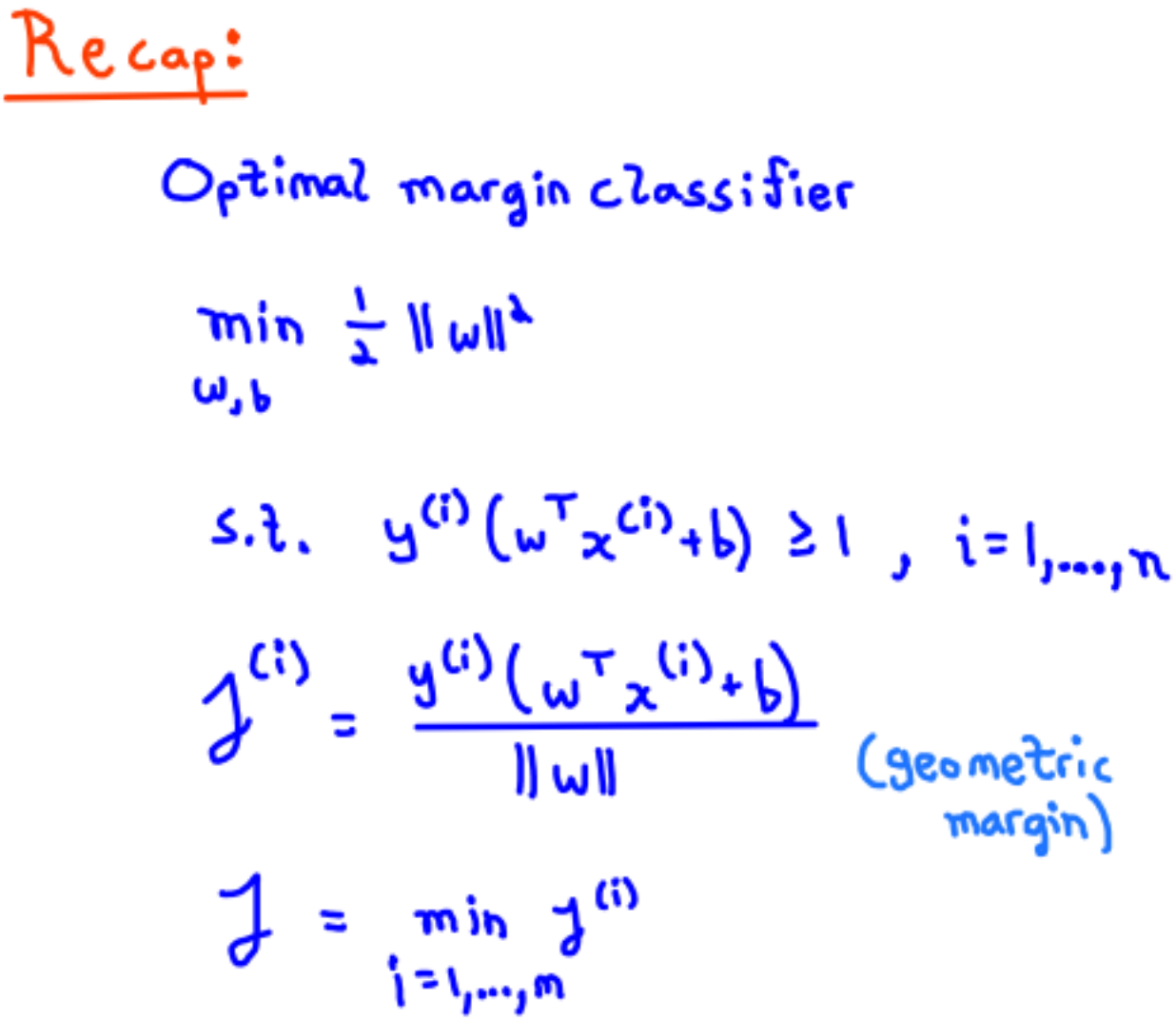
Lecture 7 [Kernels]

https://www.youtube.com/watch?v=8NYoQiRANpg&list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU&index=7

SVM:

Recap:



For the **Optimal Margin Classifier**, we want to find the decision boundary with the greatest possible ***geometric margin***

Gamma$^{(i)}$ is the formula for computing the distance from the example $\mathbf{x}^{(i)}$,$\mathbf{y}^{(i)}$ to the decision boundary governed by the parameters **w** and **b**

Gamma is the worst case **geometric margin**. Of all of your **m** training examples, which one has the least (has the worst possible) ***geometric margin***?

And we try to make the ***Optimal margin classifier*** as big as possible

The ***optimal margin classifier*** is basically this algorithm

The ***optimal margin classifier plus kernels*** (take this idea and apply it in a 100 billion dimensional feature space) is a **SVM**

Given a training set, you want to find the decision boundary, parameterized by **w** and **b**, that maximizes the ***geometric margin***. Your classifier will output (**y = g(...)**):

Recap:

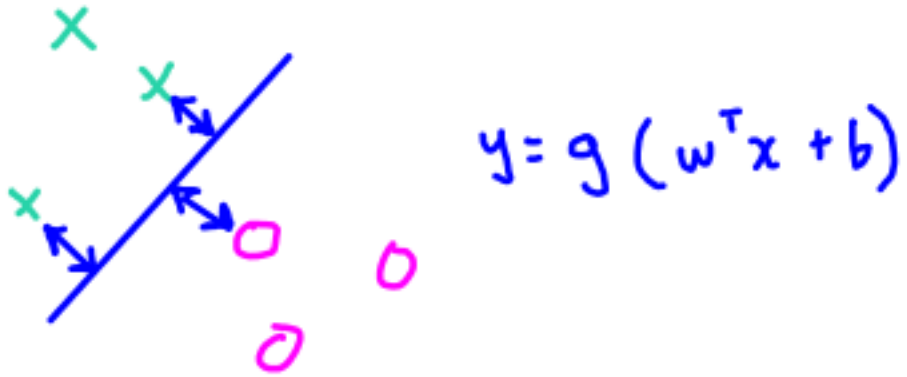
Optimal margin classifier

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

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

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

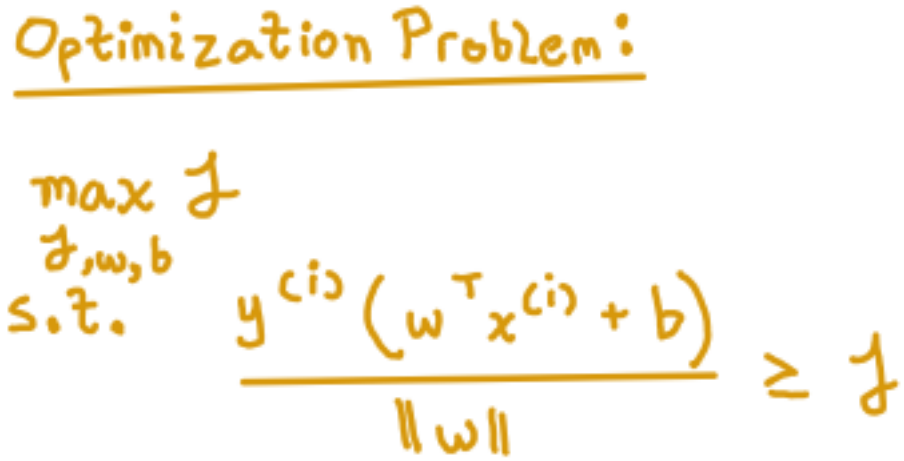
$$\gamma = \min_{i=1,\ldots,m} \gamma^{(i)}$$



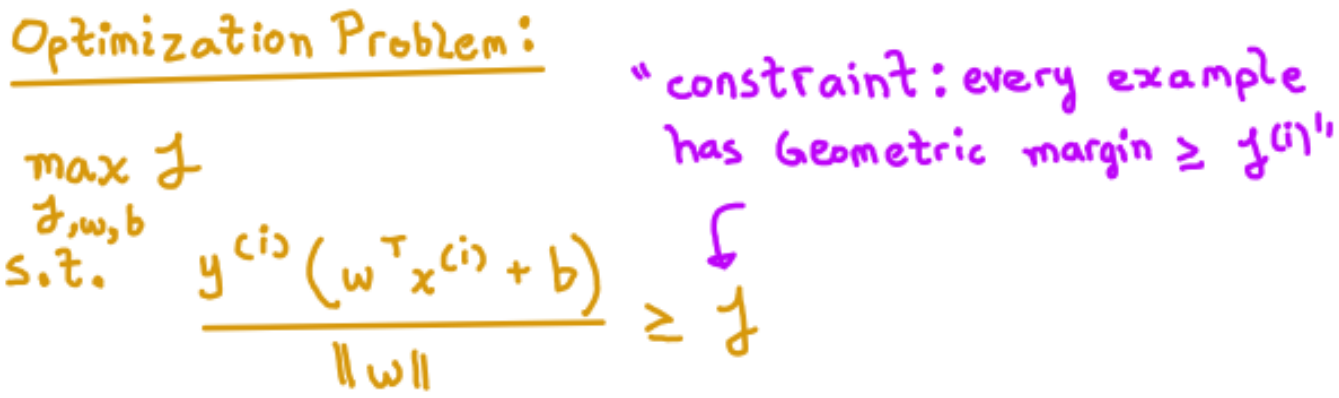
So you want to find parameters **w** and **b**. They'll define the decision boundary, where your classifications switch from positive to negative, that maximizes the ***geometric margin***

Optimization problem:

One way to pose this as an optimization problem is to try to find the biggest possible value of **Gamma**, subject to that, the ***geometric margin*** must be greater than or equal to **Gamma**

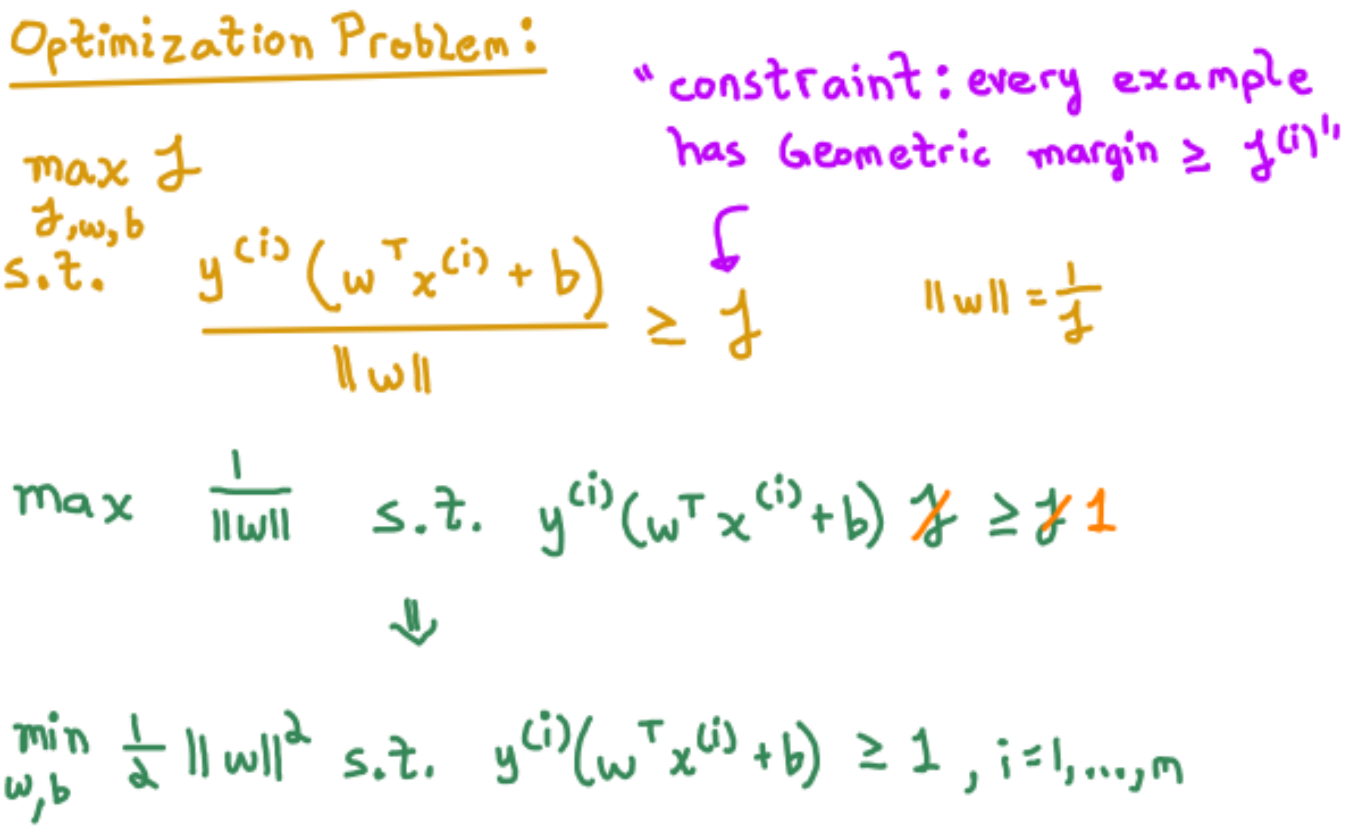


In this optimization problem, the parameters you get to fiddle with are **Gamma**, **w**, and **b**. And if you solve this optimization problem, then you are finding the values of **w** and **b** that defines a straight line (decision boundary)



This constraint is says that every example has ***geometric margin*** greater or equal to **Gamma**, and you want to set **Gamma** to be as big as possible, which means that you're maximizing the worst-case ***geometric margin***

The trick to simplifying the optimization problem equation above (notice that you could choose anything you want for the norm of **w** by scaling by a constant factor without changing the decision boundary) into $\mathbf{y = g(w^T x+b)}$ is, if you choose to scale the $\mathbf{||w||}$ to be equal to $\mathbf{1/Gamma}$, because if you do that, then this optimization objective becomes:



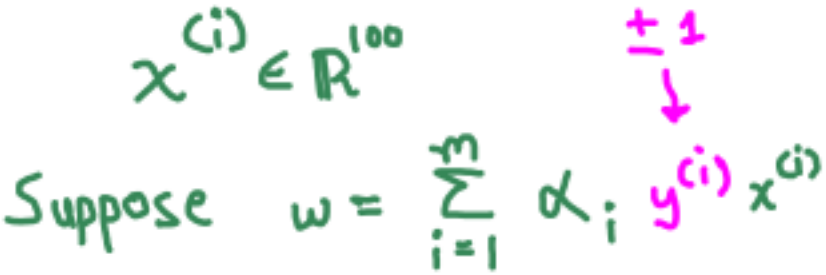
So it substitutes **||w||** to be equal to **1/Gamma**, which cancels out the **Gammas**. So you end up with this optimization problem. Instead of maximizing **1/||w||**, you minimize **(1/2)||w||**^{2}

If you solve this optimization problem, and you're minimizing over **w** and **b**, you are solving for the parameters **w** and **b** that give you the ***optimal margin classifier***

Representer theorem:

This will be a key idea in how we'll work in potentially high-dimensional feature spaces and it will teach you how to represent feature vectors and how to represent parameters that may be 100 billion or even infinite dimensional

We've been deriving this algorithm as if the features **x**$^{(i)}$ are some reasonable dimensional feature. What we will assume is that **w** could be represented as a sum as a linear combination of the training examples:



This is actually not an assumption. The **representer theorem** proves that this is just true at the optimal value of **w**

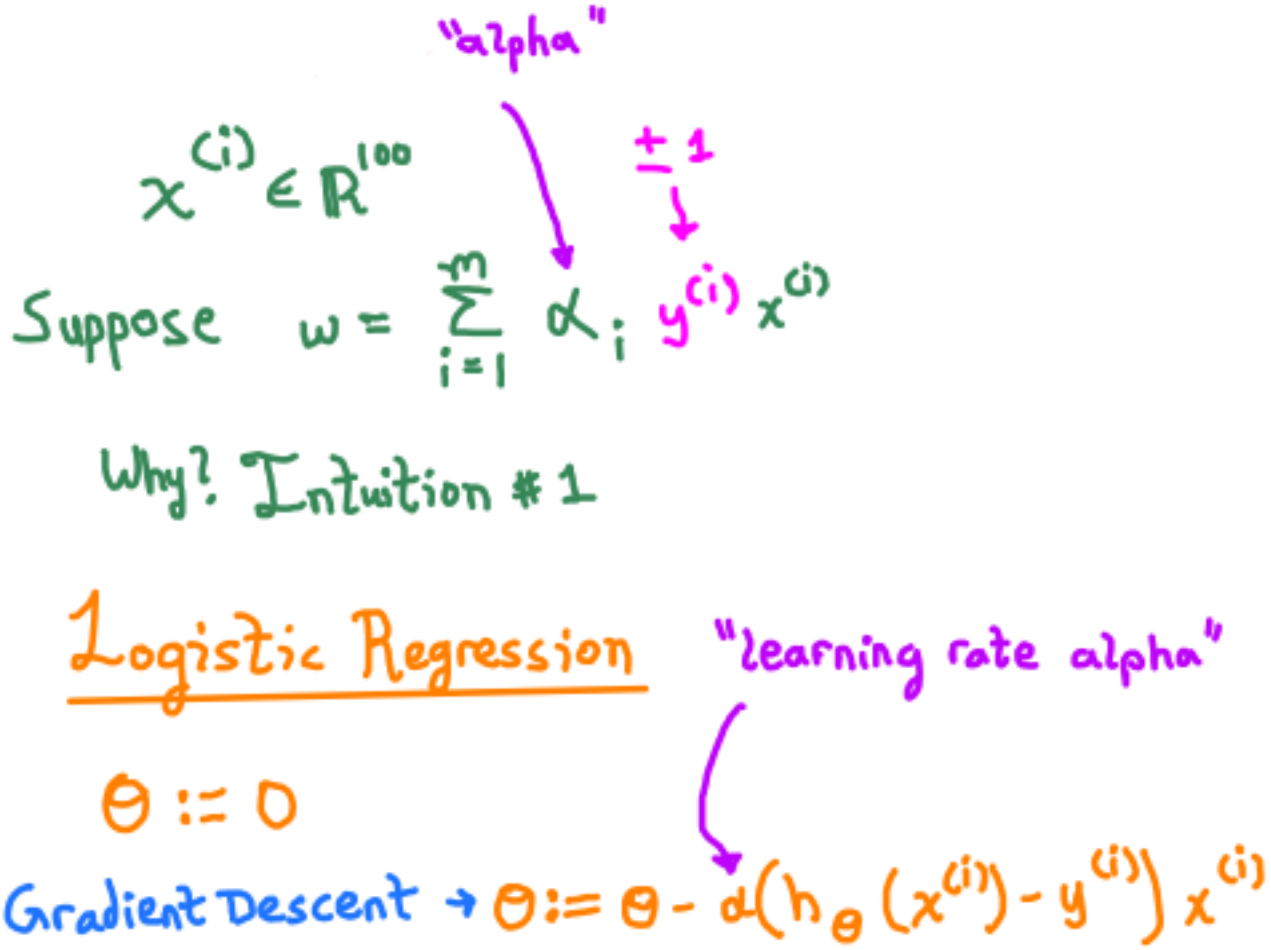
To convey in a couple ways as to why this is a reasonable thing to assume:

Intuition #1:

We're going to refer to ***Logistic Regression***

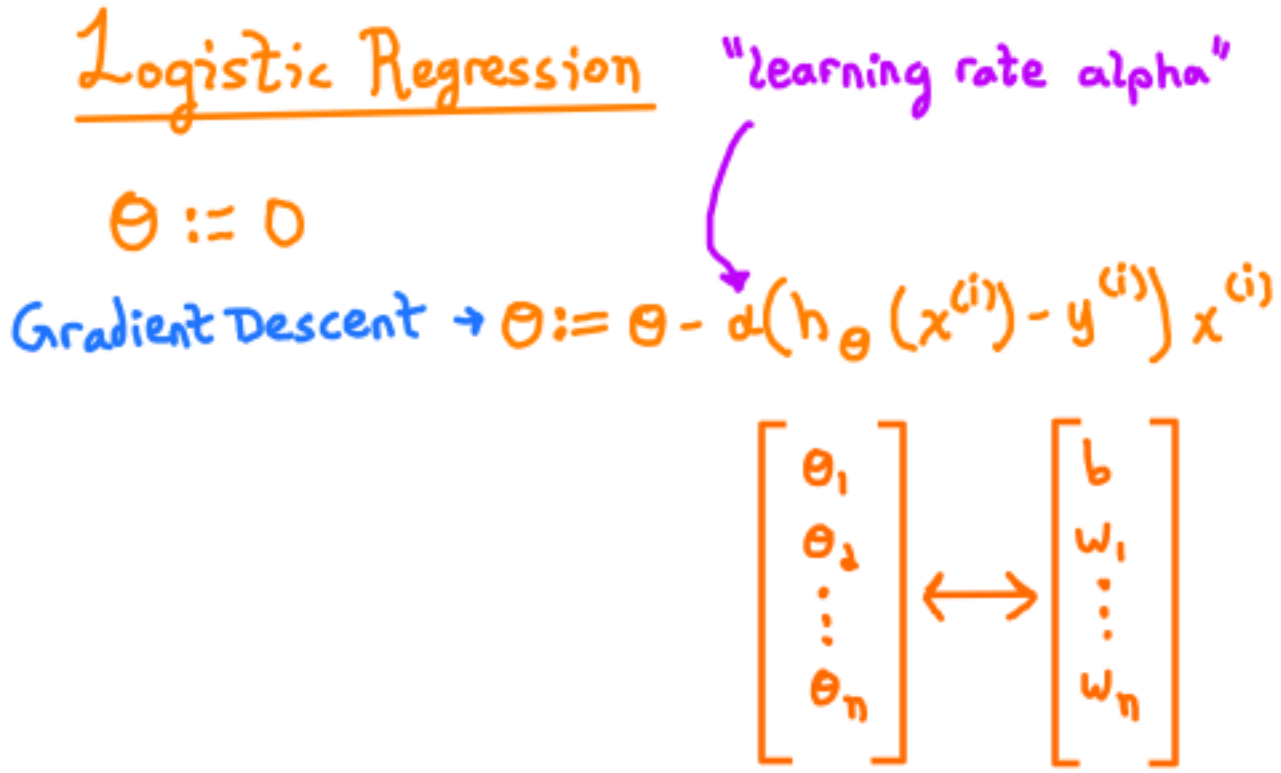
Suppose that you run ***Logistic Regression*** with **Gradient Descent** (say ***Stochastic Gradient Descent***), then you initialize the parameters to be equal to **0** at first and then for each iteration of ***stochastic gradient descent***, **Θ** gets updated as **Θ - α (...)**

#*# The alphas have nothing to do with each other, this is just overloader notation #*#



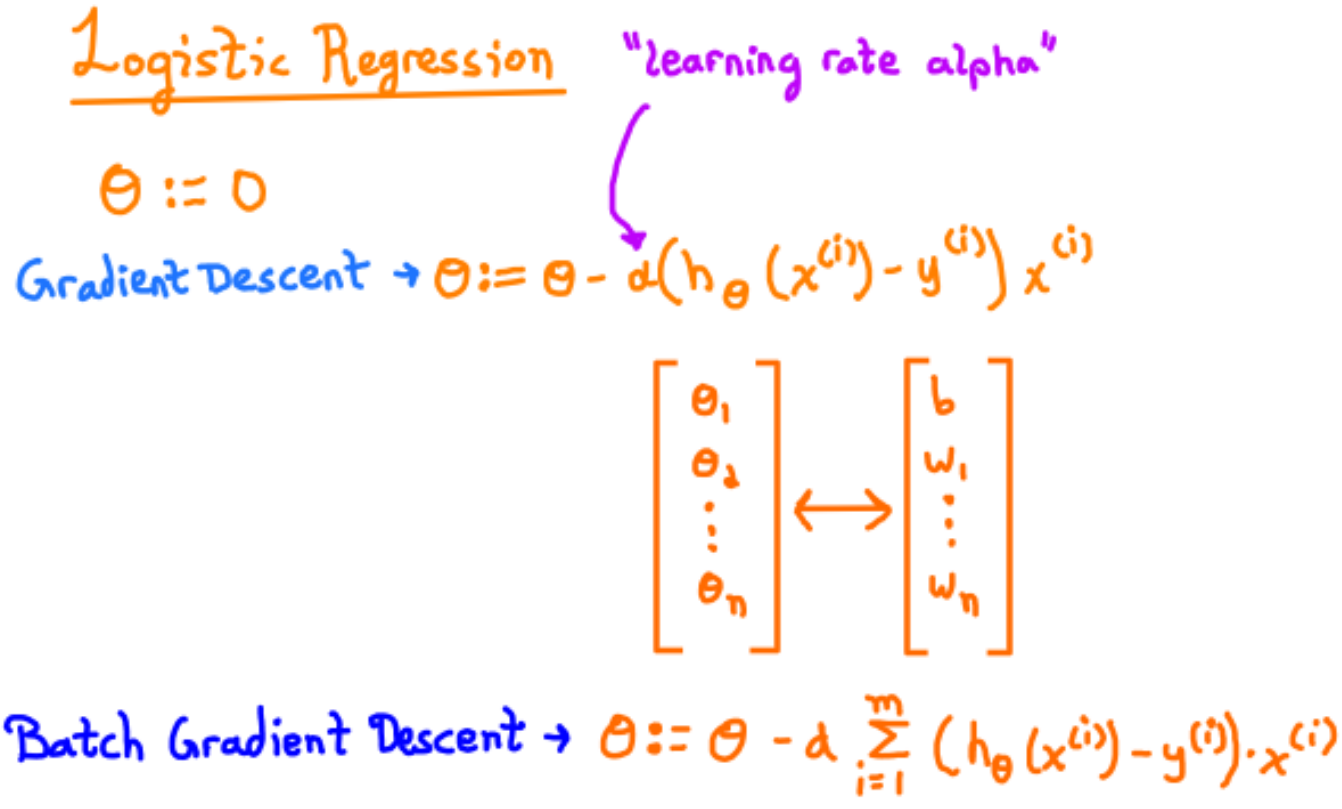
This is saying that on every iteration, you're updating the parameters **θ** by adding or subtracting some constant times some training example

Kind've *proof by induction*, if **θ** starts off **0** and if on every iteration of **gradient descent** you're adding a multiple of some training example, then no matter how many iterations you run **gradient descent**, **θ** is still a linear combination of your training examples



As you run Logistic Regression after every iteration, the parameters **θ** or the parameters **w** are always a linear combination of the training examples. This is also true if you use ***Batch Gradient Descent***

If you use ***Batch Gradient Descent***, then the update rule is:



So it turns out, you can derive ***gradient descent*** for the ***support vector machine learning algorithm*** as well. You can derive

gradient descent optimized **w** subject to this and you can have a proof by induction that, no mater how many iterations you run during descent, it will always be a linear combination of the training examples:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1, \; i = 1, \ldots, m$$

That's one intuition for how you might see that, assuming **w** is a linear combination of the training examples, is a reasonable assumption

Intuition #2:

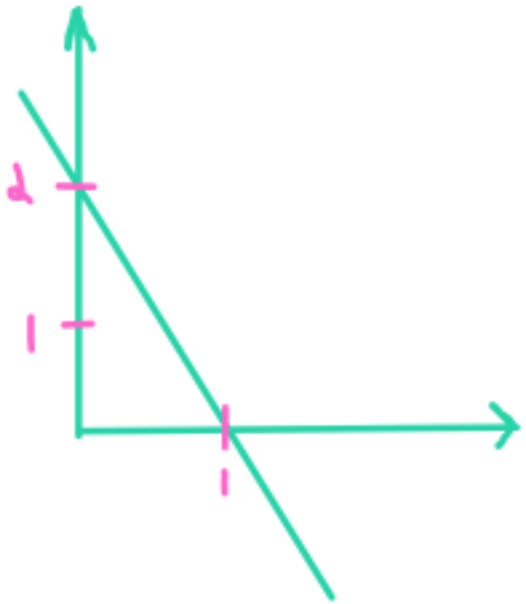
Let's say that the classifier uses this:

$$g\left(\begin{bmatrix}2\\1\end{bmatrix}x - 2\right)$$

(w = $\begin{bmatrix}2\\1\end{bmatrix}$, b = -2)

Then it turns out that the decision boundary is this:

Intuition #2

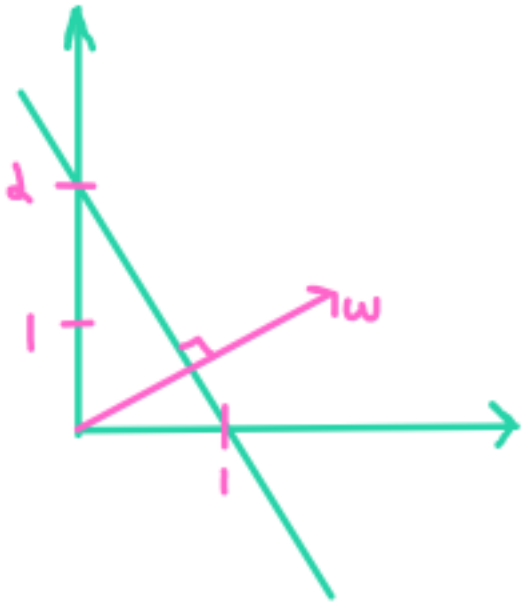


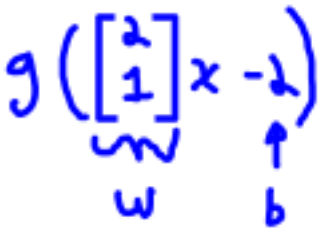
$$g\left(\begin{bmatrix}2\\1\end{bmatrix}x - 2\right)$$

(w = $\begin{bmatrix}2\\1\end{bmatrix}$, b = -2)

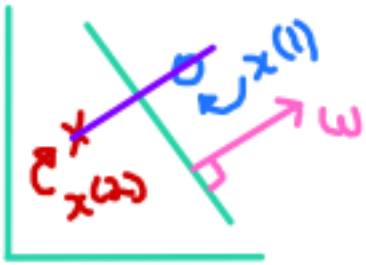
And it turns out that the vector **w** is always at 90 degrees to the decision boundary

Intuition #2





To take a simple example, let's say you have 2 training examples, a positive example and negative example, then by illustrating:

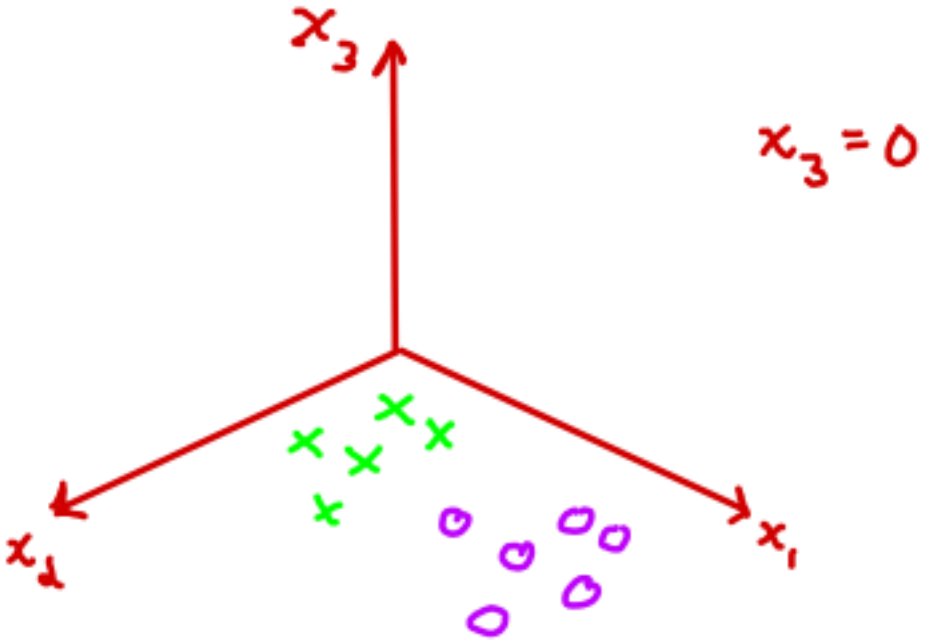


The linear algebra way of saying this is that, the vector **w** lies in the span of the training examples

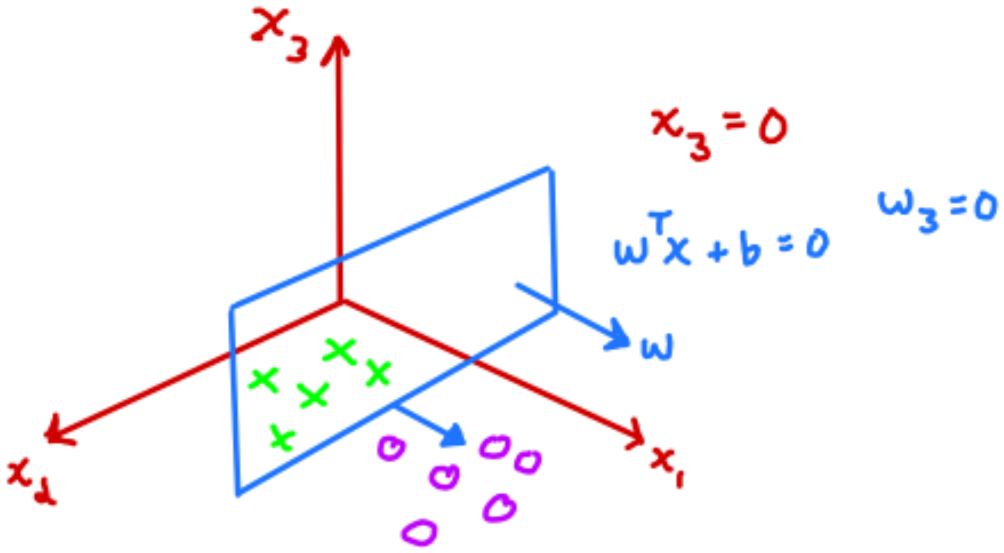
The way to picture this is that **w** sets the direction of the decision boundary, and as you vary the values of **b**, then the relative position of the decision boundary will be moved back and forth (parallel), and **w** pins the direction of the decision boundary

We're going to be working in very very high dimensional feature spaces. For this example, let's say you have 3 features, $\mathbf{x_1}$, $\mathbf{x_2}$, $\mathbf{x_3}$

Let's say, for the sake of illustration, that all of your examples lie in the plane of $\mathbf{x_1}$ and $\mathbf{x_2}$ (let's say for all of your training examples, $\mathbf{x_3 = 0}$) :



Then the decision boundary will be some sort of vertical plane that looks like this:



This is going to be the plane specifying $\mathbf{w}^T\mathbf{x}+\mathbf{b}=\mathbf{0}$, where now $\mathbf{w}$ and $\mathbf{x}$ are 3-dimensional. And so the vector $\mathbf{w}$ should have $\mathbf{w_3}=\mathbf{0}$

If one of the features is always 0 (always fixed), then $\mathbf{w_3}$ should be equal to $\mathbf{0}$. That's another way of saying that the vector $\mathbf{w}$ should

be represented as in the span of just the features $\mathbf{x}_1$, $\mathbf{x}_2$, as a span of the training examples

The formal proof of this result is called the **representer theorem**

Let's assume that **w** can be written as follows:

Optimization Problem:

$$w = \sum_{i=1}^{m} d_i y^{(i)} x^{(i)}$$

The optimization problem was this; You want to solve for **w** and **b** so that $||\mathbf{w}||^2$ is as small as possible and:

Optimization Problem:

$$w = \sum_{i=1}^{m} d_i y^{(i)} x^{(i)}$$

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

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

$||\mathbf{w}||$ is just equal to $\mathbf{w}^T\mathbf{w}$

Optimization Problem:

$$w = \sum_{i=1}^{m} d_i y^{(i)} x^{(i)}$$

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad (\|w\|^2 = w^T w)$$

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

And if you plug in this definition of **w** into these equations, you have as the **optimization objective**:

$$\min \frac{1}{2}\left(\sum_{i=1}^{m} d_i y^{(i)} x^{(i)}\right)^T \left(\sum_{j=1}^{m} d_j y^{(j)} x^{(j)}\right)$$

$$= \min \frac{1}{2} \sum_i \sum_j d_i d_j y^{(i)} y^{(j)} x^{(i)T} x^{(j)}$$

$$\min \frac{1}{2}\left(\sum_{i=1}^{m} d_i\, y^{(i)} x^{(i)}\right)^T \left(\sum_{j=1}^{m} d_j\, y^{(j)} x^{(j)}\right)$$

$$= \min \frac{1}{2} \sum_i \sum_j d_i d_j\, y^{(i)} y^{(j)} \underbrace{x^{(i)T} x^{(j)}}_{\langle x^{(i)}, x^{(j)} \rangle}$$

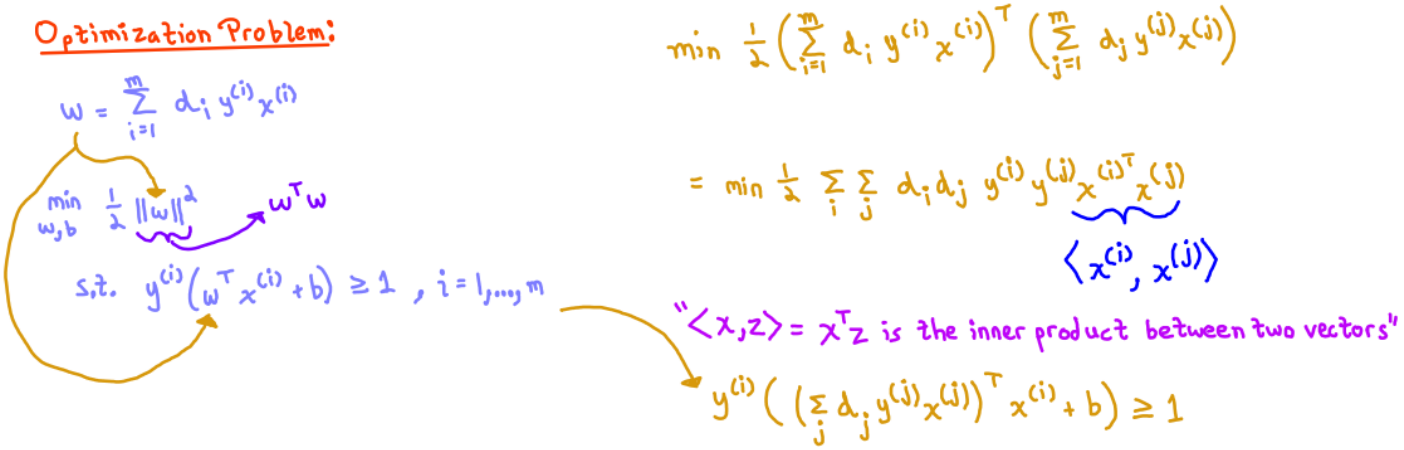
"$\langle x, z \rangle = x^T z$ is the inner product between two vectors"

Kernels:

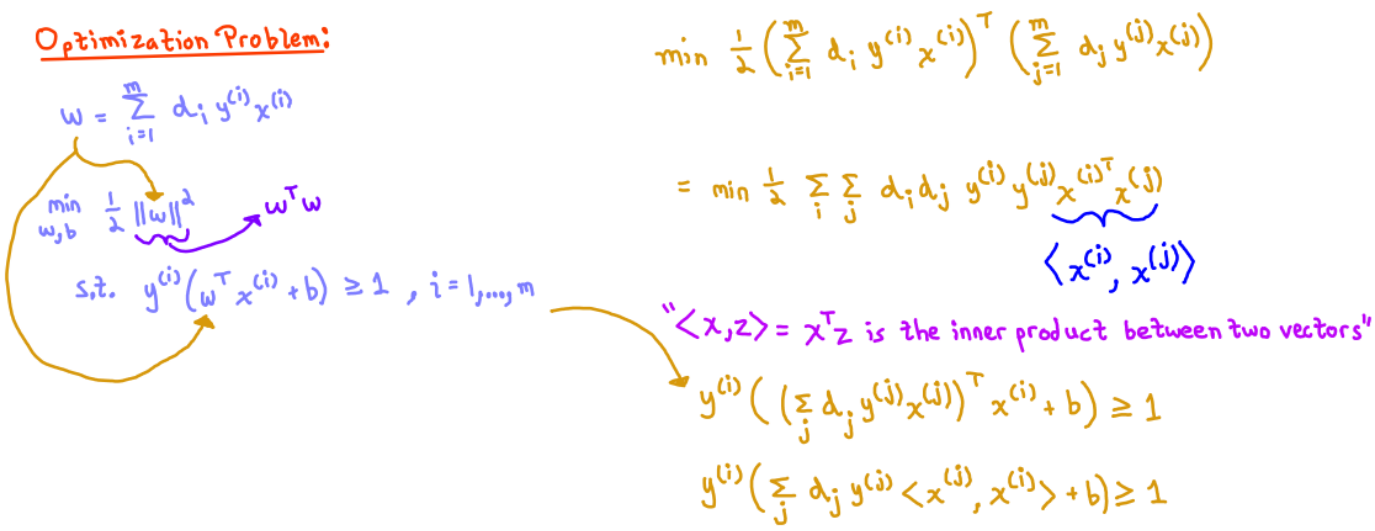
The mechanism for work on these incredibly high-dimensional feature spaces

When we derive kernels, you see that expressing your algorithm in terms of inner products between features **x** is the key mathematical step needed to derive kernels. We'll use this slightly different set of angle bracket notation to denote the inner product between two different feature vectors

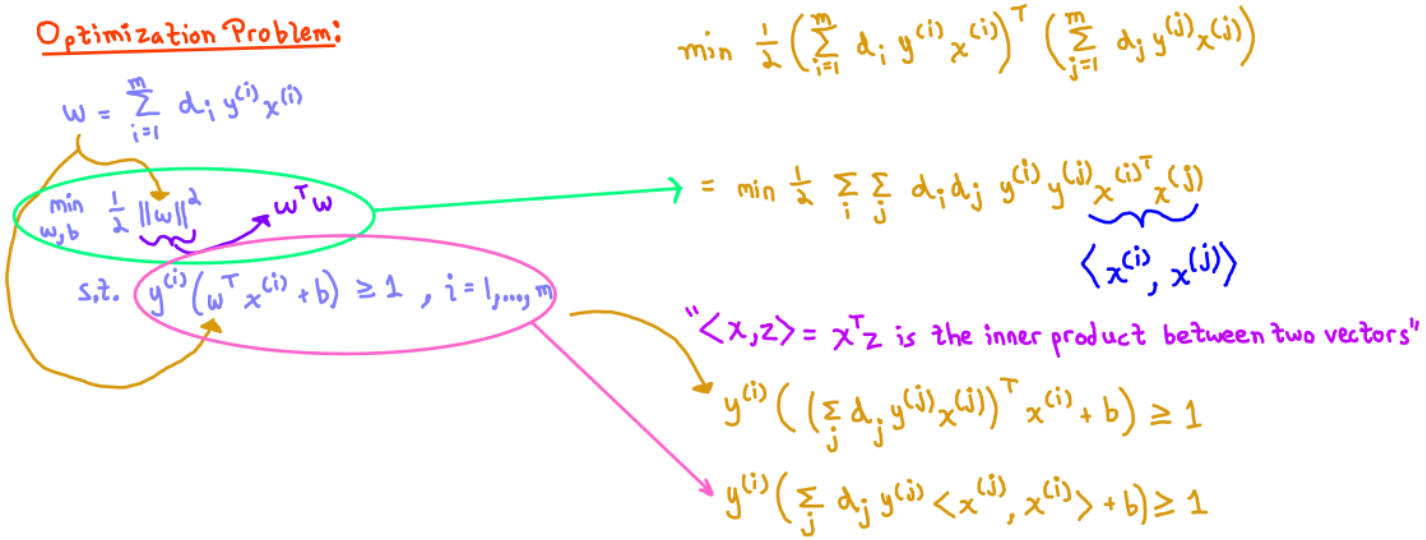
This is the optimization objective, and the previous constraint becomes:



And if you just multiply this out:



To make sure the mapping is clear:



The key property we're going to use is, if you look at these two equations in terms of how we pose the optimization problem, the only place that the feature vectors appears is in the inner product

It turns out that (when we discussed the **Kernel** trick and the application of **Kernels**), if you can compute the inner product very efficiently, that's when you can get away with manipulating even infinite-dimensional feature vectors

The reason we want to write out the whole algorithm in terms of inner products is there'll be important cases where the feature vectors are 100 trillion-dimensional or even infinite-dimensional, but you can compute the inner product very efficiently without needing to loop over a large number of elements in an array

By convention, in the way that you see **support vector machines** referred to in research papers or in textbooks, it turns out that there's a further simplification of that optimization problem, known as the **Dual Optimization Problem**, which is that you can simplify to this by using ***convex optimization theory***:

Simplified Optimization Problem:

$$\max \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle$$

$$\text{s.t. } \alpha_i \geq 0$$

$$\sum_i y^{(i)} \alpha_i = 0$$

"Dual Optimization problem"

To make a prediction:

1) Solve for α_i's, b

2) To make a prediction, compute:

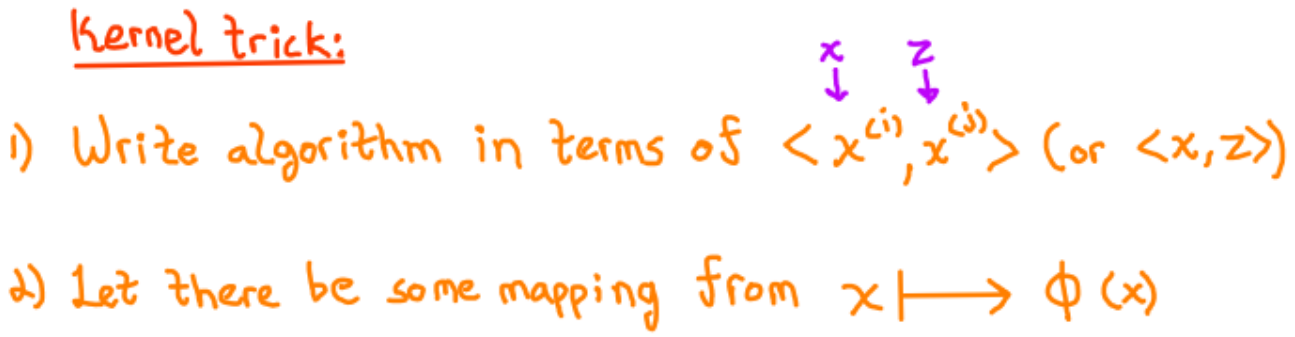
$$h_{w,b}(x) = g(w^T x + b)$$

$$= g\left(\left(\sum_i \alpha_i y^{(i)} x^{(i)}\right)^T x + b\right)$$

$$= g\left(\sum_i \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b\right)$$

Once you have stored the **α**'s in your computer memory, you can make predictions using just inner products again. And so the entire algorithm, both the optimization objective you need to deal with during training as well as how you make predictions, is expressed only in terms of inner products

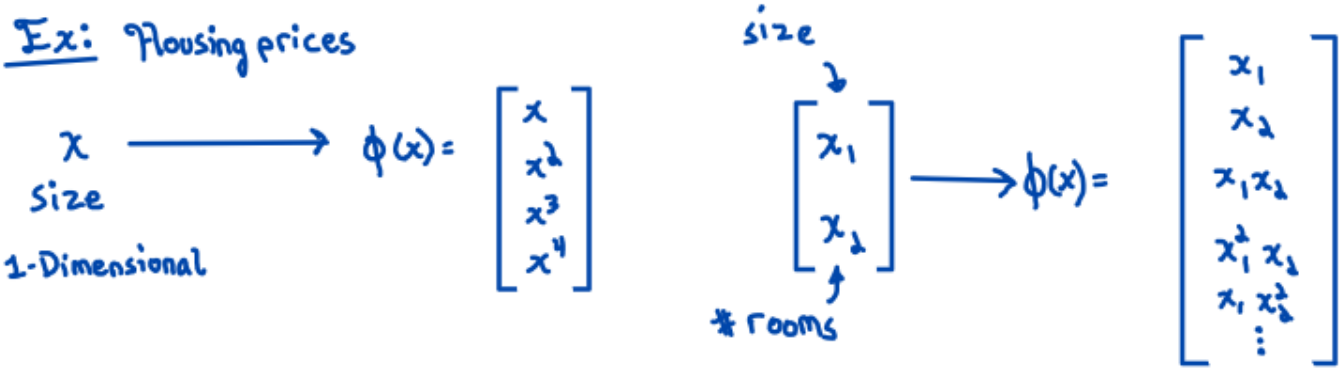
Kernel Trick (Applying Kernels):



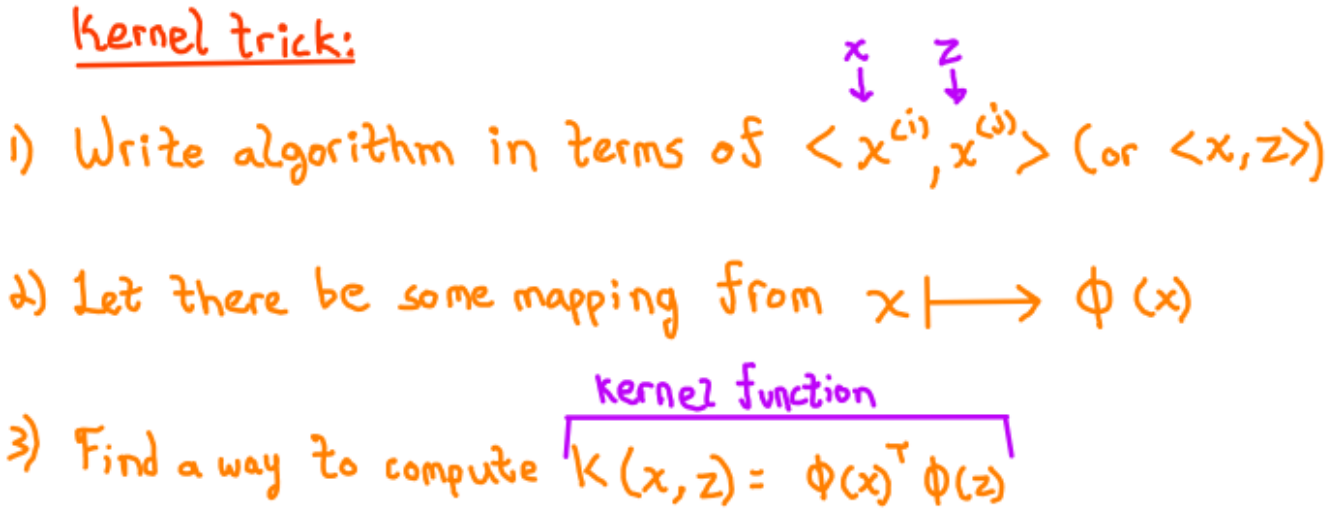
Let there be some mapping from your original input features **x** to some high-dimensional set of features **Φ**

You could take this 1D feature and expand it to a high-dimensional feature vector, so this would be one way of defining a high-dimensional feature mapping

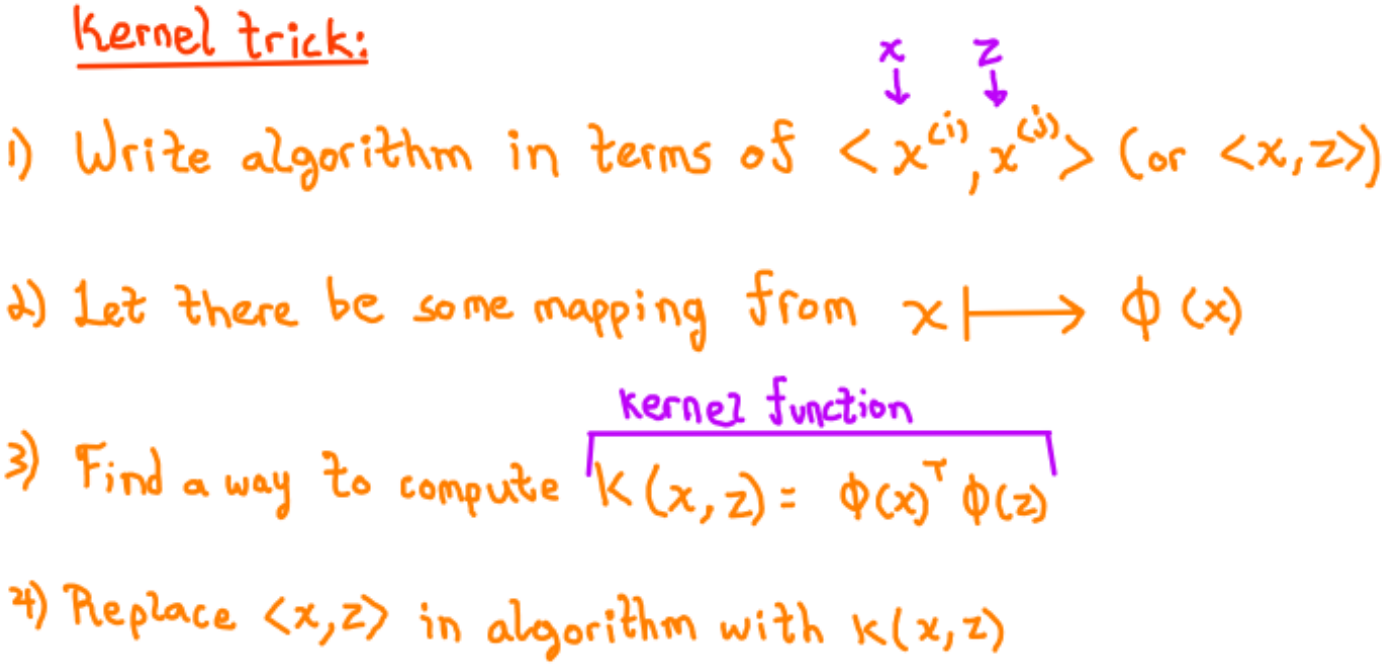
Another example:



What we'll be able to do is work with feature mappings, **Φ(x)**, where the original input **x** may be 1-dimensional, 2-dimensional, etc, and **Φ(x)** could be 100,000-dimensional or infinite-dimensional. We'll be able to do this very efficiently



What we're going to do is, we'll see that there are clever tricks so that you can compute the inner product between **x** and **z**, even when **Φ(x)** and **Φ(z)** are incredibly high-dimensional



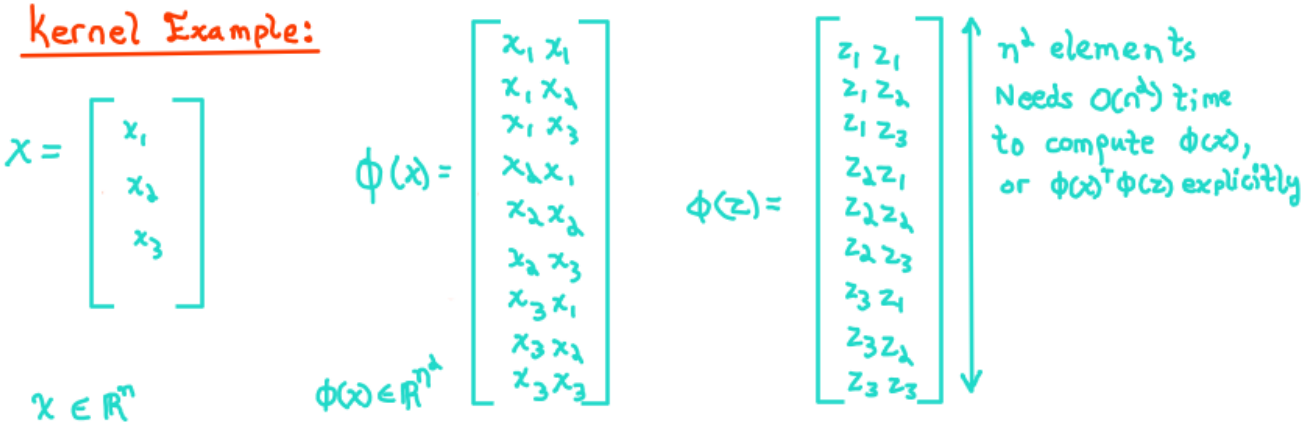
If you could do this, then what you're doing is, you're running the whole learning algorithm on this high-dimensional set of features. The problem with swapping out **x** for **Φ(x)** is that it can be very computationally expensive if you're working with 100,000 dimensional feature vectors, but that's what you would do if you were to swap in **Φ(x)** for **x** (in the naive straightforward way)

What we'll see is that if you can compute **K(x,z)**, then you could, because you've written your whole algorithm just in terms of inner

products, not need to explicitly compute **Φ(x)**. You can always just compute these **kernels**

Example of Kernels:

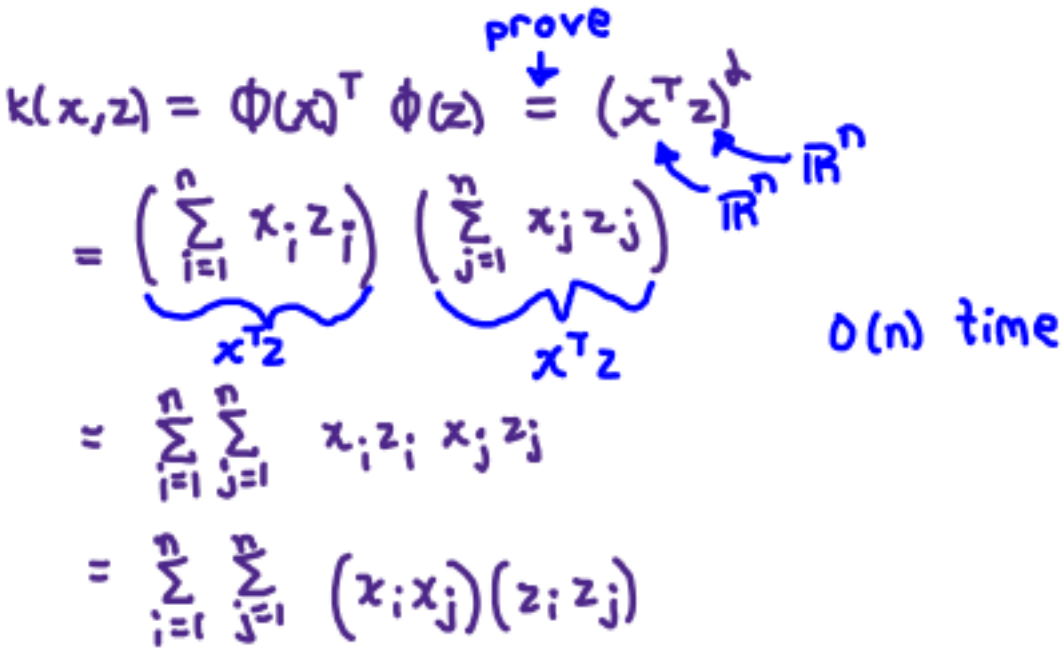
For this example, let's say that your original input features was 3-dimensional and the feature mapping **Φ(x)** to be all pair-wise monomial terms



Really it's **n/2** because a lot of these elements are duplicated

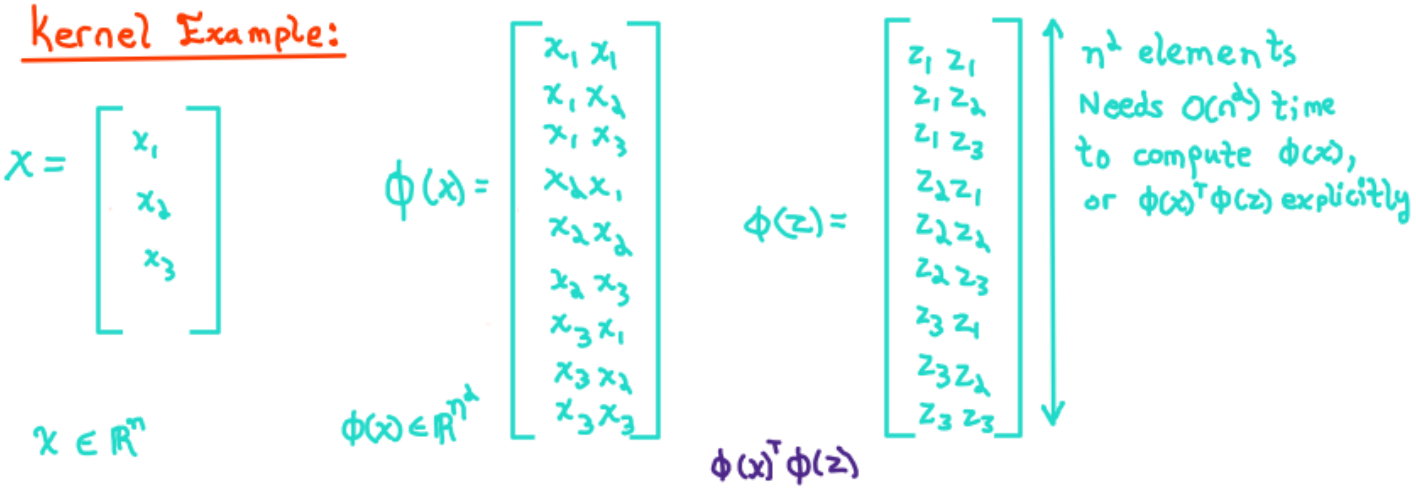
We can find a better way to do that

What we want is to write out the Kernel of **(x,z)** and prove that this can be computed as $(x^Tz)^2$:



Because **x** is n-dimensional and **z** is n-dimensional, $(x^T z)^2$ is an **O(n)** time computation because taking $x^T z$ is just the inner product of two n-dimensional vectors and then you take that number ($x^T z$ is a real number) and you just square it

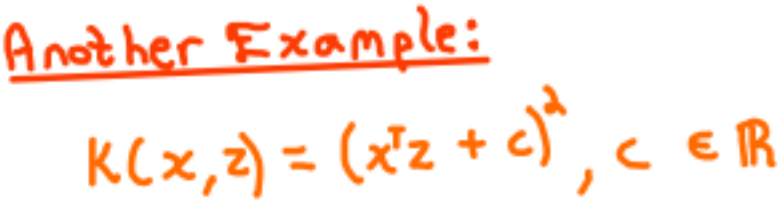
If you apply this to the earlier example:



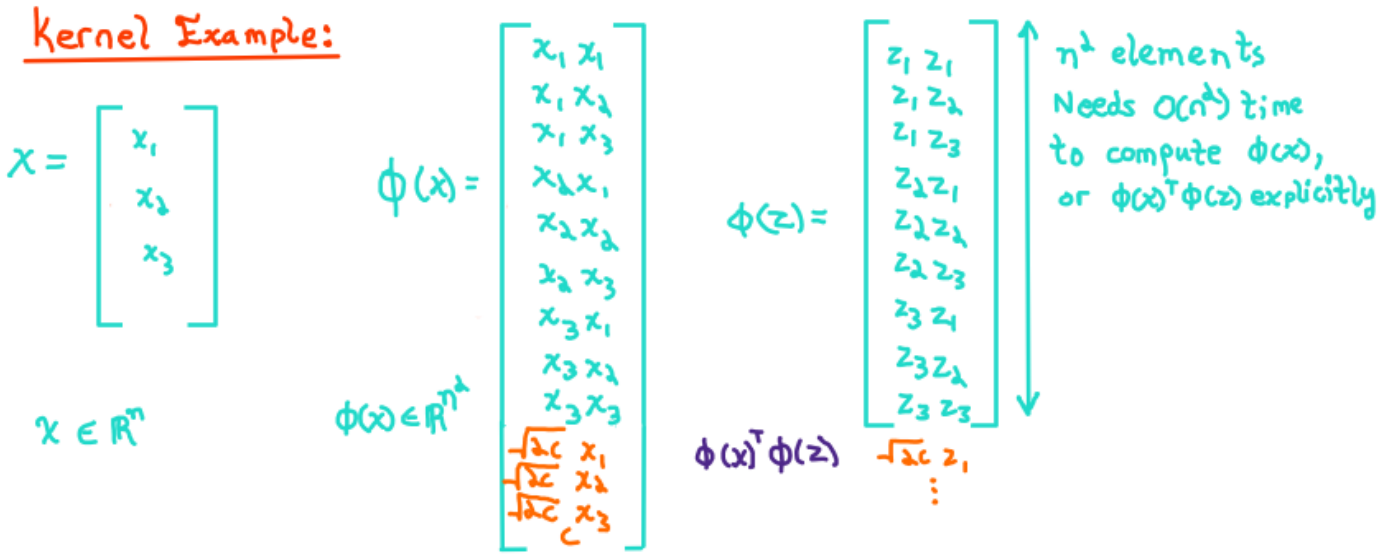
This proves that you've turned what was previously an order n^2 (**O(n^2)**) time calculation into an order **n** (**O(n)**) time calculation, which means that, if **n** was **10,000**-dimensional, instead of needing to manipulate **100,000**-dimensional vectors, you could do so by manipulating only **10,000**-dimensional vectors

Other Examples of Kernels:

It turns out that if you choose this Kernel where you now add a "**+ c**", where **c** is a constant (some fixed real number):



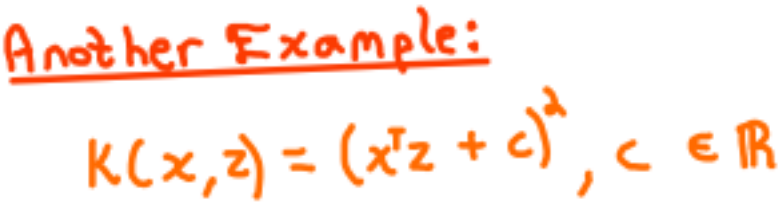
That corresponds to modifying your features as follows:



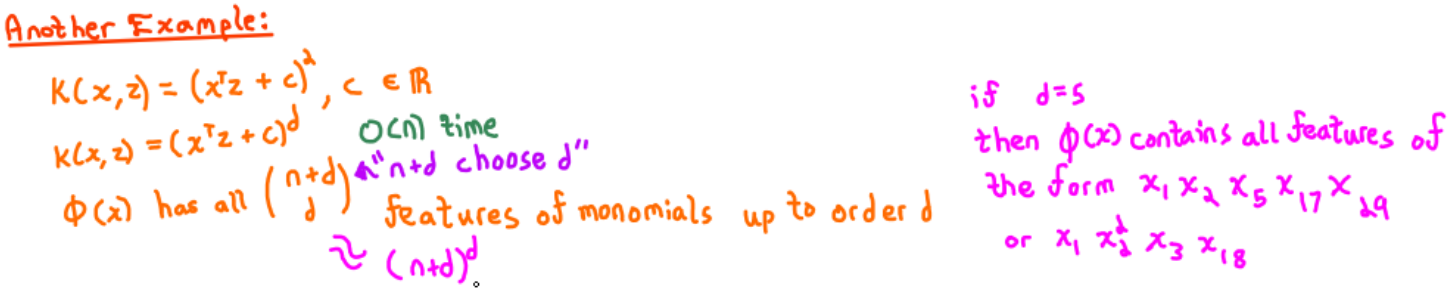
Other examples:

When you add **+c** there, it corresponds to adding the above

If this is your new definition for **Φ(x)** and make the sam change to **Φ(z)**, then if you take the inner product of these, then it can be computed as this:



The role of the constant **c** is that it trades off the relative weighting between the binomial terms (the $\mathbf{x_i x_j}$) compared to the single (first degree) terms like $\mathbf{x_1}$, $\mathbf{x_2}$, or $\mathbf{x_3}$



So if you choose this as your **kernel**, this corresponds to constructing **Φ(x)** to contain all of these features, and there are kind've exponentially many of them, the monomials (basically all the monomial terms up to a fifth degree polynomial (fifth order monomial) term)

It turns out that there are very roughly $(n+d)^d$ number of features, but your computation doesn't blow up exponentially even as **d** increases

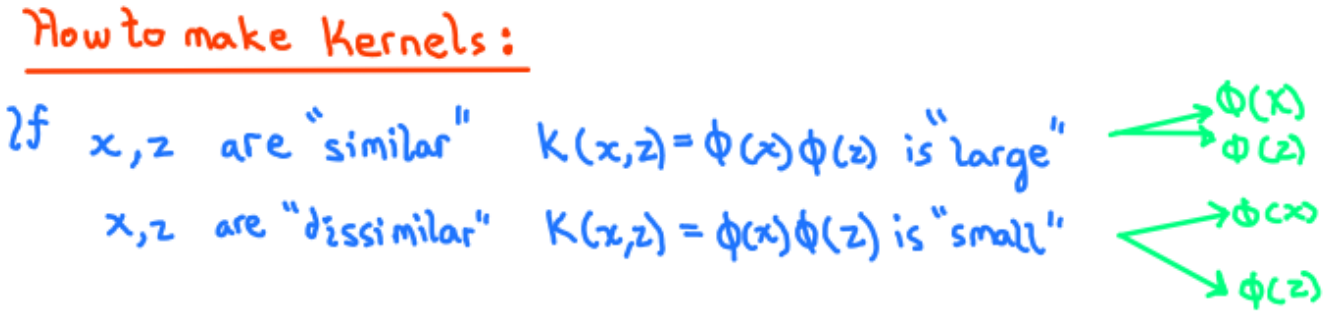
What a Support Vector Machine is is taking the Optimal Margin Classifier, that was derived earlier, and applying the **kernel trick** to it

If you choose some of these **kernels** for example, then you could run an SVM in these very very high-dimensional feature spaces, but your computational time scales only linearly as **O(n)** (as the dimension of your input features x rather than as a function of this n^2-dimensional feature space). You're actually building a linear classifier

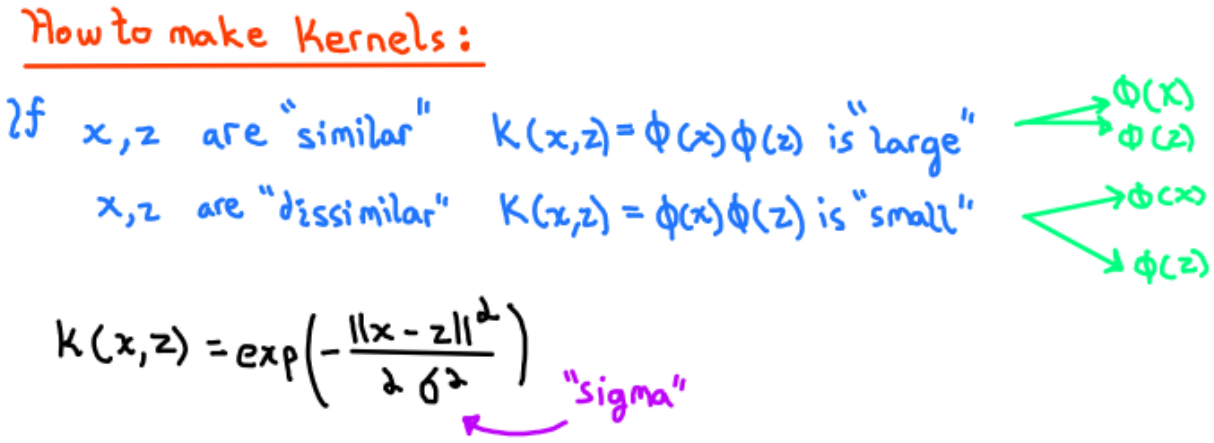
Visualization: https://www.youtube.com/watch?v=OdlNM96sHio

You're taking the data, mapping it to a much higher dimensional feature space, and then finding a linear decision boundary in that higher dimensional feature space, which is going to be a hyperplace (a straight line or plane)

How do you make Kernels?:



If you think of **Kernels** as a similarity measure of a function, this function below does have the property that if **x** and **z** are very close to each other, then this function would evaluate to $\mathbf{e}^{\mathbf{0}}$, which is about **1**, but if **x** and **z** are very far apart, then this function would be small



So this function actually satisfies the criteria

Is it okay to use this as a **kernel** function? It turns out that you can use it as a **kernel** function only if:

We've derived the whole algorithm assuming this to be true, but it turns out that if you plug in the **kernel** function for which this isn't true, then all of the derivation breaks down and the optimization problem can have very strange solutions that don't correspond to good classification (to a good classifier) at all. So this puts constraints on what **kernel** functions we could choose

For example, one thing it must satisfy is:

$$k(x,x) = \phi(x)^T \phi(z) \geq 0$$

This had better be greater than or equal to **0** because inner product of a vector with itself had better be non-negative. So if it is ever less than **0**, then this is not a valid **kernel** function

More generally, there's a theorem that proves when something is a valid **kernel**

Valid Kernel Theorem:

Let $\{x^{(1)}, \ldots, x^{(d)}\}$ be d points

Let $K \in \mathbb{R}^{d \times d}$

$K_{ij} = K(x^{(i)}, x^{(j)})$ "kernel matrix"

This is also sometimes called the **Gran (Gram?)** matrix

So just apply the **Kernel** to every pair of those **d** points and put them in a **d**x**d** matrix like that

Given the following, if **K** is a valid **kernel** function, so if there is some feature mapping **Φ**, then:

Given any vector z,

$$z^T K z = \sum_i \sum_j z_i K_{ij} z_j$$

$$= \sum_i \sum_j z_i \phi(x^{(i)})^T \phi(x^{(j)}) z_j$$

$$= \sum_i \sum_j z_i \sum_k (\phi(x^{(i)}))_k (\phi(x^{(j)}))_k z_j$$

$$= \sum_k \sum_i \sum_j z_i (\phi(x^{(i)}))_k (\phi(x^{(j)}))_k z_j$$

$$= \sum_k \left(\sum_i z_i \phi(x^{(i)})_k \right)^2 \geq 0$$

So $K \geq 0$

So this proves that the **Kernel matrix**, **K**, is *positive semi-definite*

More generally, it turns out that this is also a sufficient condition for a function **K** to be a valid **kernel** function

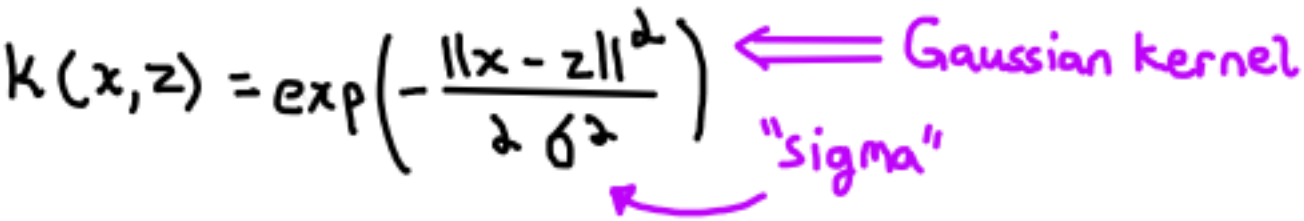
Mercer's Theorem:

Mercer's Theorem:

K is a valid kernel function (i.e. $\exists \phi$ s.t. $K(x,z) = \phi(x)^T \phi(z)$) if and only if for any d points $\{x^{(1)}, \ldots, x^{(d)}\}$, the corresponding kernel matrix $K \geq 0$ "positive semi-definite"

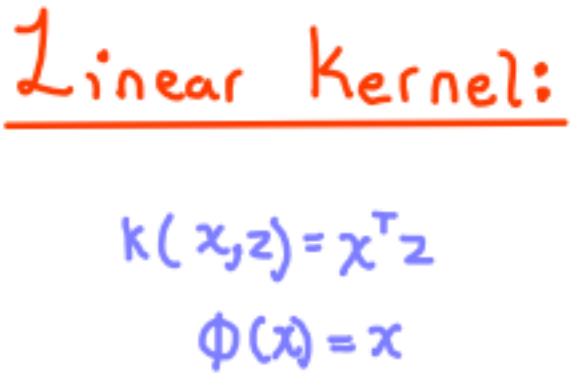
Gaussian Kernel:

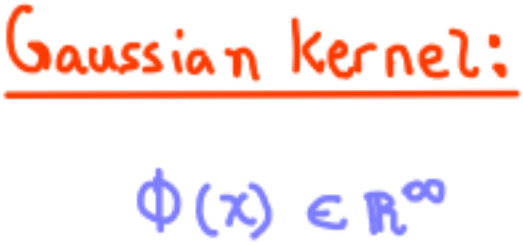
It turns out that this **kernel** is a valid **kernel** known as the **Gaussian Kernel**



This is one of the most widely used **kernels**

Linear Kernel:





The most widely used **kernel** is maybe the **Linear Kernel**. It uses **Φ(x) = x**, so there are no high-dimensional features. The **Linear Kernel** just means you're not using a high dimensional feature mapping, or the feature mapping is just equal to the original features

After the **Linear Kernel**, the **Gaussian Kernel** is probably the most widely used **kernel** which corresponds to a feature dimensional space that is infinite-dimensional. This particular **kernel function** corresponds to using all monomial features. It corresponds to using all these polynomial features without end (going to arbitrarily high-dimensional space), but giving giving a smaller weighting to the very very high-dimensional ones , which is why it's widely used

It turns out that the **kernel trick** is more general than the **support vector machine**. It was really popularized by the **support vector machine** where researchers (***Vladimir Vapnik*** and ***Corinna Cortes***) found that applying these ***kernel tricks*** to a **support vector machine** makes for a very effective learning algorithm

But the **Kernel trick** is actually more general and if you have any learning algorithm that you can write in terms of inner products like $\langle x^{(i)}, x^{(j)} \rangle$, then you can apply the ***kernel trick*** to it

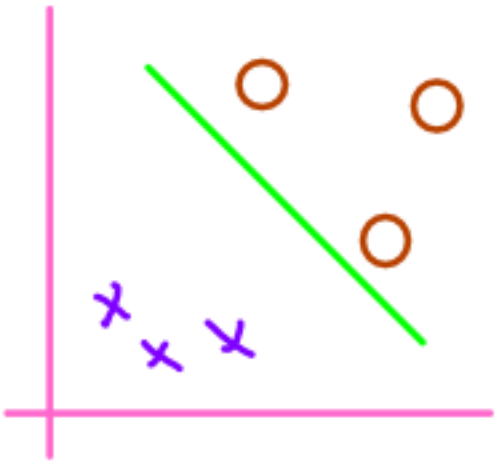
The way to apply the ***Kernel trick*** is to take a learning algorithm, write the whole thing in terms of inner products, and then replace it with **k(x,z)** for some appropriately chosen **kernel** function **k(x,z)**

All of the ***discriminative learning algorithms*** discussed so far, can be written in this way so that you can apply the ***kernel trick***. So ***Linear Regression***, ***Logistic Regression***, everything in the ***Generalized Linear Model family***, the ***perceptron algorithm***, etc. All of those algorithms, you can actually apply the ***kernel trick*** to, which means that you could apply Linear Regression in an infinite dimensional feature space if you wish

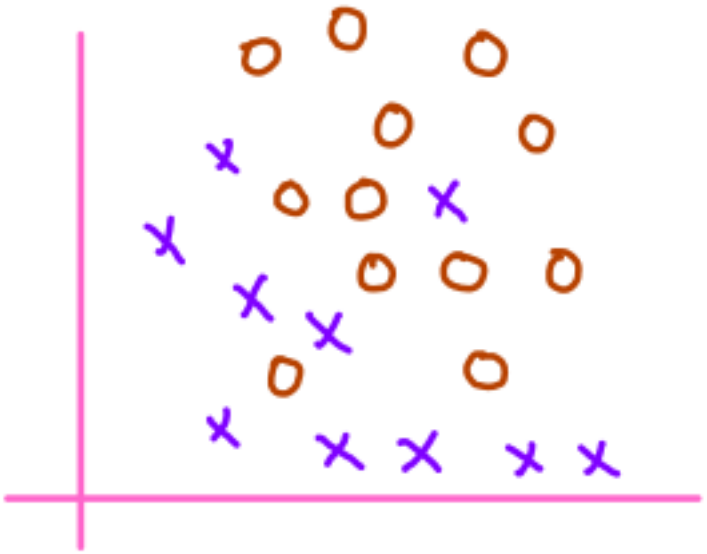
A lot of algorithms can be married with the ***kernel trick*** to implicitly apply the algorithm in even an infinite dimensional feature space but without needing your computer to have an infinite amount of memory or using infinite amounts of computation

The single place that it's most powerfully applied is the ***support vector machine***. In practice, the ***kernel trick*** is applied all the time in ***support vector machines*** and less often in other algorithms

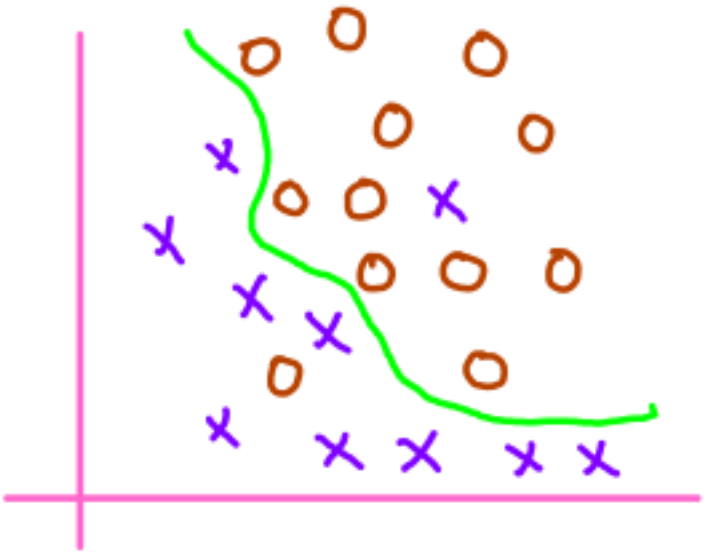
We now want to fix the assumption we had made that the data is linearly separable



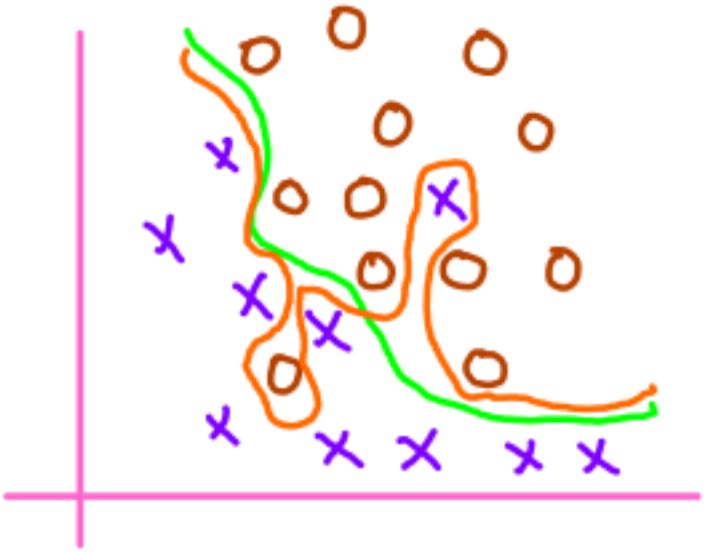
Sometimes you don't want your learning algorithm to have zero errors on the training set, so when you take this low-dimensional data and map it to a very high dimensional feature space, the data does become much more separable, but it turns out that if your data set is a little bit noisy:



If your data looks like this, you'd maybe want it to find a decision boundary like this:



And you don't want ti to try so hard to separate every little example as defined by a really complicated decision boundary like this:



So sometimes, either in the low-dimensional space or in the high-dimensional space (**Φ**), you don't actually want the algorithms to separate out your data perfectly, and then sometimes even in high-dimensional feature space, your data may not be linearly separable and you don't want the algorithm to have zero error on the training set

There's an algorithm called the ***L_1 norm soft margin SVM***, which is a modification to the basic algorithm

L_1 norm soft margin SVM algorithm:

$$L_1 \text{ norm soft margin SVM algorithm:}$$

$$\min \frac{1}{2}\|w\|^2 \quad \text{s.t.} \quad \underbrace{y^{(i)}(w^T x^{(i)} + b)}_{\text{functional margin}} \geq 1 \quad , \quad i = 1, \ldots, m$$

What the ***L_1 norm soft margin*** does is the following, the above ***optimization problem*** was saying "Let's make sure each example has function margin greater or equal to 1" and in the ***L_1 norm soft margin SVM***, we're going to relax this

We're going to say that it needs to be bigger than **1 - "Greek alphabet C"** and then modify the cost function as follows:

$$\text{min} \ \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \quad \text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i \ , \ i = 1,\dots,m \ , \ \xi_i \geq 0$$

ℓ_1 norm soft margin SVM algorithm:

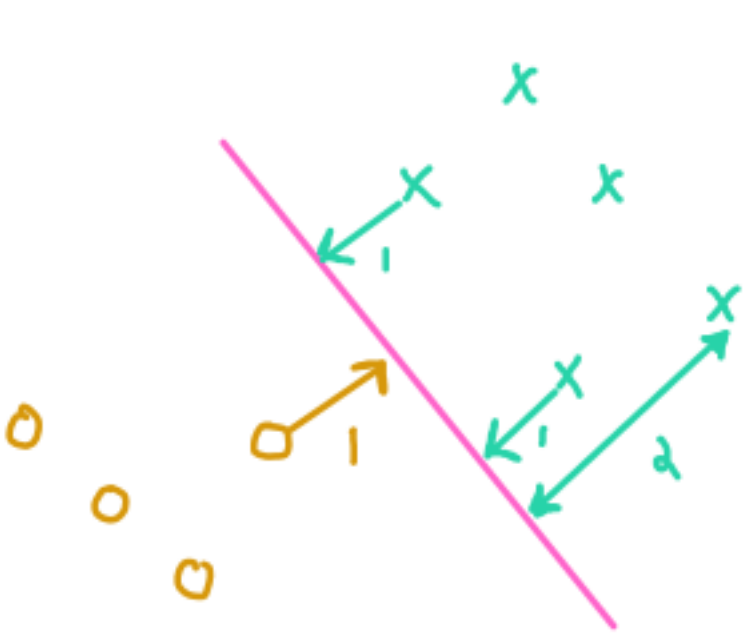
"Greek alphabet C"

functional margin

Remember, if the ***functional margin*** is greater than or equal to 0, it means the algorithm is classifying that example correctly and the **SVM** is asking for it to not just classify correctly but to classify it correctly with the ***functional margin*** of at least 1, and if you allow ξ_i to be positive ($\xi_i > 0$), then that's relaxing that constraint. You don't want the ξ_i's to be too big, which is why you add to the ***optimization cost function*** a cost for making ξ_i too big. So you optimize this as a function of **w**, **b**, **ξ**

This parameter **C** is something you need to choose as it trades off how much you want to insist on getting the training examples right versus saying "it's okay if you label a few terms out of this one"

If you draw a picture, it turns out that in this example below with that being the optimal decision boundary, these three examples would be equidistant from the this straight line, because if they weren't then you can fiddle the straight line to improve the margin even a little bit more



These few examples have **functional margin** exactly equal to 1 and the other example will have **functional margin** equal to 2, and

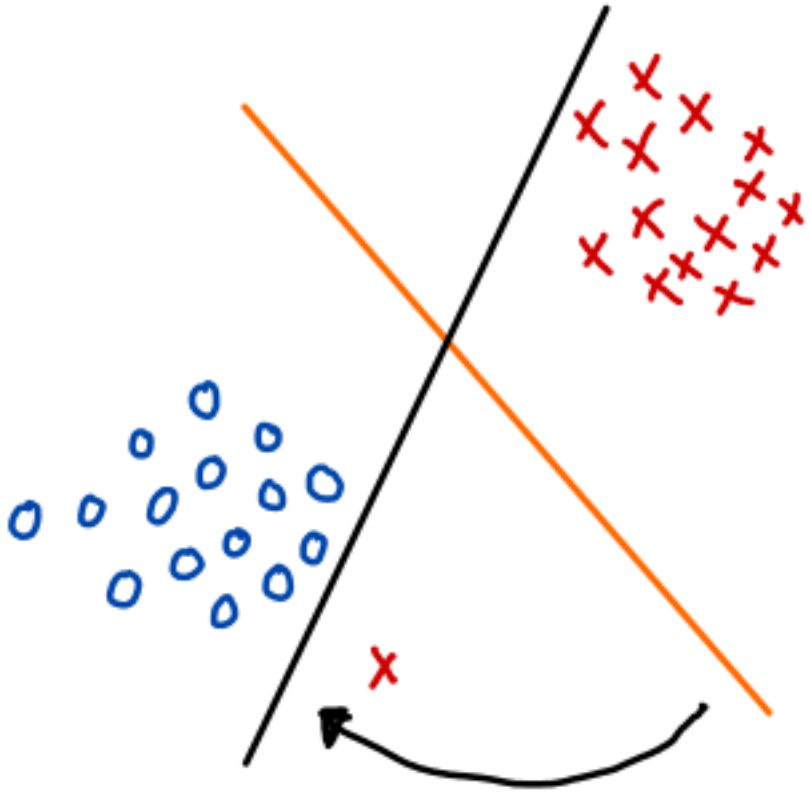
the further away examples are, they'll have even bigger **functional margins**

What this optimization objective is saying is that it is okay if you have an example with **functional margin** a little bit less than 1 and by setting ξ_i to 0.5 (for example), is letting you get away with having **functional margin** a little bit less than 1

ξ_i

One other reason why you might want to use L_1 **norm soft margin SVM** is the following. Let's say you have a data set that looks like this with a decision boundary like this (**orange**)

If you have just one outlier, then technically the data set is still linearly separable. If you really want to separate out this data set, you can actually choose this decision boundary (**black**):



But the **basic optimal margin classifier** will allow the presence of one training example to cause you to have this dramatic swing in the position of the decision boundary. The original ***optimal margin classifier***, it optimizes for the worst-case margin. The concept of optimizing for the worst-case margin allows one example, by being the worst case training example, to have a huge impact on your decision boundary

So the L_1 **norm soft margin SVM** allows the **SVM** to still keep the decision boundary closer to the (**orange**) line even when there's one outlier, and it makes it much more robust to outliers

If you go through the ***Representer theorem*** derivation, representing **w** as a function of the alphas and so on, it turns out that the problem then simplifies to the following:

$$\max \sum_{i=1}^{m} d_i - \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} d_i d_j \langle x^{(i)}, x^{(j)} \rangle$$

$$\text{s.t.} \quad \sum_{i=1}^{m} y^{(i)} d_i = 0$$

$$0 \le d_i \qquad i = 1, \ldots, m$$

The only change we make to this is we end up with an additional condition on the authorize. So if you go for that simplification, now that you've changed the algorithm to have this extra term, then this new form is called the ***Dual form of the optimization problem***

$$\max \sum_{i=1}^{m} d_i - \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} d_i d_j \langle x^{(i)}, x^{(j)} \rangle$$

$$\text{s.t.} \quad \sum_{i=1}^{m} y^{(i)} d_i = 0$$

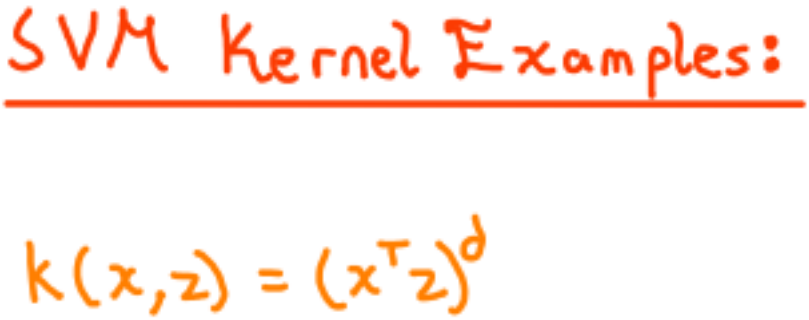
$$0 \le d_i \le C \quad , \; i = 1, \ldots, m$$

It turns out today there are very good numerical software packages which are solving that for you and you can just code without worrying about the details that much

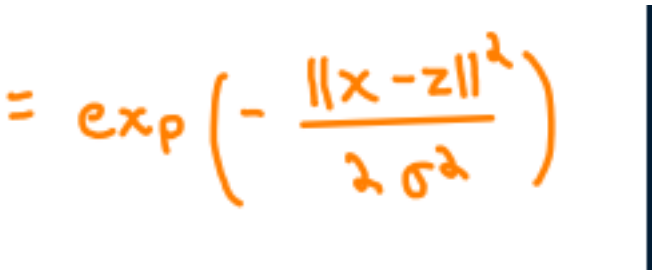
Examples of SVM Kernels:

It turns out the **SVM** with a polynomial **Kernel** works quite well

This is called a ***polynomial kernel***:



and this is called a ***Gaussian kernel***:

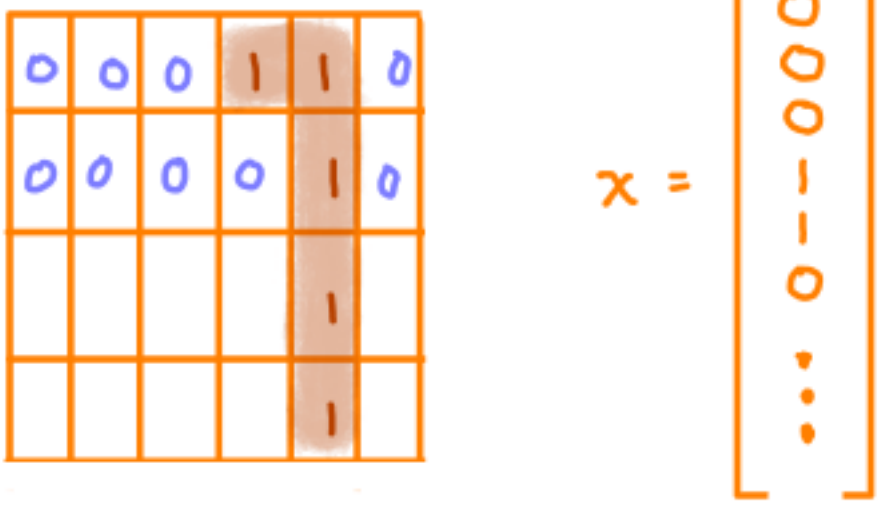


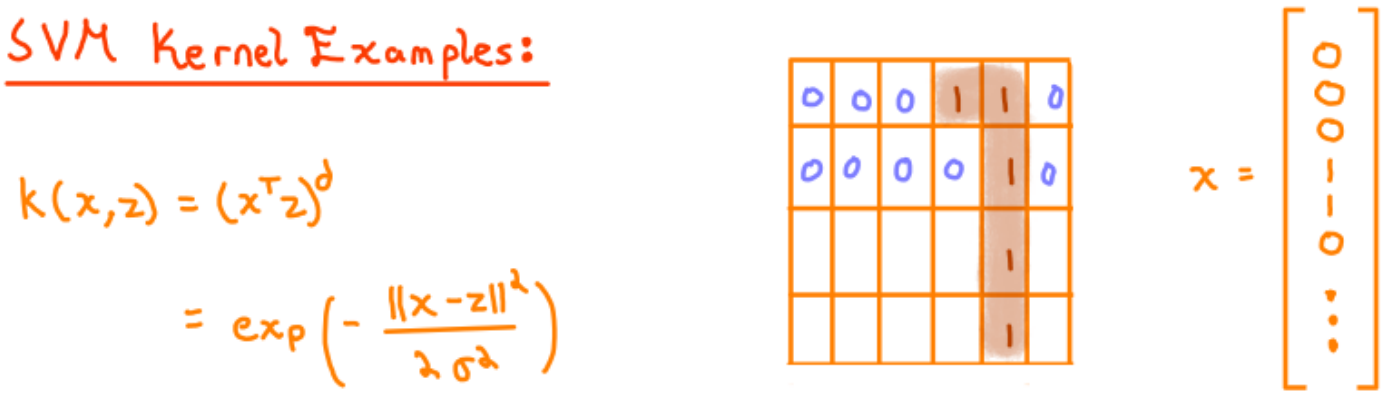
SVM Kernel Examples:

$$k(x,z) = (x^T z)^d$$

$$= \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right)$$

In the early days of ***SVM***s, one of the proof points of ***SVM***s that the field of Machine Learning was doing a lot of work on was on handwritten digit classification. A digit is a matrix of pixels with values that are 0 or 1 or maybe grayscale values and if you take a list of pixel intensity values and just list out all the pixel intensity values then that can be your feature **x**, and if you feed it to a **SVM** using either of the two above ***kernels*** (***polynomial kernel, Gaussian kernel***), it'll do not too badly as a handwritten digit classification





There's a classic data set called **MNIST** which is a classic benchmark in the history of Machine Learning and it was a very surprising result many years ago that ***SVM*** with a **kernel** like this does very well on handwritten digit classification

In the past several years, we've found that ***deep learning algorithms***, specifically ***Convolutional Neural Networks* (CNN)**, do even better than the ***SVM***, but for some time ***SVM***s were the best algorithm, and they're very easy to use in turnkey. There aren't a lot of parameters to fiddle with so that's the one very nice property about them

More generally, a lot of the most innovative work in ***SVM***s has been into the design of ***kernels***

#Knuth-Morris-Pratt algorithm#

As you apply **SVM**s, one of the things you see is that depending on the input data you have, there can be innovative ***kernels*** to use in order to measure the similarity of two amino acid sequences or the similarity of two of whatever else and then to use that to build a classifier even on a very strange shaped object which do not come as a feature

Another example is if the input **x** is a histogram, maybe of two different countries (you have histograms of people's demographics), it turns out that there is a ***kernel*** that's taking the ***min*** of the two histograms and then summing up to compute a ***kernel function*** that inputs two histograms that measures how similar they are

So there are many different ***kernel*** functions for many different unique types of inputs you might want to classify

#?#***Delta Margin Classifier***#?#