Learning theory, Kernel Methods

Sriram Sankararaman

The instructor gratefully acknowledges Fei Sha, Ameet Talwalkar, Eric Eaton, and Jessica Wu whose slides are heavily used, and the many others who made their course material freely available online.

Outline

What concept is learnable?

Learning Conjunctions

Learning Conjunctions -- Algorithm

Training data

- <(1,1,1,1,1,1,…,1,1), 1>
- <(1,1,1,0,0,0,…,0,0), 0>
- <(1,0,1,1,1,0,...0,1,1), 1>
- <(1,1,1,1,1,0,...0,0,1), 1>
- <(1,0,1,0,0,0,...0,1,1), 0>
- <(1,1,1,1,1,1,…,0,1), 1>
- <(0,1,0,1,0,0,...0,1,1), 0>

$f = x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

A simple learning algorithm (*Elimination*)

- Discard all negative examples
- Build a conjunction using the features that are common to all positive conjunctions

h= $x_1 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

$x_1, x_3, x_4, x_5, x_{100}$

0, 1, 1, 1, 1

Positive examples ***eliminate*** irrelevant features

Learning Conjunctions: Analysis

Theorem: Suppose we are learning a conjunctive concept with n dimensional Boolean features using m training examples. If

$$m > \frac{n}{\epsilon}\left(\log(n) + \log\left(\frac{1}{\delta}\right)\right)$$

Poly in n, $1/\delta$, $1/\epsilon$

then, with probability > $1 - \delta$, the error of the learned hypothesis $err_D(h)$ will be less than ϵ.

n: # literals

If we see these many training examples, then the algorithm will produce a conjunction that, with high probability, will make few errors

Requirements of Learning

- Cannot expect a learner to learn a concept exactly
  - There will generally be multiple concepts consistent with the available data
  - Unseen examples could *potentially* have any label
  - We may misclassify *uncommon* examples that do not show up in the training set

Requirements of Learning

- Cannot expect a learner to learn a concept exactly
  - There will generally be multiple concepts consistent with the available data
  - Unseen examples could *potentially* have any label
  - We may misclassify *uncommon* examples that do not show up in the training set

- Cannot always expect to learn a close approximation to the target concept
  - Sometimes the training set will not be representative

Probably approximately correctness

- The only realistic expectation of a good learner is that with high probability it will learn a close approximation to the target concept
- In Probably Approximately Correct (PAC) learning, one requires that
  - given small parameters ϵ and δ,
  - With probability at least $1 - \epsilon$, a learner produces a hypothesis with error at most δ
- The reason we can hope for this is the *consistent distribution assumption*

PAC Learnability

Consider a concept class C defined over an instance space X (containing instances of length n), and a learner L using a hypothesis space H

The concept class C is PAC learnable by L using H if for all $f \in C$, for all distribution D over X, and fixed $\epsilon > 0$, $\delta < 1$, given m examples sampled i.i.d. according to D, the algorithm L produces, with probability at least $(1-\delta)$, a hypothesis h ∈ H that has error at most ε, where m is *polynomial* in $1/\varepsilon$, $1/\delta$, n and size(H)

example: conjunction: $m > \frac{n}{\epsilon}\left(\log(n) + \log\left(\frac{1}{\delta}\right)\right)$

efficiently learnability

- The concept class C is *efficiently learnable* if L can produce the hypothesis in time that is polynomial in 1/ ε, 1/ δ, n and size(H)

PAC Learnability

- We impose two limitations
- Polynomial *sample complexity* (information theoretic constraint)
  - Is there enough information in the sample to distinguish a hypothesis h that approximate f ?
- Polynomial *time complexity* (computational complexity)
  - Is there an efficient algorithm that can process the sample and produce a good hypothesis h ?

Worst Case definition: the algorithm must meet its accuracy

  - for every distribution (The distribution free assumption)
  - for every target function f in the class C

Example: Learning Conjunctions

Suppose we are learning a conjunctive concept with n dimensional Boolean features using m training examples. If

$$m > \frac{n}{\epsilon}\left(\log(n) + \log\left(\frac{1}{\delta}\right)\right)$$

then, with probability > 1 - δ, the error of the learned hypothesis $err_D(h)$ will be less than ϵ.

m is *polynomial* in 1/ ε, 1/ δ, n and size(H)

A general result

Let H be any hypothesis space.

With probability $1-\delta$ a hypothesis h that is consistent with a training set of size m will have an error $< \epsilon$ on future examples if

$$m > \frac{1}{\epsilon}\left(\ln(|H|) + \ln\frac{1}{\delta}\right)$$

1. Expecting lower error increases sample complexity (i.e more examples needed for the guarantee)

2. If we have a larger hypothesis space, then we will make learning harder (i.e higher sample complexity)

3. If we want a higher confidence in the classifier we will produce, sample complexity will be higher.

A general result

Let H be any hypothesis space.

With probability 1 -δ a hypothesis h $\rightarrow$ H that is consistent with a training set of size m will have an error < ϵ on future examples if

$$m > \frac{1}{\epsilon}\left(\ln(|H|) + \ln\frac{1}{\delta}\right)$$

It expresses a preference towards smaller hypothesis spaces

Next question: What if size(H) is infinity?

Complicated/larger hypothesis spaces are not necessarily bad. But simpler ones are unlikely to fool us by being consistent with many examples!

Infinite Hypothesis Space

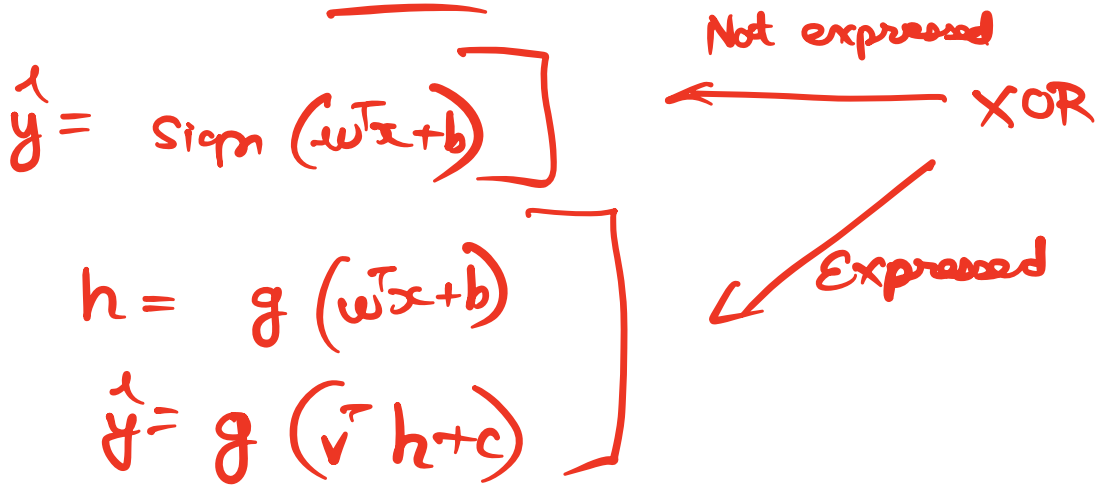
- The previous analysis was restricted to finite hypothesis spaces
- Some infinite hypothesis spaces are more expressive than others
  - Linear threshold function vs. a combination of LTUs
- Need a measure of the expressiveness of an infinite hypothesis space other than its size

$h(x) = f(w^T x + b)$

$\text{sign}(w^T x + b)$

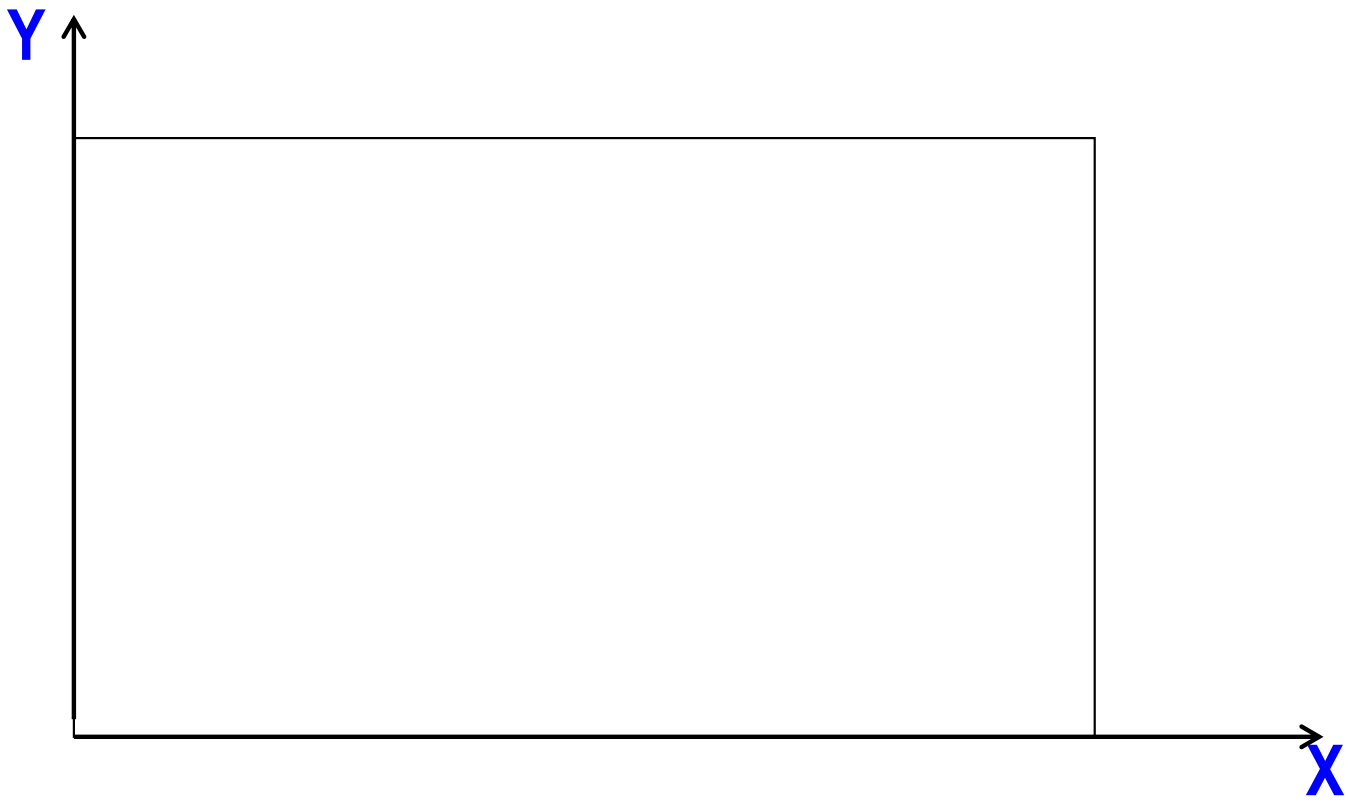
Vapnik-Chervonenkis dimension

- The Vapnik-Chervonenkis dimension (VC dimension) provides such a measure
  - "What is the expressive *capacity* of a set of functions?"
- Analogous to |H|, there are bounds for sample complexity using VC(H)



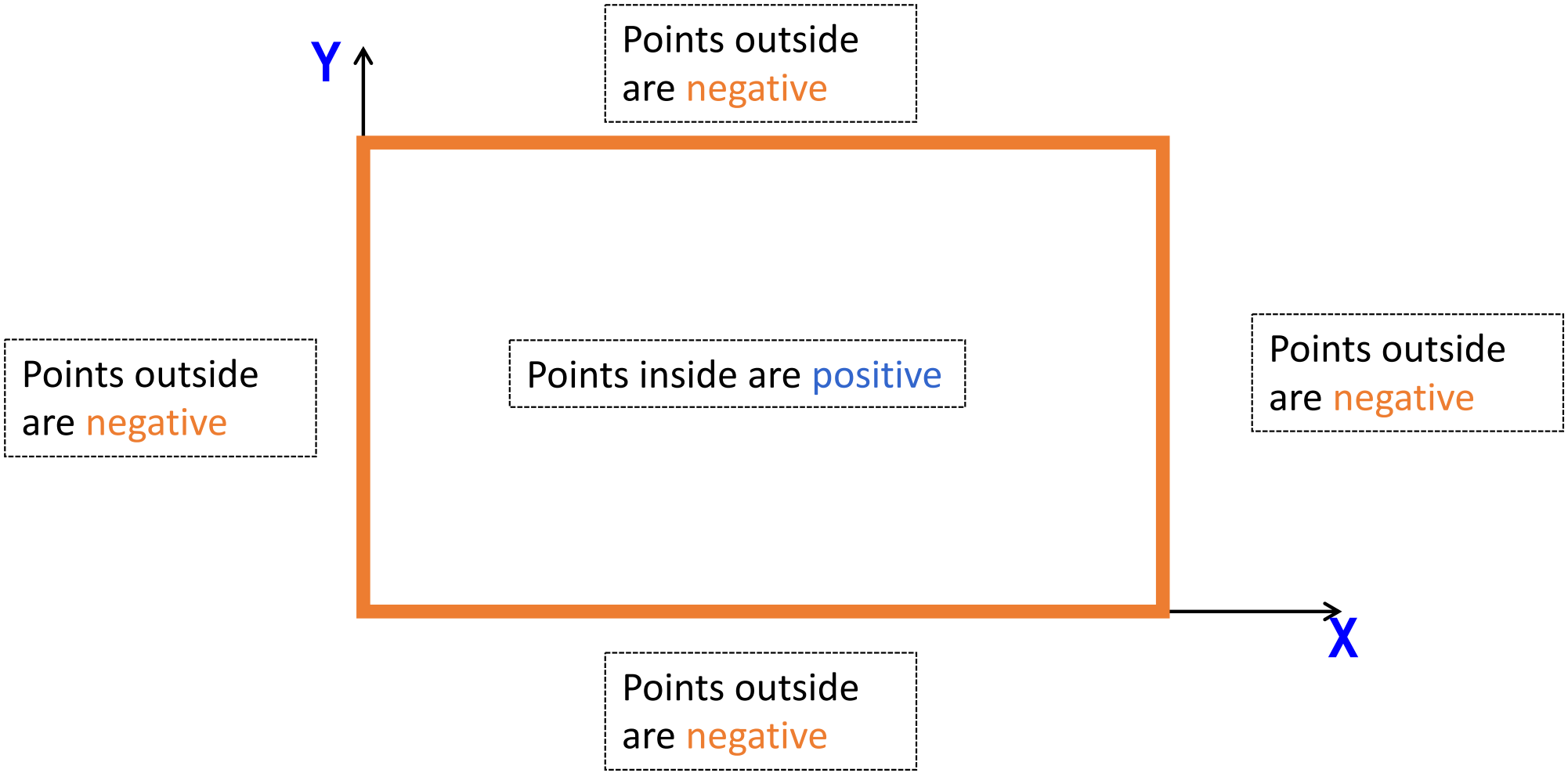
Learning Rectangles

Assume the target concept is an axis parallel rectangle



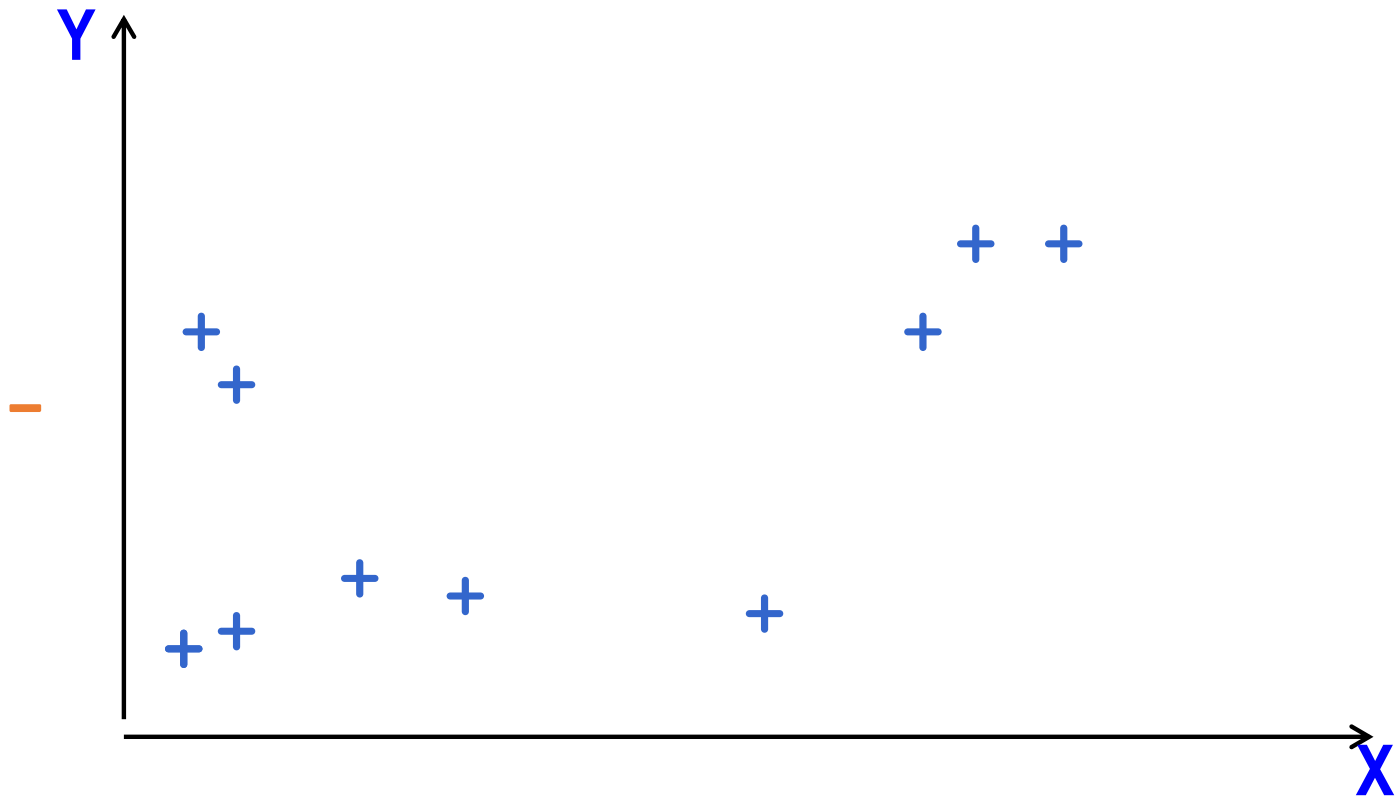
Learning Rectangles

Assume the target concept is an axis parallel rectangle



Learning Rectangles

Assume the target concept is an axis parallel rectangle

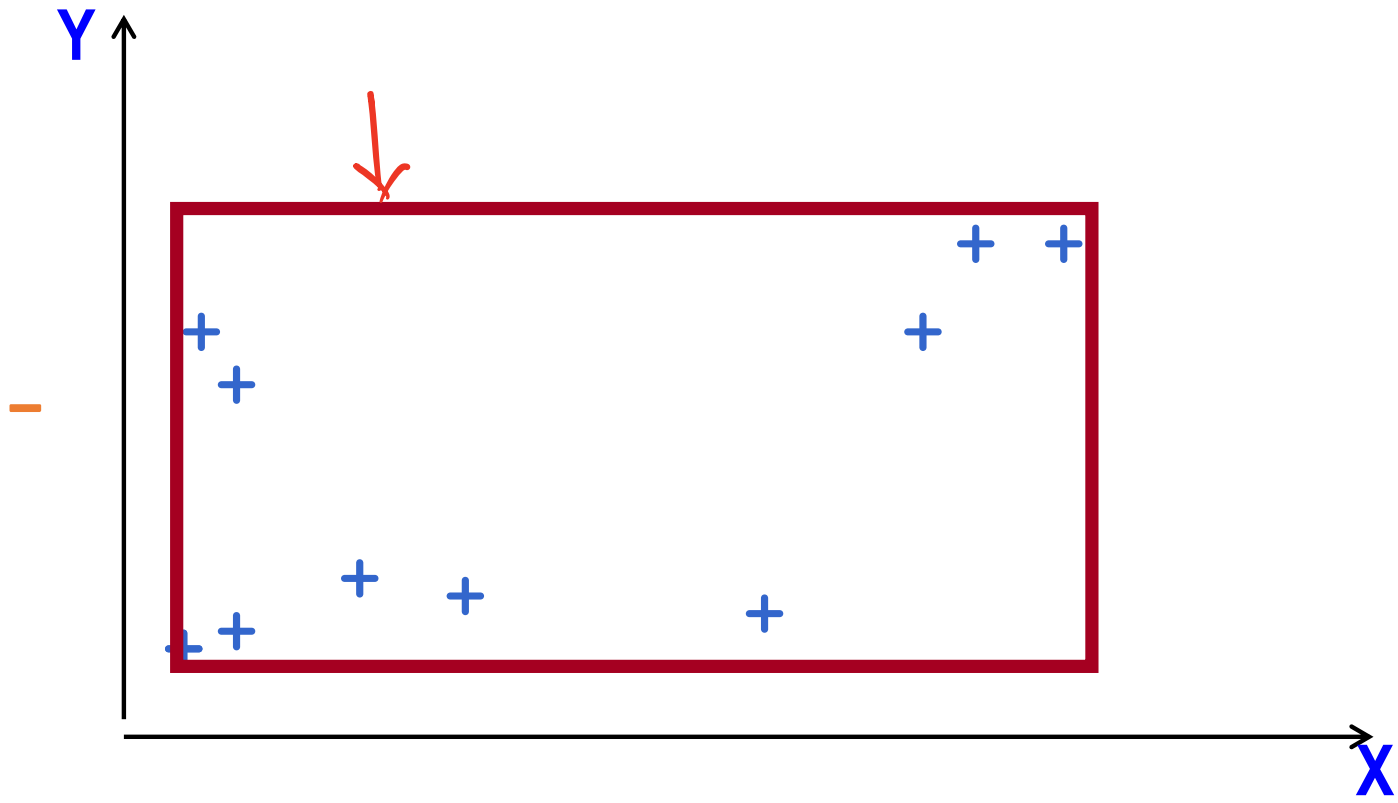


Will we be able to learn the target rectangle?

Can we come close?

Learning Rectangles

Assume the target concept is an axis parallel rectangle

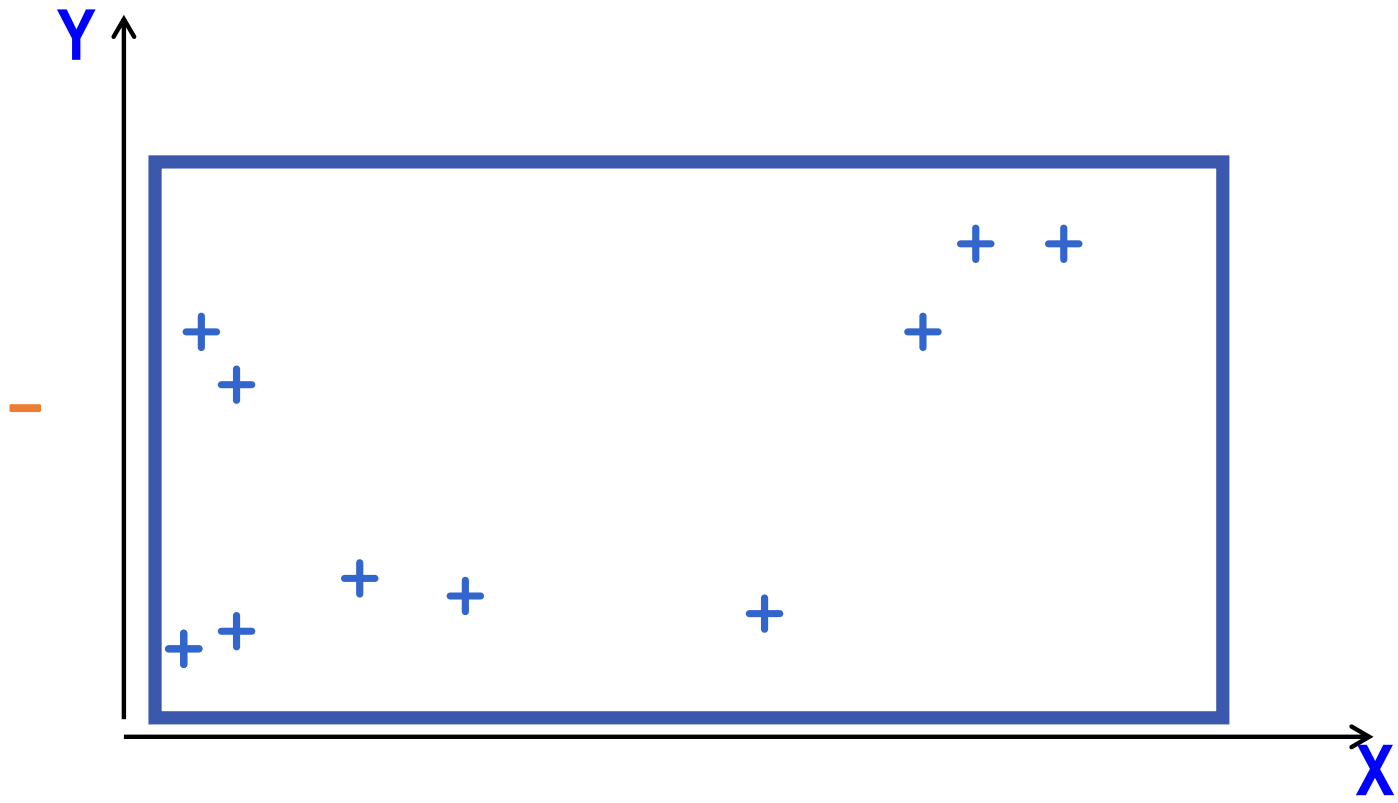


Key observation: Despite there are infinite # hypothesis
The blue & red rectangles have the same predictions

Can we come close?

Learning Rectangles

Assume the target concept is an axis parallel rectangle

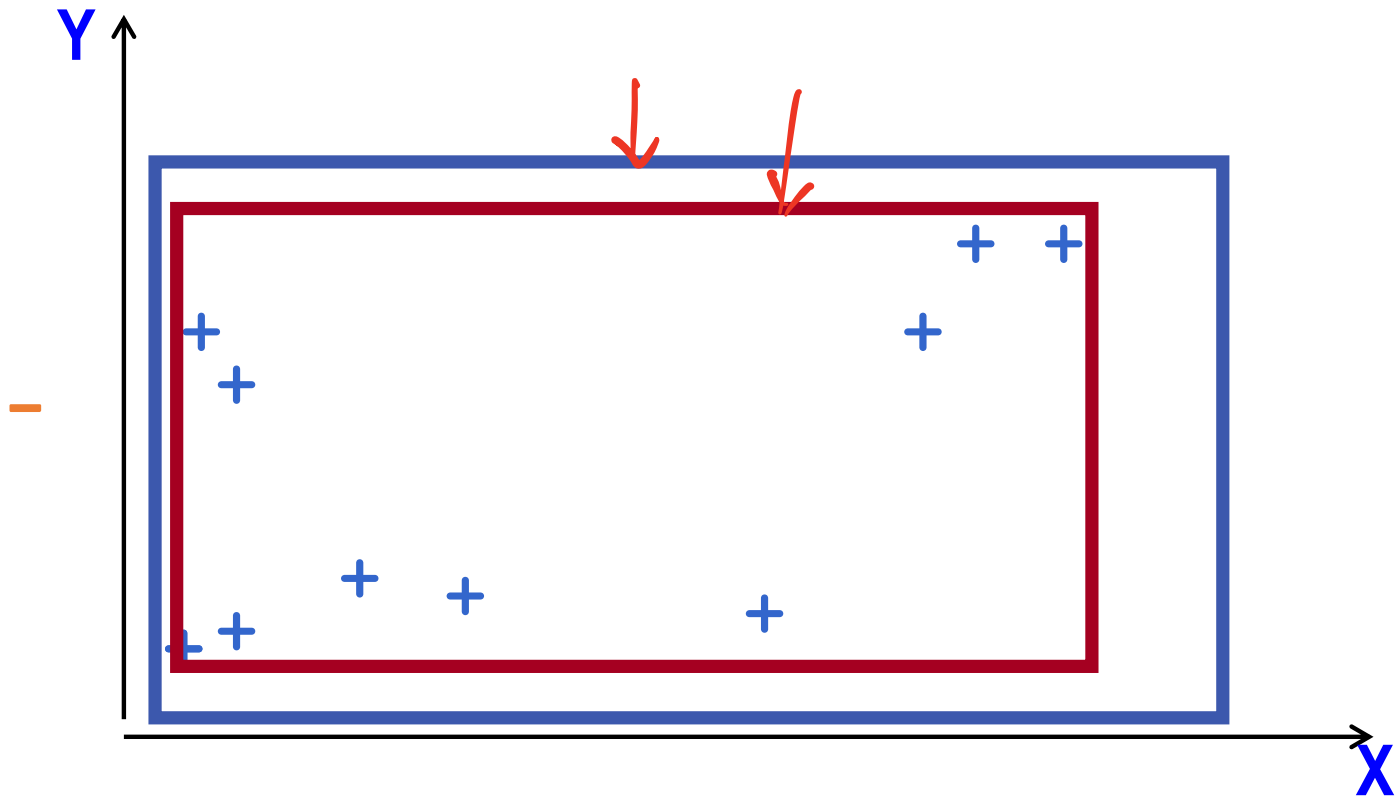


Key observation: Despite there are infinite # hypothesis
The blue & red rectangles have the same predictions

Can we come close?

Learning Rectangles

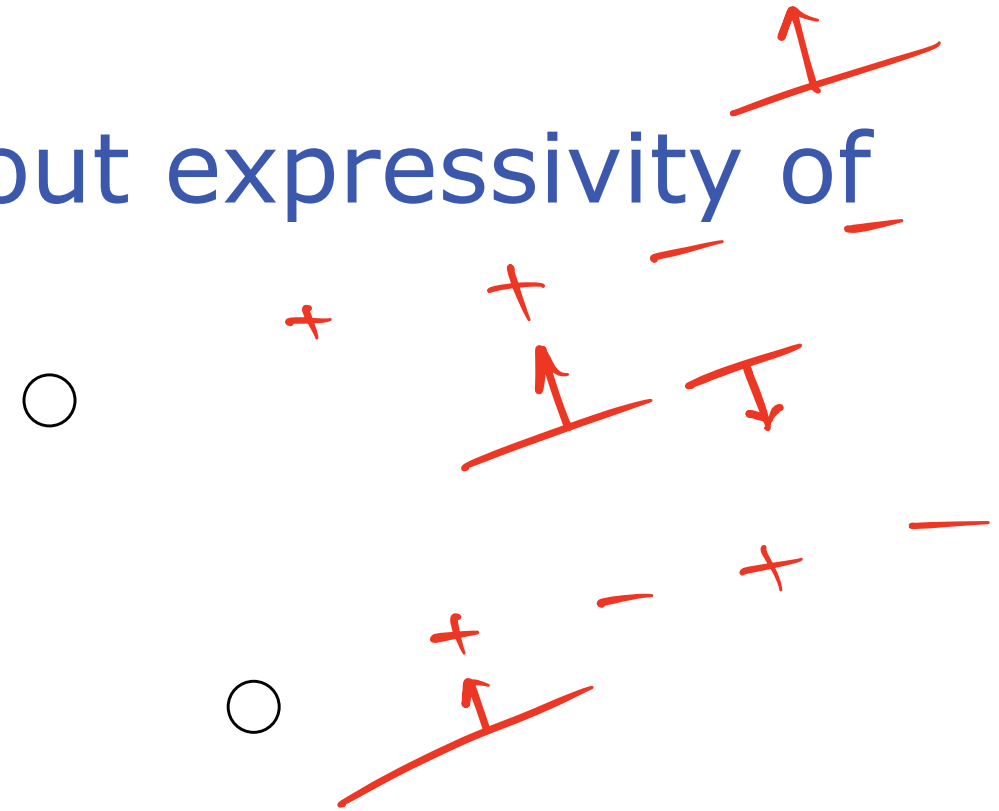
Assume the target concept is an axis parallel rectangle



Key observation: Despite there are infinite # hypothesis
The blue & red rectangles have the same predictions

Can we come close?

Let's think about expressivity of functions

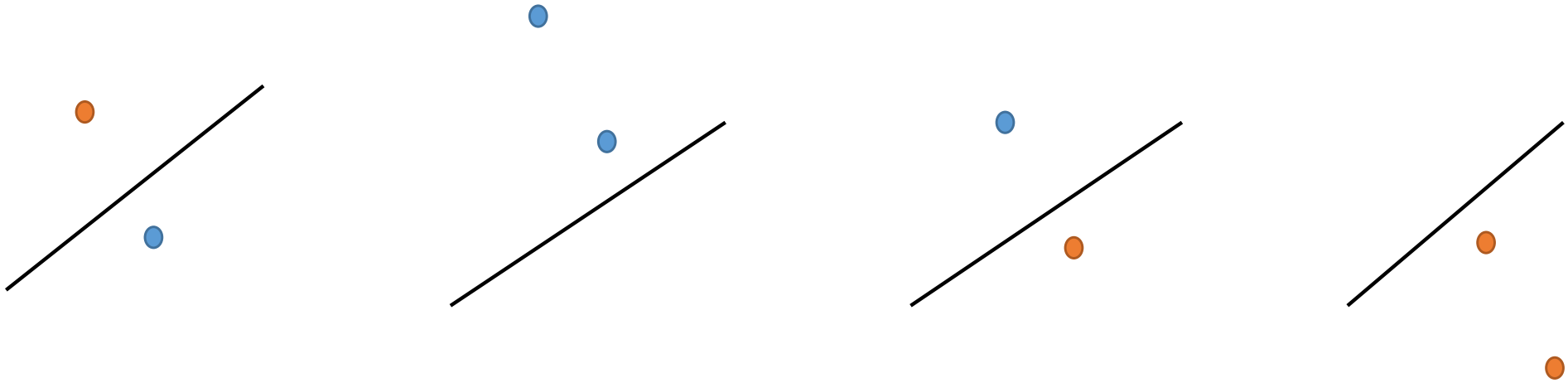


Suppose we have two points.

Can linear classifiers correctly classify any labeling of these points?

Linear functions are expressive enough to ***shatter*** 2 points

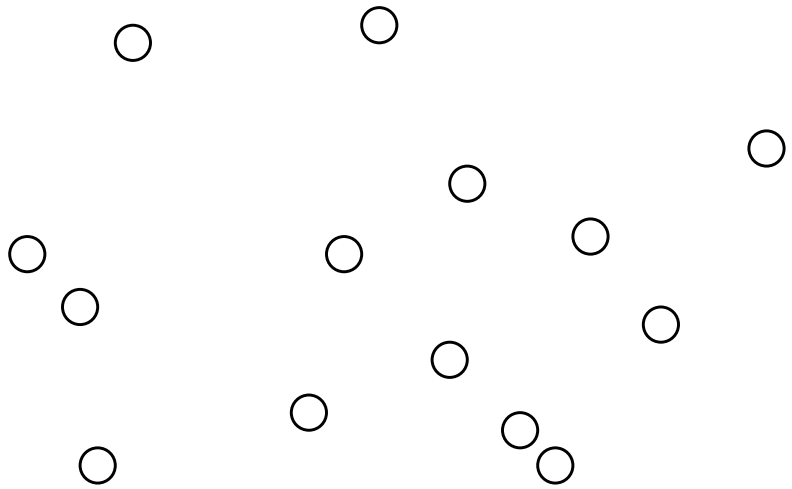
Let's think about expressivity of functions



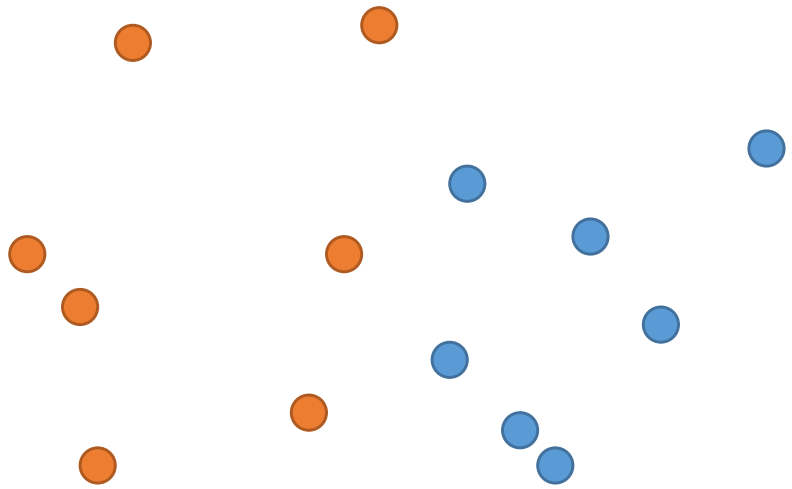
Suppose we have two points.
Can linear classifiers correctly classify any labeling of these points?

Linear functions are expressive enough to ***shatter*** 2 points

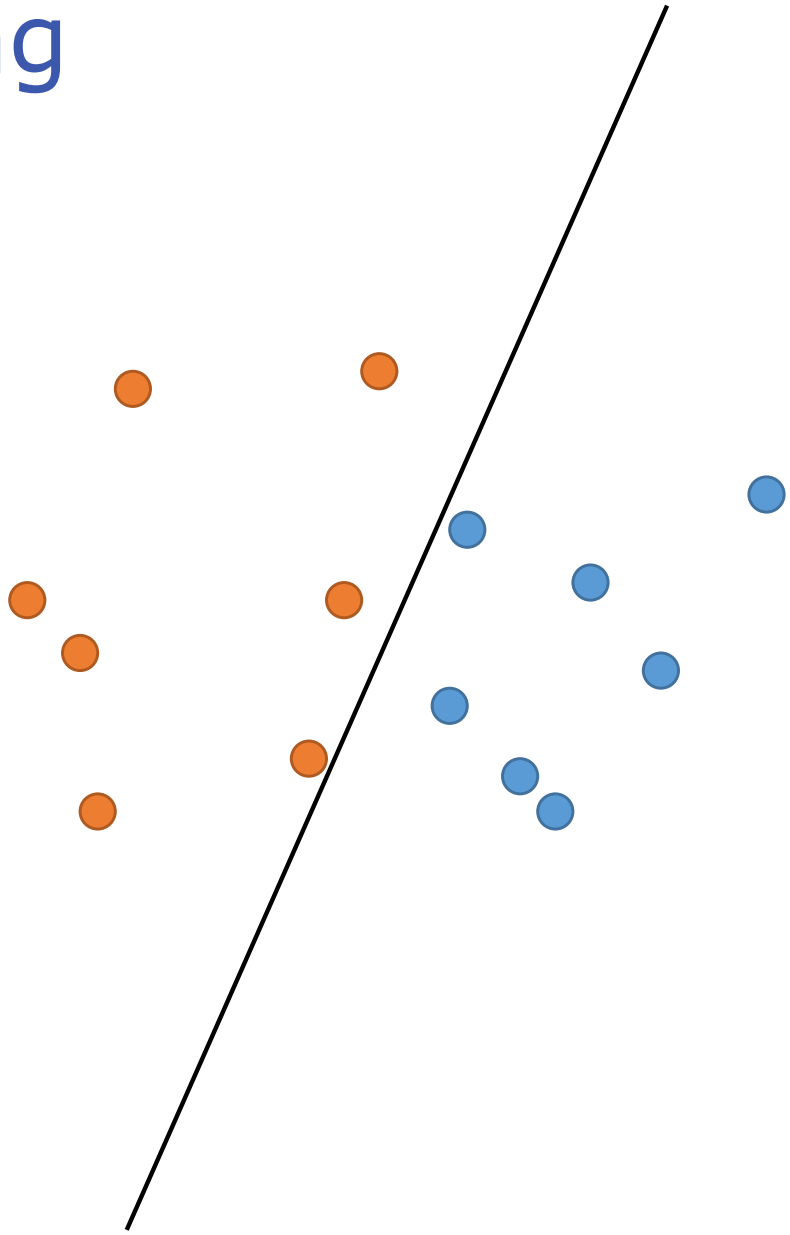
Shattering



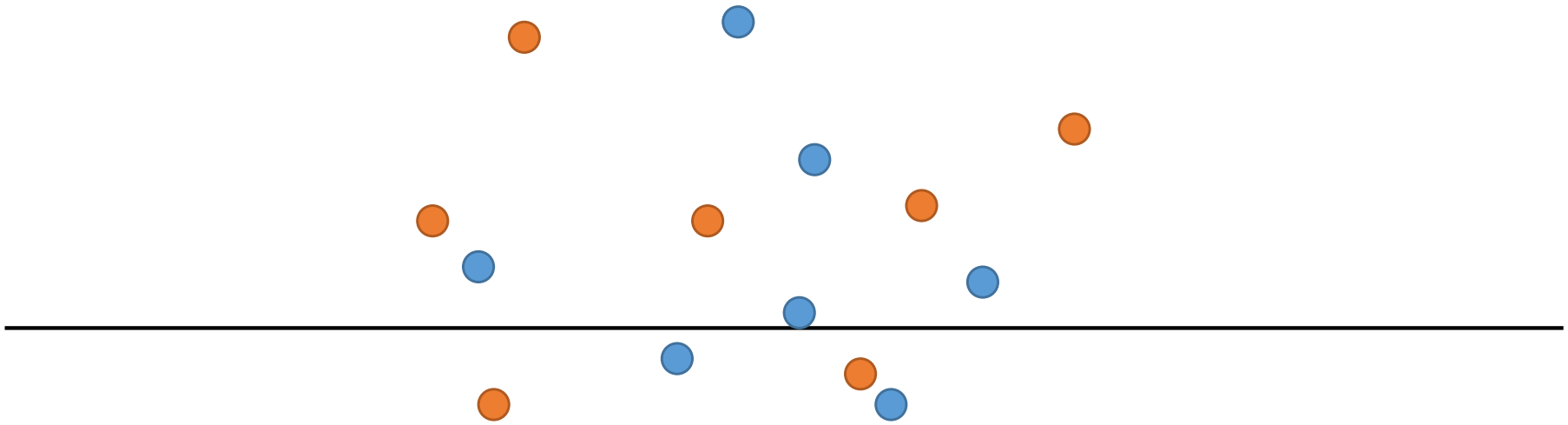
Shattering



Shattering

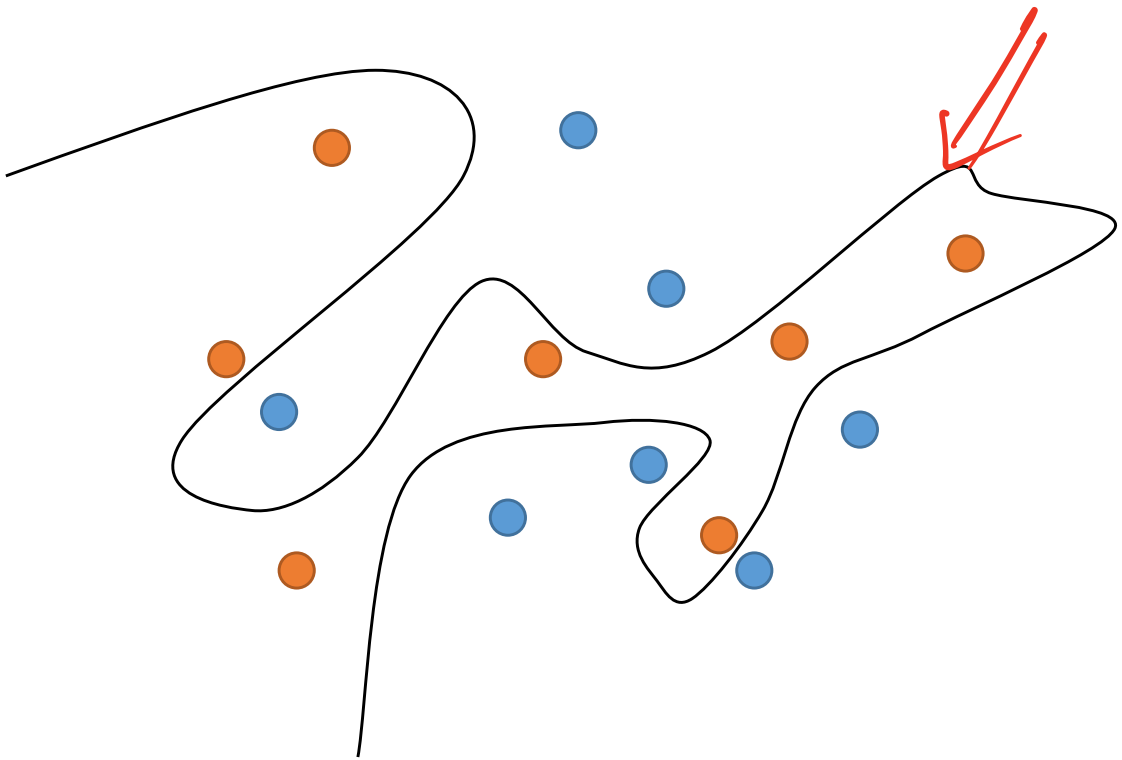


Shattering



This particular labeling of the points can not be separated by *any* line

Shattering



Linear functions are not expressive to shatter fourteen points

Because there is a labeling that can not be separated by them

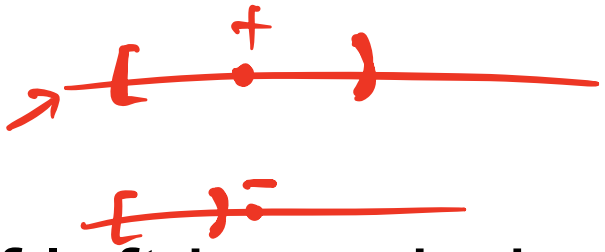
Of course, a more complex function could separate them

Shattering

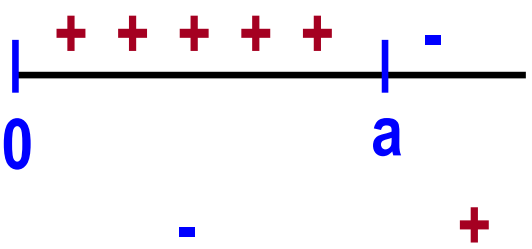
Definition: A *set S of examples* is shattered by a *set of functions* H if for *every* partition of the examples in S into positive and negative examples there is a function in H that gives exactly these labels to the examples

Intuition: A rich set of functions shatters large sets of points

Left bounded intervals



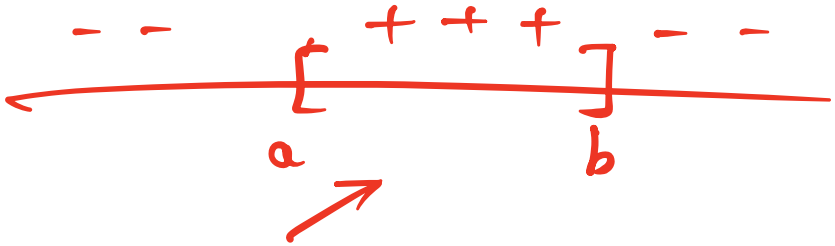
Example 1: Hypothesis class of left bounded intervals on the real axis: [0,a) for some real number a>0



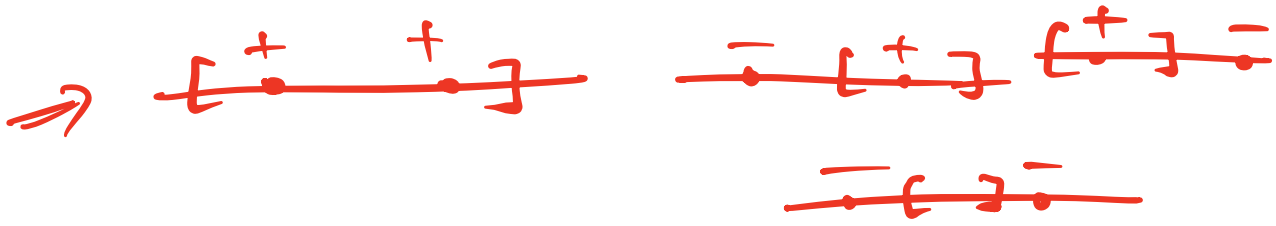
Sets of **two** points cannot be shattered

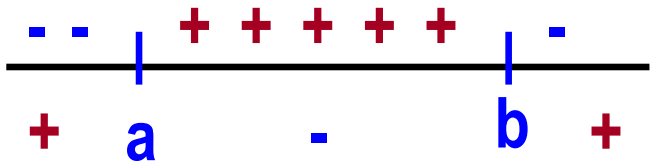
That is: given two points, you can label them in such a way that no concept in this class will be consistent with their labeling

Real intervals



Example 2: Hypothesis class is the set of intervals on the real axis: [a,b],for some real numbers b>a





All sets of one or two points can be shattered

But some sets of three points cannot be shattered

Shattering

Definition: A *set S of examples* is shattered by a *set of functions* H if for *every* partition of the examples in S into positive and negative examples there is a function in H that gives exactly these labels to the examples

Shattering: The adversarial game

You

An adversary

You: Hypothesis class H can shatter these d points

You: Aha! There is a function h ∈ H that correctly predicts your evil labeling

Adversary: That's what you think! Here is a labeling that will defeat you.

Adversary: Argh! You win this round. I will find another one

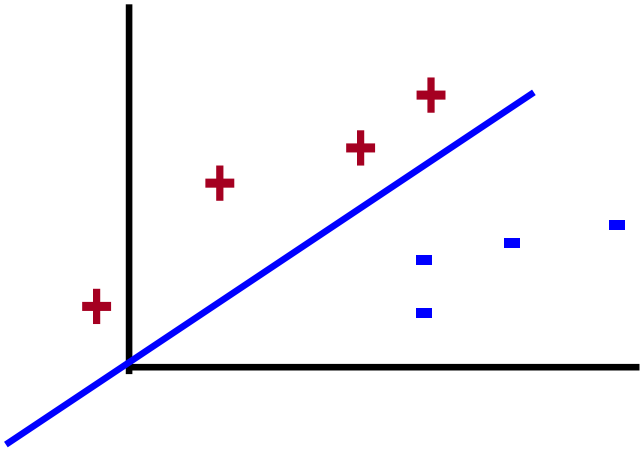
Vapnik-Chervonenkis Dimension

Definition: The VC dimension of hypothesis space H over instance space X is the size of the largest *finite* subset of X that is shattered by H

- If there exists any subset of size d that can be shattered, VC(H) >= d
  - Even one subset will do
- If no subset of size d can be shattered, then VC(H) < d

Example 3: 2-D Half spaces in a plane

Lines



Can one point be shattered?

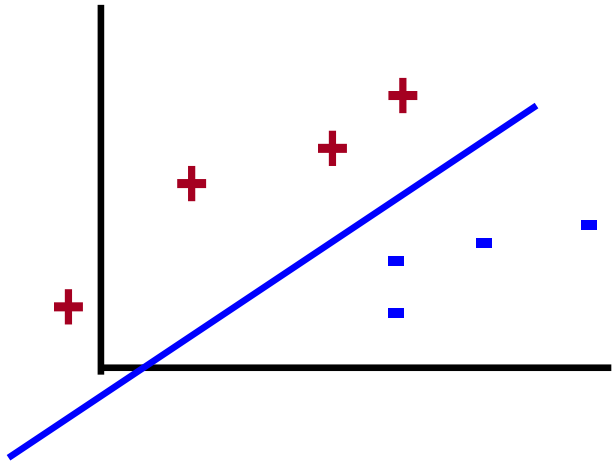
Is there any two points can be shattered?

Is there any three points?
Can any three points be shattered?

$VC(\text{lines}) \geq 3$

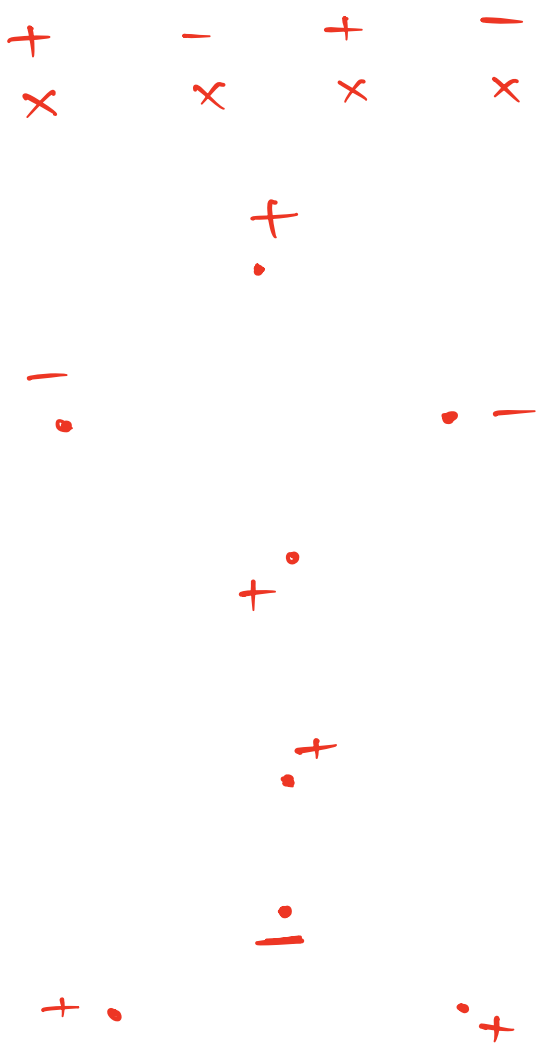
Example Half spaces in a plane

- Prove VC >=1
  - Show any point can be shattered



- Prove VC >=2
  - Show there exists 2 points can be shattered

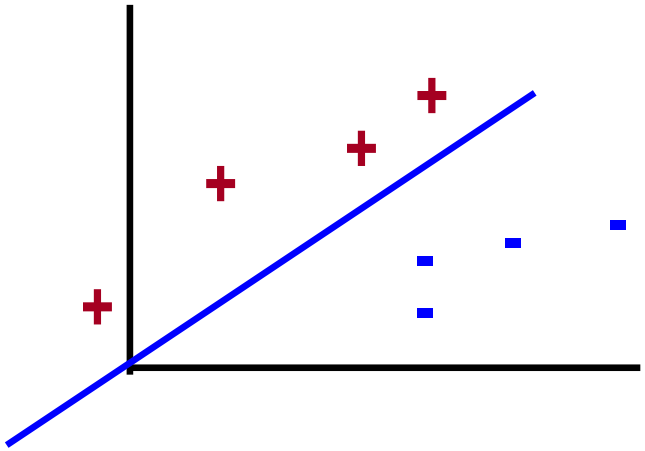
- Prove VC >=3
  - Show there exists 3 points can be shattered



VC (line) < 4

Example Half spaces in a plane

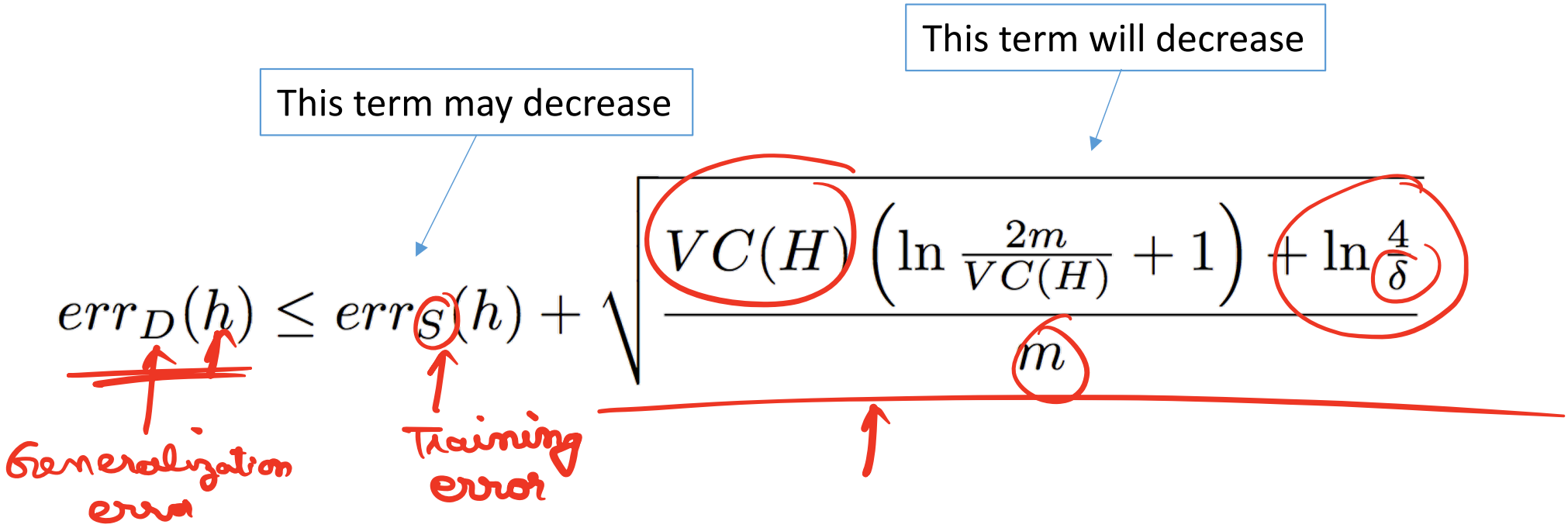
- Prove VC <4
  - Show no 4 points can be shattered
- Therefore, VC = 3



- Suppose three of them lie on the same line, label the outside points + and the inner one –
- Otherwise, make a convex hull. Clockwisely, label them + - + - ;
- Otherwise, one point inside the convex hall of the other three points; label outside + and the inner one –
- Four points cannot be shattered!

VC dimension of Half spaces

- In general, the VC dimension of an n-dimensional linear function is n+1



This formulation will be provided in the exam

Computational Learning Theory

- The Theory of Generalization
  - Using training instance to rule out incorrect hypotheses
- Probably Approximately Correct (PAC) learning
  - How many examples you need to see to obtain a learned function with error $\leq \epsilon$
- Shattering and the VC dimension

Outline

Motivation

- Linear models are convenient.
  - Computationally efficient for learning (training) and prediction.
- We would like our models to be "expressive".
  - If it is not expressive enough, it will underfit.
  - Too expressive models can overfit.

Motivation

How to increase the expressive power of linear models?

- Map the feature vector $\boldsymbol{x}$ to an expanded version $\phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^D \rightarrow \boldsymbol{z} \in \mathbb{R}^M$.
- Use a non-linear basis function $\phi(\boldsymbol{x})$ as input to linear model.

Motivation

How to increase the expressive power of linear models?

- Map the feature vector $\boldsymbol{x}$ to an expanded version $\phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^D \rightarrow \boldsymbol{z} \in \mathbb{R}^M$.
- Use a non-linear basis function $\phi(\boldsymbol{x})$ as input to linear model.
- Difficulty: When M is large, computational difficulty.

Motivation

How to choose nonlinear basis function for regression?

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})$$

where $\boldsymbol{\phi}(\cdot)$ maps the original feature vector $\boldsymbol{x}$ to a M-dimensional *new* feature vector. In the following, we will show that we can sidestep the issue of choosing which $\boldsymbol{\phi}(\cdot)$ to use — instead, we will choose *equivalently* a *kernel function* that are often computationally efficient to work with.

Example

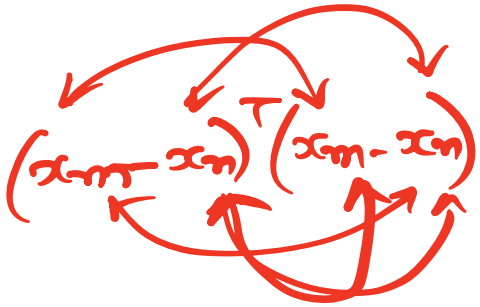
$$\phi(\boldsymbol{x}) = \begin{pmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \vdots \\ \sqrt{2}x_D \\ {x_1}^2 \\ x_1x_2 \\ \vdots \\ x_1x_D \\ x_2x_1 \\ {x_2}^2 \\ \vdots \\ x_2x_D \\ \vdots \\ x_Dx_1 \\ \vdots \\ {x_D}^2 \end{pmatrix}$$

$x = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix}$

$O(D^2)$

Outline

Kernelized nearest neighbors



In nearest neighbor classification, the most important quantity to compute is the (squared) distance between two data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$

$$d(\boldsymbol{x}_m, \boldsymbol{x}_n) = \|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 = \boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_m + \boldsymbol{x}_n^{\mathrm{T}}\boldsymbol{x}_n - 2\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n$$

To perform classification in a transformed feature space, we only need to compute distances in this space:

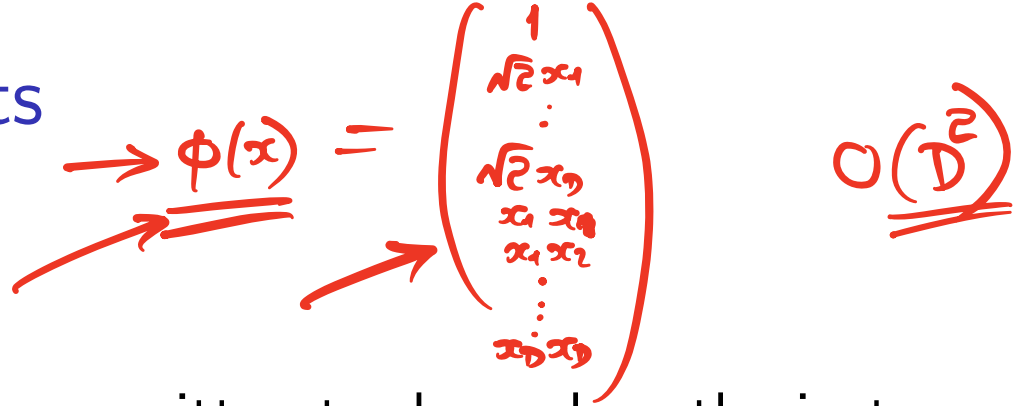
$$d(\boldsymbol{\phi}(\boldsymbol{x}_m), \boldsymbol{\phi}(\boldsymbol{x}_n)) = \|\boldsymbol{\phi}(\boldsymbol{x}_m) - \boldsymbol{\phi}(\boldsymbol{x}_n)\|_2^2 = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_m) + \boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n) - 2\boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)$$

Computing inner products

Many learning algorithms can be rewritten to depend on the instances $\boldsymbol{x}_i, \boldsymbol{x}_j$ only through inner products $\phi(\boldsymbol{x}_i)^{\mathrm{T}}\phi(\boldsymbol{x}_j)$.

Why is this helpful ?

Computing inner products



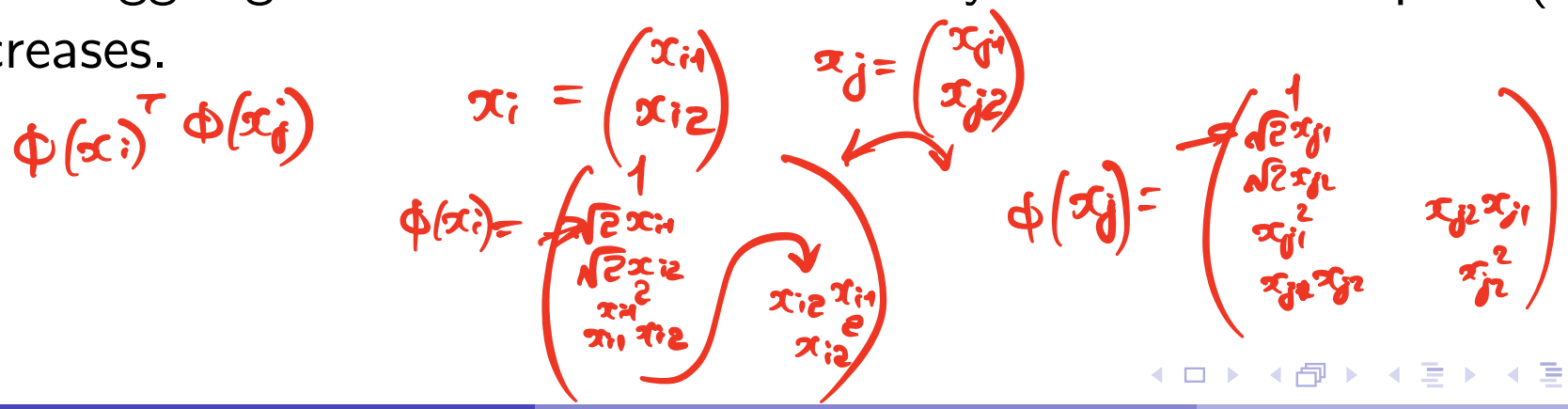
Many learning algorithms can be rewritten to depend on the instances $\boldsymbol{x}_i, \boldsymbol{x}_j$ only through inner products $\phi(\boldsymbol{x}_i)^{\mathrm{T}}\phi(\boldsymbol{x}_j)$.

Why is this helpful ? Computing $\phi(\boldsymbol{x})$ is $\mathrm{O}(D^2)$.

$$\phi(\boldsymbol{x}_i)^{\mathrm{T}}\phi(\boldsymbol{x}_j) = (1 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j)^2$$

However, inner product can be computed in $\mathrm{O}(D)$.

Even bigger gains when the dimensionality of the feature space (M) increases.

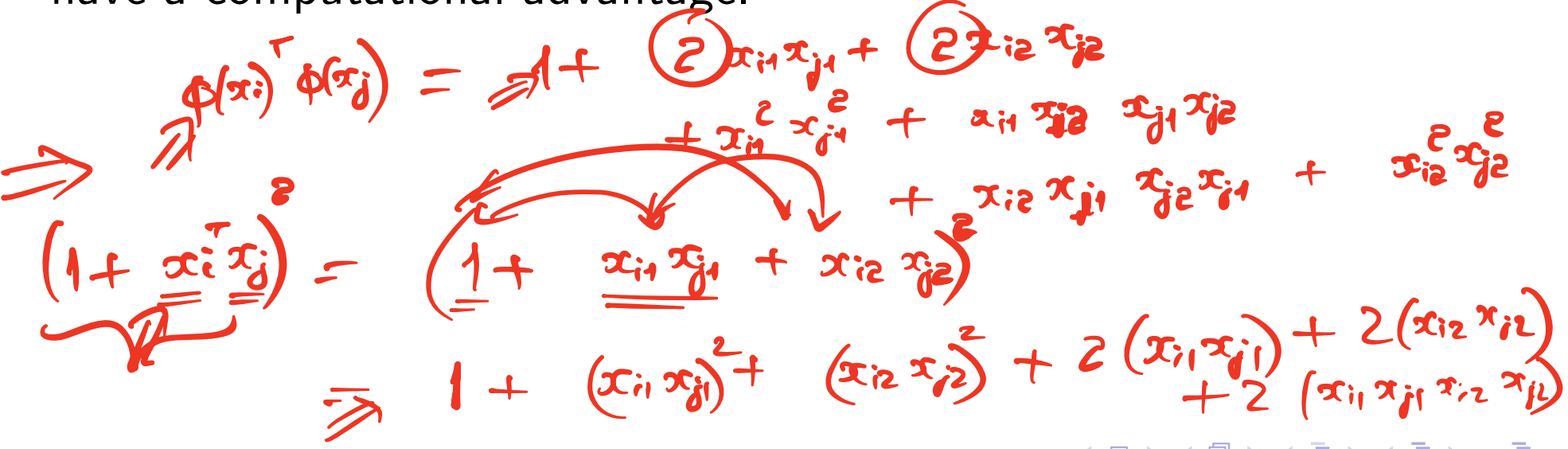


Computing inner products

Let us define a function that computes inner products in the transformed feature space

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_j)$$

If we can compute the value of k without explicitly computing ϕ, we will have a computational advantage.



Kernelized nearest neighbors

We replace all the inner products in the distance with a kernel function $k(\cdot, \cdot)$, arriving at the kernel distance

$$d^{\text{KERNEL}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_m, \boldsymbol{x}_m) + k(\boldsymbol{x}_n, \boldsymbol{x}_n) - 2k(\boldsymbol{x}_m, \boldsymbol{x}_n)$$

The distance is equivalent to compute the distance between $\boldsymbol{\phi}(\boldsymbol{x}_m)$ and $\boldsymbol{\phi}(\boldsymbol{x}_n)$

$$d^{\text{KERNEL}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = d(\boldsymbol{\phi}(\boldsymbol{x}_m), \boldsymbol{\phi}(\boldsymbol{x}_n))$$

where the $\boldsymbol{\phi}(\cdot)$ is the nonlinear mapping function implied by the kernel function. The nearest neighbor of a point $\boldsymbol{x}$ is thus found with

$$\arg\min_n d^{\text{KERNEL}}(\boldsymbol{x}, \boldsymbol{x}_n)$$

Outline

Inner products between features

Let us examine more closely the inner products $\phi(\boldsymbol{x}_m)^{\mathrm{T}}\phi(\boldsymbol{x}_n)$ for a pair of data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$.

Polynomial-based nonlinear basis functions consider the following $\boldsymbol{\phi}(\boldsymbol{x})$:

$$\boldsymbol{\phi} : \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \boldsymbol{\phi}(\boldsymbol{x}) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Inner products between features

Let us examine more closely the inner products $\boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)$ for a pair of data points $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$.

Polynomial-based nonlinear basis functions consider the following $\boldsymbol{\phi}(\boldsymbol{x})$:

$$\boldsymbol{\phi} : \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \boldsymbol{\phi}(\boldsymbol{x}) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

This gives rise to an inner product in a special form,

$$\begin{aligned} \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n) &= x_{m1}^2x_{n1}^2 + 2x_{m1}x_{m2}x_{n1}x_{n2} + x_{m2}^2x_{n2}^2 \\ &= (x_{m1}x_{n1} + x_{m2}x_{n2})^2 = (\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n)^2 \end{aligned}$$

Namely, the inner product can be computed by a function $(\boldsymbol{x}_m^{\mathrm{T}}\boldsymbol{x}_n)^2$ defined in terms of the original features, *without computing $\boldsymbol{\phi}(\cdot)$.*

Kernel functions

Some intuition

- To use kernelized nearest neighbors, we need to be able to compute an inner product between a test point and any training point.
- Since we need this for any possible pair of points, we need a function that takes a pair of points and computes an inner product.
- This is the kernel function $k(\cdot, \cdot)$.
- Given two inputs, $k(\cdot, \cdot)$ tells us how "similar" or "close" these inputs are in the space defined by the function ϕ.

Common kernel functions

Polynomial kernel function with degree of d

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n + c)^d$$

for $\boldsymbol{x}_m, \boldsymbol{x}_n \in \mathbb{R}^D$, $c \geq 0$ and d is a positive integer.

Common kernel functions

Polynomial kernel function with degree of d

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n + c)^d$$

for $\boldsymbol{x}_m, \boldsymbol{x}_n \in \mathbb{R}^D$, $c \geq 0$ and d is a positive integer.

Gaussian kernel, RBF kernel, or Gaussian RBF kernel

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = e^{-\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 / 2\sigma^2}$$

- Only depends on difference between two inputs
- Corresponds to a feature space with *infinite* dimensions (but we can work directly with the original features)!

Common kernel functions

Polynomial kernel function with degree of d

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n + c)^d$$

for $\boldsymbol{x}_m, \boldsymbol{x}_n \in \mathbb{R}^D$, $c \geq 0$ and d is a positive integer.

Gaussian kernel, RBF kernel, or Gaussian RBF kernel

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = e^{-\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 / 2\sigma^2}$$

- Only depends on difference between two inputs
- Corresponds to a feature space with *infinite* dimensions (but we can work directly with the original features)!

These kernels have hyperparameters to be tuned: d, c, σ^2

Kernel functions

Definition: a kernel function $k(\cdot, \cdot)$ is a bivariate function that satisfies the following properties. For any $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$,

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_n, \boldsymbol{x}_m) \text{ and } k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)$$

for *some* function $\boldsymbol{\phi}(\cdot)$.

Not very useful though

Kernel functions

Definition: a kernel function $k(\cdot, \cdot)$ is a bivariate function that satisfies the following properties. For any $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$,

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_n, \boldsymbol{x}_m) \text{ and } k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)$$

for *some* function $\boldsymbol{\phi}(\cdot)$.

Not very useful though

Examples we have seen

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n)^2$$

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = e^{-\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 / 2\sigma^2}$$

Kernel functions

Definition: a kernel function $k(\cdot, \cdot)$ is a bivariate function that satisfies the following properties. For any $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$,

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = k(\boldsymbol{x}_n, \boldsymbol{x}_m) \text{ and } k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)$$

for *some* function $\boldsymbol{\phi}(\cdot)$.

Not very useful though

Examples we have seen

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m^{\mathrm{T}} \boldsymbol{x}_n)^2$$
$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = e^{-\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2 / 2\sigma^2}$$

Example that is not a kernel

$$k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2$$

Conditions for being a positive semidefinite kernel function

Mercer theorem (loosely), a bivariate function $k(\cdot, \cdot)$ is a kernel function, if and only if, for *any N and any $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., and $\boldsymbol{x}_N$*, the matrix

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & k(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1) & k(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots & \vdots & \vdots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & k(\boldsymbol{x}_N, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix}$$

is positive semidefinite.

Why $\|\boldsymbol{x}_m - \boldsymbol{x}_n\|_2^2$ is not a positive semidefinite kernel?

Use the definition of positive semidefinite kernel function. We choose $N = 2$, and compute the matrix

$$\boldsymbol{K} = \begin{pmatrix} 0 & \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 \\ \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 & 0 \end{pmatrix}$$

This matrix cannot be positive semidefinite as it has both *negative* and positive eigenvalues.

Recap: why use kernel functions?

Without specifying $\phi(\cdot)$, the kernel matrix

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & k(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1) & k(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots & \vdots & \vdots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & k(\boldsymbol{x}_N, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix}$$

is exactly the same as

$$\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$$

$$= \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \end{pmatrix}$$

'Kernel trick'

Many learning methods depend on computing *inner products* between features — we have seen the example of regularized least squares. For those methods, we can use a kernel function in the place of the inner products, i.e., *"kernelizing"* the methods, thus, introducing nonlinear features.

When we talk about support vector machines, we will see the trick one more time.

There are infinite numbers of kernels to use!

Rules of composing kernels (this is just a partial list)

- if $k(\boldsymbol{x}_m, \boldsymbol{x}_n)$ is a kernel, then $ck(\boldsymbol{x}_m, \boldsymbol{x}_n)$ is also if $c > 0$.
- if both $k_1(\boldsymbol{x}_m, \boldsymbol{x}_n)$ and $k_2(\boldsymbol{x}_m, \boldsymbol{x}_n)$ are kernels, then $\alpha k_1(\boldsymbol{x}_m, \boldsymbol{x}_n) + \beta k_2(\boldsymbol{x}_m, \boldsymbol{x}_n)$ are also if $\alpha, \beta \geq 0$
- if both $k_1(\boldsymbol{x}_m, \boldsymbol{x}_n)$ and $k_2(\boldsymbol{x}_m, \boldsymbol{x}_n)$ are kernels, then $k_1(\boldsymbol{x}_m, \boldsymbol{x}_n)k_2(\boldsymbol{x}_m, \boldsymbol{x}_n)$ are also.
- if $k(\boldsymbol{x}_m, \boldsymbol{x}_n)$ is a kernel, then $e^{k(\boldsymbol{x}_m, \boldsymbol{x}_n)}$ is also.
- ...

In practice, choosing an appropriate kernel is an "art"

People typically start with polynomial and Gaussian RBF kernels or incorporate domain knowledge.

Outline

Kernelized perceptron

Algorithm 1 PerceptronTrain $(\mathcal{D}, MaxIter)$

$\boldsymbol{w} \leftarrow 0$
for $iter = 1 \ldots MaxIter$ **do**
  for all $(\boldsymbol{x}, y) \in \mathcal{D}$ **do**
    $a \leftarrow \boldsymbol{w}^{\mathrm{T}} \phi(\boldsymbol{x})$
    if $ya \leq 0$ **then**
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + y\phi(\boldsymbol{x})$
    end if
  end for
end for
return $\boldsymbol{w}$

Prediction: $\hat{y} = sign(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}))$.

How to kernelize?

- At any iteration, $\boldsymbol{w} = \sum_n \alpha_n \phi(\boldsymbol{x}_n)$
- Here α_n is the number of mistakes made on example n.
- What will the prediction on a new sample $\boldsymbol{x}$ be ?
- The activation a can then be computed as

$$\begin{aligned} a &= \boldsymbol{w}^{\mathrm{T}} \phi(\boldsymbol{x}) \\ &= (\sum_n \alpha_n \phi(\boldsymbol{x}_n))^{\mathrm{T}} \phi(\boldsymbol{x}) \\ &= \sum_n \alpha_n \phi(\boldsymbol{x}_n)^{\mathrm{T}} \phi(\boldsymbol{x}) \end{aligned}$$

We can kernelize the prediction so that:

$$\hat{y} = \sum_n \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$$

Kernelized perceptron

Algorithm 2 KernelizedPerceptronTrain $(\mathcal{D}, MaxIter)$

$\boldsymbol{\alpha} \leftarrow 0$
for $iter = 1 \dots MaxIter$ **do**
  for all $(\boldsymbol{x}, y) \in \mathcal{D}$ **do**
    $a \leftarrow \sum_n \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$
    if $ya \leq 0$ **then**
      $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + y$
    end if
  end for
end for
return $(\boldsymbol{\alpha})$

Summary

- Kernels allow us to design algorithms that use rich set of features while being computationally efficient.
- Many machine learning algorithms can be "kernelized".
- Picking kernels is an art.
- We still need to tune hyperparameters.