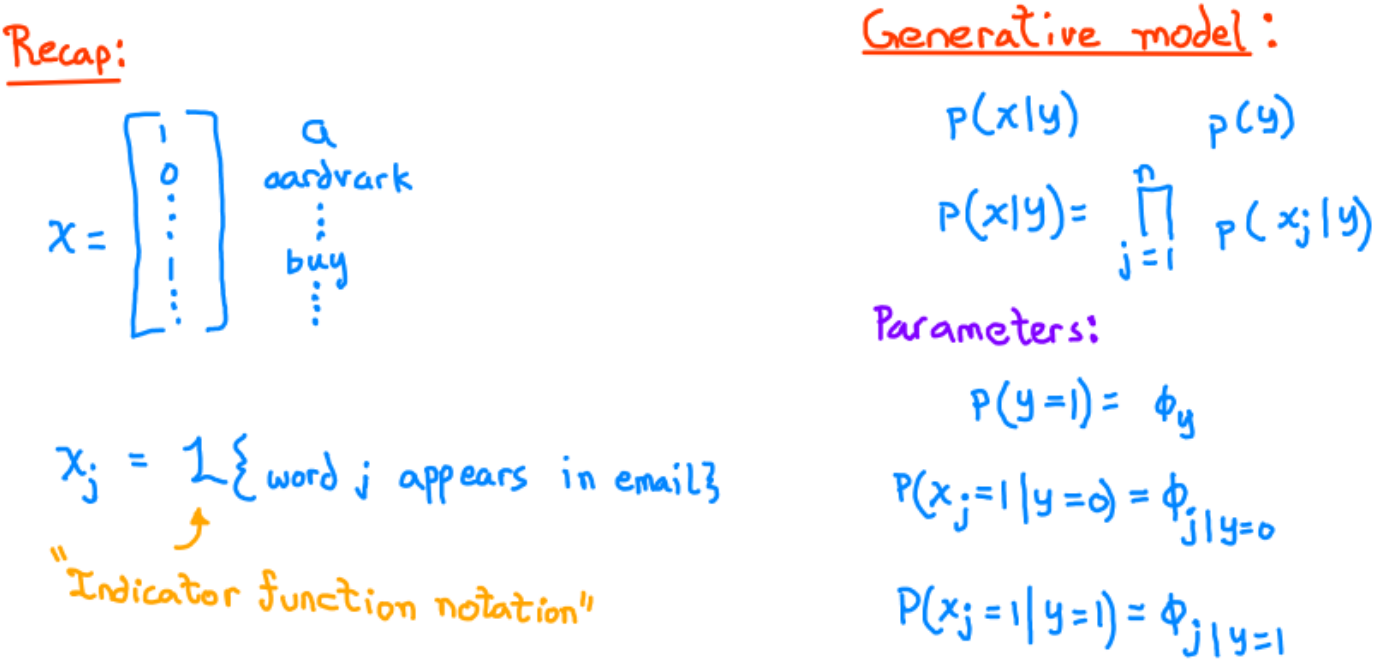
Lecture 6 [Support Vector Machines]

Naive Bayes:



The Naive Bayes algorithm is a ***Generative Learning algorithm*** in which, given a piece of email or Twitter message or some piece of text, takes a dictionary and puts in 0's and 1's in your feature vector (feature representation) depending on whether different words appear in a particular email, and so that becomes your feature representation for an email that you're trying to classify as spam or not spam, for example

#subscript j is supposed to denote the indexes and the features, and subscript i to denote the index in the training examples

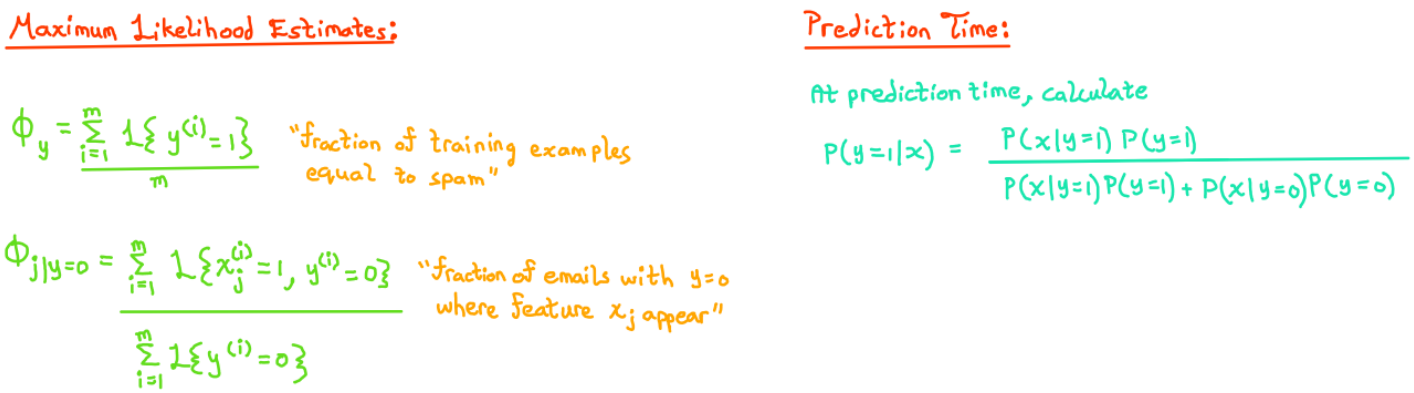
To build a **Generative model** for this, we need to model **p(x|y)** and **p(y)**. **Gaussian Discriminant Analysis** models these two terms with a ***Gaussian*** and a ***Bernoulli*** respectively, and Naive Bayes uses a different model. With Naive Bayes in particular, **p(x|y)** is modeled as a product of the conditional probabilities of the individual features given the class label **y**

Φ_y is the class prior (What's the chance that y = 1 before you've seen any features)

$\Phi_{j|y=0}$ (The chance of that word appearing in a non-spam email)

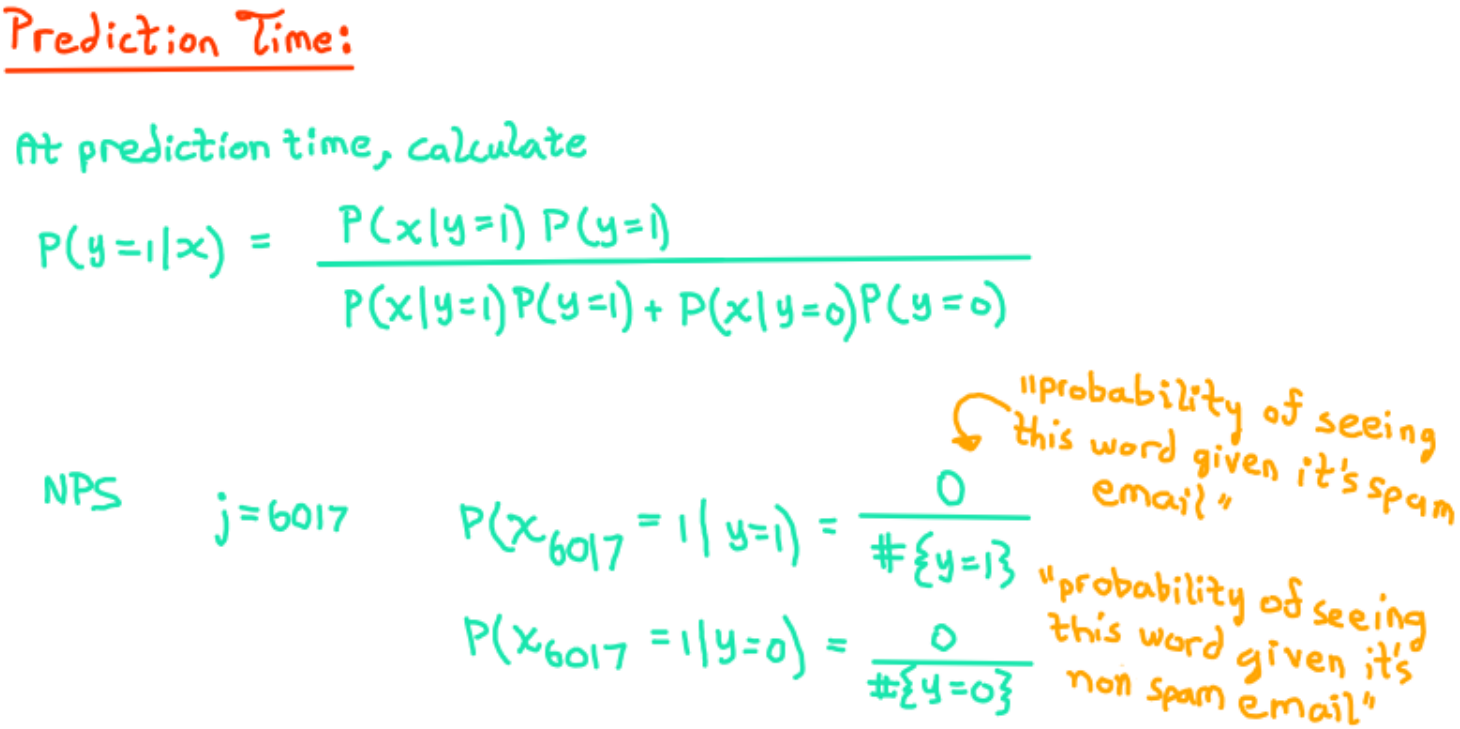
$\Phi_{j|y=1}$ (The chance of that word appearing in a spam email)

If you derive the ***Maximum Likelihood Estimates***, you will find that the Maximum Likelihood Estimates of Φ_y is:

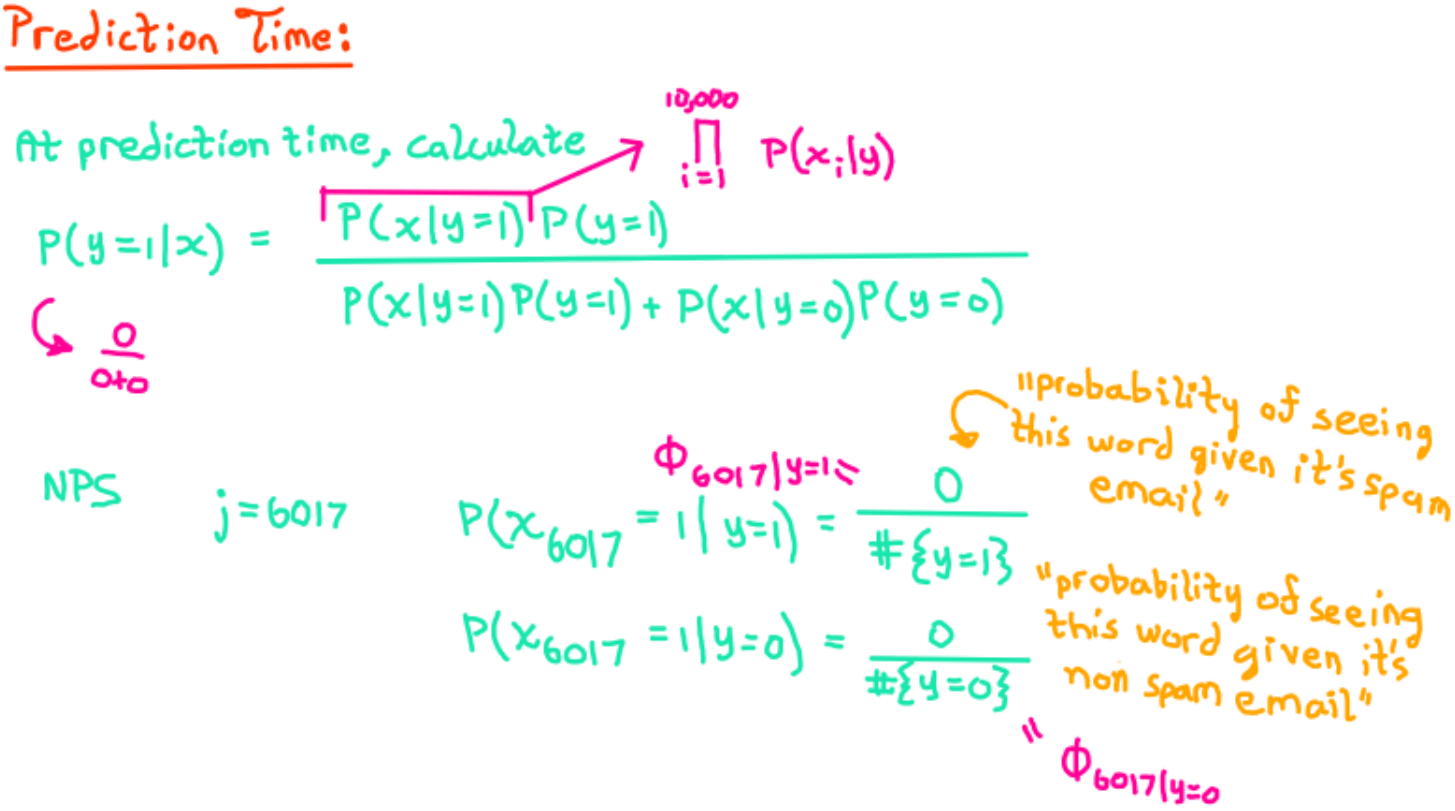


#One of the top Machine Learning conferences is the NIPS (Neural Information Processing Systems) conference

It turns out that this algorithm will almost work. when you haven't seen a word before in any of your past emails, you calculate the chance of it being spam and nonspam as being 0

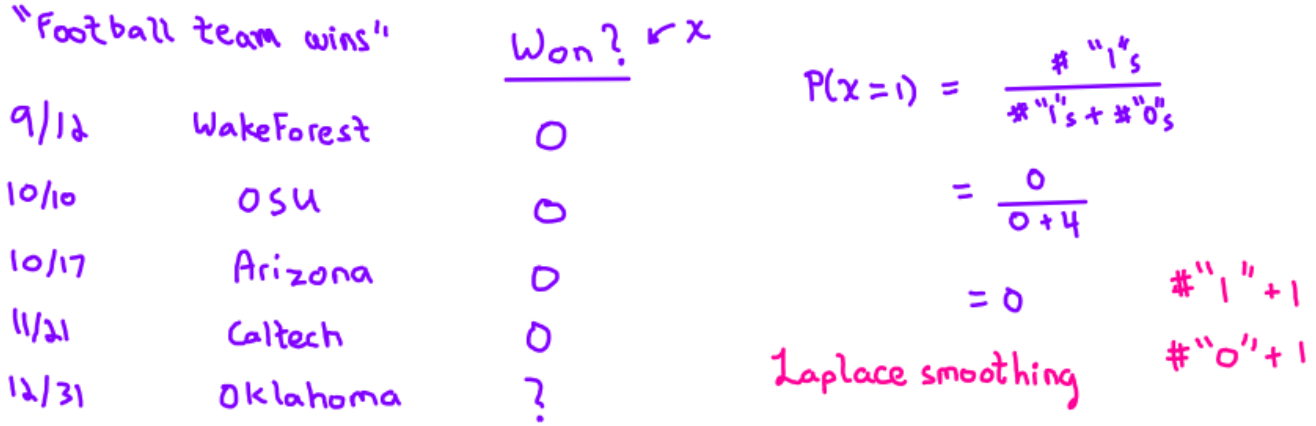


Statistically it is bad to say that that chance of something is 0 just because you haven't seen it yet and where this will cause the Naive Bayes algorithm to break down is, if you use these as estimates of the parameters:

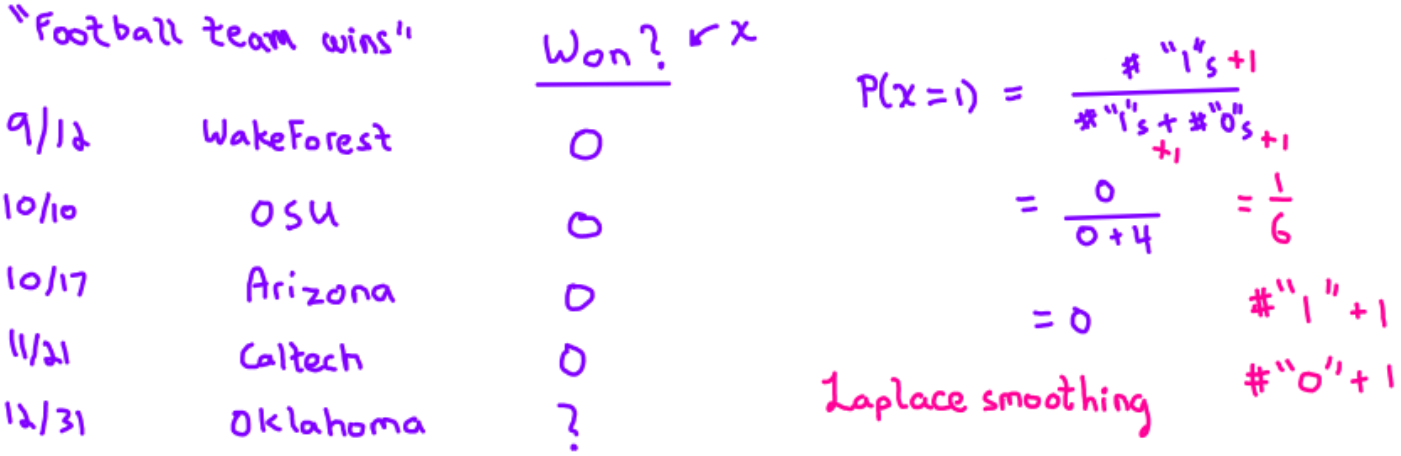


Apart from the Divide-by-0 error, it turns out that statistically, it's just a bad idea. Laplace smoothing is a technique that helps address this problem

{"Lah"-"plaws"}
Laplace smoothing:



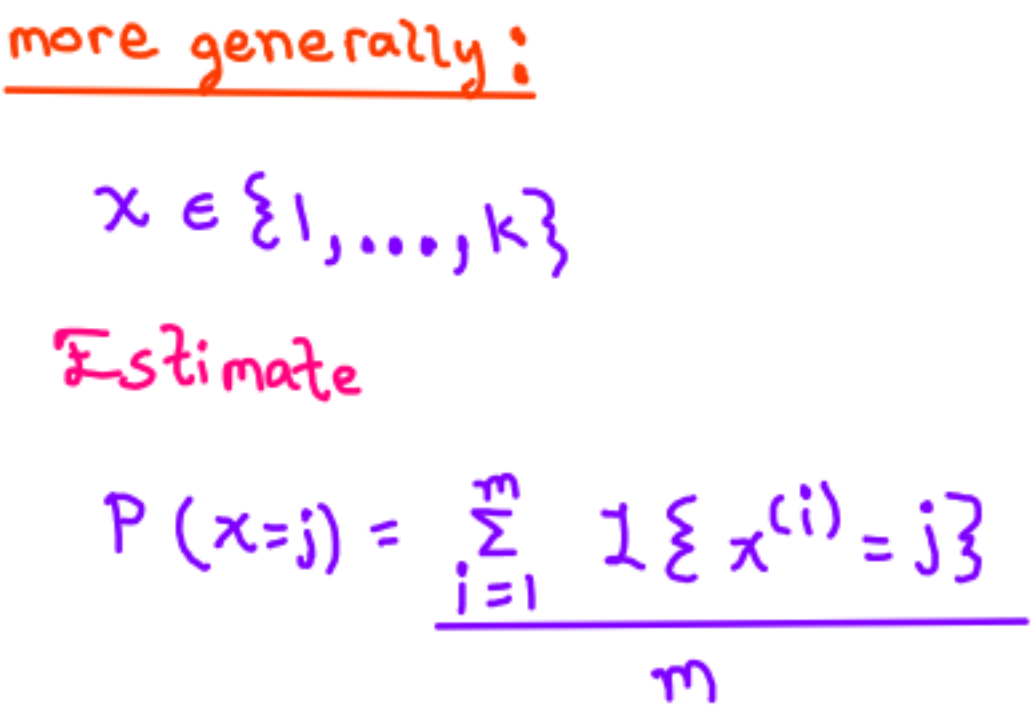
With Laplace smoothing, what we're going to do is imagine that we saw the positive outcomes (the number of wins) as #"1"s + 1 and the negative outcomes (the number of losses) as #"0"s + 1



Which gives a more reasonable estimate for the chance of the football team winning or losing the next game

There's a certain set of circumstances under which there's an optimal estimate. If you assume that you are Bayesian, with a uniform Bayesian prior on the chance of the sun rising tomorrow, given the chance the sun rising tomorrow is uniformly distributed in the unit interval anywhere from 0 to 1, then after a set of observations of the coin toss of whether the sun rises, it will be a Bayesian optimal estimate of the chances of the sun rising tomorrow

If you're estimating probabilities for a k-way random variable, then you estimate the chance of (**x = j**) to being equal to:



This is the ***Maximum Likelihood Estimate*** and with Laplace smoothing, you'd add 1 to the numerator and add k to the denominator

more generally:

$x \in \{1, \ldots, k\}$

Estimate

$$P(x=j) = \frac{\sum_{i=1}^{m} 1\{x^{(i)} = j\} + 1}{m + k}$$

For Naive Bayes, the way this modifies your parameter estimates is this:

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1, y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}$$

This is the **Maximum Likelihood Estimate**, and with **Laplace smoothing** you add 1 to the numerator and 2 to the denominator

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1, y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} + 2}$$

This means that your estimates are never exactly 0 or exactly 1, which takes away the problem of 0 divided by 0

In the examples discussed so far, the features were binarly valued

*Quick Generalization:

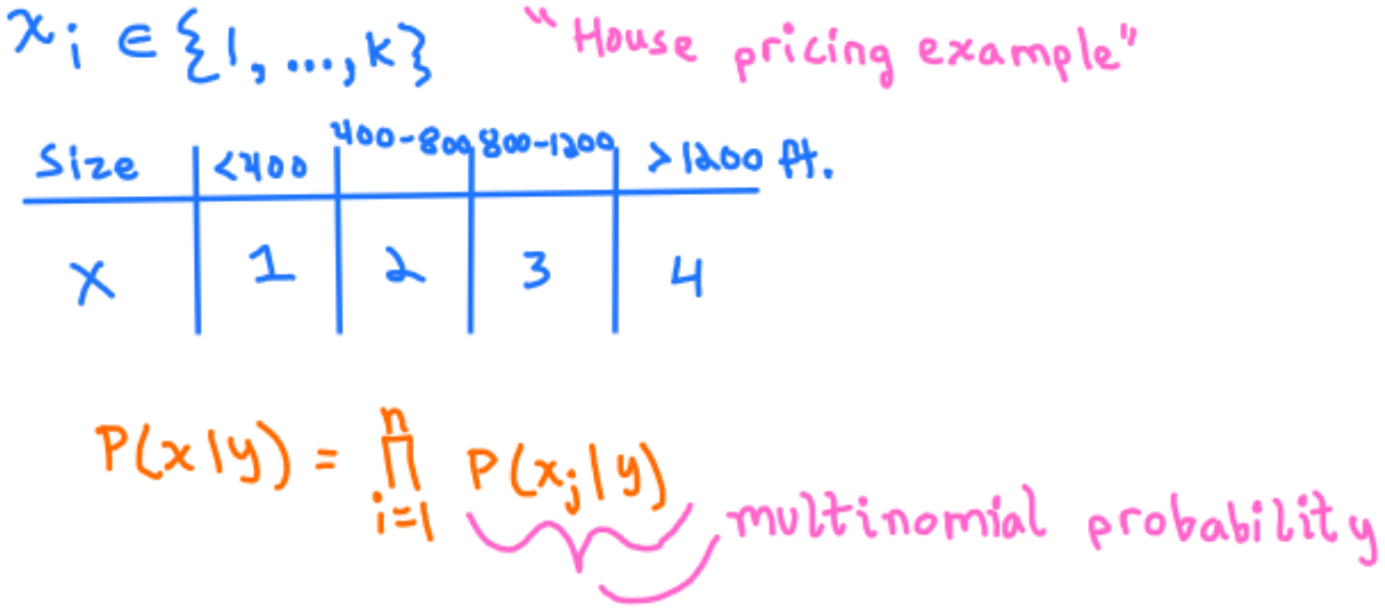
When the features are multinomial valued, then the generalization:

$x_i \in \{1, \ldots, k\}$ "House pricing example"

Size	<400	400-800	800-1200	>1200 ft.
x	1	2	3	4

This is how you discretize a continuous valued feature to a discrete value feature

If you want to apply Naive Bayes to this problem:

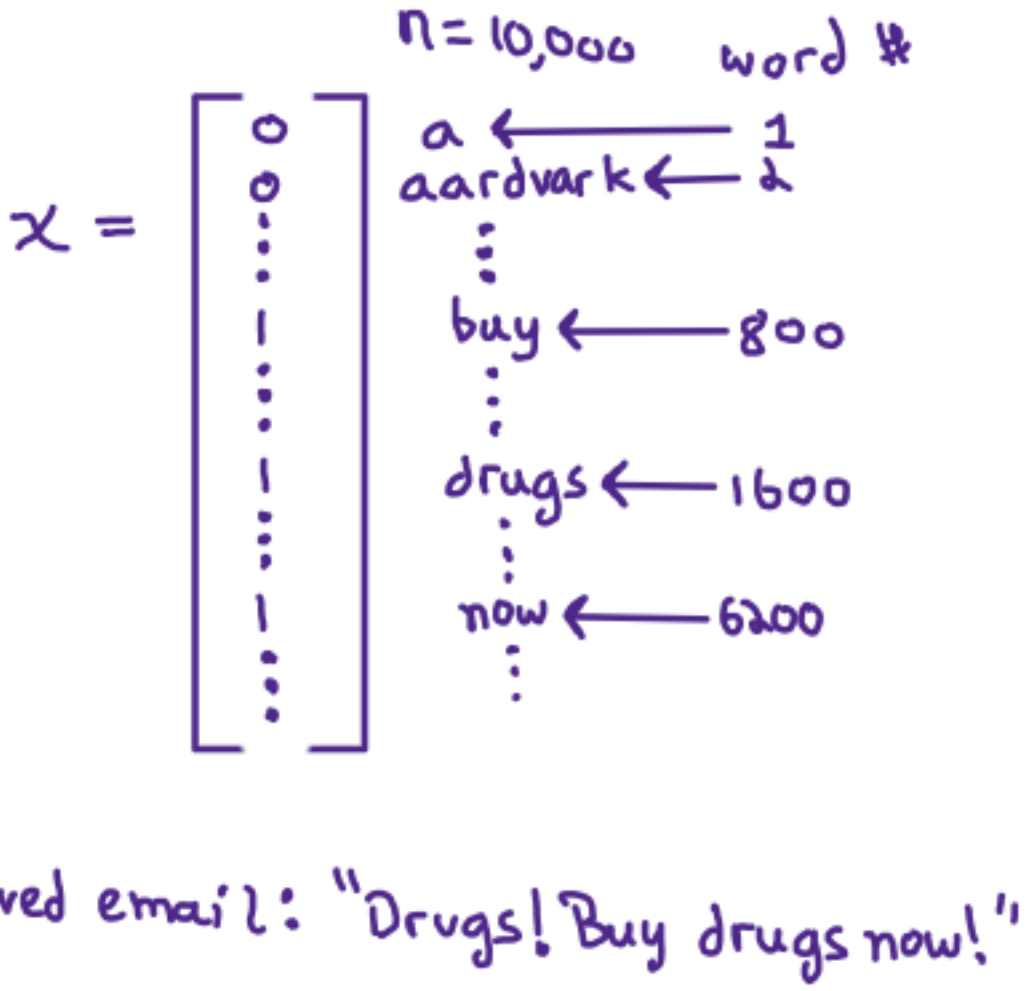


If **x** now takes on one of four values there, then this probability can estimators of multinomial probability. So instead of a Bernoullie distribution over two possible outcomes, this can be a probability mass function probably over four possible outcomes if you discretize the size of a house into four values

If you've ever discretized variables, a typical **rule of thumb** in machine learning, often we discretize variables into 10 values (into 10 buckets), as that just often seems to work well enough

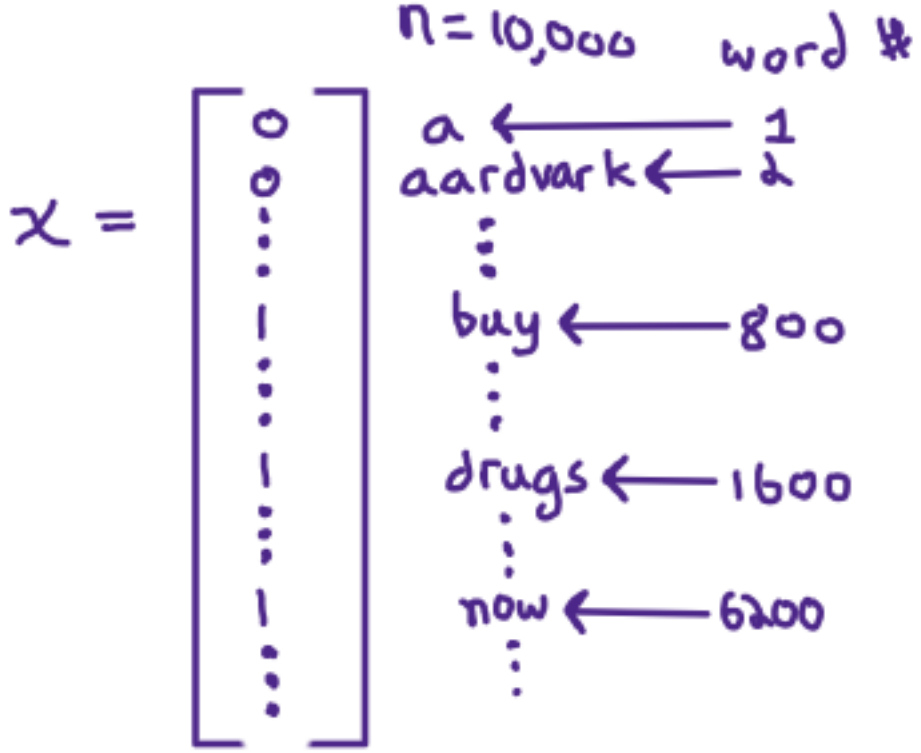
There's a different variation on Naive Bayes that is actually much better for the specific problem of text classification

Our feature representation for x so far was the following:



An interesting thing about Naive Bayes is that it throws away the fact that word “drugs” has appeared twice, which is losing a little bit of information

In this feature representation, each feature is either 0 or 1, which is part of why it throws away the information where the one word “drugs” appears twice and maybe should be given more weight in your classifier

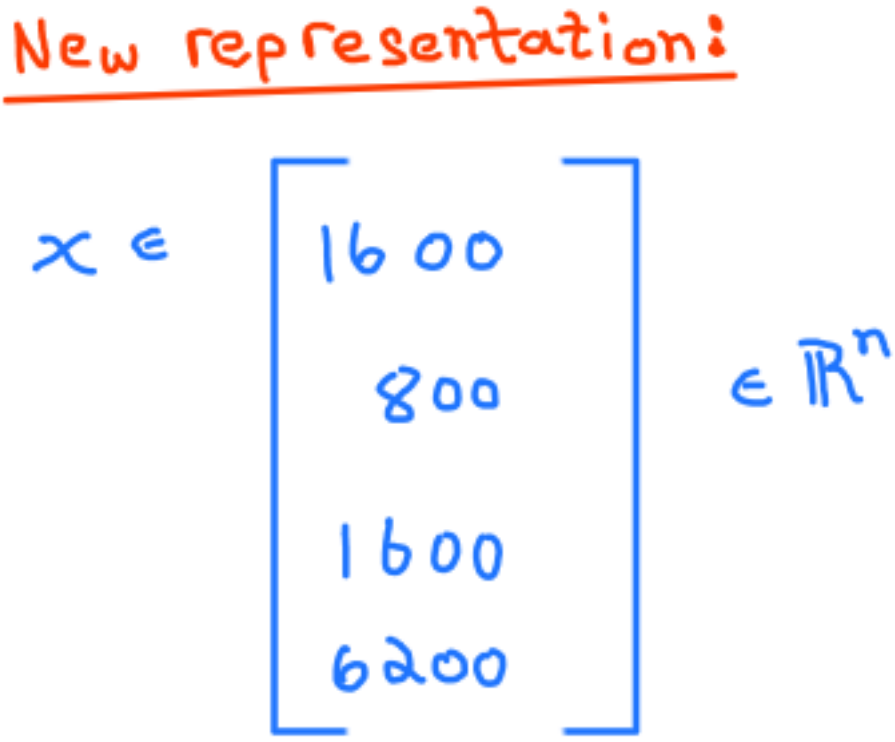


Received email: "Drugs! Buy drugs now!"

$x_i \in \{0, 1\}$

There's a different representation which is specific to text

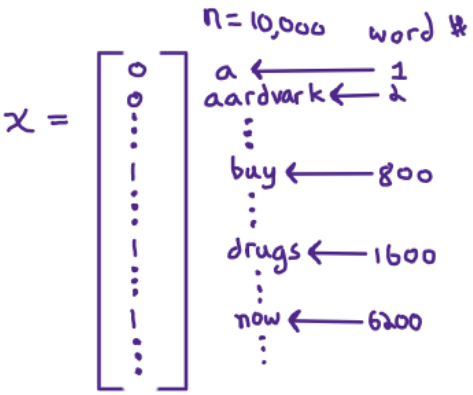
Text data has a property that they can be very long or very short



For that email is going to be represented as a 4-dimensional feature vector. It is going to be **n-dimensional** for an email of length **n**

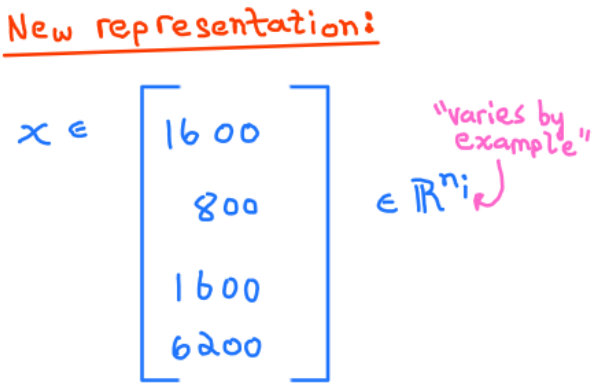
So rather than a 10,000-dimensional feature vector, we now have a 4-dimensional feature vector, but now $\mathbf{x_j}$ is an index from 1 to 10,000 instead of just being 0 or 1

n varies by training, so $\mathbf{n_i}$ is the length of email **i**. So for a longer email, the feature vector **x** will be longer, and if you have a shorter email, this feature vector will be shorter



Received email: "Drugs! Buy drugs now!"

$x_i \in \{0,1\}$

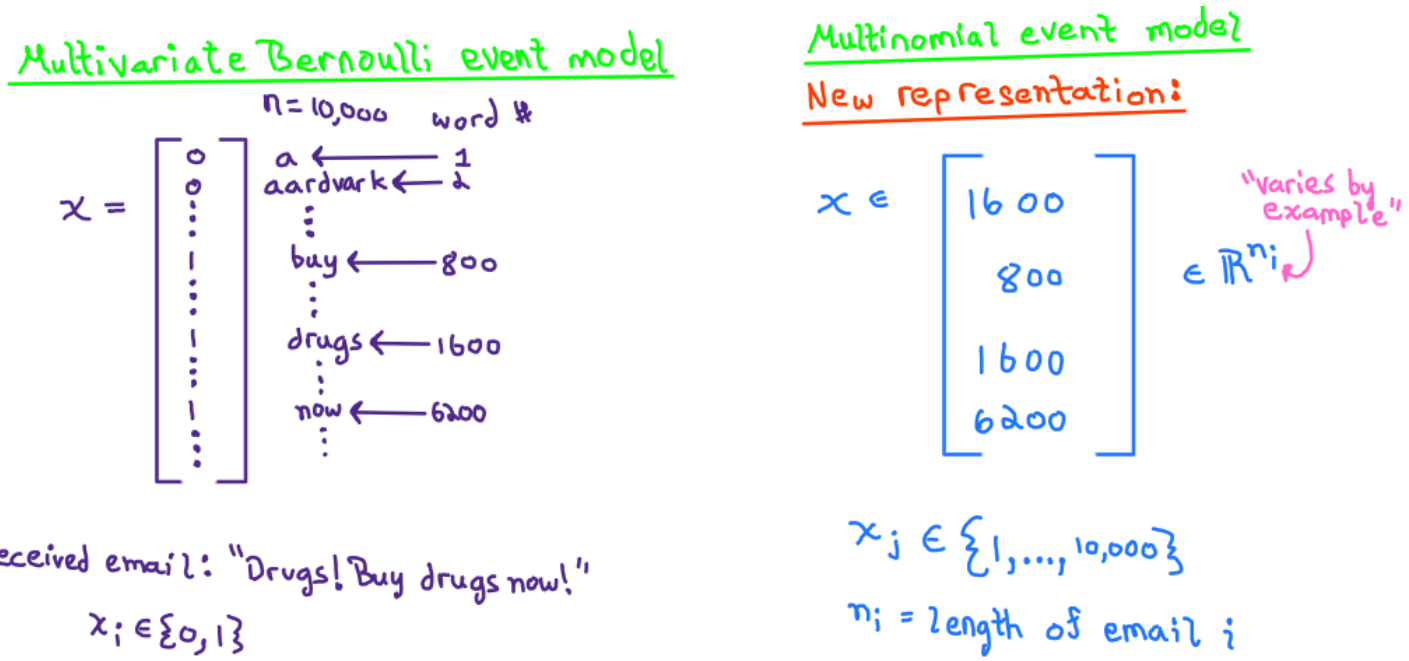


$x_j \in \{1,\ldots,10{,}000\}$

n_i = length of email i

Event models:

To give names to the algorithms we're going to develop:



Bernoulli means “coin tosses” and multivariate means there are **n** number of Bernoulli random variables in this model, and event comes from statistics

With this new representation called the Multinomial event model, we're going to build a generative model:

$$P(x,y) = P(x|y)P(y)$$

$$\text{“Assume Naive Bayes”} = \prod_{j=1}^{n} P(x_j|y)P(y)$$

One of the reasons these two models were very confusing to the machine learning community is because this is exactly the equation that was discussed in Lecture 5 [GDA & Naive Bayes], so this equation looks cosmetically identical. But with this new model, **Multinomial Event Model**, the definition of $\mathbf{x_j}$ and the definition of **n** is very different

So instead of a product from 1 through 10,000, it's a product from 1 through the number of words in the email, and it is now a

multinomial probability rather than a binary or Bernoulli probability

It turns out that with this model, the parameters are:

$$P(x,y) = P(x|y)P(y)$$

"Assume Naive Bayes" $$= \prod_{j=1}^{n} P(x_j|y)P(y)$$

Parameters :

$$\phi_y = P(y=1)$$

$$\phi_{k|y=0} = P(x_j = k|y=0)$$

"Chance of word j being k if label y=0"

So, it's the chance of the third word in the email being the word “drugs” or the chance of the second word in the email being “buy”, or etc.

We assume that this probability doesn't depend on **j**, that for every position in the email, for the chance of the first word being “drugs”, is the same as the chance of the second word being “drugs”, same as the third word being “drugs”, and so on and so forth. Which is why on the left-hand side, **j** doesn't actually appear on the left-hand side

So the way that, given a new (test) email, you would calculate this probability is by plugging these parameters that you estimate from the data into the formula $P(\mathbf{x}_j=k|y=0)$

$$P(x,y) = P(x|y)P(y)$$

"Assume Naive Bayes" $$= \prod_{j=1}^{n} P(x_j|y)P(y)$$

Parameters:

$$\phi_y = P(y=1)$$

$$\phi_{k|y=0} = P(x_j = k|y=0)$$

"Chance of word j being k if label $y=0$"

$$\phi_{k|y=1} = P(x_j = k|y=1)$$

M.L.E. of parameters:

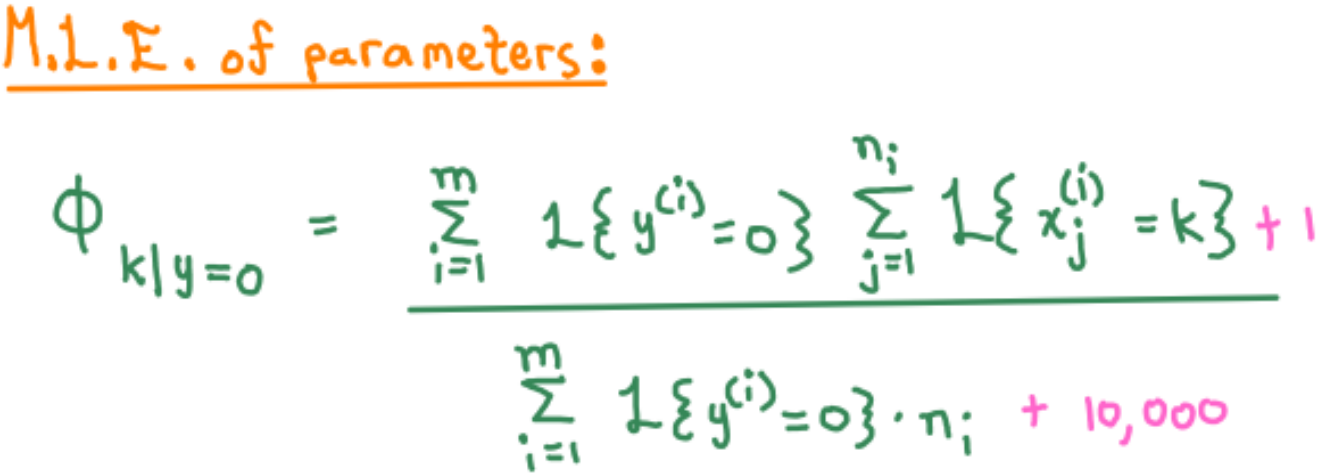
$$\phi_{k|y=0} = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=0\} \sum_{j=1}^{n_i} 1\{x_j^{(i)}=k\}}{\sum_{i=1}^{m} 1\{y^{(i)}=0\} \cdot n_i}$$

The english meaning of this formula basically says, “Look at all the words in all of your non-spam emails, and look at all of the words in all of the emails, and so of all of those words, what fraction of those words is the word ‘drugs’?” And that's your estimate of the chance of the word “drugs” appearing in the non-spam email in some position in that email

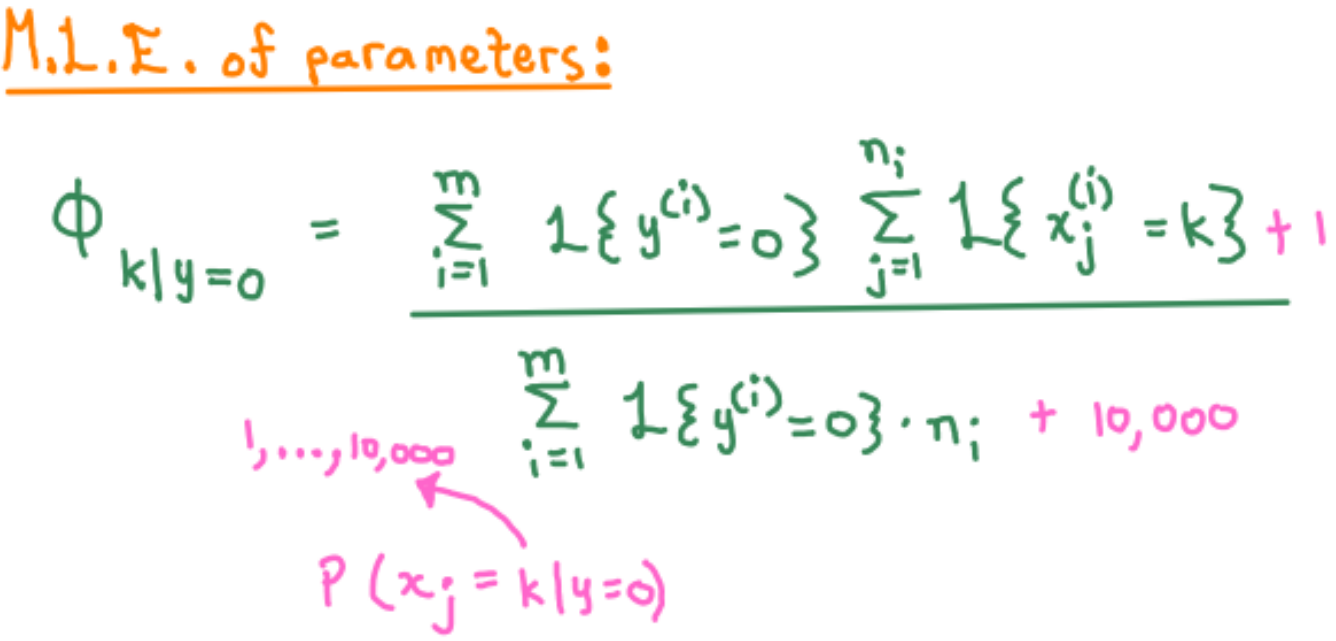
In math, the denominator is the sum of your training set (indicator is not spam) times the number of words in that email. So the denominator ends up being the total number of words in al of your non-spam emails in your training set and the numerator as some of your training set, sum from **i=1** through **m**, indicator **y=0** (so count up only the things for non-spam email, and for the non-spam email, **j=1** through $\mathbf{n_i}$, go over the words in that email and see how many words are that word **k**

So if in your training set, you have 100,000 words in your non-spam emails and 200 of them are the word drugs (occurs 200 times), then this ratio will be 200/100,000

Lastly, to implement **Laplace smoothing** with this, you would add 1 to the numerator as usual and then add the number of possible outcomes (10,000) to the denominator



So this is the probability of **x** being equal to the value of **k**, where **k** ranges from 1-10,000 (if you have a dictionary size / list of 10,000 words you're modeling). And so the number of possible values for **x** is 10,000, so you add 10,000 to the denominator



#What do you do if the word's not in the dictionary?

There are two approaches to that. One approach is to just throw it away, ignore it, disregard it, etc. The second approach is to take the rare words and map them to a special token which traditionally is denoted **UNK** for ***unknown words***

So if in your training set, you decide to take just the top 10,000 words into your dictionary, then everything that's not in the top 10,000 words can map to your unknown word token or the unknown words special symbol

The indicator function evaluates to 1 if true and 0 if false based on its parameters

Comments on applying Machine Learning algorithms:

When would you use the Naive Bayes algorithm?

It turns out the Naives Bayes algorithm is actually not very competitive with other learning algorithms. So for most problems you find that logistic regression will work better in terms of delivering a higher accuracy than Naive Bayes

But the advantages of Naive Bayes is, first it's computationally very efficient, and second it's relatively quick to implement. It also doesn't require an iterative gradient descent thing and the number of lines of code needed to implement Naive Bayes is relatively small

So if you're facing a problem where your goal is to implement something quick and dirty, then Naive Bayes is maybe a reasonable choice

If your goal is not to invent a brand new learning algorithm but to take the existing algorithms and apply them, then a rule of thumb that's suggested is, when you get started on a machine learning project, start by implementing something quick and dirty. Instead of implementing the most complicated possible learning algorithms, start by implementing something quickly and train the algorithm, look at how it performs, and then use that to deep out the algorithm and keep iterating on that

If your goal is to make something work for an application rather than inventing a new learning algorithm and publishing a paper on a new technical contribution, such as if your working on an application on understandings news better, improving the environment, or estimating prices, etc, and your primary objective is to just make an algorithm work, then rather than building a very complicated algorithm at the onset, it is recommended to implement something quickly so that you can better understand how it's performing, and then do error analysis and use that to drive your development

It turns out that when you're starting out on a new application problem, it's hard to know what's the hardest part of the problem. So if you want to build an anti-spam classifier, there's lot of things you could work on. For example, spammers will deliberately misspell words which might cause the spam filter to map the word to an unknown word. Another idea might be that a lot of spam emails spoofs email headers, and another thing you might do is try to fetch the URLs that are referred to in the email and then analyze the web pages that you get to. There are a lot of things that you could do to improve a spam filter, and any one of these topics could easily be three months or six months of research. But when you're building, for example, a new spam filter for the first time, how do you actually know which of these is the best investment of your time

So some advice to those who work on projects, if your primary goal is to just to get things to work, is to not somewhat arbitrarily dive in and spend six months on improving spam filter handling of misspelled words or spend six months on trying to analyze email headers, but to instead implement a more basic algorithm. Almost implement something quick and dirty, and then look at the examples that your learning algorithm is still misclassifying

You'll find that, if after you've implemented a quick and dirty algorithm you find that your anti-spam algorithm is misclassifying a lot of examples with (for example) deliberately misspelled words, it's only then that you have more evidence that it's worth spending a bunch of time solving the deliberately misspelled words problem

When you implement a spam filter and you see that it's not misclassifying a lot of examples of these misspelled words, then you most likely should not bother. Go work on something else instead or at least treat that as a low priority

One of the uses of **GDA (Gaussian Discriminant Analysis)** as well as **Naive Bayes** is that they're not going to be the most accurate algorithms. If you want the highest classification accuracy, there are other algorithms like **Logistic Regression** or **SVM (Support Vector Machines)**, or **Neural Networks**, which will almost always give you higher classification accuracy than these algorithms. But the advantage of **Gaussian Discriminant Analysis** and **Naive Bayes** is that they are very quick to train or it's non-iterative (this is just counting) and **GDA** is just computing means and covariances. So it's very computation efficient and they're also simple to implement, so it can help you implement that quick and dirty thing that helps you get going more quickly

As you start working on your project, it is advised to not spend weeks designing exactly what you're going to do if you have an applied project, but instead get a data set and apply something simple. Start with **Logistic Regression** and not a **Neural Network** or not something more complicated. Or start with **Naive Bayes** and then see how that performs and then go from there

Can you use Logistic Regression with discrete variables?

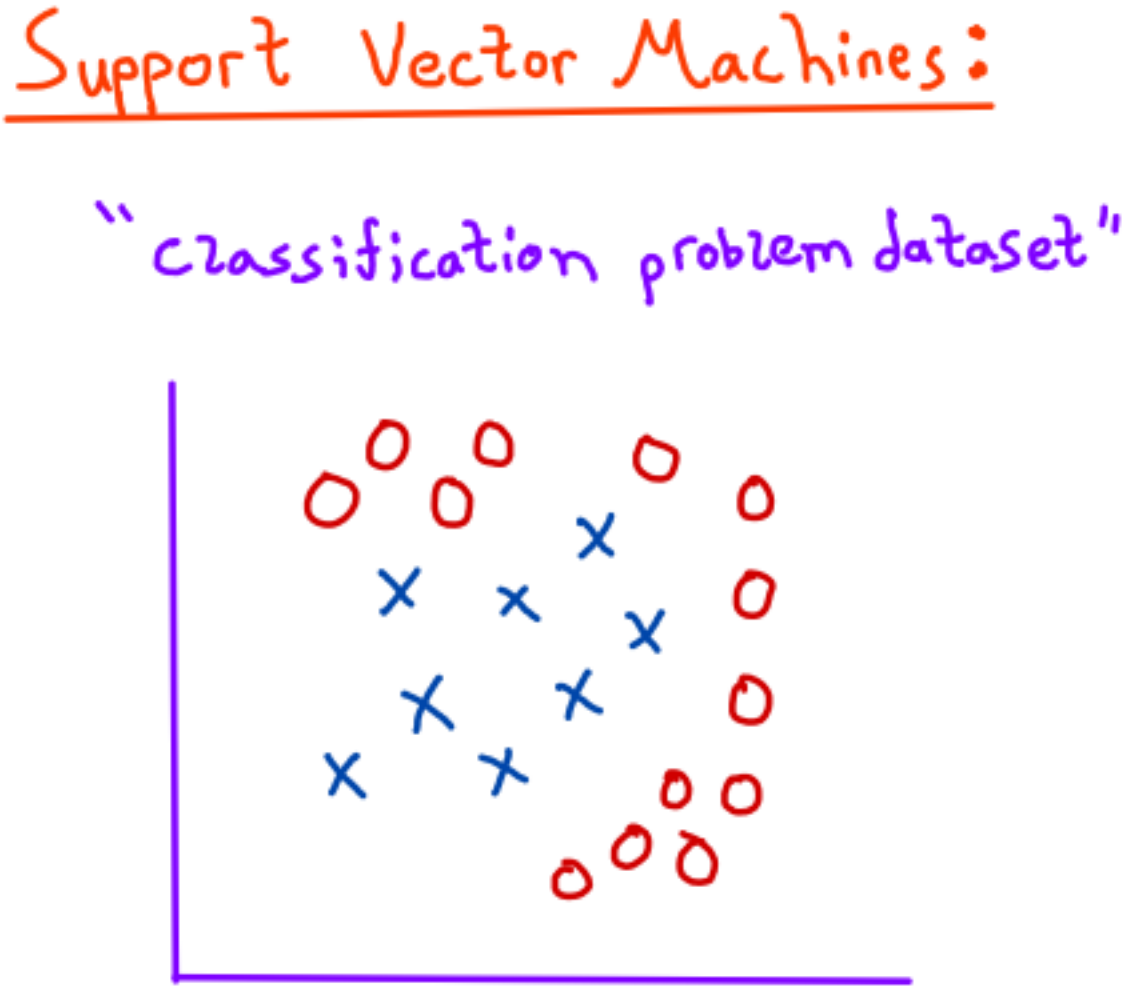
One of the weaknesses of the Naive Bayes algorithm is that it treats all of the words as completely separate from each other. And so the words “one” and “two” are quite similar and the words “mother” and “father” are quite similar, and so with this feature representation, it doesn't know the relationship between these words.

So in machine learning, there are other ways of representing words. There's a technique called **Word Embeddings** in which you choose the feature representation that encodes the fact that the words “one” and “two” are quite similar to each other, the words “mother” and “father” are quite similar to each other, the words “London” and “Tokyo” are quite similar to each other because they're both city names, etc. And so this is a technique from **Neural Networks** which will reduce the number of training examples you need so they are a good text classifier because it comes in with more knowledge baked in

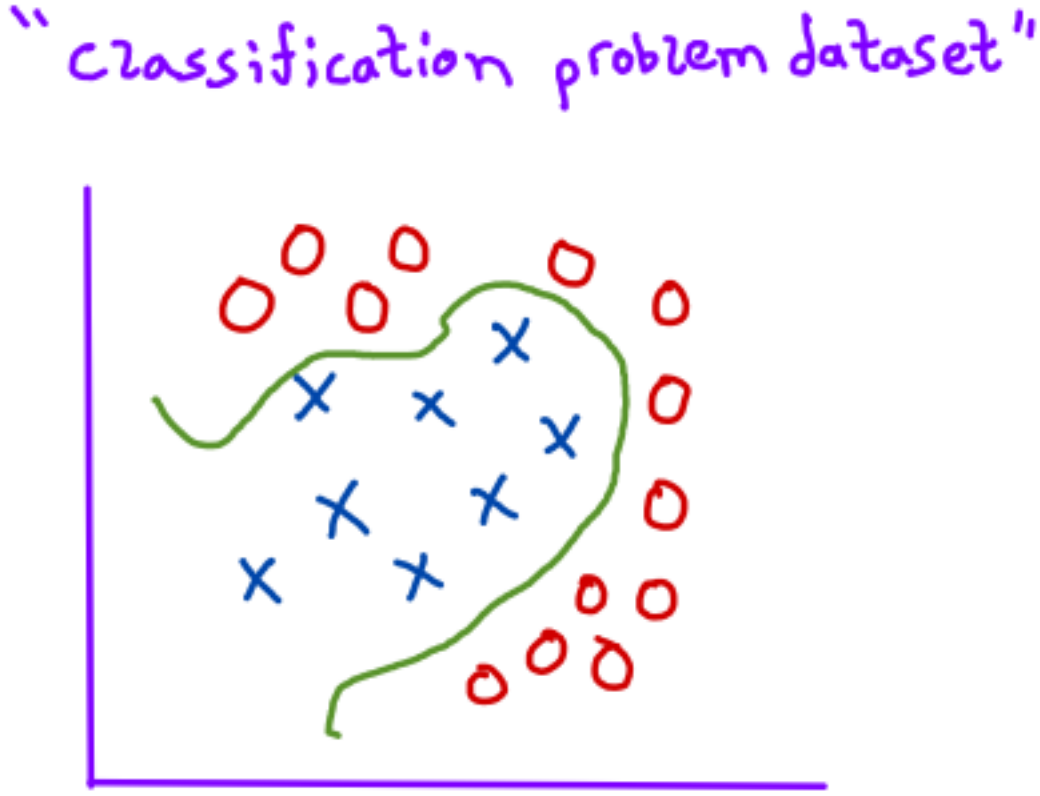
Support Vector Machines Intro:

A different type of classifier is **Support Vector Machines**

Let's say the classification problem, where the datasetlooks like this:



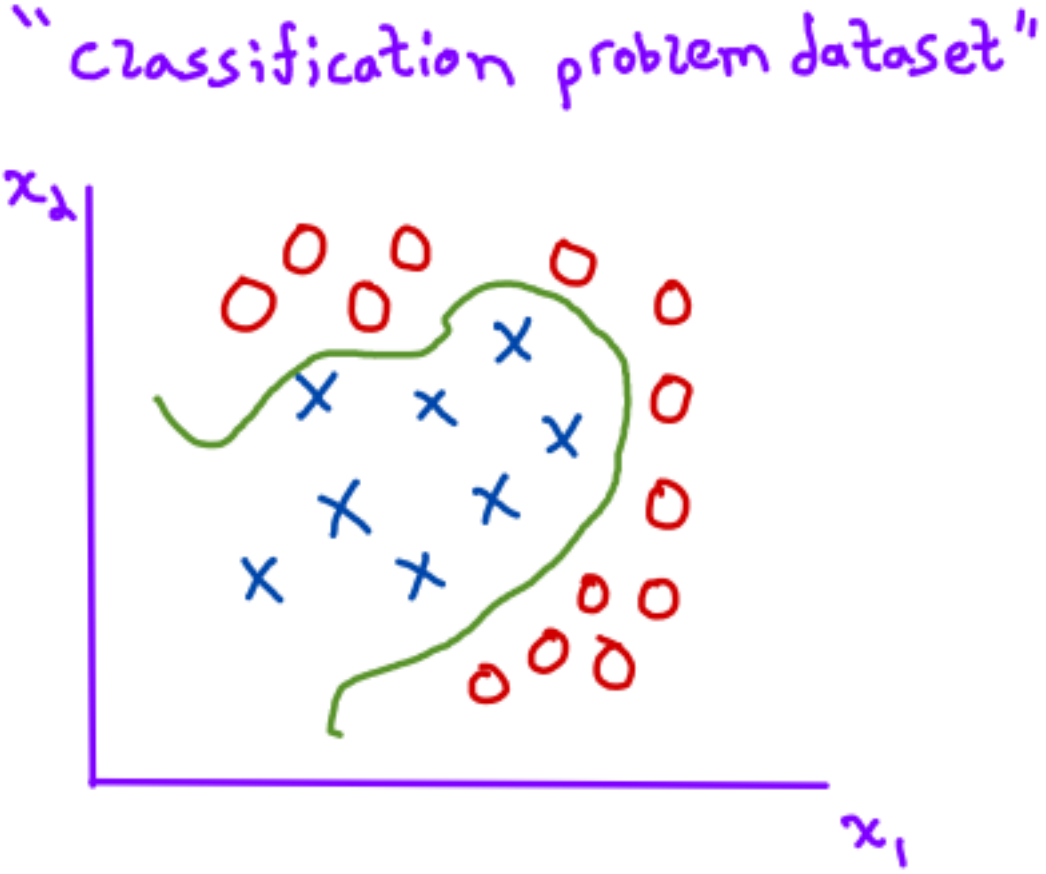
And you want an algorithm to find a non-linear decision boundary:



The **Support Vector Machine (SVM)** will be an algorithm to help us find potentially very very non-linear descision boundaries like this

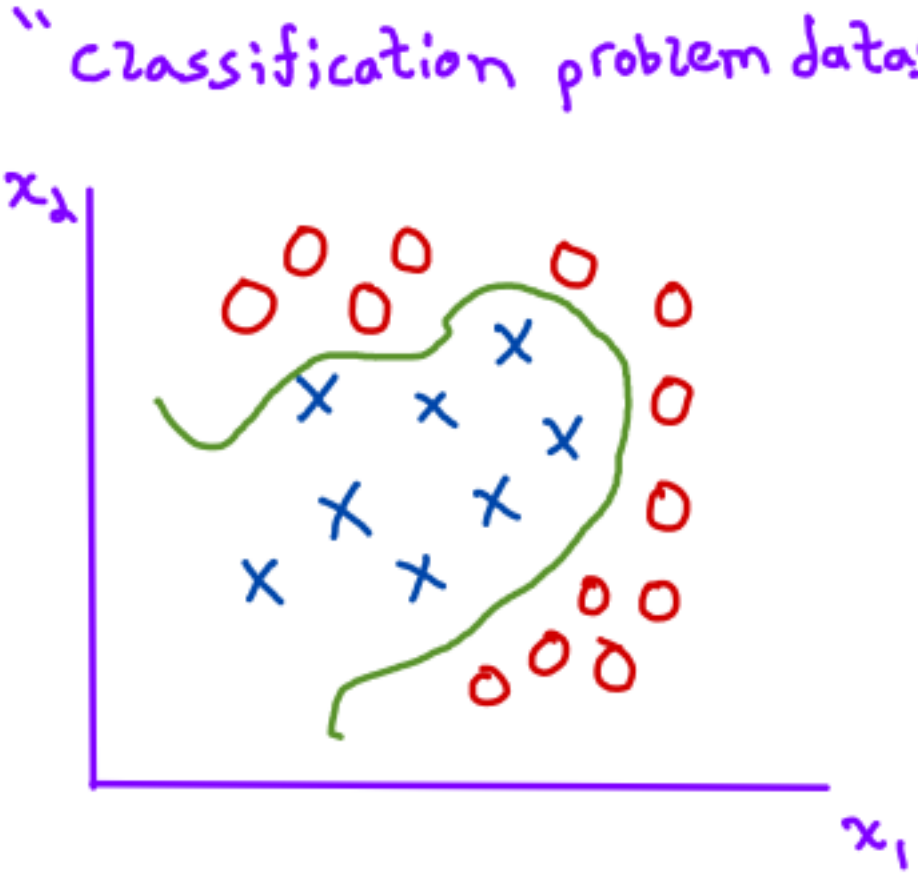
Now one way to build a classifier like this would be to use Logistic Regression

Labelling the axes:



Logistic Regression will fit a straight line to the data. Gaussian Discriminant Analysis will end up with a straight line decision boundary

So one way to apply Logistic Regression like this would be to take your feature vector, $\mathbf{x_1}$ $\mathbf{x_2}$, and map it to a new high dimensional feature vector called $\mathbf{\Phi(x)}$ that has these high-dimensional features

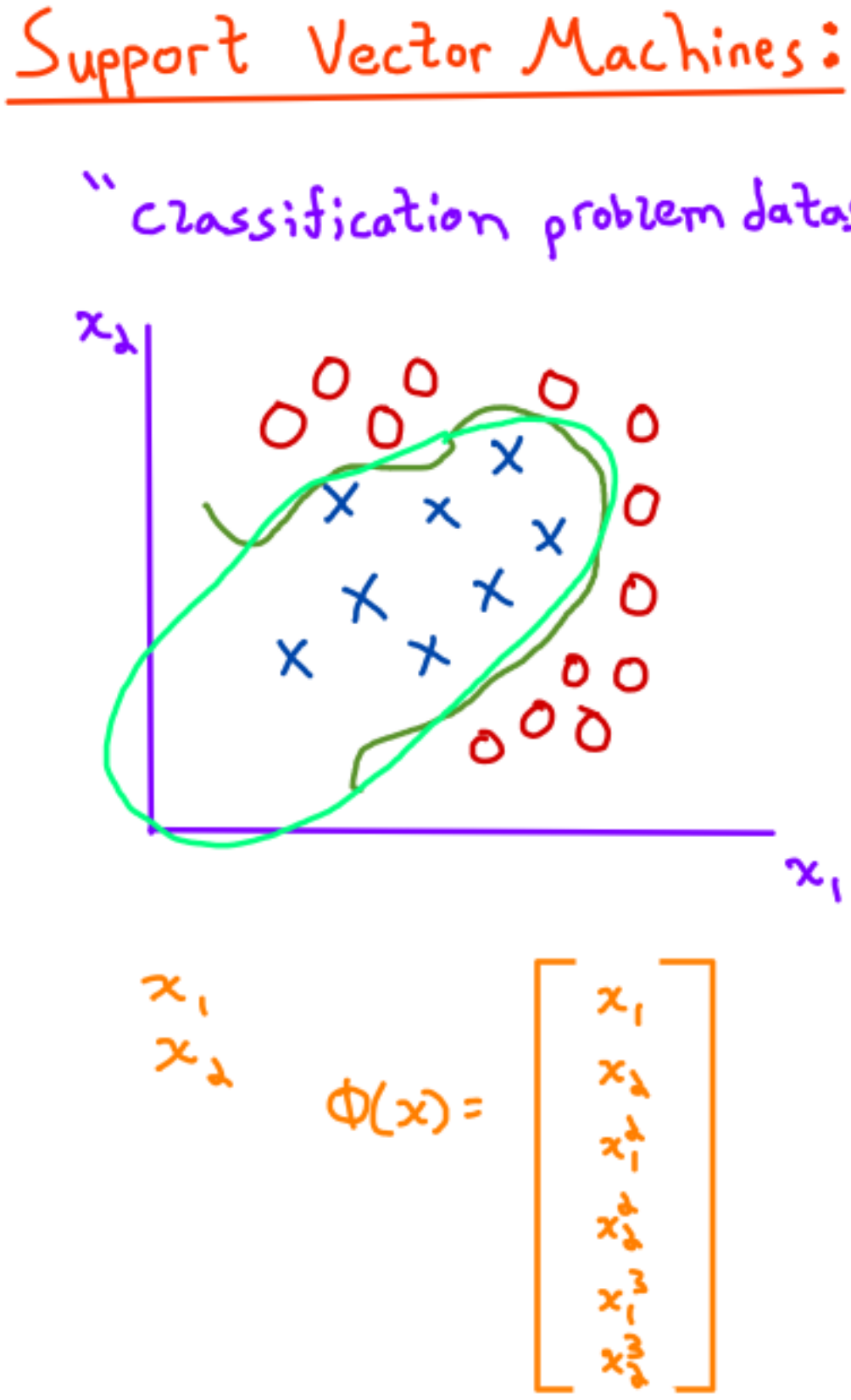


x_1
x_2

$$\phi(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1^3 \\ x_2^3 \end{bmatrix}$$

It turns out that if you do this and then apply **Logistic Regression** to this augmented feature vector, then **Logistic Regression** can learn non-linear decision boundaries

With these other features **Logistic Regression** could actually learn the decision boundary that's the shape of an ellipse:



But randomly choosing these features is a little bit of a pain. What will be seen with **Support Vector Machines** is that we will be able to derive an algorithm that can take, say input features $\mathbf{x}_1$ $\mathbf{x}_2$, map them to a much higher dimensional set of features, and then apply a linear classifier in a way similar to **Logistic Regression** but different in details that allows you to learn very non-linear decision boundaries

One of the reasons **Support Vector Machines** are used today is it's a relatively turn-key algorithm in the sense that it doesn't have too many parameters to fiddle with. Even for **Logistic Regression** or for **Linear Regression**, you might have to tune the learning

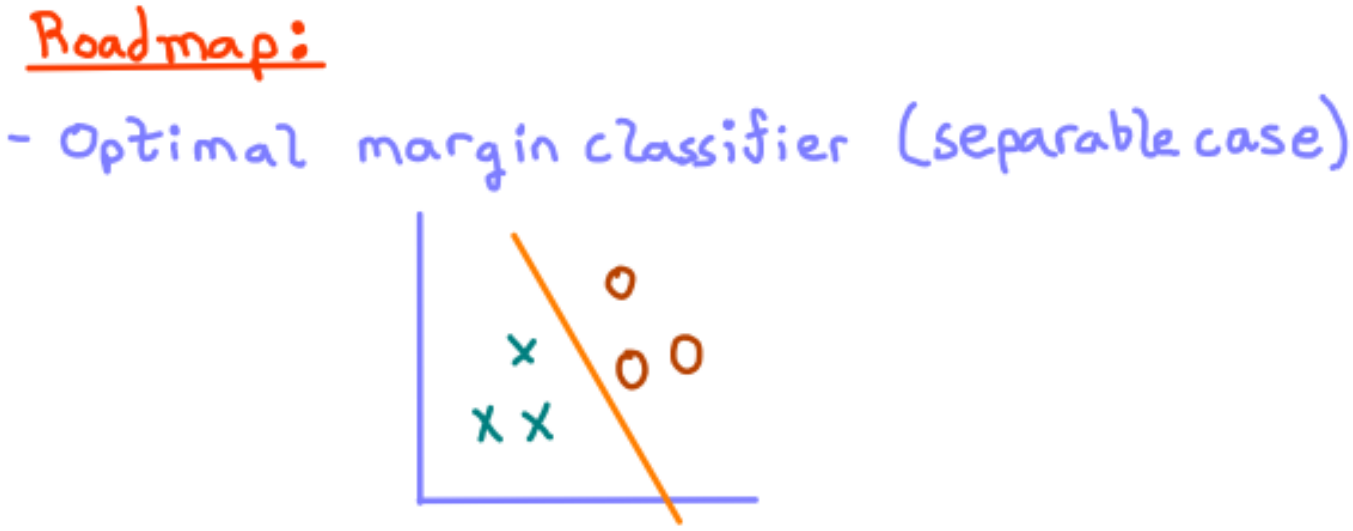
rate **α** parameter, and that's just another thing to fiddle with. You'll try a few values and hope you didn't mess up how you set that value

Support Vector Machine today has a very robust, very mature software packages that you can just download to train the **Support Vector Machine** on a problem, and you just run it and the algorithm will kind of converge without you having to worry too much about the details

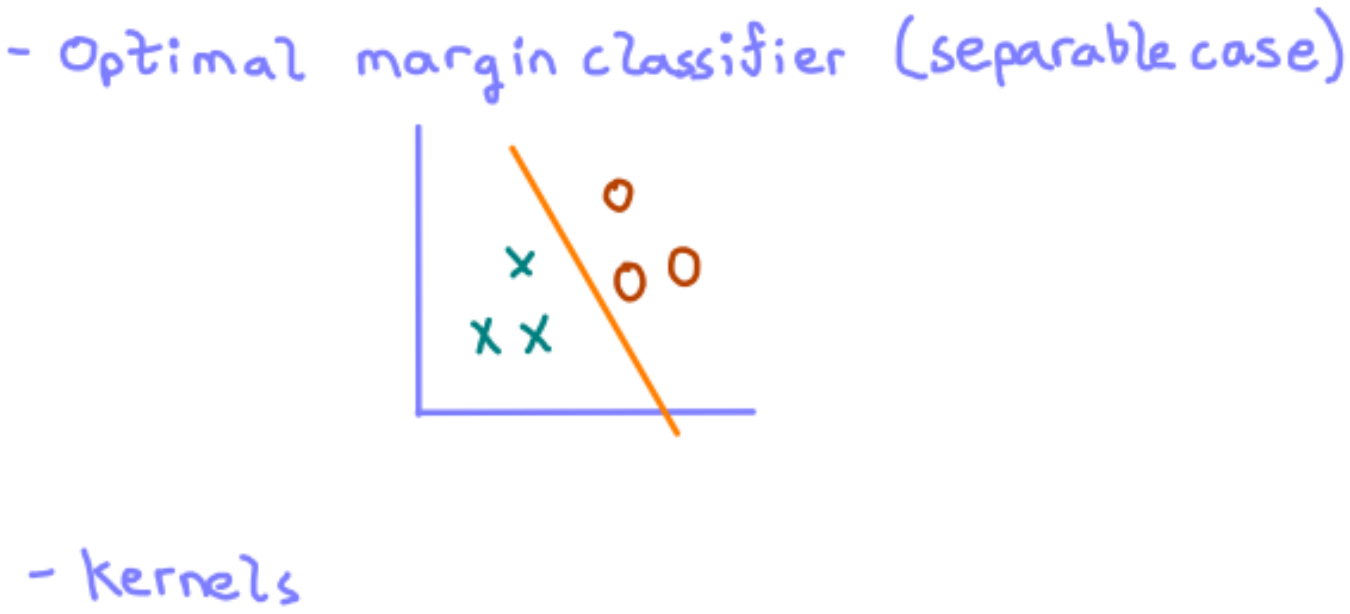
On the grand scheme of things, **Support Vector Machines** are not as effective as **Neural Networks** for many problems, but one great property of **Support Vector Machines** is it's turn-key. You kind of just turn the key and it works and there isn't as many parameters, like the learning rate and other things, that you have to fiddle with

Roadmap:

We're going to develop the following set of ideas:

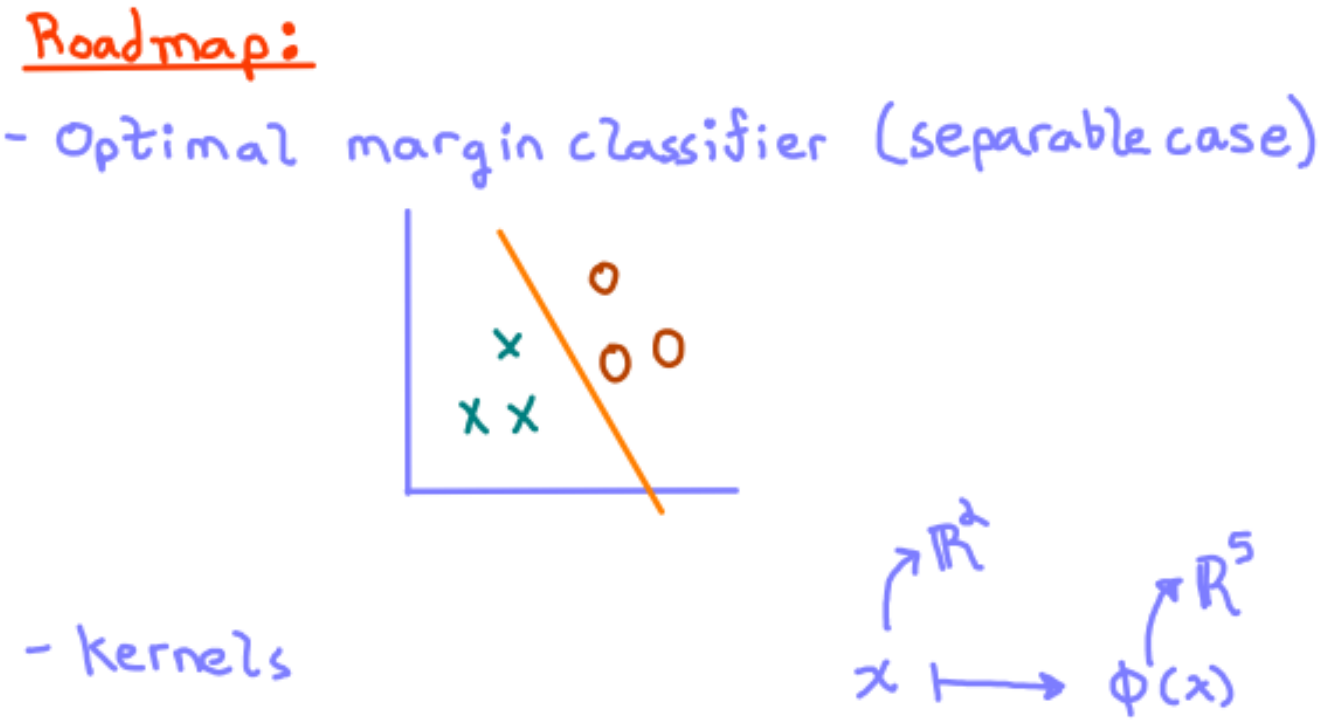


What **separable case** means is that we're going to start off with datasets that we assume look like this and that are linearly separable. So the **Optimal Margin Classifier** is the basic building block for the **Support Vector Machine** and we'll first derive an algorithm that'll have some similarities to **Logistic Regression** but that allows us to scale in important ways, that, to find a linear classifier for training sets like this that we assume for now can be linearly separated



The **Kernel** idea is one of the most powerful ideas in machine learning

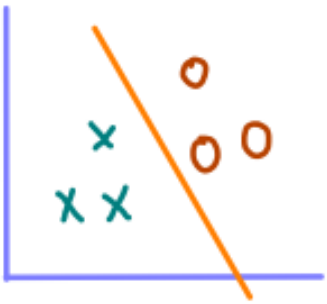
How do you take a feature vector **x** and map it to a much higher dimensional set of features, and then train an algorithm on this high dimensional set of features?



The cool thing about **Kernels** is that this high dimensional set of features may not be 5-dimensional, it might be 100,000-dimensional or it might even be infinite-dimensional

Roadmap:

- Optimal margin classifier (separable case)



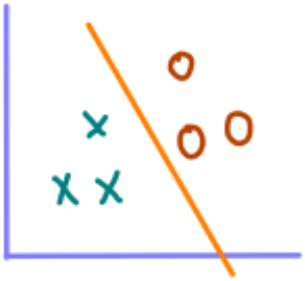
- Kernels

So, with the **Kernel** formulation, we're going to take our original set of features that you are given for the houses you're trying to sell or for medical conditions you're trying to predict, and map this 2-dimensional feature vector space into maybe an infinite-dimensional set of features

What this does is it relieves us from a lot of the burden of manually picking features. You don't have to fiddle with these features too much because the **Kernels** allow you to choose an infinitely large set of features

Roadmap:

- Optimal margin classifier (separable case)



- Kernels

- Inseparable case

Functional Margin:

The **Functional Margin** is, informally, the **functional margin** of the classifier is how well (confidently and accurately) do you classify an example

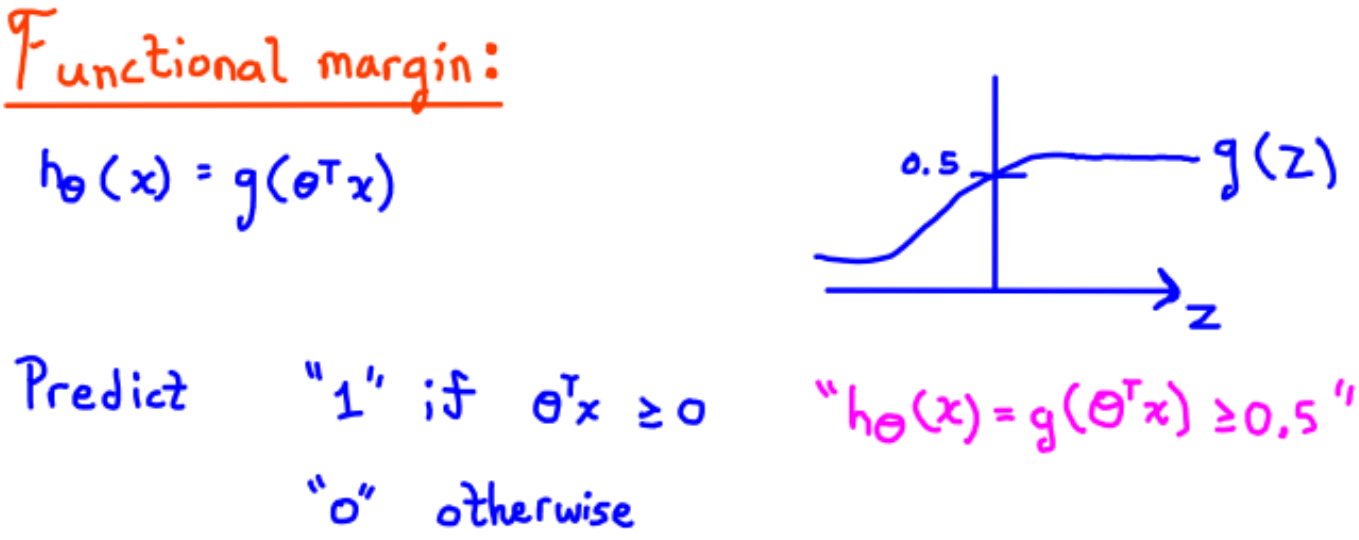
Binary Classification using Logistic Regression:

Functional margin:

$h_\theta(x) = g(\theta^T x)$

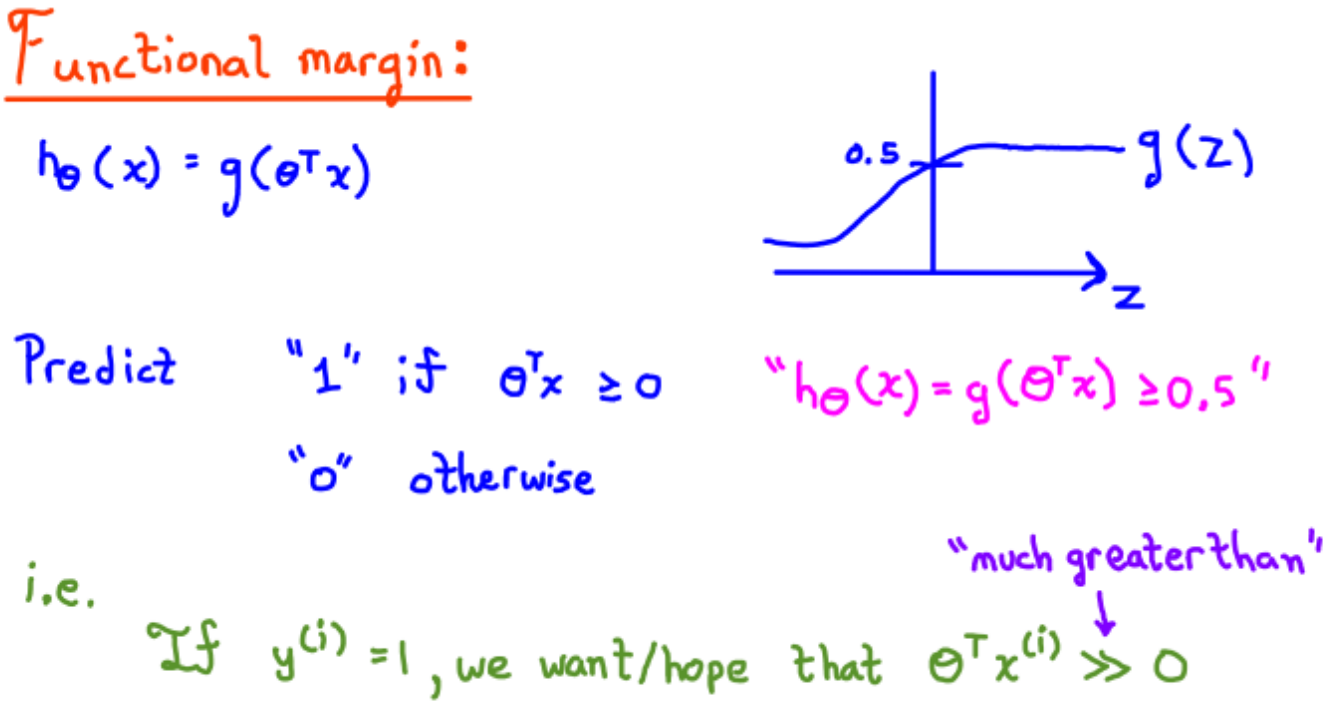
This is a classifier, $\mathbf{h_\Theta}$, where it equals the logistic function applied to $\mathbf{\Theta^T x}$

So, if you turn this into a **binary classification** (if you have this algorithm predict, not a probability, but predict 0 or 1), then what this classifier will do is predict **1** if $\Theta^T x > 0$ and predict **0** otherwise



So, if you predict 1 (if $\Theta^T x \geq 0$), meaning that the estimated (output) probability of a class being **1** is greater than 50/50, so you predict **1**. And if $\Theta^T x < 0$, then you predict that this class is **0**

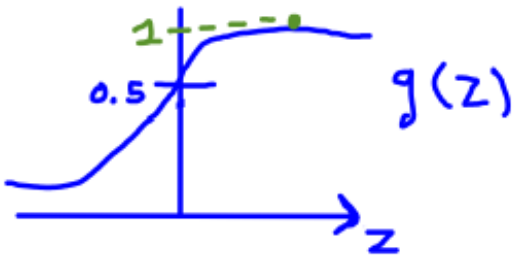
This is what will happen if you have **Logistic Regression** output 1 or 0 rather than output a probability



If the true label is **1**, then if the algorithm is doing well, hopefully $\Theta^T x$ will be far to the right. So the output probability is very very close to 1

Functional margin:

$h_\theta(x) = g(\theta^T x)$



Predict "1" if $\theta^T x \geq 0$ "$h_\theta(x) = g(\theta^T x) \geq 0.5$"

"0" otherwise

i.e. "much greater than"

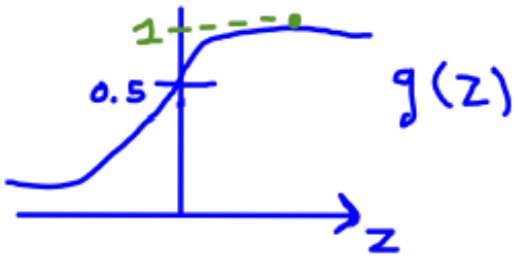
If $y^{(i)} = 1$, we want/hope that $\theta^T x^{(i)} \gg 0$

#!!#**# ^ADJUSTED GRAPH PROPERLY^ #**#!!#

If indeed, $\boldsymbol{\Theta^T x \gg 0}$, then $\boldsymbol{g(\Theta^T x)}$ will be very close to **1** which means that it's giving a very good (very accurate) (very correct and confident) prediction that equals 1

Functional margin:

$h_\theta(x) = g(\theta^T x)$



Predict "1" if $\theta^T x \geq 0$ "$h_\theta(x) = g(\theta^T x) \geq 0.5$"

"0" otherwise

i.e.

"much greater than"

If $y^{(i)} = 1$, we want/hope that $\theta^T x^{(i)} \gg 0$

If $y^{(i)} = 0$, we want/hope that $\theta^T x^{(i)} \ll 0$

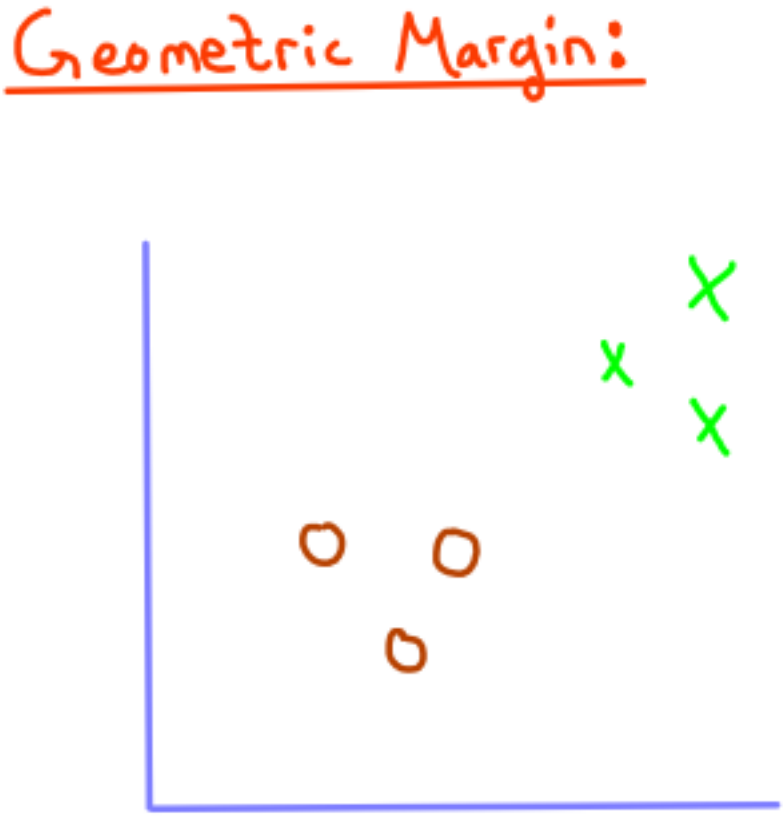
"much less than"

And if $\mathbf{y^{(i)}=0}$, then what we want/hope is that $\mathbf{\Theta^T x \ll 0}$ because it is true, then the algorithm is doing very well on that example

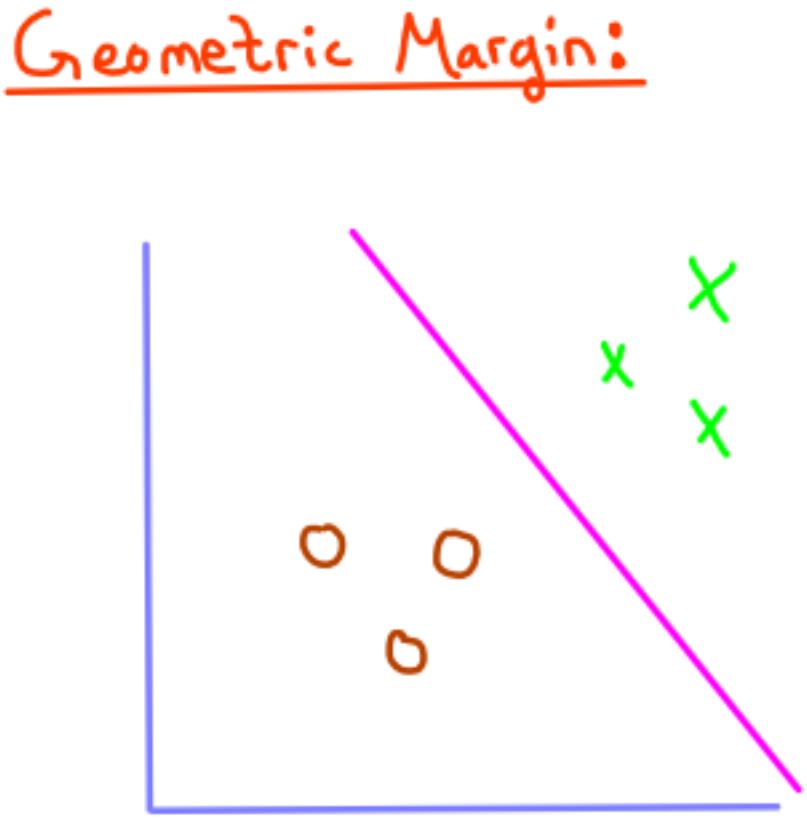
The **Functional Margin** captures this idea that if a classifier has a large ***functional margin***, it means that those two (if-statements above) statements are true

Geometric Margin:

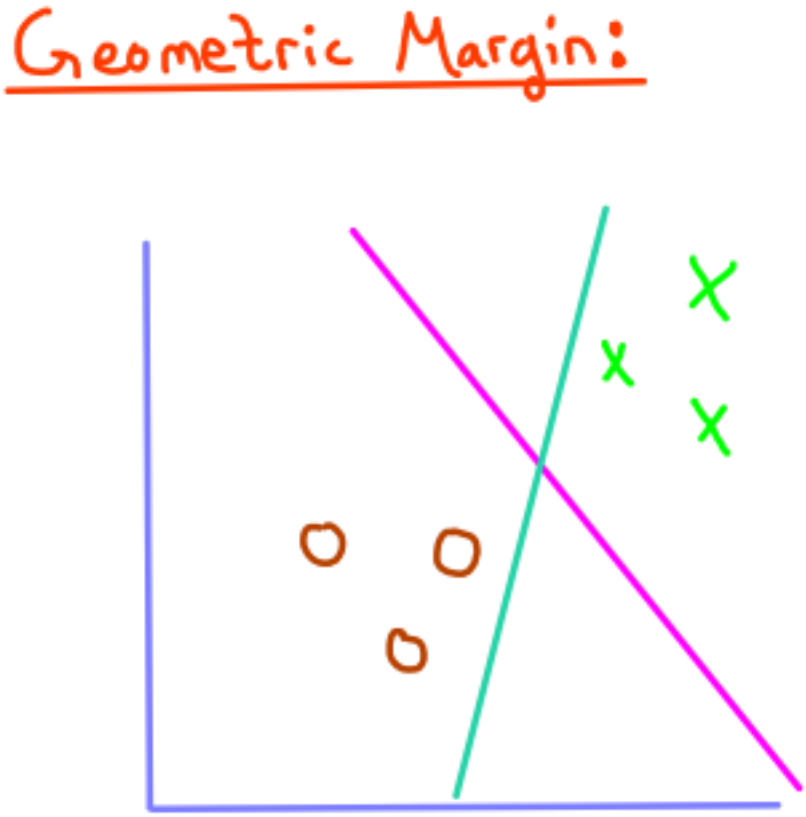
For now, assume the data is linearly separable



Assuming this is your dataset:



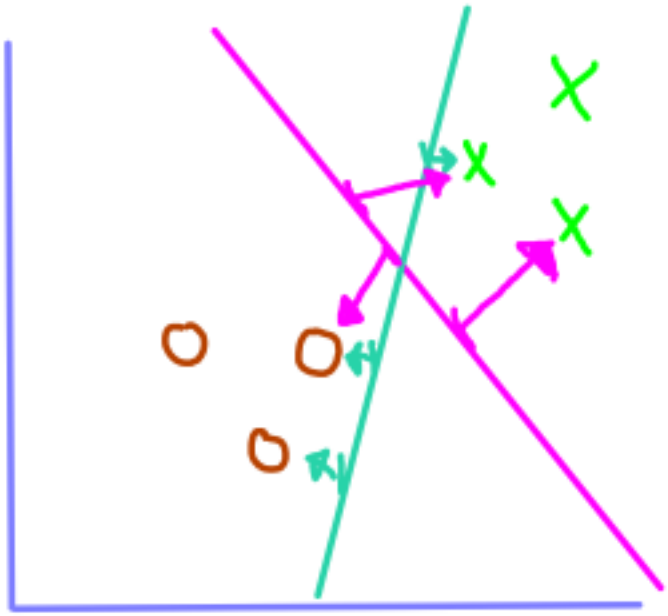
This seems like a pretty good decision boundary for separating positive and negative examples



This is another decision boundary that also separates the positive and negative examples, but somehow the first line (magenta) looks much better than the second line (teal)

Why is that? The (teal) line comes really close to a few of the training examples, whereas the (magenta) line has a much bigger separation (a much bigger distance from the positive and negative examples)

Geometric Margin:

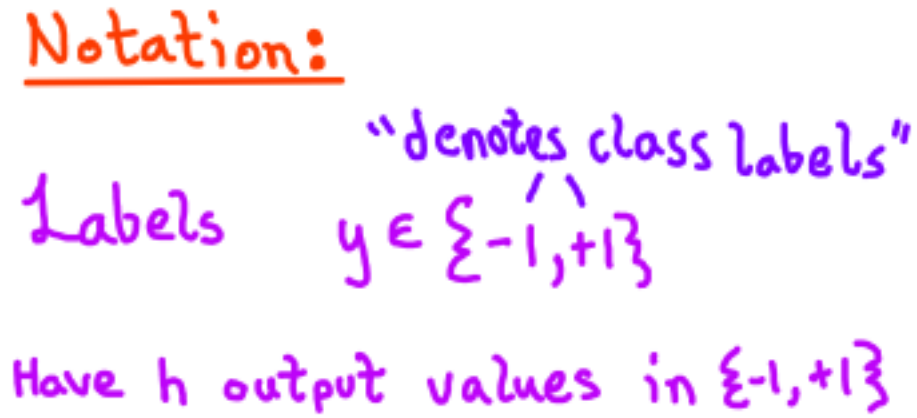


So even thought the (teal) and (magenta) line both perfectly separate the positive and negative examples, the (magenta) line has a much bigger separation, which is called the **Geometric Margin**, but there's a much bigger ***geometric margin***, meaning a physical separation from the training examples even as it separates them

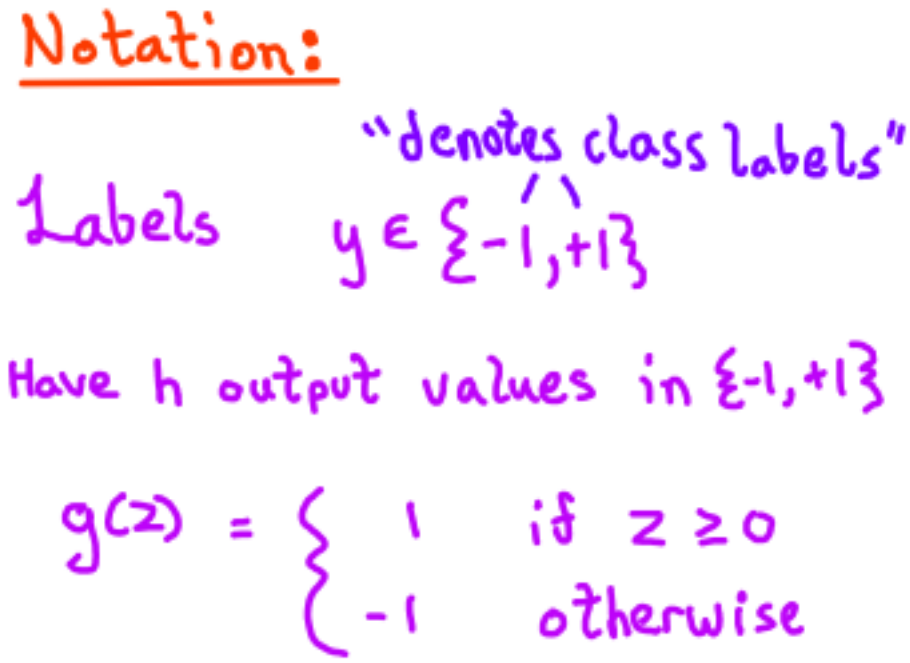
Notation to develop SVMs:

Because these algorithms have different properties, using slightly different notation to describe them makes some of the math a little bit easier

When developing **SVM**s, we're going to use **-1** and **+1** to denote the class labels

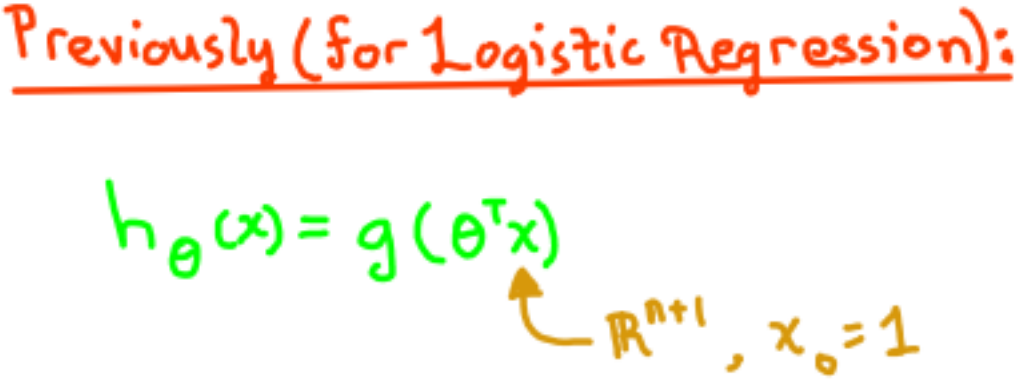


Rather than having a hypothesis output a probability, like what was seen in ***Logistic Regression***, the **Support Vector Machine** will output either a **-1** or **+1**

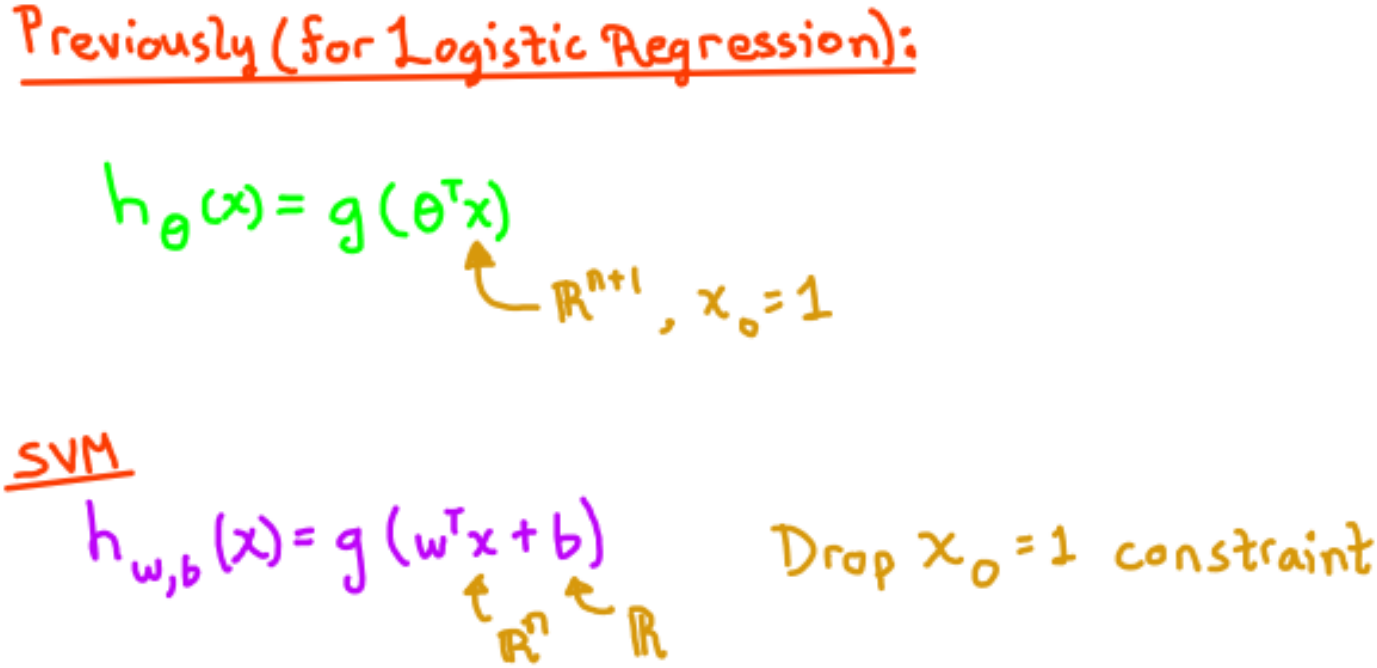


So instead of a *smooth transition* from 0 to 1, we have a hard transition (abrupt transition) from **-1** to **+1**

Where previously we had Logistic Regression:



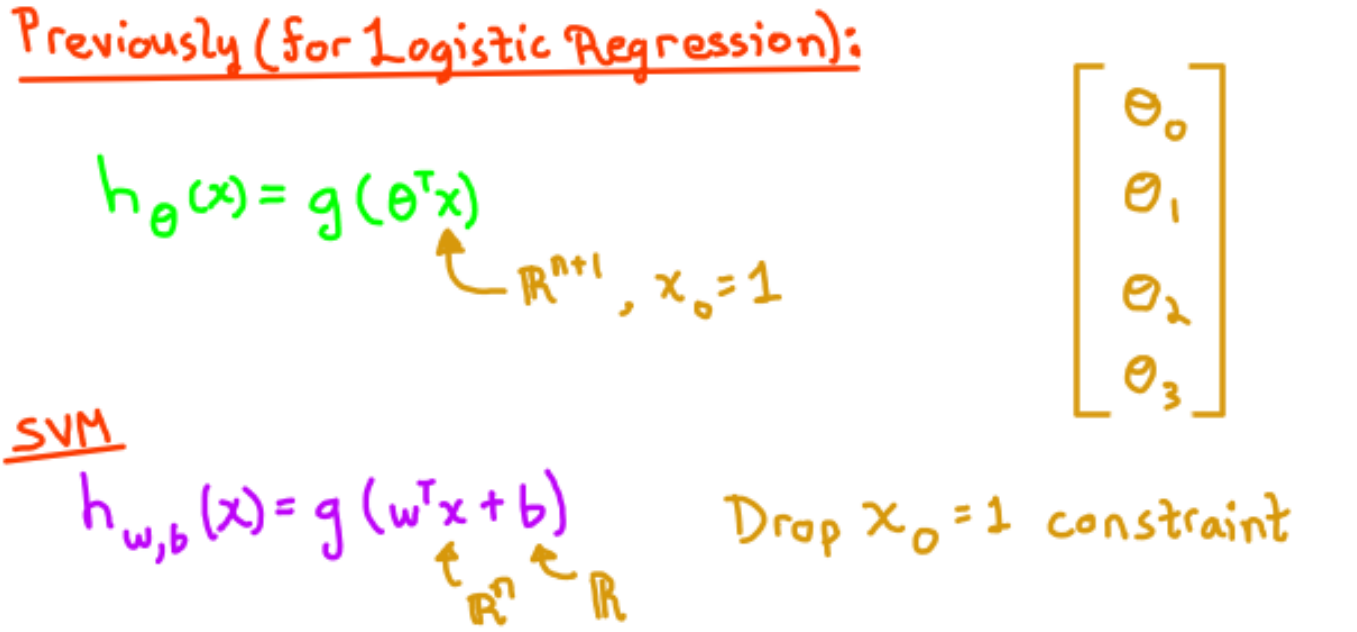
For the SVM, we will have:



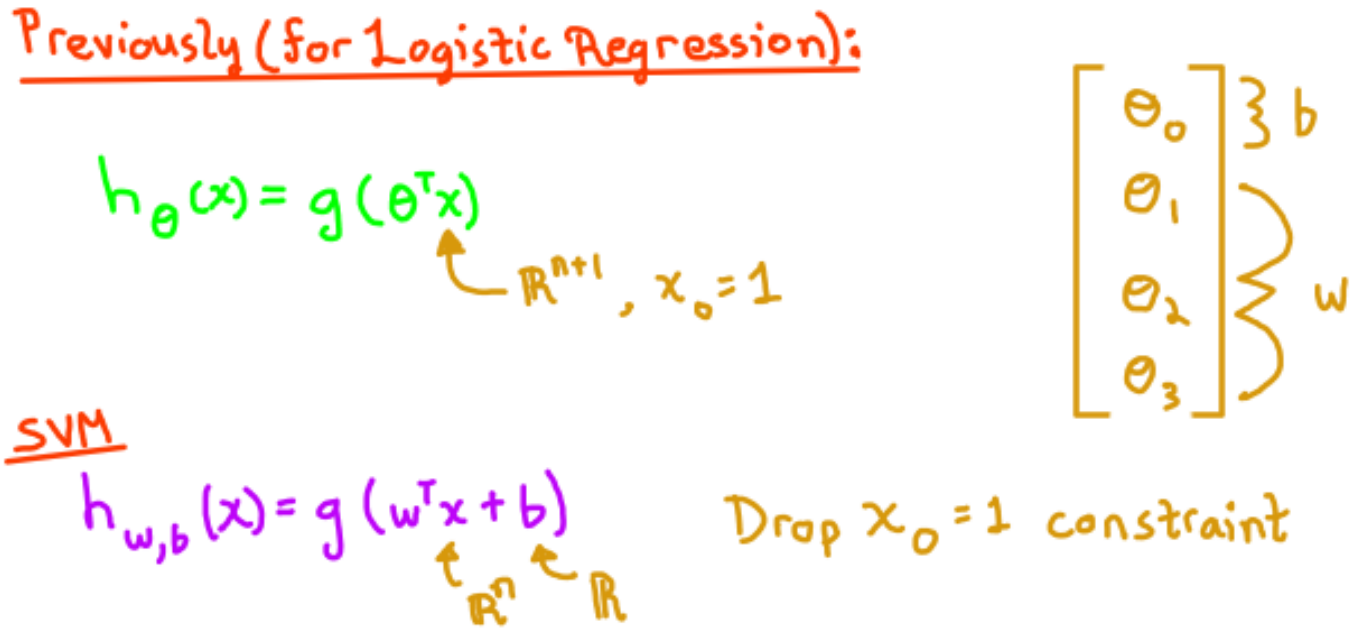
So for the **SVM**, the parameters of the ***SVM*** will be the parameters **w** and **b**, and the hypothesis applied to **x** will be $\mathbf{g(w^T x + b)}$ and we're dropping the $\mathbf{x_0 = 1}$ constraint. So separate out **w** and **b** as follows

This is a standard notation used to develop **Support Vector Machines**

One way to think about this is, if the parameters are:

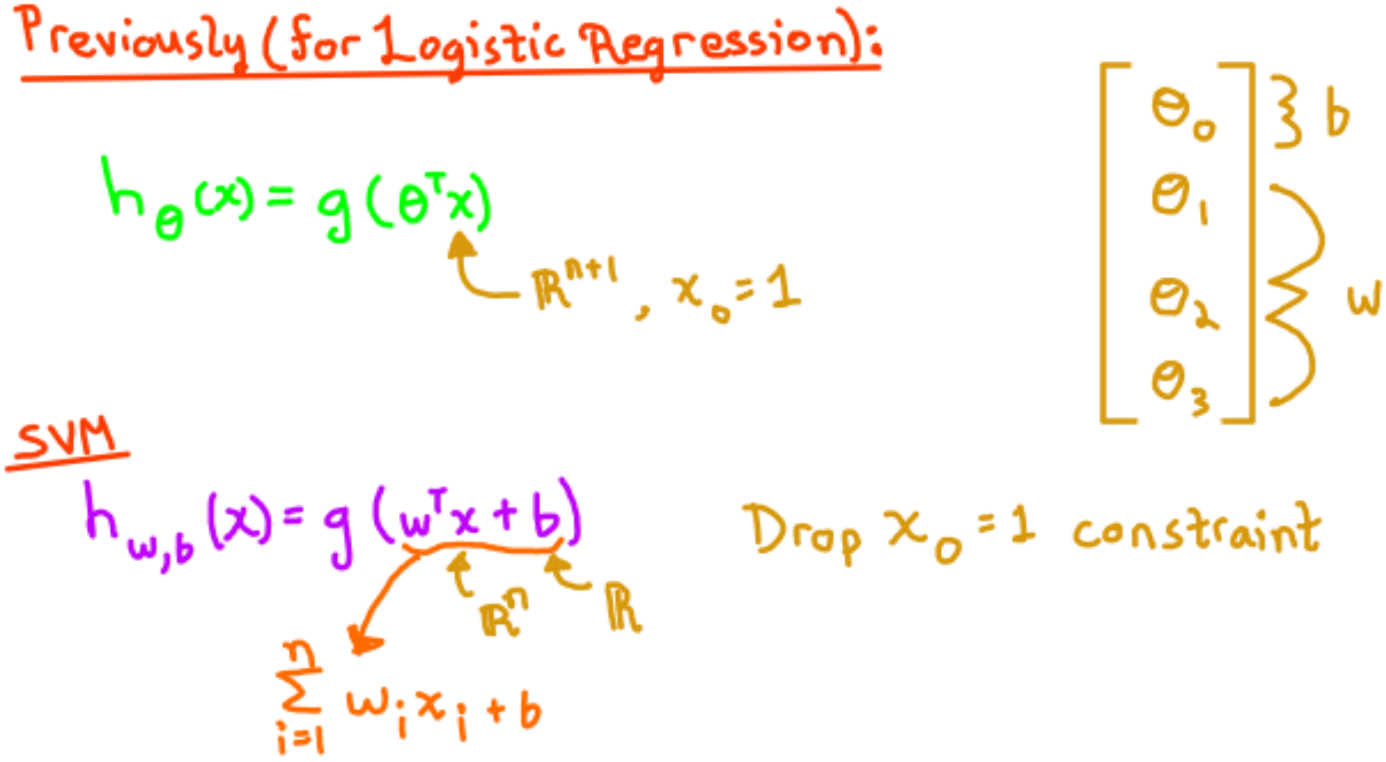


Then these will be the new **b** and the new **w**:



So you just separate out the Θ_0 which was previously multiplying to x_0

And this term becomes the following since we're gotten rid of x_0:



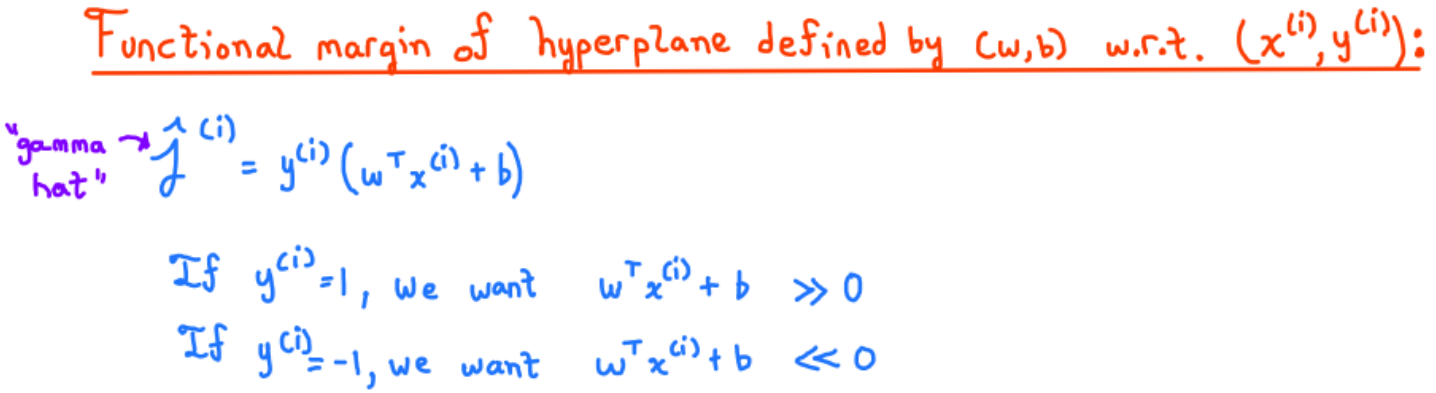
Formalized Definition of a Functional Margin:

The parameters **w** and **b** are defined as ***linear classifier***. So with the formulas above, the parameters **w** and **b** defines a **line** (really it defines a ***hyperplane***), or in high dimensions it'd be a ***plane*** or a ***hyperplane*** that defines a **straight line**, separating out the positive and negative examples

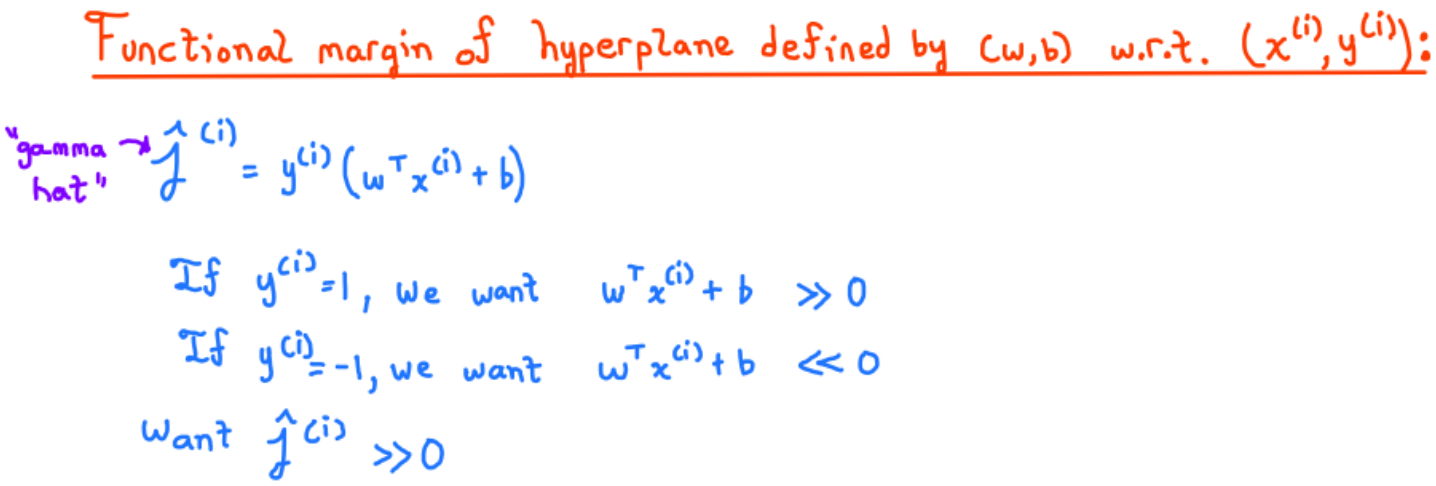
So the functional margin of a hyperplane, defined by this with respect to one training example

Hyperplane just means straight line in high-dimension. This is a **Linear Classifier** so it's just the **Functional Margin** of this classifier with respect to one training example

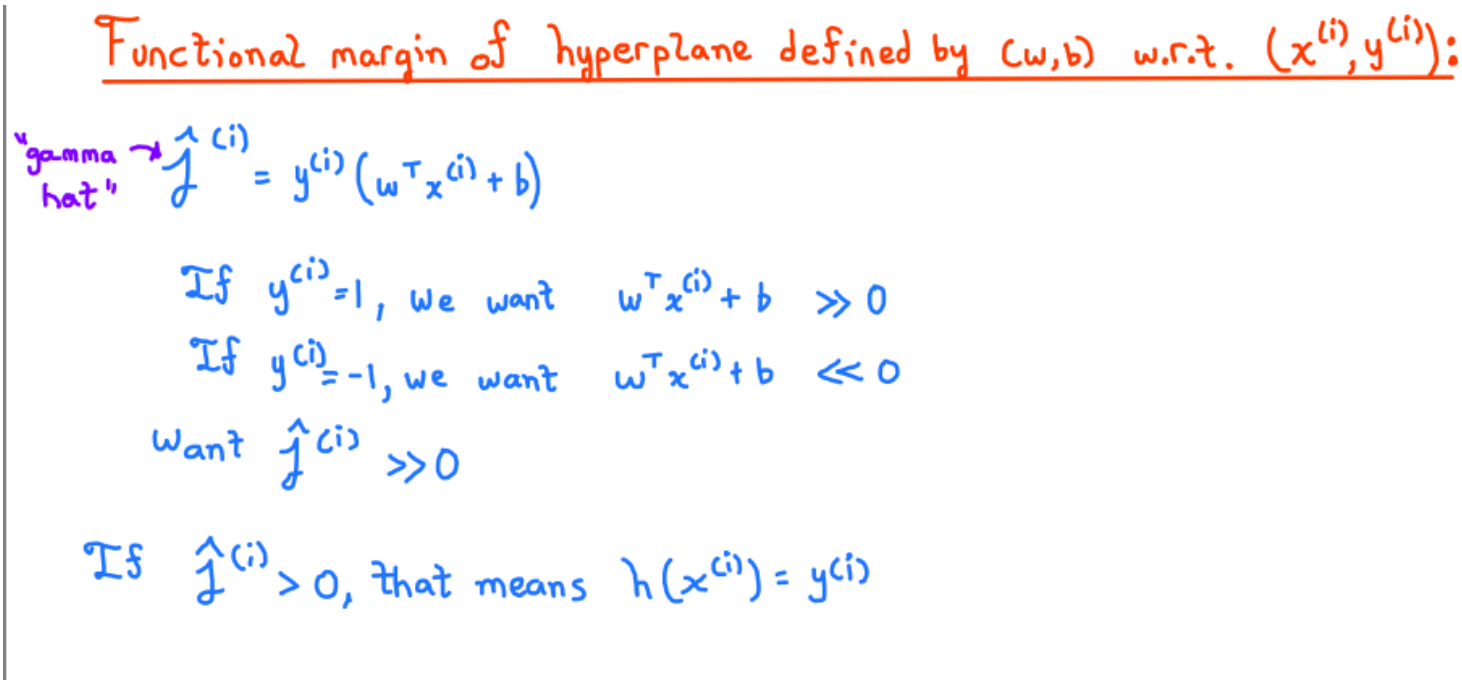
What we hope for is for our classifier to achieve a large functional margin



If you kind've combine these two statements, you're basically saying that you hope that ***Gamma-hat*$^{(i)}$** is $\gg$ **0** because $\mathbf{y^{(i)}}$ is now **{-1,+1}** and so if $\mathbf{y^{(i)}=1}$ you'd want $\mathbf{(w^T x^{(i)} + b)}$ to be very very large, and if $\mathbf{y^{(i)}=-1}$ you'd want $\mathbf{(w^T x^{(i)} + b)}$ to be a very very large negative number. Either way it's just saying that you hope Gamme-hat would be very large

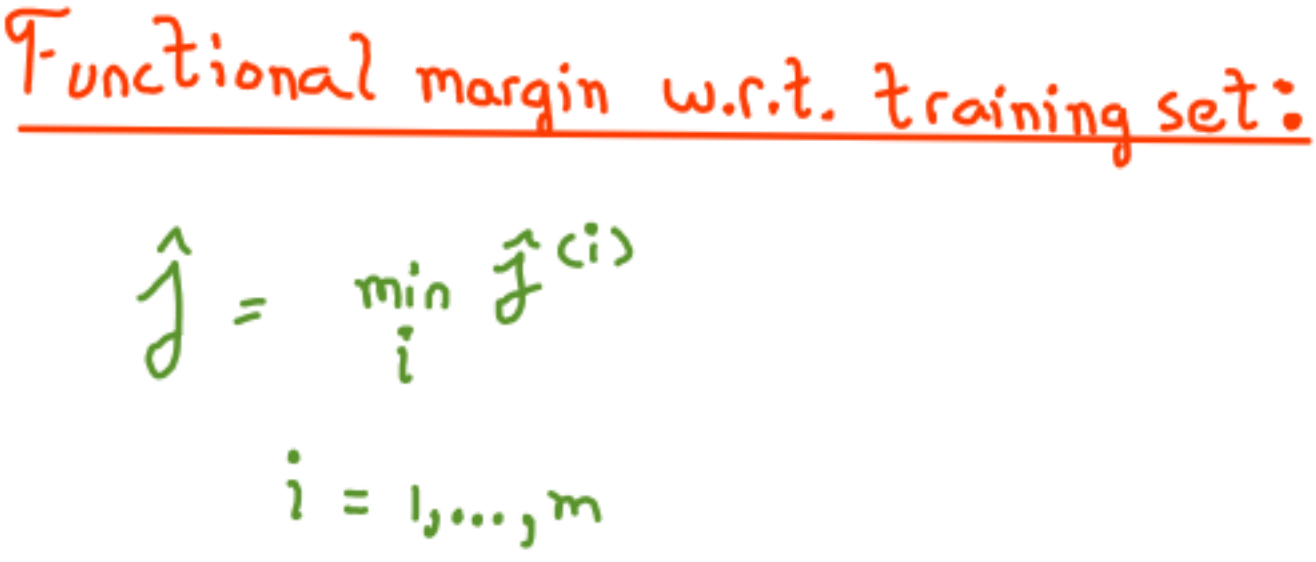


One property of this as well is that:



So as long as the **Functional Margin** (**Gamma-hat$^{(i)}$**) is greater than 0, it means that either **($w^T x^{(i)}$ + b)** is bigger than **0** or **($w^T x^{(i)}$ + b)** is less than **0**, depending on the sign of the label, and it means that the algorithm gets this one example correct at least

If it's much greater than **0** then it means, in the **Logistic Regression** case, that the prediction is at least a little bit above **0.5** or a little bit below **0.5** probability, so that at least gets it right, and if it's much greater than **0** or much less than **0** then that means the probability output in the **Logistic Regression** case is either very close to **1** or very close to **0**



i equals ranges over your training examples

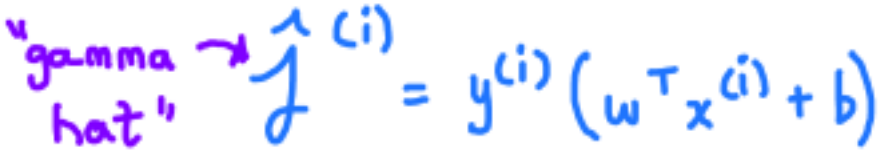
This is a worst case notion; but so this (previous definition *w.r.t.* a single training example) definition of a **function margin** is how well

you are doing on that one training example. Now we're defining the **function margin** *w.r.t.* the entire training set as how well you are doing on the worst example in your training set

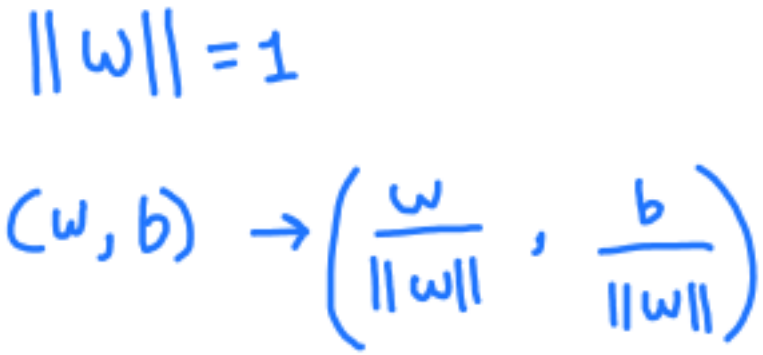
For now, because we're assuming that the training set is linearly separable, we're using this kind of worst-case notion and defining the **functional margin** to be the **functional margin** of the worst training example

One thing about the definition of the **functional margin** is it's actually really easy to cheat and increase the **functional margin**, and one thing you can do in regards to this formula (below) is if you take **w** and multiply it by **2** and take **b** and multiply it by **2**, then everything here **($w^T x^{(i)}$ + b)** just multiplies by **2** and you've doubled the **functional margin**, but you haven't actually changed anything meaningful

So one way to cheat on the **functional margin** is just by scaling the parameters by 2, 10, etc., but this doesn't actually change the decision boundary (it doesn't actually change any classification) just to multiply all of your parameters by a factor of 2, 10, etc.



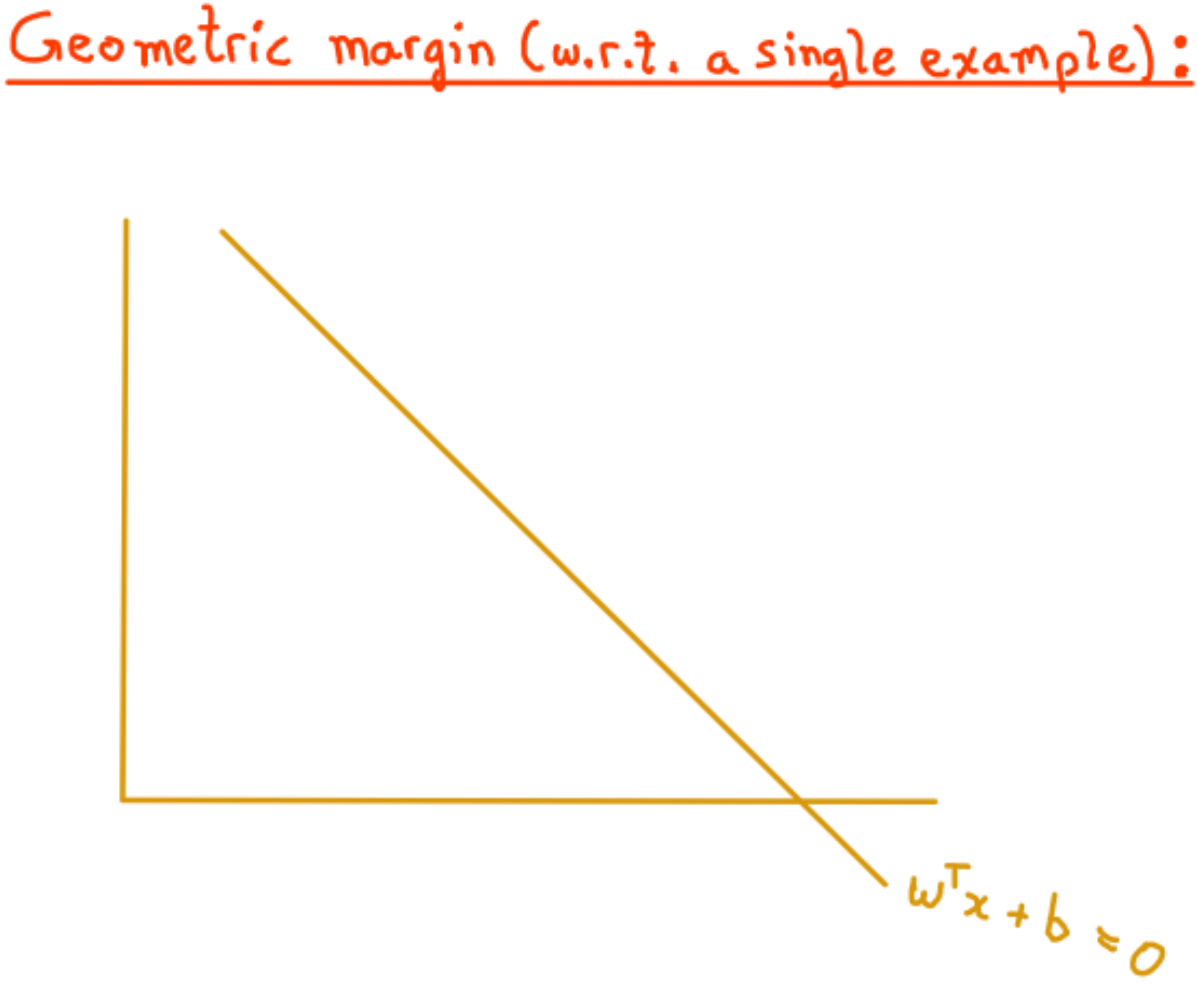
One thing you could do would be to normalize the length of your parameters. So for example, hypothetically you could impose a constraint, that norm ||**w**|| = 1 . Another way to do that would be to take **w** and **b** and replace it with just the value of parameters through by the magnitude (by the **Euclidean length** of the parameter vector **w**). This doesn't change any classification, it's just rescaling the parameters but that it prevents the previously discussed way of cheating on the **functional margin**



In fact, more generally you could actually scale **w** and **b** by any other values you want and it doesn't matter as the classification will stay the same

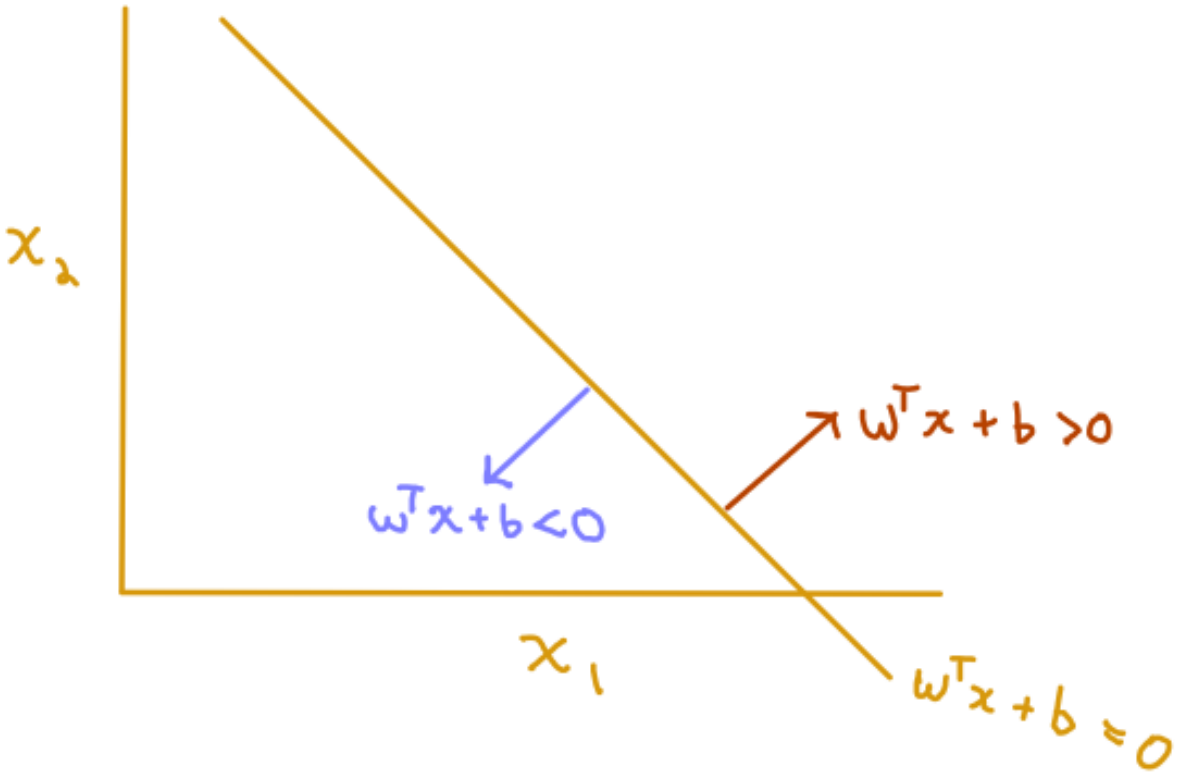
Formalized Definition of a Geometric Margin:

Let's say you have a classifier and given parameters **w** and **b** that defines a ***linear classifier*** and the equation **$w^T x+b=0$** defines the equation of a straight line

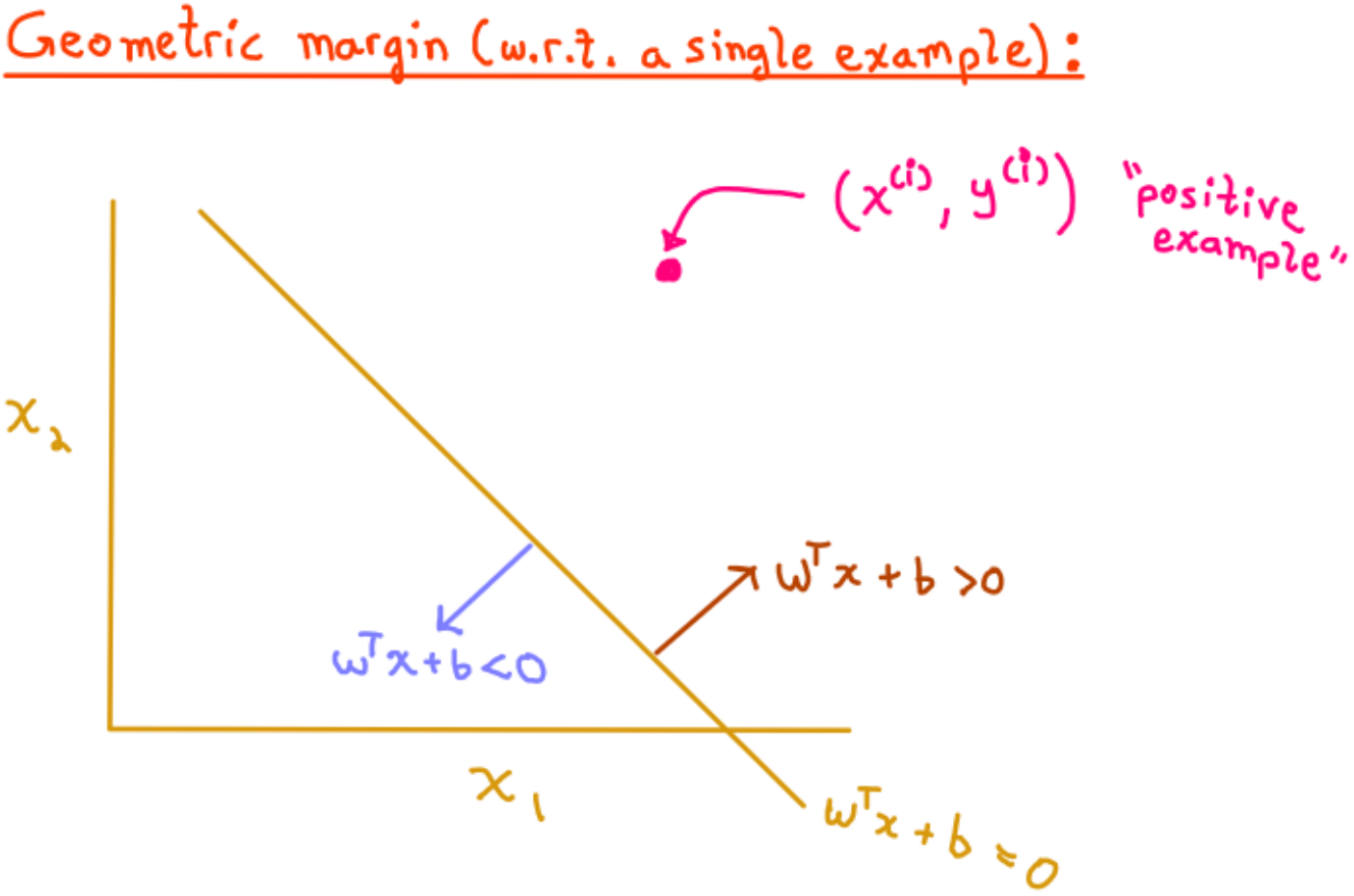


So given parameters **w** and **b**, the *upper-right* is where your classifier will predict **y=1** and the *lower-left* is where it'll predict **y=-1**

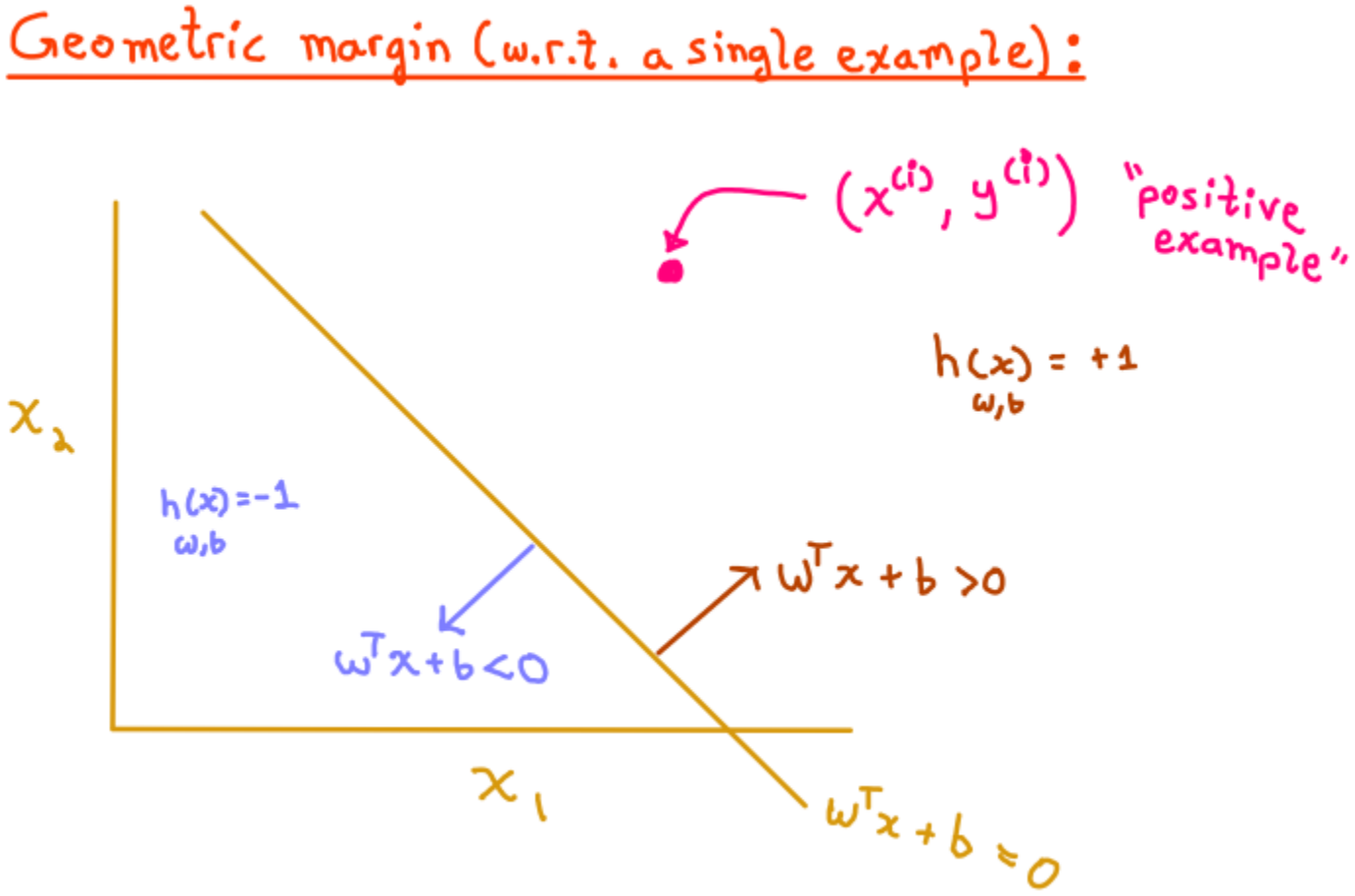
Geometric margin (w.r.t. a single example):



Say you have a training example here:



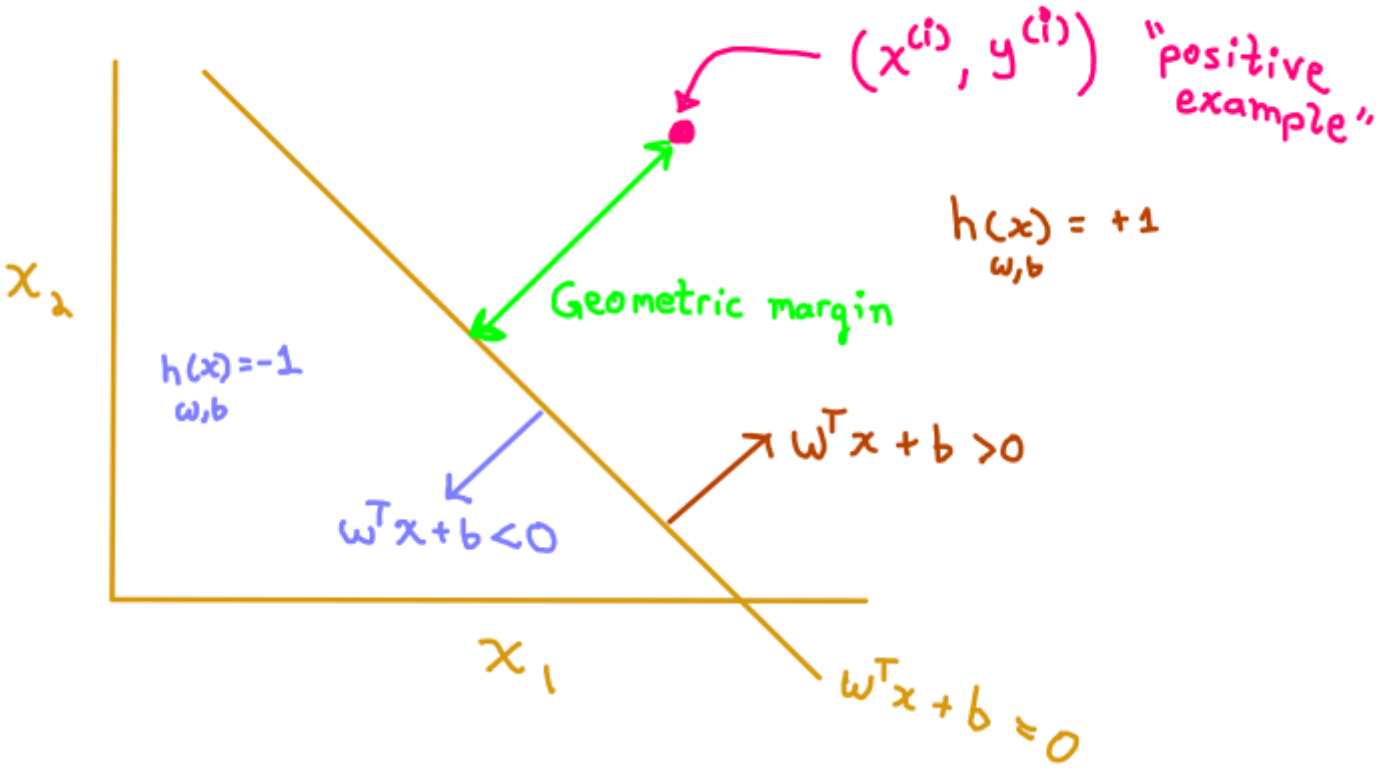
And so your classifier is classifying this example correctly because in the *upper-right* half plane $\mathbf{w^T x+b>0}$ and so in this *upper-right* region your classifier is predicting **+1**, whereas in the *lower-left* region it would be predicting **h(x)=-1**



And that's why this straight line, where it switches from predicting negative to positive is the decision boundary

What we're going to do is define this distance (the Euclidean distance) to be the **geometric margin** of this training example:

Geometric margin (w.r.t. a single example):



Geometric margin of hyperplane (w,b) w.r.t. $(x^{(i)}, y^{(i)})$:

"$\|w\|$ is the norm of w"

$$\gamma^{(i)} = \frac{w^T x^{(i)} + b}{\|w\|}$$

This is for the positive example

More generally, we're going to define the geometric margin to be equal to this and this definition applies to both the positive examples and the negative examples:

Geometric margin of hyperplane (w,b) w.r.t. $(x^{(i)}, y^{(i)})$:

"$\|w\|$ is the norm of w"

$$\gamma^{(i)} = \frac{y^{(i)}(w^T x^{(i)} + b)}{\|w\|}$$

The relationship between the ***geometric margin*** and the ***functional margin*** is that the ***geometric margin*** is equal to the ***functional margin*** divided by the norm of **w**

Geometric margin of hyperplane (w,b) w.r.t. $(x^{(i)}, y^{(i)})$:

"$\|w\|$ is the norm of w"

$$\gamma^{(i)} = \frac{y^{(i)}(w^T x^{(i)} + b)}{\|w\|}$$

$$\gamma^{(i)} = \frac{\hat{\gamma}^{(i)}}{\|w\|}$$

Geometric margin w.r.t. training set:

$$\gamma = \min_i \gamma^{(i)}$$

Where again, it uses ***worst-case notion*** (look through all your training examples and pick the worst possible training example and that is your ***geometric margin*** on the training set)

Reminder: $\hat{\gamma}$ = functional margin

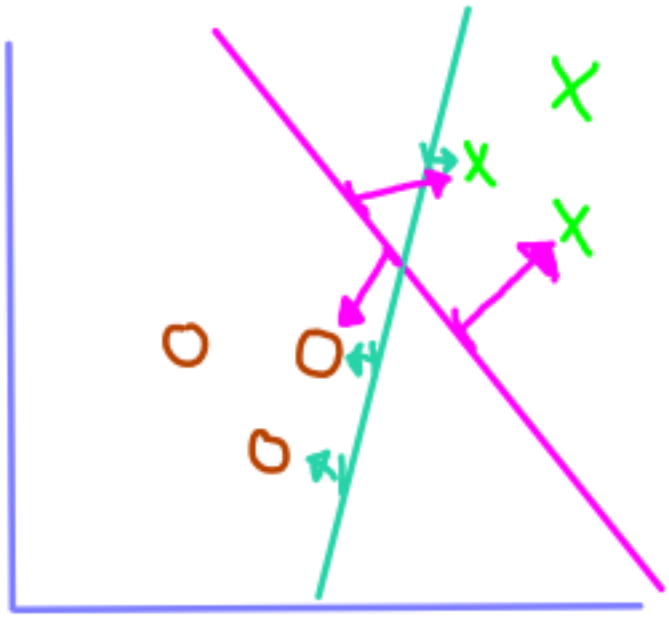
γ = geometric margin

Optimal Margin Classifier:

The **Optimal Margin Classifier** is basically an algorithm that tries to maximize the **geometric margin**

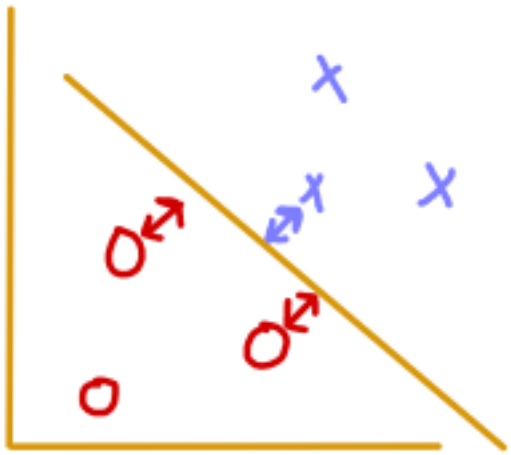
What the rudimentary **SVM (Support Vector Machine)** does (what the ***SVM*** and low-dimensional spaces will do), also called the **Optimal Margin Classifier**, is pose an optimization problem to try to find the (magenta) line to classify these examples

Geometric Margin:



What the **Optimal Margin Classifier** does is choose the parameters **w** and **b** to maximize the geometric margin

In other words, the ***optimal margin classifier*** is the baby **SVM**. It's like a ***SVM*** for linearly separable data (at least for now)



And so the **optimal margin classifier** will choose that straight line because that straight line maximizes the distance or maximizes the **geometric margin** to all of these examples

How you pose this mathematically, there are a few steps of this derivation (leaving out the middle steps)

It turns out one way to pose this problem is to maximize **gamma**, **w**, and **b** of **gamma**. So you want to maximize the **geometric margin** subject to that every training example must have **geometric margin** greater than or equal to **gamma**

Optimal margin classifier:

Choose w, b to maximize γ ("geometric margin")

$$\max_{\gamma, w, b} \gamma$$

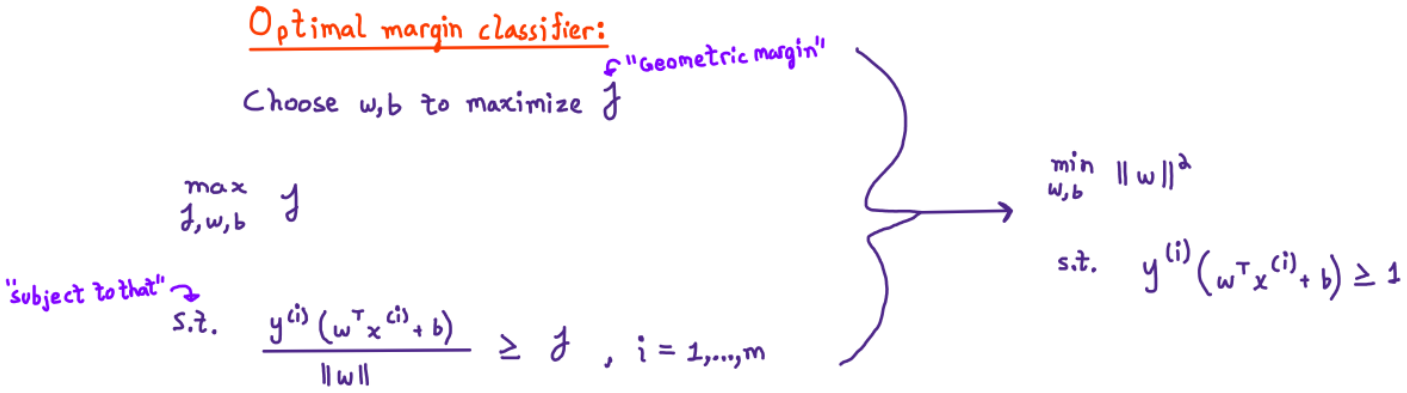
"subject to that"

$$\text{s.t.} \quad \frac{y^{(i)}(w^T x^{(i)} + b)}{\|w\|} \geq \gamma \quad , \quad i = 1, \ldots, m$$

So you want **gamma** to be as big as possible, subject to that every single training example must have at least that ***geometric margin***.

This causes you to maximize the worst-case ***geometric margin***

It turns out this is, not in this form, this is in a convex optimization problem, so it's difficult to solve this without a (without running?) gradient descent and initially known local optima and so on, but it turns out that via a few steps of rewriting, you can reformulate this problem into the equivalent problem which is a minimizing norm of **w**, subject to the **geometric margin**



This problem (maximizing **gamma**, **w**, **b**) is just solving for **w** and **b** to make sure that every example has a **geometric margin** greater or equal to **gamma**, and you want **gamma** to be as big as possible. So this is the way to formulate an optimization problem that says "Maximize the ***geometric margin***"

One piece of intuition to take away is "the smaller **w** is, the less of a normalization division effect you have"

This (minimize **w**, **b**) turns out to be a convex optimization problem and if you optimize this, then you will have the **optimal margin classifier**. They're very good numerical optimization packages to solve this optimization problem, and if you give this a dataset then, assuming your data's separable, then you have the optimal margin classifier (which is really a baby **SVM**). When we add **Kernels** to it, then you have the full complexity of the **SVM** norm