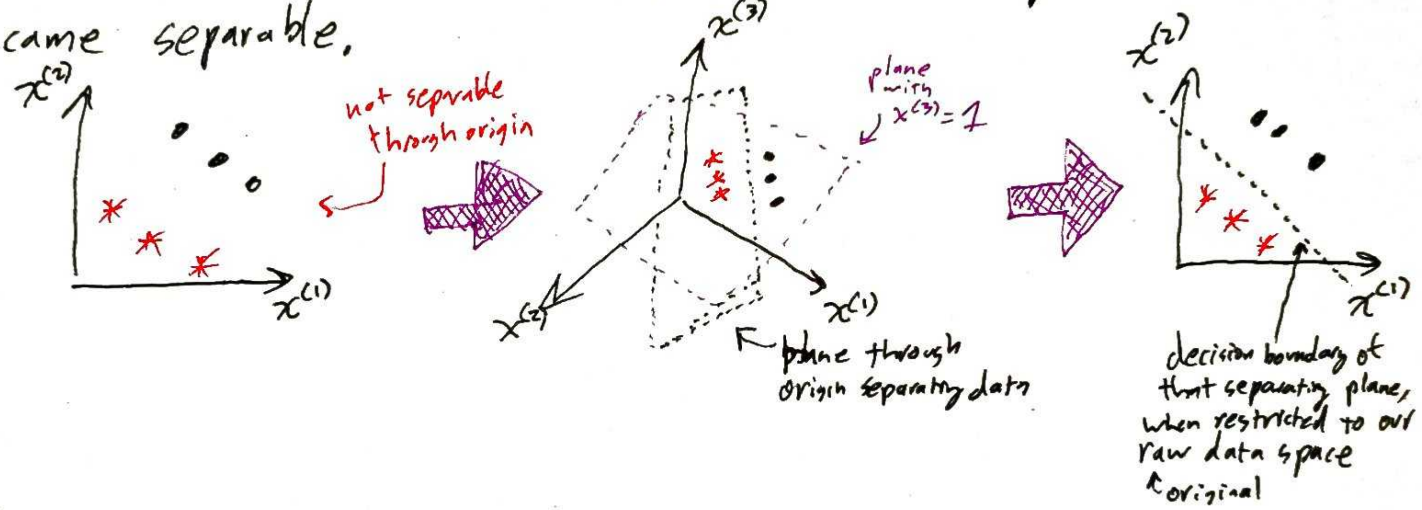
Extension to Perceptrons: The Kernel Trick

Let's explore the kernel trick in detail.
First, recall our change to make the perceptron algorithm find decision boundaries that did not go through the origin. We modified our feature vectors:

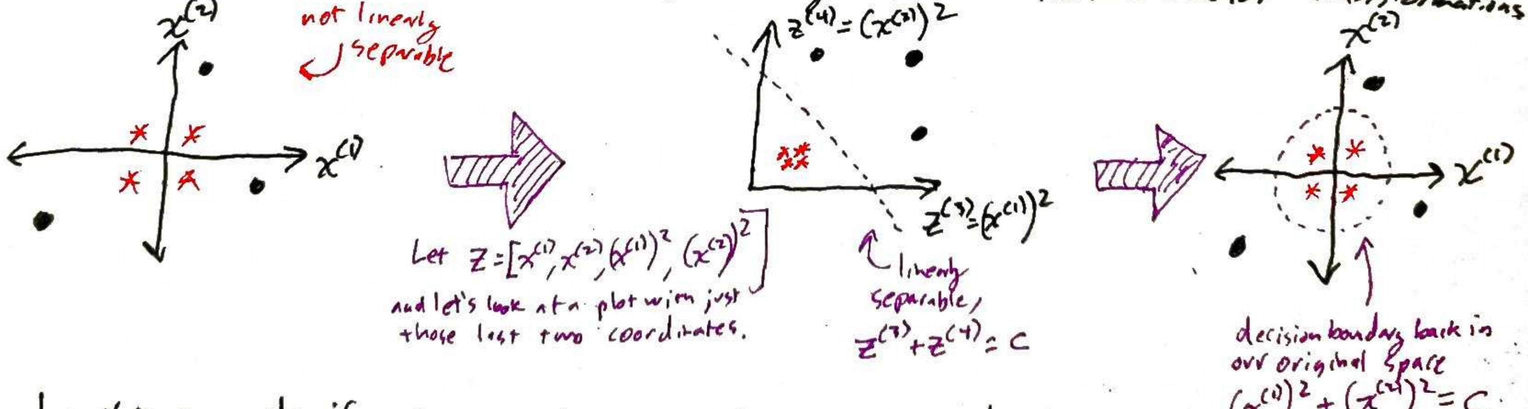
$$x = [x^{(1)}, \dots, x^{(d)}]^T \implies z = [1, x^{(1)}, \dots, x^{(d)}]^T$$

raw feature vector — transformed feature vector

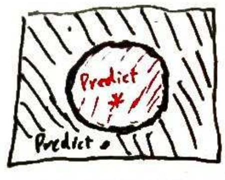
With this transformation, data which was not separable became separable.



We can take this further, using other feature vector transformations



In this example, if we were to run our perceptron algorithm on the transformed data z, it might be able to find a <u>non-linear</u> decision boundary w.r.t. our original space!



We call these transformations <u>feature maps</u>

$$\Phi : \mathbb{R}^d \to \mathbb{R}^m$$

$\mathbb{R}^d$ ↳ input is a d-dimensional feature vector

$\mathbb{R}^m$ ↳ output is an m-dimensional feature vector.

Our goal is to apply our learning algorithms not on our original data vectors x, but instead on the mapped vectors $\Phi(x)$.

One way is to just preprocess all our data with Φ, but sometimes there is a better way. Sometimes the feature map Φ is huge or complicated to compute while the inner products $\langle \Phi(x), \Phi(z) \rangle$ can be computed efficiently in another way.

The <u>Kernel trick</u> is to rewrite an algorithm to only interact with the feature vectors through dot products, and then use a kernel function that computes the same values as $\langle \Phi(x), \Phi(y) \rangle$

<u>Kernel function</u>: For a feature map Φ, K is a Kernel function for Φ if

$$K(x,z) = \langle \Phi(x), \Phi(z) \rangle \text{ for all } x, z \text{ in the domain of } \Phi.$$

The kernel trick cannot always be done, but it can for several common algorithms, including perceptrons and SVMs.

Examples of Kernels

what Φ corresponds to this K?

$\boxed{1}$ $K(x,z) = (\langle x, z\rangle)^2$

$$= \left(\sum_{i=1}^{d} x^{(i)} z^{(i)}\right)^2$$

$$= \left(\sum_{i=1}^{d}\sum_{j=1}^{d} x^{(i)}x^{(j)}z^{(i)}z^{(j)}\right)$$

$$= \langle \Phi(x), \Phi(z)\rangle$$

where $\Phi([x^{(1)},\dots,x^{(d)}]) = [x^{(1)}x^{(1)}, x^{(1)}x^{(2)}, \dots, x^{(1)}x^{(d)}, x^{(2)}x^{(1)}, \dots, x^{(d)}x^{(d)}]$

d^2 terms, one for each pair of coordinates

In 2-dimensions, this looks like
$$K(x,z) = (x^{(1)})^2(z^{(1)})^2 + (x^{(1)}x^{(2)})(z^{(1)}z^{(2)}) + (x^{(2)}x^{(1)})(z^{(2)}z^{(1)}) + (x^{(2)})^2(z^{(2)})^2$$

and $\Phi(x) = [(x^{(1)})^2, x^{(1)}x^{(2)}, x^{(2)}x^{(1)}, (x^{(2)})^2]$

[with a little care, you could simplify and combine like terms]

Some Observations:

- computing $\Phi(x)$ is $\Theta(d^2)$, in time and space
- computing $K(x,z)$ is just $\Theta(d)$, since after taking the dot product $\langle x, z\rangle$, squaring it is $O(1)$

$\boxed{2}$ $K(x,z) = (\langle x, z\rangle + c)^2$ $\quad c \geq 0$

$$= \left(\sum_{i=1}^{d} x^{(i)} z^{(i)} + c\right)^2$$

$$= \Bigg(\underbrace{\sum_{i=1}^{d}\sum_{j=1}^{d} x^{(i)}x^{(j)}z^{(i)}z^{(j)}}_{\text{same set of features as in example } \boxed{1}} + \underbrace{2c\sum_{i=1}^{d} x^{(i)}z^{(i)}}_{2c\langle x,z\rangle = \langle\sqrt{2c}\,x, \sqrt{2c}\,z\rangle} + \underbrace{c^2}_{\substack{\langle c,c\rangle \\ \text{1-d dot product} \\ \text{one feature}}}\Bigg)$$

If we let $\Phi(x)$ be the feature map from $\boxed{1}$

the $\Psi(x) = [\Phi(x), \sqrt{2c}\,x, c]$ is a feature map

with $K(x,z) = \langle \Psi(x), \Psi(z)\rangle$

- Ψ has $d^2 + d + 1$ output dimensions
- K is still $\Theta(d)$ to compute

$\boxed{3}$ $K(x,z) = (\langle x,z\rangle + c)^k$, $\quad c \geq 0 \quad$ k is an integer

... has a feature map with $d^{O(k)}$ coordinates

but $K(x,z)$ can be computed in $O(d+k)$ time $\quad$ [$O(k)$ time to raise to k^{th} power]

For a weights vector w, the decision boundary $\langle w, \Phi(x)\rangle = 0$ for this kernel would be a degree k polynomial.

We call $\boxed{3}$ the polynomial kernel.

$\boxed{4}$ $K(x,z) = e^{-\frac{\|x-z\|^2}{c^2}}$ is called the Gaussian Kernel

the feature map is related to the Taylor expansion of e^{-y^2}, and is an <u>infinite dimensional feature map</u>.

We could not compute $\Phi(x)$ directly, but we can still perform the perceptron algorithm in this infinite dimensional space thanks to the Kernel trick.

$\boxed{5}$ String Kernels. If we never touch our original data outside of $K(x,z)$, then x and z don't even have to be vectors.

Let s and t be strings using the same alphabet Σ, for any positive integer p, we can define the following Kernel function:

$$K_p(s,t) = \#\text{ of } p\text{-length substrings in common between } s \text{ and } t$$

With this we can use the perceptron algorithm on classifying strings.

What is the feature map?

$\Phi(s)$ has a coordinate for every possible substring of length p.

$$\Phi^{(u)}(s) = \begin{cases} 1 & \text{if } u \text{ is a substring of } s \\ 0 & \text{otherwise} \end{cases}$$

then $\langle \Phi(s), \Phi(t) \rangle = \#$ coordinates nonzero in both $\Phi(s)$ and $\Phi(t)$

With some dynamic programming, $K_p(s,t)$ can be computed somewhat quickly

$\boxed{6}$ Graph Kernels. Let G_1, G_2 be two graphs, and p an integer ≥ 2

$K(G_1, G_2) = \#$ of subgraphs of size p in common between G_1 and G_2

e.g. for $p=3$, possible subgraphs are:

There are many more possibilities for kernels!

Kernelizing the Perceptron

Let's start by just inserting $\Phi(x)$ into the perceptron algorithm

1. Initialize $w_1 = 0$

2. For $t = 1, 2, 3, \ldots$

   If $y_t \langle w_t, \Phi(x_t) \rangle \leq 0$

   then

   $w_{t+1} = w_t + y_t \Phi(x_t)$

   else

   $w_{t+1} = w_t$

These are the two places where we interact with the ~~deature~~ feature vectors.

How can we avoid explicitly working with the potentially intractable feature vectors $\Phi(x)$? How can we handle our weights vector ~~the~~ w also being intractable?

- First, observe that our update rule ensures w is always a linear combination of terms $y_t \Phi(x_t)$, where each term corresponds to a time where a mistake was made

  → If we just keep track of where our mistakes happened, we will have all the info held by w.

- During training and making predictions, we only need w in so far as computing dot products of the form $\langle w_t, \Phi(x) \rangle$

  → If we can compute those dot products, we can fully run our algorithm.

Now let $M = \{i_1, i_2, \dots, i_k\}$ be a sequence storing the indices of all the times so far we made a mistake during training, so that

$$w_t = y_{i_1}\Phi(x_{i_1}) + \dots + y_{i_k}\Phi(x_{i_k}) = \sum_{i \in M} y_i \Phi(x_i)$$

then
$$\langle w_t, \Phi(x) \rangle = \left\langle \sum_{i\in M} y_i \Phi(x_i), \Phi(x) \right\rangle$$
$$= \sum_{i \in M} \langle y_i \Phi(x_i), \Phi(x) \rangle$$
$$= \sum_{i \in M} y_i \langle \Phi(x_i), \Phi(x) \rangle$$
$$= \sum_{i \in M} y_i \underbrace{K(x_i, x)}_{\Phi(\cdot) \text{ no longer appears!}}$$

So we can make predictions using only $K(x,z)$, and never using $\Phi(\cdot)$.

And our update rule is just recording that we made a mistake, adding an index into M.

Kernelized Perceptron

1. Initialize M as empty
2. For $t = 1, 2, \dots$

   If $y_t \sum_{i \in M} y_i K(x_i, x_t) \leq 0$

   then add t to M

   else (do nothing)

Prediction rule

$$f(x) = \text{sign}\left(\sum_{i \in M} y_i K(x_i, x)\right)$$

to make a prediction, we need to store both the set M, and all the training points (x_i, y_i) corresponding to the indices in M

A slightly different approach: If we are doing multiple passes, the same index t might be added to M multiple times. We can save on growing M, and save on recomputing $K(\cdot,\cdot)$, if we instead just count how many times a mistake was made for each training point

1. Init $\alpha_i = 0$ for $i = 1$ to n

2. For pass $= 1, 2, \dots$

   For $i = 1, \dots, n$

   If $y_t \sum_{j=1}^{n} \alpha_j y_j K(x_j, x_i) \leq 0$

   then $\alpha_i \mathrel{+}= 1$

n = size of training data

At first, each training point was used to update the model zero times.

when a mistake is found, increment the counter

Kernel Properties: Not all functions $K(x,z)$ are kernels. We need them to at least act like dot products

These conditions are necessary and sufficient for K to be a kernel:

1] Symmetry: For all x, z, $K(x,z) = K(z,x)$

2] Positive Semi-Definiteness: For any set of points $x^1, \dots, x^m$, we can build a matrix called the kernel matrix:

$$K = \begin{bmatrix} K(x^1,x^1) & \cdots & K(x^1,x^m) \\ \vdots & \ddots & \vdots \\ K(x^m,x^1) & \cdots & K(x^m,x^m) \end{bmatrix} \quad \text{such that} \quad K_{ij} = K(x^i, x^j)$$

This matrix must be PSD, for all sets of points $x^1, \dots, x^m$ and all values of m.

Ways to show a function is a kernel:

1] Find a feature map Φ s.t. $K(x,z) = \langle \Phi(x), \Phi(z) \rangle$ ← shows it acts like a dot product

2] Show both properties, symmetry and PSD, hold. ← usually harder, since PSD property is over all sets of points of any size.

Example $K(x,z) = (\langle x,z\rangle)^2$

we could use the fact we found a feature map earlier, and be done.

Let's try the other way.

Symmetry: holds since $(\langle x,z\rangle)^2 = (\langle z,x\rangle)^2$

PSD: Let $x^1, \dots, x^m$ be any m vectors, and let K be the $m \times m$ kernel matrix. We must show K is PSD. We already have that K is symmetric, so we must show for all m-dim. vectors t, we have $t^T K t \geq 0$

$$t^T K t = \sum_{i=1}^{m}\sum_{j=1}^{m} t_i t_j K_{ij} = \sum_{i=1}^{m}\sum_{j=1}^{m} t_i t_j \langle x^i, x^j\rangle^2$$

$$= \sum_{i=1}^{m}\sum_{j=1}^{m} t_i t_j \left(\sum_{\ell=1}^{d}\sum_{p=1}^{d} x^i_{(\ell)} x^i_{(p)} x^j_{(\ell)} x^j_{(p)}\right)$$

we can just see this as a huge summation, pulling the $\sum_{\ell=1}^{d}\sum_{p=1}^{d}$ to the front

$$= \sum_{\ell=1}^{d}\sum_{p=1}^{d}\sum_{i=1}^{m}\sum_{j=1}^{m} t_i t_j x^i_{(\ell)} x^i_{(p)} x^j_{(\ell)} x^j_{(p)}$$

$$= \sum_{\ell=1}^{d}\sum_{p=1}^{d} \underbrace{\left(\sum_{i=1}^{m} t_i x^i_{(\ell)} x^i_{(p)}\right)\left(\sum_{j=1}^{m} t_j x^j_{(\ell)} x^j_{(p)}\right)}_{\text{the exact same sum, just with different local index variable}}$$

$$= \sum_{\ell=1}^{d}\sum_{p=1}^{d} \left(\sum_{i=1}^{m} t_i x^i_{(\ell)} x^i_{(p)}\right)^2 \geq 0$$

we have shown this holds for any m, any points $x^1, \dots, x^m$, and any vector t. thus the kernel matrix K is always PSD, and therefore we have both sufficient properties: K is a kernel.

How to show K is <u>not</u> a kernel?

Show a counter-example to either property.

<u>Ex.</u> $K(x,z) = -\langle x, z \rangle$ is not a kernel.

Pick any non-zero vector x. We will build a 1-by-1 Kernel matrix.

$$K = \left[K(x,x) \right] = \left[-\langle x,x \rangle \right] = \left[-\|x\|^2 \right]$$

For any 1-dim non-zero vector t, we have

$$t^T K t = -t^2 \|x\|^2 < 0$$

So PSD is violated.

If we wanted to, we could give a concrete counterexample, $m=1$, $x^1=(1,0)$ $K=[-1]$, $t=[1]$, $t^T K t = -1 < 0$

Note: In this case we we able to find a counter-example with just one point x $(m=1)$. Some non-kernels need a 2-by-2 counterexample $(m=2)$.

If you suspect K is not a kernel, you can usually find a small 1-by-1 or 2-by-2 counter example

---

<u>Kernels to Distances:</u> With the ability to act like dot products, kernels can also give distances.

$$D_K(x,z) = \sqrt{K(x,x) + K(z,z) - 2K(x,z)}$$

If K was not PSD, this value in the square root could be negative. Negative (or imaginary) distances would cause problems for our learning algorithms.

If $K(x,z) = \langle \Phi(x), \Phi(z) \rangle$ then

$$D_K^2(x,z) = \langle \Phi(x), \Phi(x) \rangle + \langle \Phi(z), \Phi(z) \rangle - 2\langle \Phi(x), \Phi(z) \rangle$$
$$= \langle \Phi(x) - \Phi(z), \Phi(x) - \Phi(z) \rangle = \|\Phi(x) - \Phi(z)\|^2$$

We can use kernel functions to abstractly compute distances as well as dot products.

Even k-NN can be kernelized.