elif y[i]-y\_hat == -1:*

*W[0] -= X[i][0]\*learn\_rate*

*W[1] -= X[i][1]\*learn\_rate*

*b -= learn\_rate*

*return W, b*

*# This function runs the perceptron algorithm repeatedly on the dataset,*

*# and returns a few of the boundary lines obtained in the iterations,*

*# for plotting purposes.*

*# Feel free to play with the learning rate and the num\_epochs,*

*# and see your results plotted below.*

*def trainPerceptronAlgorithm(X, y, learn\_rate = 0.01, num\_epochs = 25):*

*x\_min, x\_max = min(X.T[0]), max(X.T[0])*

*y\_min, y\_max = min(X.T[1]), max(X.T[1])*

*W = np.array(np.random.rand(2,1))*

*b = np.random.rand(1)[0] + x\_max*

*# These are the solution lines that get plotted below.*

*boundary\_lines = []*

*for i in range(num\_epochs):*

*# In each epoch, we apply the perceptron step.*

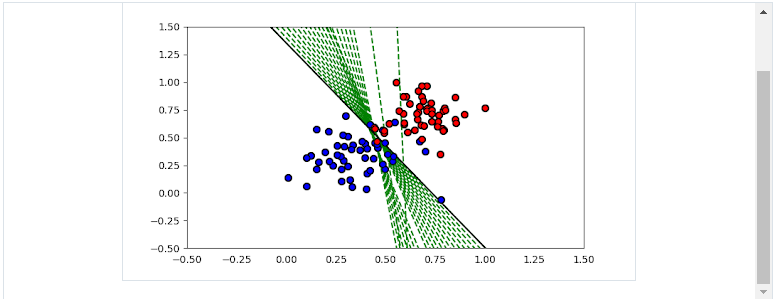
*W, b = perceptronStep(X, y, W, b, learn\_rate)*

*boundary\_lines.append((-W[0]/W[1], -b/W[1]))*

*return boundary\_lines*

We could change the “learn\_rate” and “num\_epoch” parameters above to give us different outputs:

Learn\_rate=0.01 and num\_epoch=25:



Learn\_rate=0.01, num\_epoch = 10

