CSE 446
Kernel Methods; Bootstrap

Natasha Jaques

CSE 446 2025 spring topics overview

https://courses.cs.washington.edu/courses/cse446/25sp/schedule/

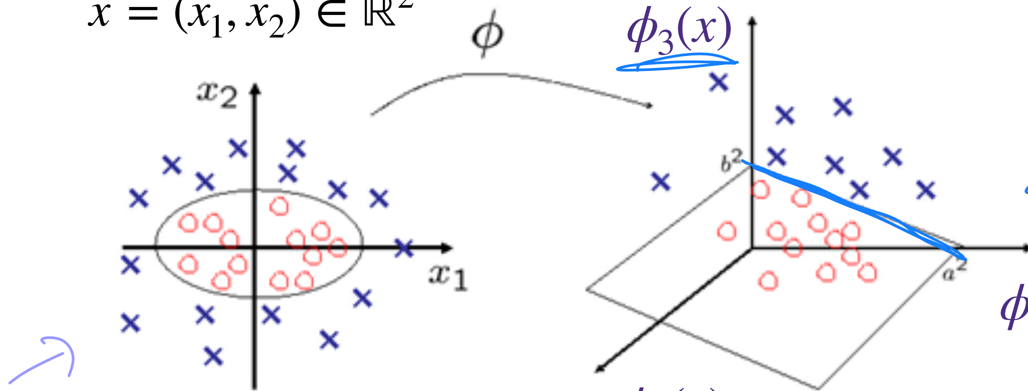
- Supervised learning
  - Linear models (lecture topics 1-9)
    - Linear regression
    - Ridge regression
    - LASSO regression
    - Logistic regression
  - **Non-linear models (lecture topics 10-13)**
    - **Kernel methods**
    - Neural Networks
    - Non-parametric methods
- Unsupervised learning (lecture topics 14-16)
  - PCA/SVD
  - Clustering
- Advanced topics (lecture topics 18-19)

Kernel methods

Lifting to very large dimensional features

What if the data is not linearly separable?

- Use features, for example,



This data is not linearly separable

circle
$\frac{x_1^2}{a^2} + \frac{x_2^2}{b^2} \leq 1$

$\frac{\phi_1(x)}{a^2} + \frac{\phi_2(x)}{b^2} \leq 1$

// linear in the new feature space

Can you suggest some features $\phi_1(x_1, x_2), \phi_2(x_1, x_2), \phi_3(x_1, x_2)$ such that this data is linearly separable in this 3-dimensional space?

feature engineering

$\phi_1(x_1, x_2) = x_1^2$

$\phi_2(x_1, x_2) =$ don't care

$\phi_3(x_1, x_2) = x_2^2$

want high dimensions to separate the data

∞ d ??

What if the data is not linearly separable?

- Generally, **high dimensional feature spaces make it easier to linearly separate different classes**
- However, hard to know which feature map will work for given data
- So, **why not just use super high-dimensional features** and hope that the algorithm will automatically pick the right ones?

↳ unwieldy, computational $$

↳ projecting d feats to p feats if p is very large,

overfitting

$p > n > d$

↳ rank deficient not invertible

Creating Features

- Feature mapping $\phi : \mathbb{R}^d \to \mathbb{R}^p$ maps original data into a rich and high-dimensional feature space (usually $d \ll p$)

For example, in d=1, one can use

$$\phi(x) = \begin{bmatrix} \phi_1(x) \\ \phi_2(x) \\ \vdots \\ \phi_k(x) \end{bmatrix} = \begin{bmatrix} x \\ x^2 \\ \vdots \\ x^k \end{bmatrix}$$

For example, for d>1,
one can generate vectors $\{u_j\}_{j=1}^{p} \subset \mathbb{R}^d$

and define features:

$$\phi_j(x) = \cos(u_j^T x)$$

$$\phi_j(x) = (u_j^T x)^2$$

$$\phi_j(x) = \frac{1}{1 + \exp(u_j^T x)}$$

Creating Features

- Feature space can get really large really quickly!

- How many coefficients/parameters are there for degree-k polynomials for $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$?

- At a first glance, it seems inevitable that we need memory (to store the features $\{\phi(x_i) \in \mathbb{R}^p\}_{i=1}^n$) and run-time that increases with p where $d < n \ll p$

How do we deal with high-dimensional lifts/data?

A fundamental trick in ML: use kernels

defn

A function $K : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ is a ***kernel*** for a map $\phi(\cdot)$
if $K(x, x') \triangleq \phi(x)^T\phi(x') = \phi(x) \cdot \phi(x') = \langle \phi(x), \phi(x') \rangle$

This notation is for dot product (which is the same as inner product)

→ mean the same thing

- Main idea: computing inner products can be much more efficient than explicitly commuting the (very high-dimensional) features

↳ compute kernel $K(x, x')$ not the ϕ

↳ bypass operating on $\phi \in \mathbb{R}^p$

$K(x, x') = \|x - x'\|^2$

$K(x, x') = x^T x'$

How do we deal with high-dimensional lifts/data?

A fundamental trick in ML: use kernels

- So, if we can represent our
  - training algorithms and
  - decision rules for prediction
- as functions of dot products of feature maps (i.e. $\{\phi(x) \cdot \phi(x')\}$)

  and if we can find a **kernel** for our feature map such that

$$K(x, x') = \phi(x)^T \phi(x')$$

then we can avoid explicitly computing and storing (high-dimensional) $\{\phi(x_i)\}_{i=1}^n$ and instead only work with the kernel matrix of the training data $\{K(x_i, x_j)\}_{i,j \in \{1,\ldots,n\}}$

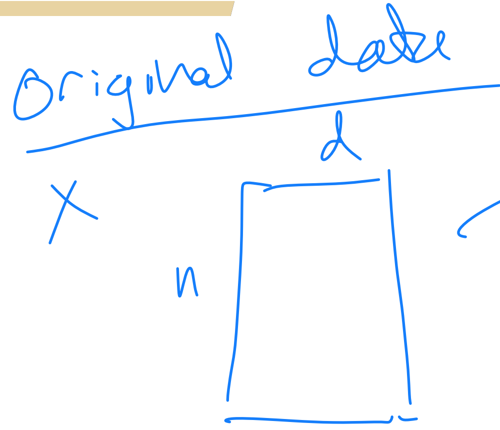
Kernels are much more efficient to compute than features

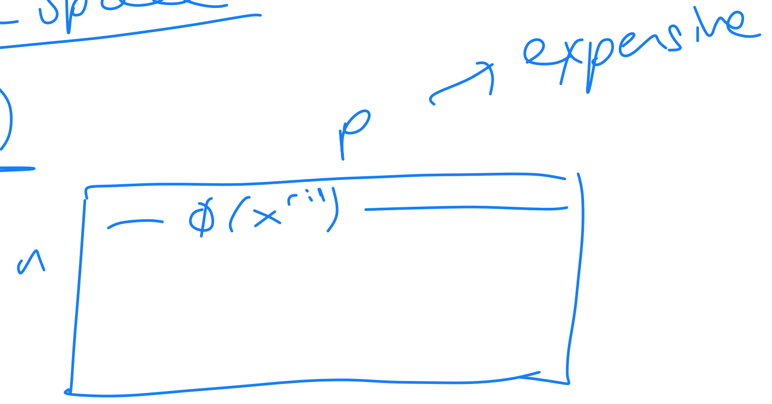
- As illustrating examples, consider polynomial features of degree exactly k
  - $\phi(x) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ for $k = 1$ and $d = 2$, then $K(x, x') = x_1 x_1' + x_2 x_2'$
  - $\phi(x) = \begin{bmatrix} x_1^2 \\ x_2^2 \\ x_1 x_2 \\ x_2 x_1 \end{bmatrix}$ for $k = 2$ and $d = 2$, then $K(x, x') = (x^T x')^2$

Kernel Matrix vs Feature Space

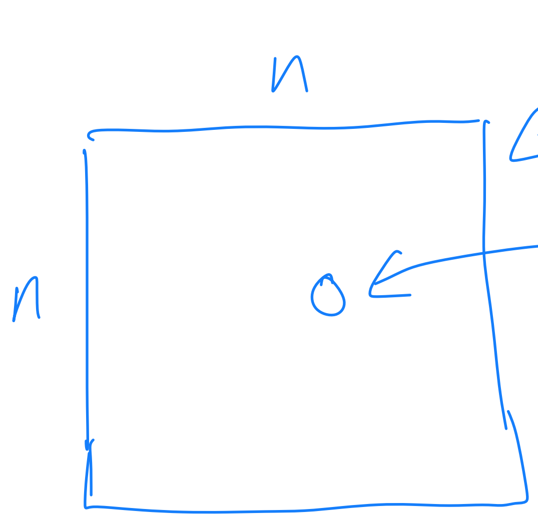
$x_1 \rightarrow$ feature 1

$x^{(1)} \rightarrow$ data point 1





kernel



K is the kernel matrix

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

Kernels are much more efficient to compute than features

- As illustrating examples, consider polynomial features of degree exactly k
  - $\phi(x) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ for $k = 1$ and $d = 2$, then $K(x, x') = x_1 x_1' + x_2 x_2'$
  - $\phi(x) = \begin{bmatrix} x_1^2 \\ x_2^2 \\ x_1 x_2 \\ x_2 x_1 \end{bmatrix}$ for $k = 2$ and $d = 2$, then $K(x, x') = (x^T x')^2$
- Note that for a data point x_i, **explicitly** computing the feature $\phi(x_i)$ takes memory/time $p = d^k$
- For a data point x_i, if we can make predictions by only computing the kernel, then computing $\{K(x_i, x_j)\}_{j=1}^n$ takes memory/time dn
  - The features are **implicit** and accessed only via kernels, making it efficient

Examples of popular Kernels

- Polynomials of degree exactly k

$$K(x, x') = (x^T x')^k$$

- Polynomials of degree up to k

$$K(x, x') = (1 + x^T x')^k$$

- Gaussian (squared exponential) kernel (a.k.a RBF kernel for Radial Basis Function)

$$K(x, x') = \exp\left(- \frac{\|x - x'\|_2^2}{2\sigma^2} \right)$$

- Sigmoid

$$K(x, x') = \tanh(\gamma x^T x' + r)$$

- All these kernels are efficient to compute, but the corresponding features are in high-dimensions

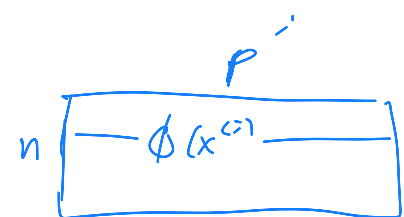
Example: feature vs. kernel

- Ridge regression with feature map $\phi(\cdot) \in \mathbb{R}^p$
  - Solve for $\widehat{w} = \arg\min_{w \in \mathbb{R}^p} \frac{1}{2} \sum_{i=1}^{n} \left(y_i - w^T \phi(x_i)\right)^2 + \lambda \frac{1}{2} \|w\|_2^2$
  - Slow when $p \gg d$

- For now, suppose we are solving this using gradient descent with
  - $w_0 = 0$ and
  - $w_{t+1} \leftarrow w_t + \eta \sum_{i=1}^{n} \phi(x_i)\left(y_i - w_t^T \phi(x_i)\right) - \eta \lambda w_t$

- Claim: For any t, w_t can be represented as $w_t = \sum_{i=1}^{n} \alpha_i \phi(x_i)$

  for some n-dimensional parameter $\alpha = (\alpha_1, \ldots, \alpha_n)$



- Prediction $\hat{y}_{\text{new}} = \hat{w}^T \phi(x_{\text{new}}) = \sum_{i=1}^{n} \hat{\alpha}_i \phi(x_i)^T \phi(x_{\text{new}})$

Kernel ridge regression

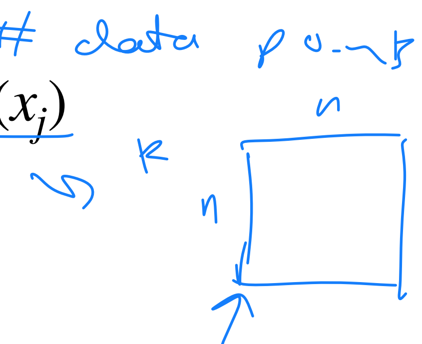
- Ridge regression with feature map $\phi(\,\cdot\,) \in \mathbb{R}^p$

$$\widehat{w} = \arg\min_{w \in \mathbb{R}^p} \frac{1}{2}\|y - \phi(X)w\|_2^2 + \lambda \frac{1}{2}\|w\|_2^2$$

- $$w = \sum_{i=1}^{n} \alpha_i\, \phi(x_i) \;=\; \phi(X)^T \alpha$$

  with now an n-dimensional parameter $\alpha = (\alpha_1, \ldots, \alpha_n)$, instead of p

- Let $K \in \mathbb{R}^{n \times n}$ be the kernel matrix, where $K_{i,j} = \phi(x_i)^T \phi(x_j)$
- The new objective is



Thus, $\widehat{\alpha}_{\text{kernel}} = (\mathbf{K} + \lambda \mathbf{I}_{n \times n})^{-1}\mathbf{y}$

- Prediction $\hat{y}_{\text{new}} \;=\; \hat{w}^T \phi(x_{\text{new}}) \;=\; \sum_{i=1}^{n} \alpha_i \phi(x_i)^T \phi(x_{\text{new}}) = \sum_{i=1}^{n} K(x_i, x_{\text{new}})\alpha$

Kernel regression

- This holds for more general class of algorithms other than GD:

$$w = \sum_{i=1}^{n} \alpha_i \phi(x_i) = \phi(X)^T \alpha$$

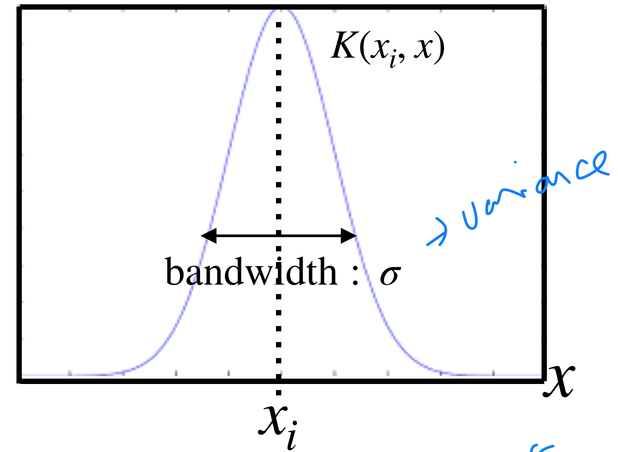
with now an n-dimensional parameter $\alpha = (\alpha_1, \ldots, \alpha_n)$, instead of p

- Kernel method is not limited to linear Ridge regression, but also applies to a broad class of methods including the kernel logistic regression

Gaussian

RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i^{(i)} - x\|_2^2}{2\sigma^2} \right\}$

training data point

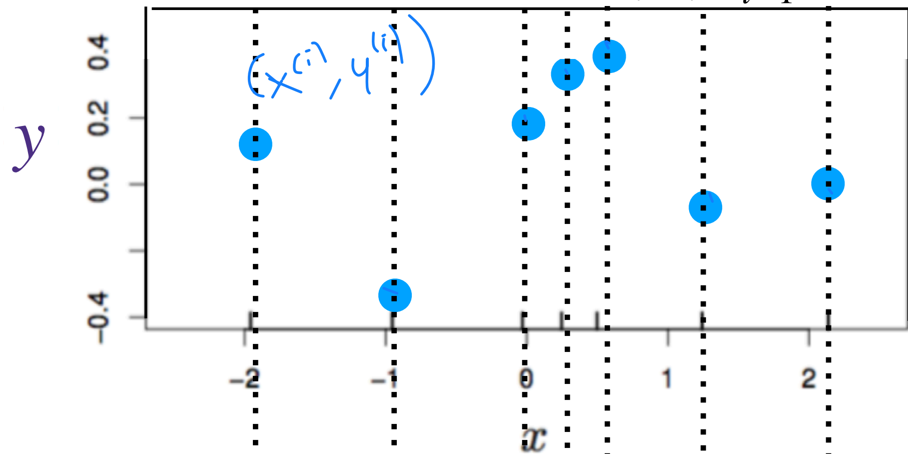


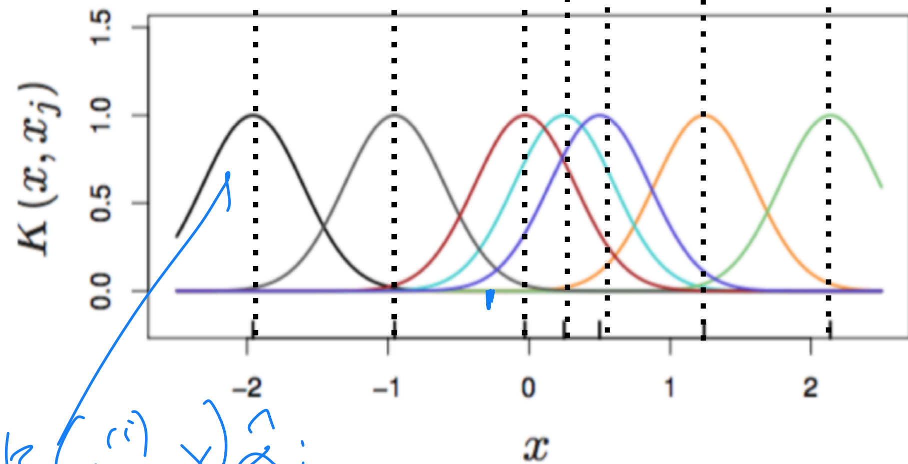
define Gaussian at one data point

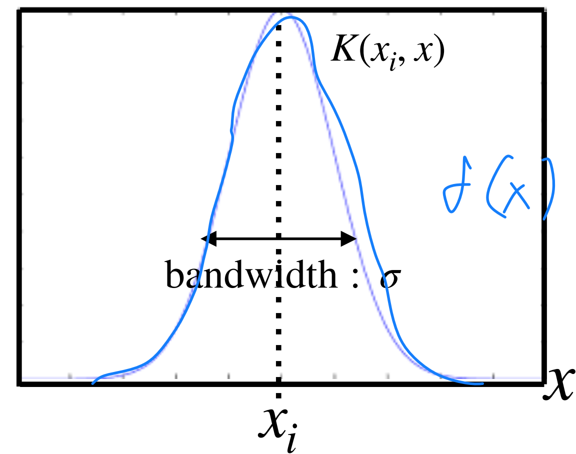
compute distance to it

RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$

samples $\{(x_i, y_i)\}_{i=1}^n$



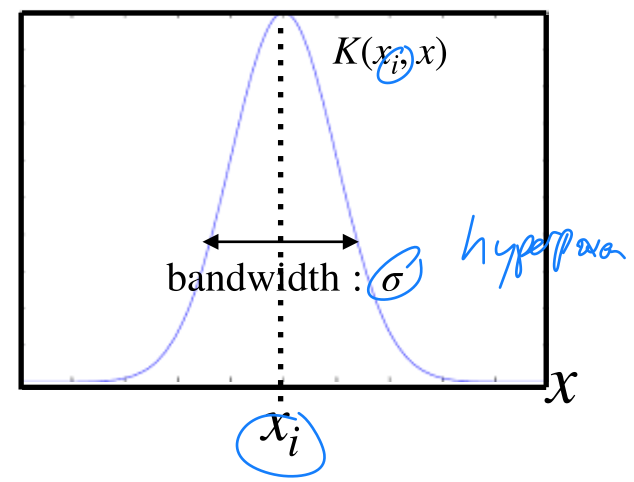


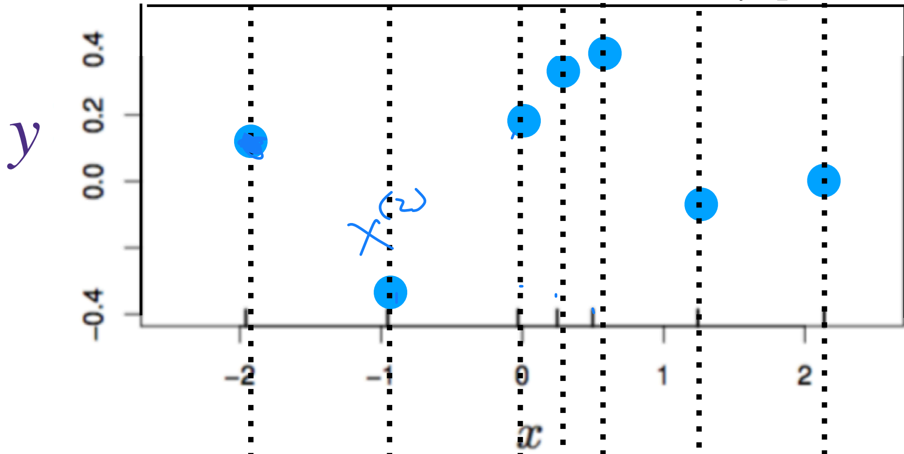


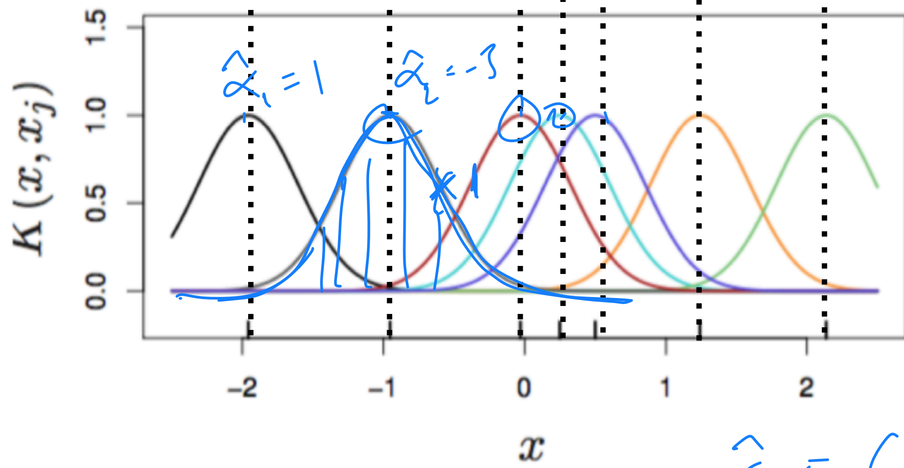
- predictor $f(x) = \sum_{i=1}^{n} \alpha_i K(x_i, x)$ is taking weighted sum of n kernel functions centered at each sample points

RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$

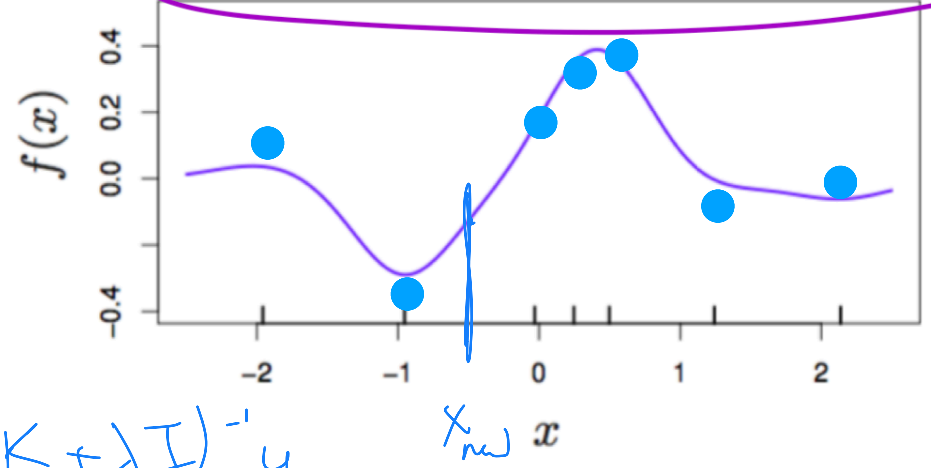
samples $\{(x_i, y_i)\}_{i=1}^n$





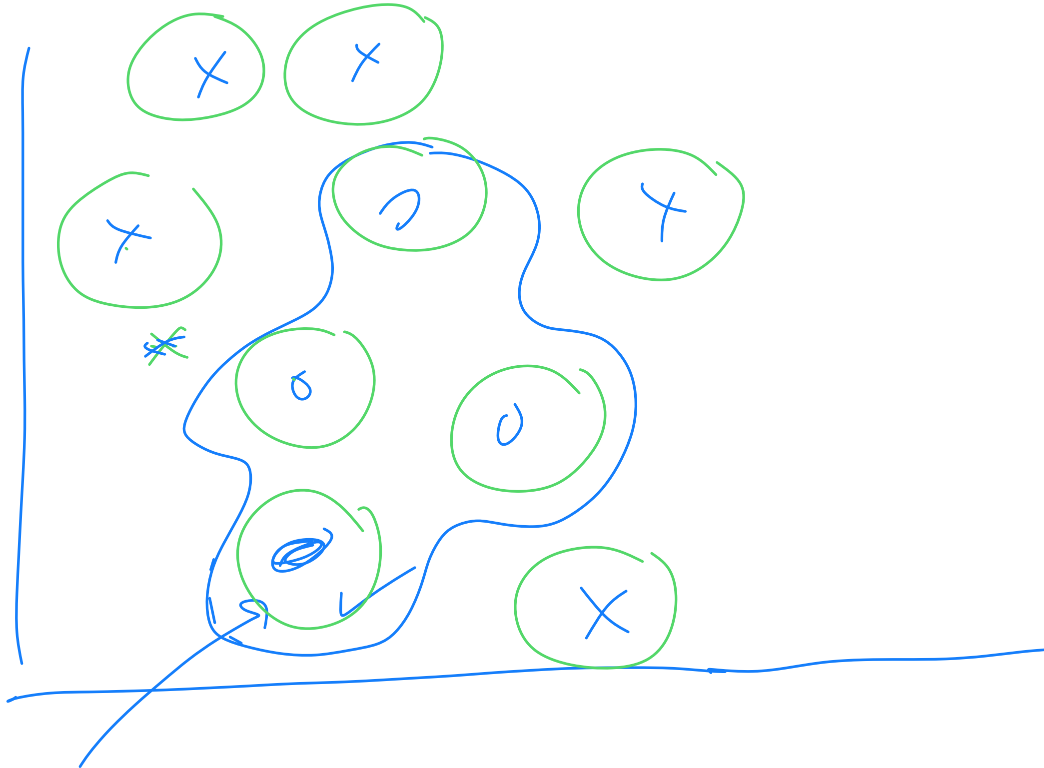


$f(x) = \alpha_0 + \sum_j \alpha_j K(x, x_j)$



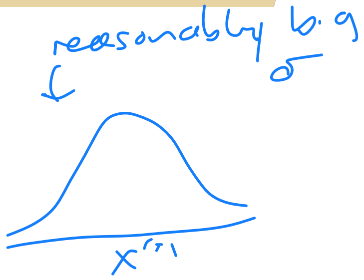
- predictor $f(x) = \sum_{i=1}^{n} \alpha_i K(x_i, x)$ is taking weighted sum of n kernel functions centered at each sample points

KERNELS

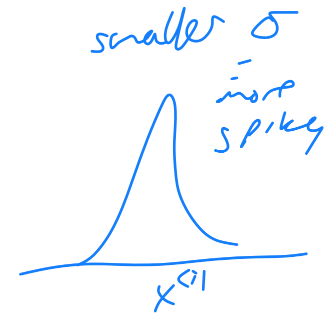


o ← ?

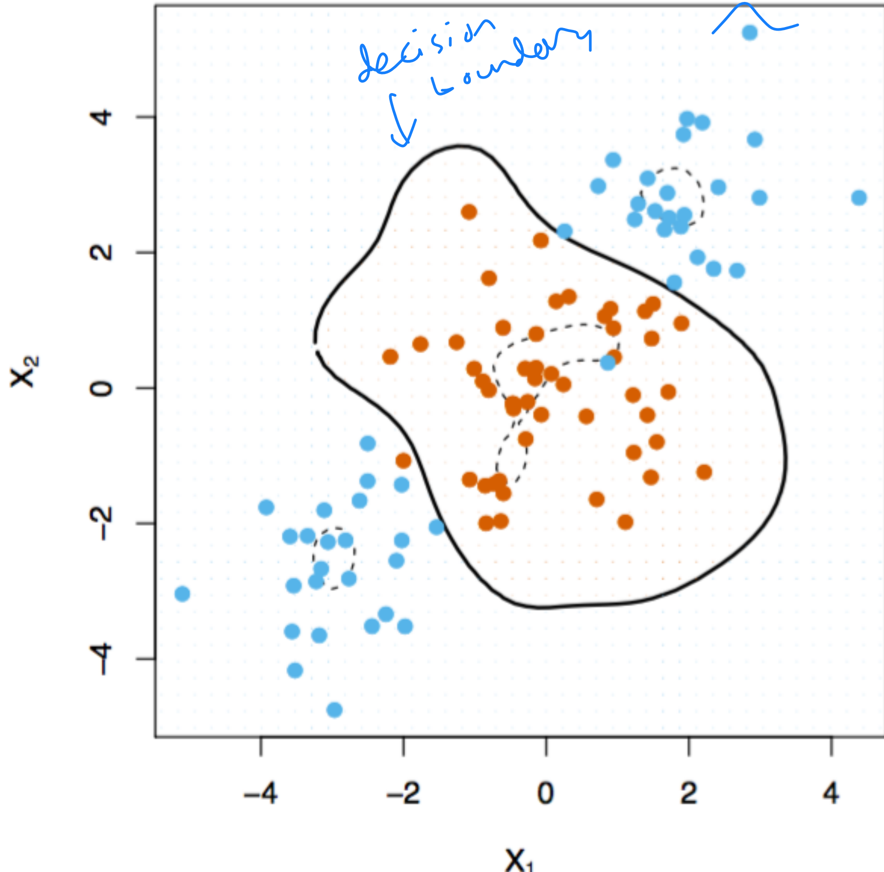
RBF kernel and random features



$$k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$$



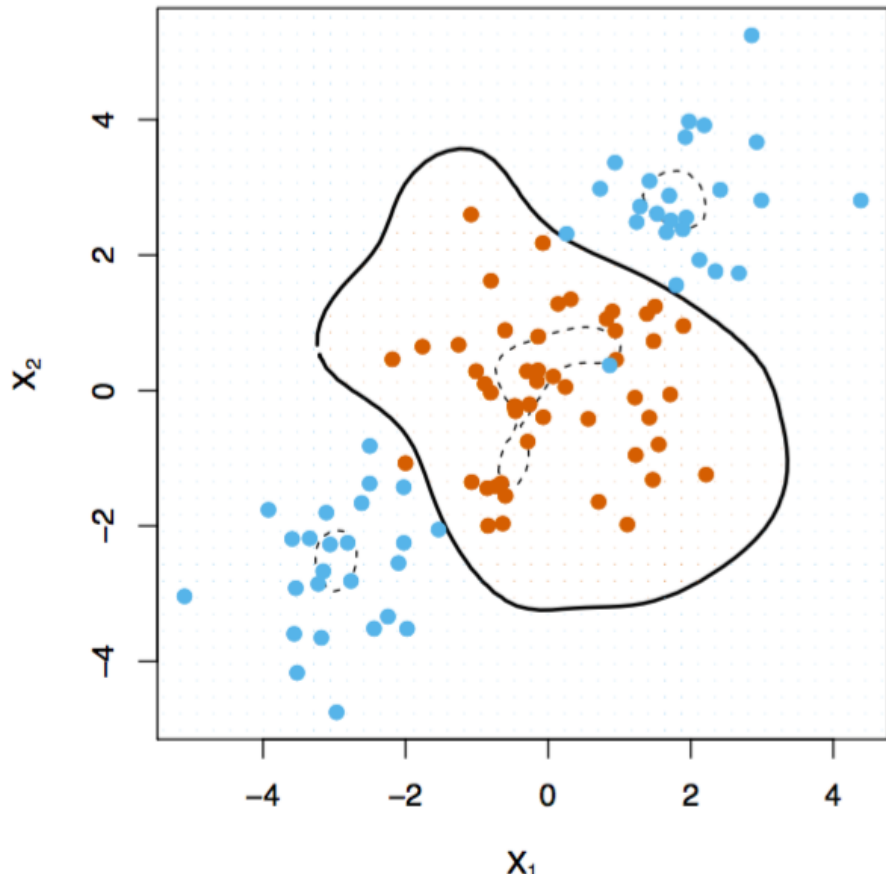
Bandwidth σ is large enough



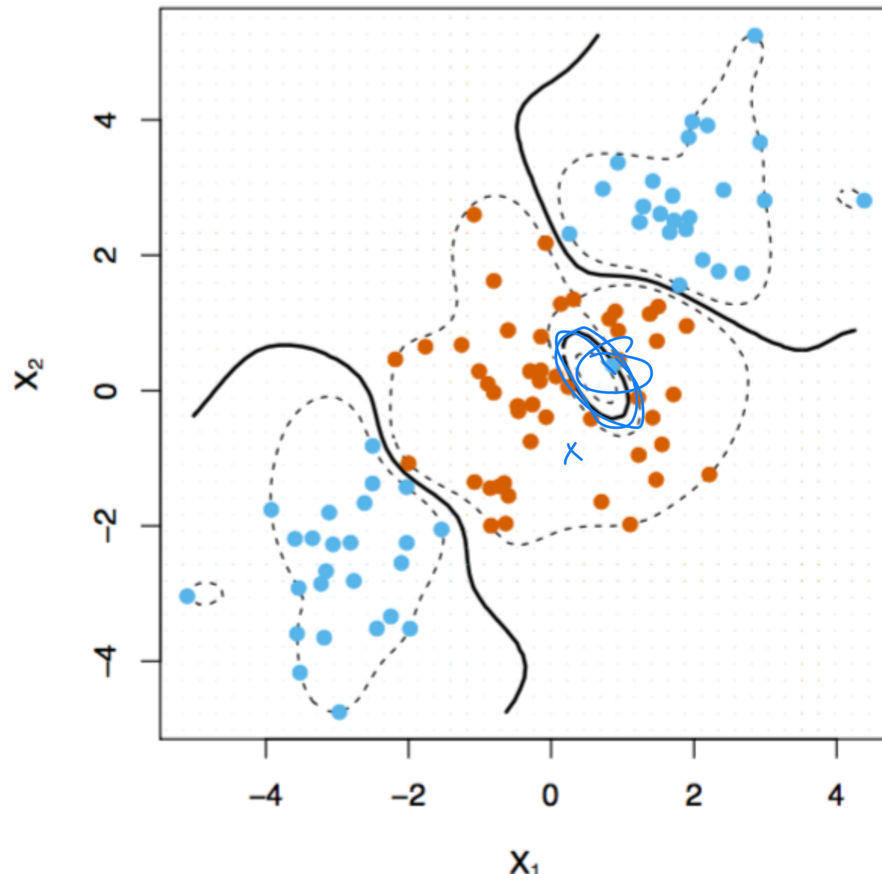
RBF kernel and random features

$$k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$$

Bandwidth σ is large enough



Bandwidth σ is small



Why do we need regularization when using kernels?

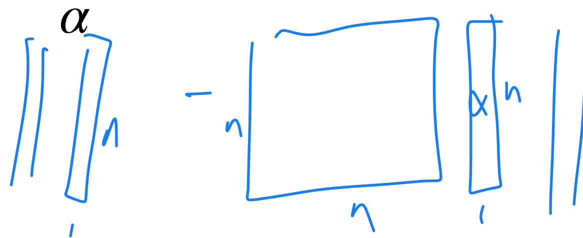
→ PSD

- Typically, $p \gg d$ and $\mathbf{K} \succ 0$. Why?

$K = \phi(x)\,\phi(x)^T$ → squared → non-negative

- So $\mathbf{K}$ is invertible and $\widehat{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_{n\times n})^{-1}\mathbf{y}$ is well defined.
- What if $\lambda = 0$? What goes wrong?

$$\arg\min_{\alpha} \|\mathbf{y} - \mathbf{K}\alpha\|_2^2$$



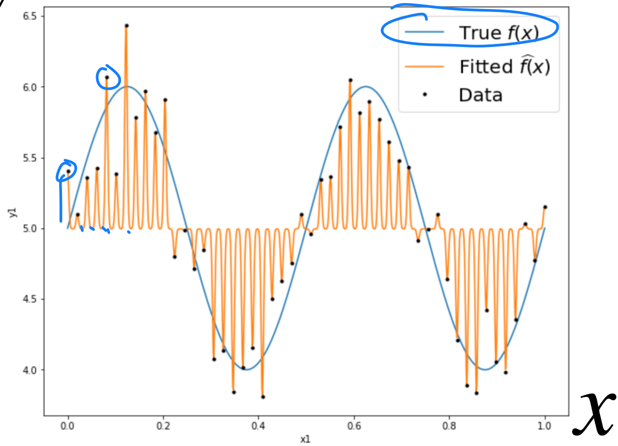
fitting $\alpha \in \mathbb{R}^n$ (n params) with n training data points

exactly fit each training data point → overfitting

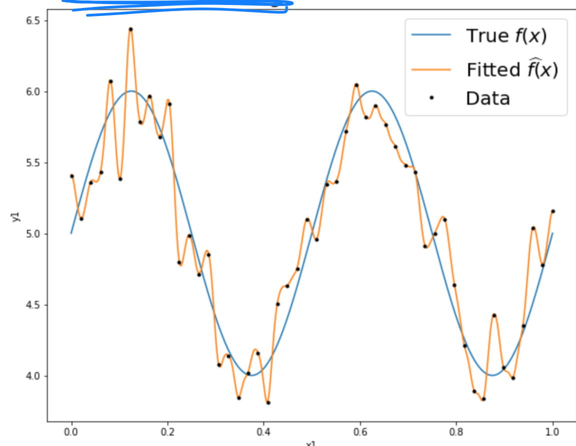
RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$

- $\mathcal{L}(\alpha) = \|\mathbf{K}\alpha - \mathbf{y}\|_2^2 + \lambda\|w\|_2^2$
- The bandwidth σ^2 of the kernel regularizes the predictor, and the regularization coefficient λ also regularizes the predictor

$\sigma = 10^{-3}$ $\lambda = 10^{-4}$



$\sigma = 10^{-2}$ $\lambda = 10^{-4}$

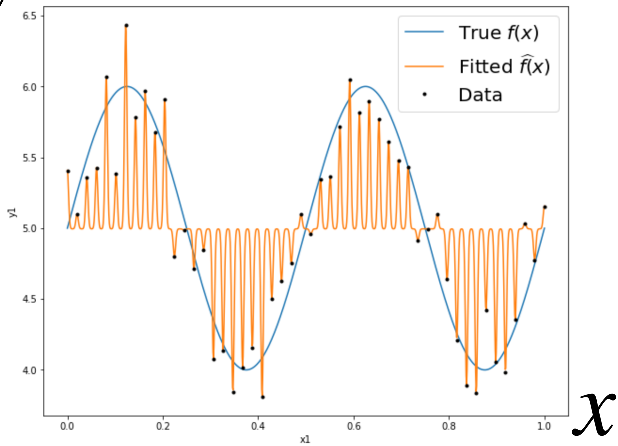


$$\widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i K(x_i, x)$$

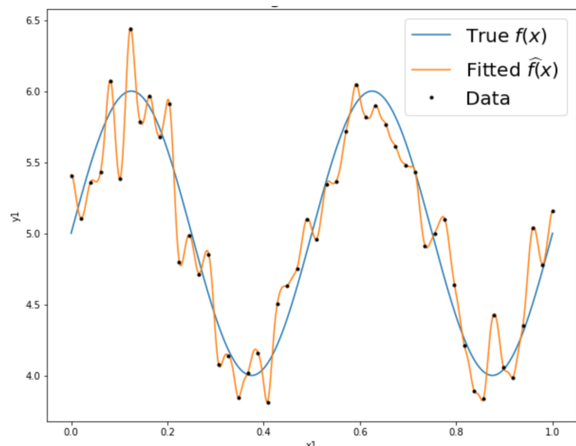
RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$

- $\mathscr{L}(\alpha) = \|\mathbf{K}\alpha - \mathbf{y}\|_2^2 + \lambda\|w\|_2^2$
- The bandwidth σ^2 of the kernel regularizes the predictor, and the regularization coefficient λ also regularizes the predictor

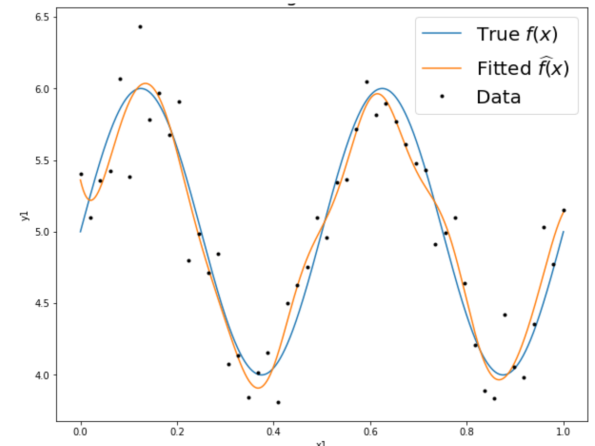
$\sigma = 10^{-3}$ $\lambda = 10^{-4}$



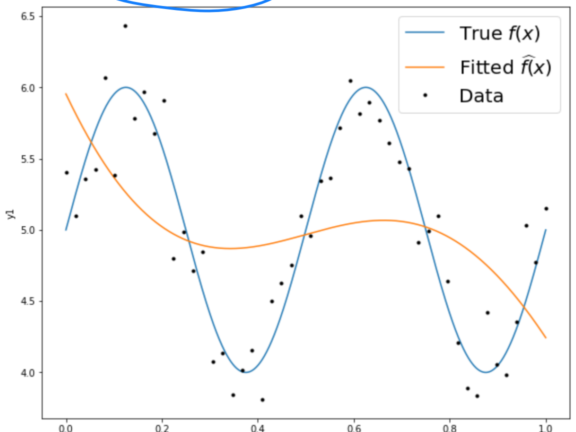
$\sigma = 10^{-2}$ $\lambda = 10^{-4}$



$\sigma = 10^{-1}$ $\lambda = 10^{-4}$



$\sigma = 10^{-0}$ $\lambda = 10^{-4}$



$$\widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i K(x_i, x)$$

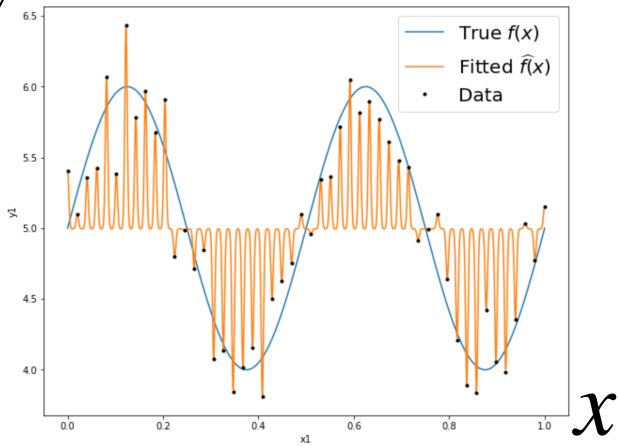
larger bandwidth σ → more smooth → move from overfitting to underfitting

RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$

- $\mathcal{L}(\alpha) = \|\mathbf{K}\alpha - \mathbf{y}\|_2^2 + \lambda\|w\|_2^2$
- The bandwidth σ^2 of the kernel regularizes the predictor, and the regularization coefficient λ also regularizes the predictor

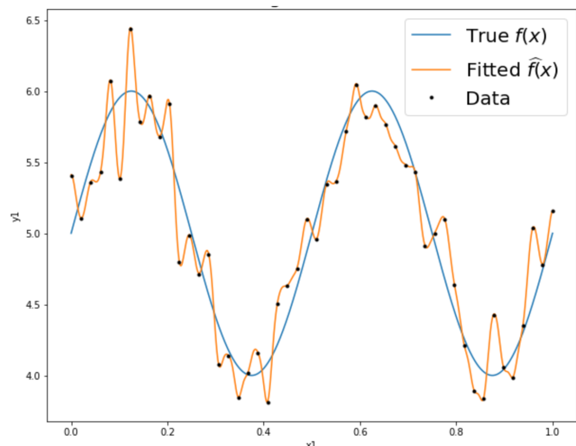
y

$\sigma = 10^{-3}$ $\lambda = 10^{-4}$

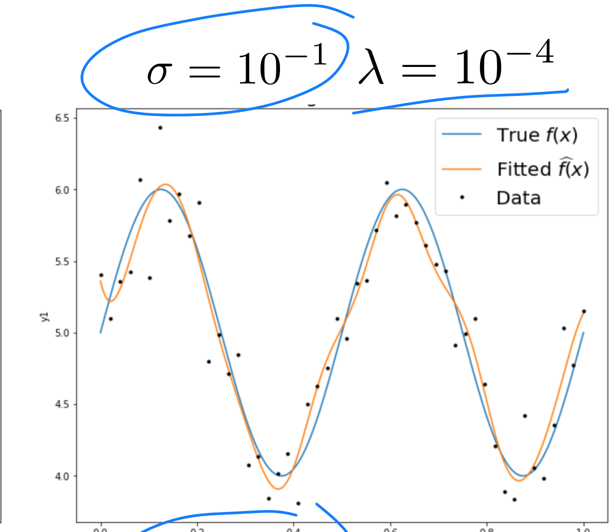


x

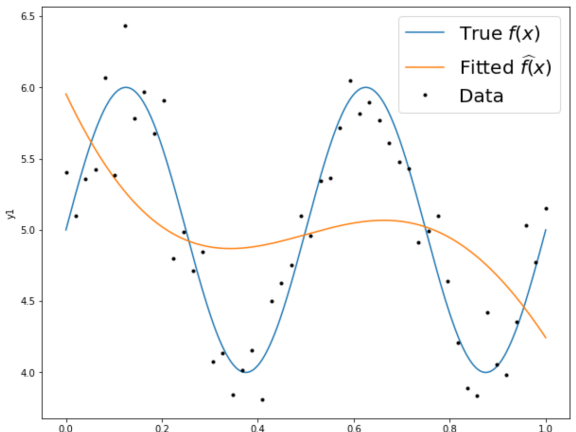
$\sigma = 10^{-2}$ $\lambda = 10^{-4}$



$\sigma = 10^{-1}$ $\lambda = 10^{-4}$

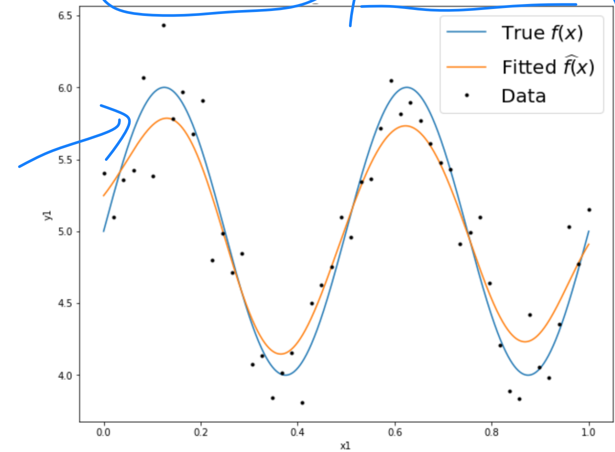


$\sigma = 10^{-0}$ $\lambda = 10^{-4}$



$$\widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i K(x_i, x)$$

$\sigma = 10^{-1}$ $\lambda = 10^{-0}$



Kernel trick finds the optimal solution for linear models under a feature map $\phi(\cdot)$

- Once we have chosen to use a feature map $\phi(\cdot) \in \mathbb{R}^p$, what we want to solve is

$$\hat{w} = \arg\min_{w \in \mathbb{R}^p} \sum_{i=1}^{n} \ell\left(y_i, w^T \phi(x_i)\right)$$ for some convex loss $\ell(,)$

- Gradient descent update (from initialization $w^{(0)} = 0$) that find the optimal solution is

$$w^{(t+1)} = w^{(t)} - \eta \sum_{i=1}^{n} \ell'(y_i, w^T \phi(x_i))\phi(x_i)$$

- One crucial observation is that all $w^{(t)}$'s (including the optimal solution $w^{(\infty)}$) lie on the subspace spanned by $\{\phi(x_1), \ldots, \phi(x_n)\}$, which is an n-dimensional subspace in $\mathbb{R}^p$
- Hence, it is sufficient to look for a solution that is represented as

$$\hat{w} = \sum_{i=1}^{n} \alpha_i \phi(x_i)$$ to find the optimal solution

- Kernel trick finds the optimal solution efficiently, by searching over the model that can be represented as $\hat{w} = \sum_{i=1}^{n} \alpha_i \phi(x_i)$

Fixed Feature V.S. Learned Feature

- Kernel method works well if we choose a good kernel such that the data is linearly separable in the corresponding (possibly infinite dimensional) feature space

- In practice, it is hard to choose a good kernel for a given problem

- Can we **learn** the feature mapping $\phi : \mathbb{R}^d \to \mathbb{R}^p$ from data also?

$x \in \mathbb{R}^d$

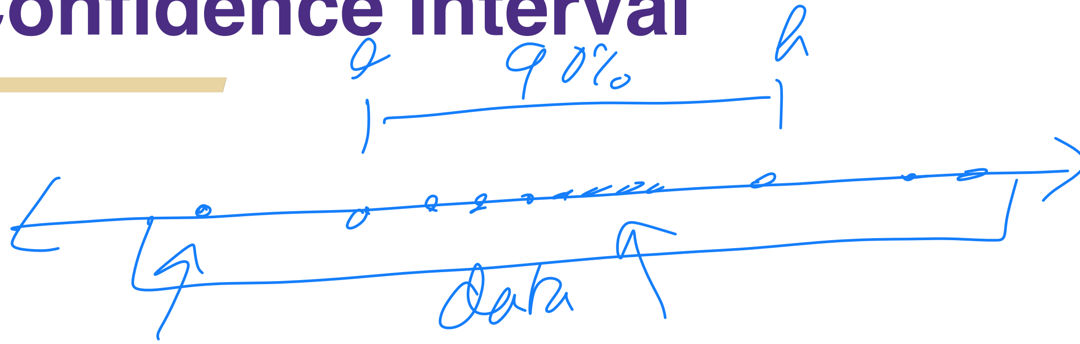
neural networks!

Questions?

Bootstrap

measuring uncertainty for your model

Confidence interval



$\hat{mean}(x) = \frac{1}{n}\sum_{i=1}^{n} x_i$

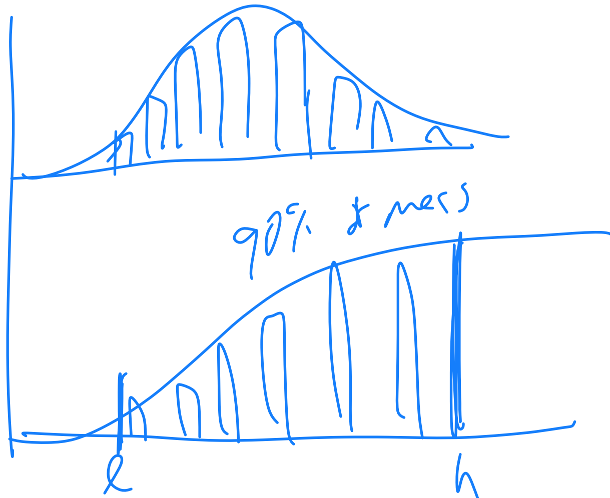
"I'm 90% certain the mean is between l, h"

How to get?

run experiment 10,000 times. get $\hat{m}_1, \hat{m}_2, \ldots \hat{m}_{10000}$

PPF

CDF



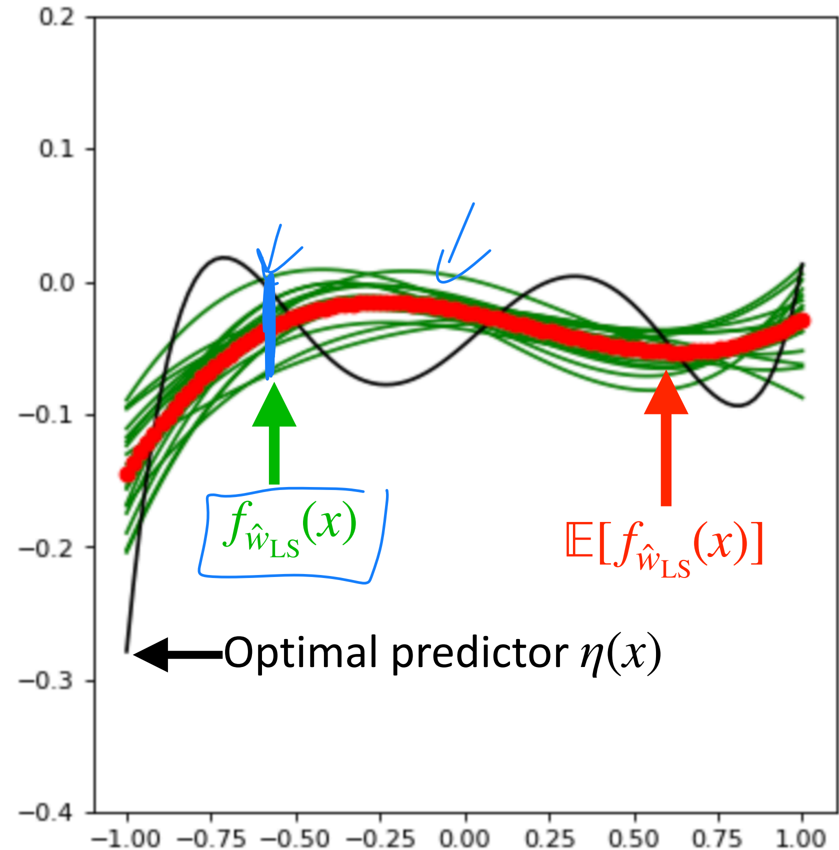
Bootstrapping: doing something seemingly impossible

- pull oneself up by one's bootstraps — improve one's position by one's own efforts without help from others.
- In our context: A general method of calculating uncertainty estimates without any additional data. (Efron, 1979)

Bootstrapping: doing something seemingly impossible

to pull oneself up by one's bootstraps

Remember bias-variance trade-off?



It is seemingly impossible to compute **Variance**, for example, with a single dataset.

Confidence interval

- suppose you have training data $\{(x_i, y_i)\}_{i=1}^n$ drawn i.i.d. from some true distribution $P_{x,y}$
- we train a kernel ridge regressor, with some choice of a kernel $K : \mathbb{R}^{d\times d} \to \mathbb{R}$

$$\text{minmize}_\alpha \; \|\mathbf{K}\alpha - \mathbf{y}\|_2^2 + \lambda \alpha^T \mathbf{K}\alpha$$

- the resulting predictor is

$$f(x) \;=\; \sum_{i=1}^n K(x_i, x)\hat{\alpha}_i,$$

where

$$\hat{\alpha} \;=\; (\mathbf{K} + \lambda \mathbf{I})^{-1}\mathbf{y} \quad \in \mathbb{R}^n$$

- we wish to build a confidence interval for our predictor $f(x)$, using 5% and 95% percentiles

Confidence interval

- suppose you have training data $\{(x_i, y_i)\}_{i=1}^n$ drawn i.i.d. from some true distribution $P_{x,y}$
- we train a kernel ridge regressor, with some choice of a kernel $K : \mathbb{R}^{d \times d} \to \mathbb{R}$

$$\text{minmize}_\alpha \ \|\mathbf{K}\alpha - \mathbf{y}\|_2^2 + \lambda \alpha^T \mathbf{K} \alpha$$

- the resulting predictor is

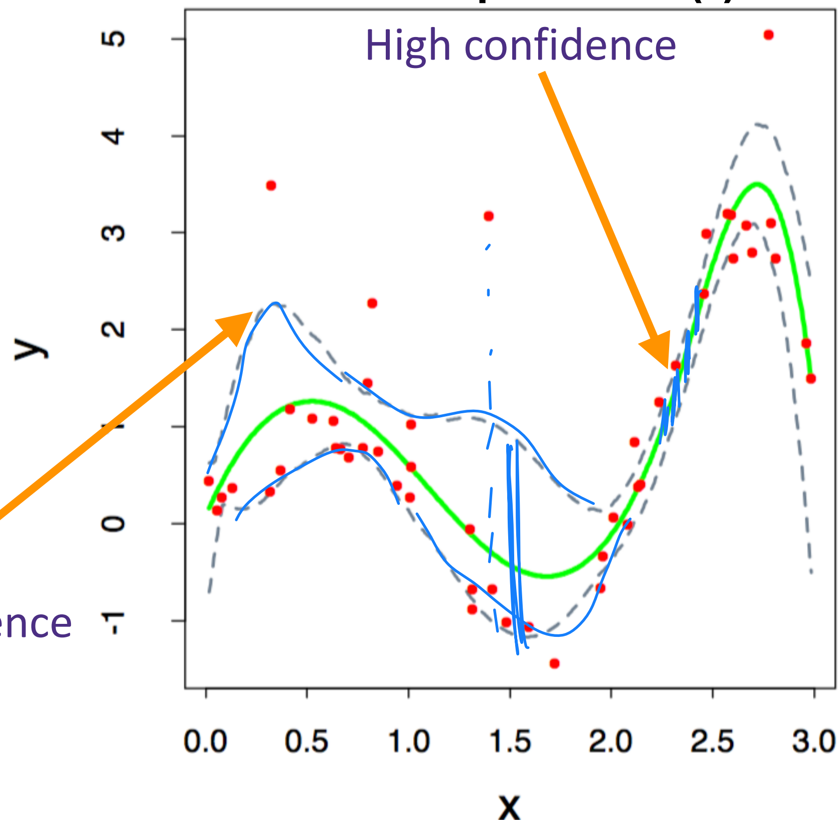
$$f(x) \ = \ \sum_{i=1}^{n} K(x_i, x)\hat{\alpha}_i,$$

where

$$\hat{\alpha} \ = \ (\mathbf{K} + \lambda \mathbf{I})^{-1}\mathbf{y} \quad \in \mathbb{R}^n$$

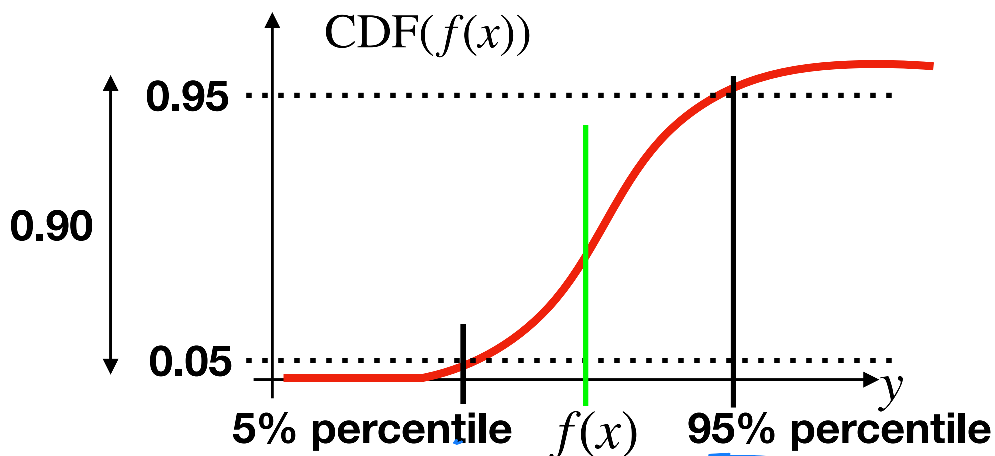
- we wish to build a confidence interval for our predictor $f(x)$, using 5% and 95% percentiles

Example of 5% and 95% percentile curves for predictor f(x)



Confidence interval

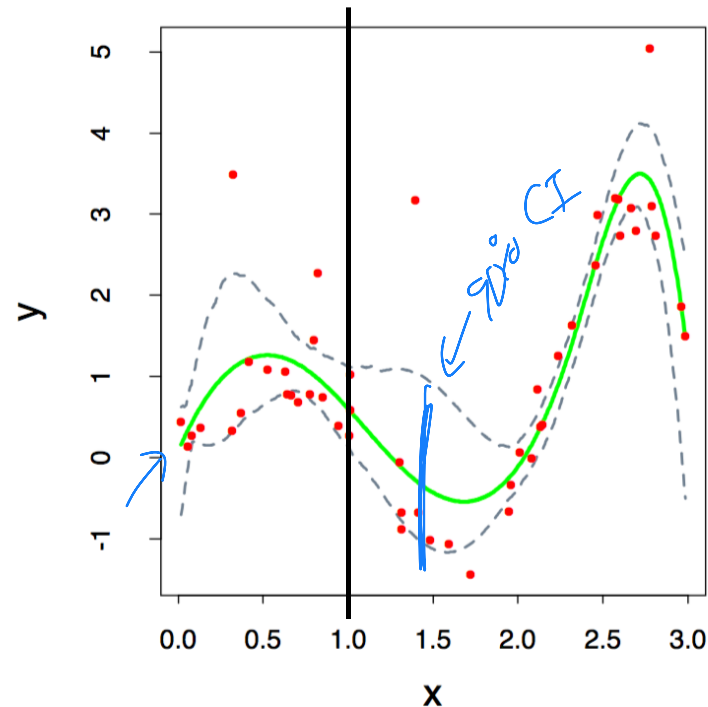
- let's focus on a single $x \in \mathbb{R}^d$
- note that our predictor $f(x)$ is a random variable, whose randomness comes from the training data $S_{\text{train}} = \{(x_i, y_i)\}_{i=1}^n$
- if we know the statistics (in particular the CDF of the random variable $f(x)$) of the predictor, then the **confidence interval** with **confidence level 90%** is defined as

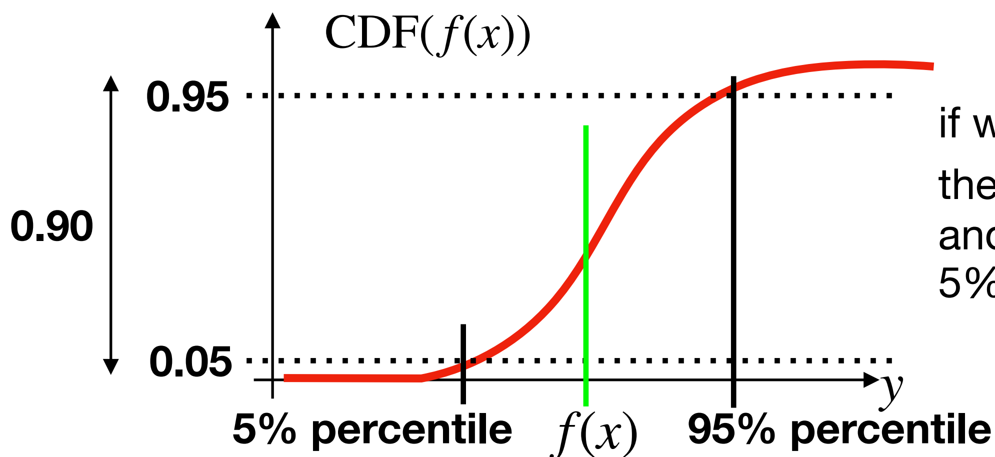


- as we do not have the cumulative distribution function (CDF), we need to approximate them

Confidence interval

- let's focus on a single $x \in \mathbb{R}^d$
- note that our predictor $f(x)$ is a random variable, whose randomness comes from the training data $S_{\text{train}} = \{(x_i, y_i)\}_{i=1}^n$
- if we know the statistics (in particular the CDF of the random variable $f(x)$) of the predictor, then the **confidence interval** with **confidence level 90%** is defined as





if we know the distribution of our predictor $f(x)$, the green line is the expectation $\mathbb{E}[f(x)]$ and the black dashed lines are the 5% and 95% percentiles in the figure above

- as we do not have the cumulative distribution function (CDF), we need to approximate them

Bootstrap

- as we cannot sample repeatedly (in typical cases), we use **bootstrap samples** instead
- bootstrap is a general tool for assessing statistical accuracy
- we learn it in the context of confidence interval for trained models

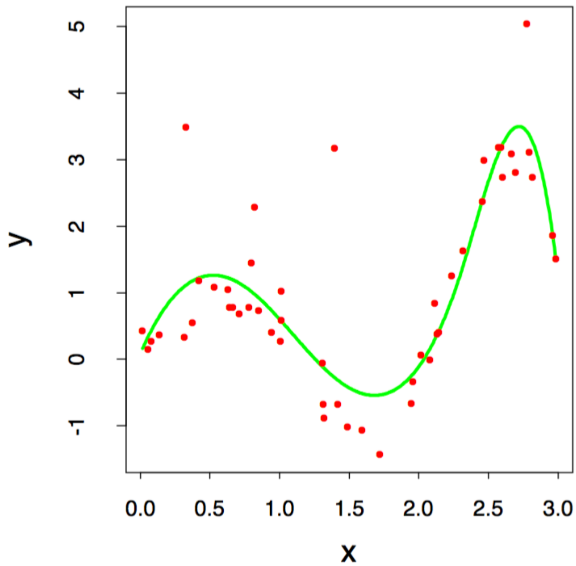
- a **bootstrap dataset** is created from the training dataset by taking n (the same size as the training data) examples uniformly at random **with replacement** from the training data $\{(x_i, y_i)\}_{i=1}^n$

- for b=1,…,B
  - create a bootstrap dataset $S^{(b)}_{\text{bootstrap}}$
  - train a regularized kernel regression $\alpha^{*(b)}$
  - predict $\hat{y}^{(b)} = \sum_{i=1}^{n} K(x_i^{(b)}, x)\alpha_i^{*(b)}$

- compute the empirical CDF from the bootstrap datasets, and compute the confidence interval

bootstrap

training a single predictor



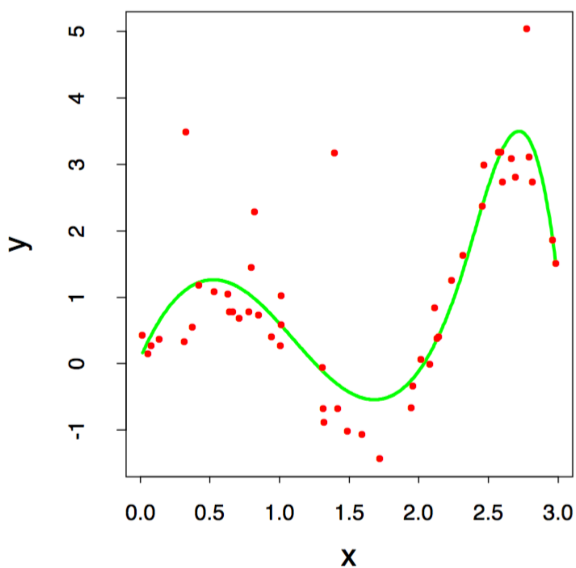
multiple bootstrapped predictors

90% confidence interval

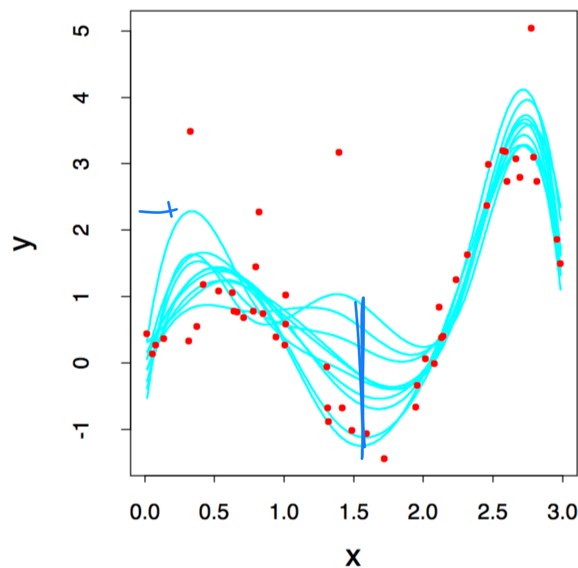
Figures from Hastie et al

bootstrap

training a single predictor



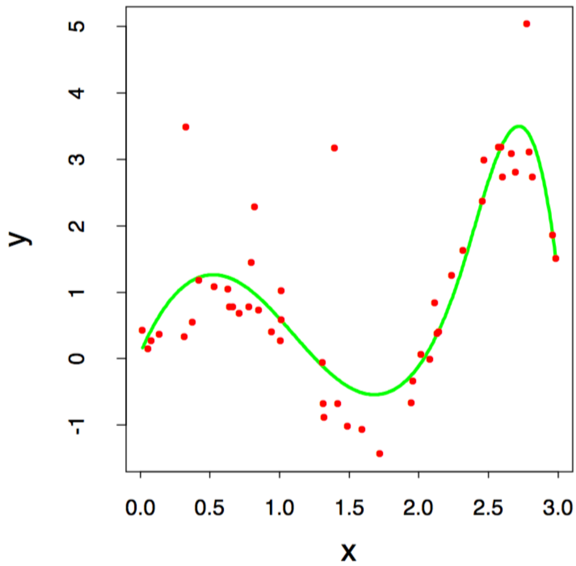
multiple bootstrapped predictors

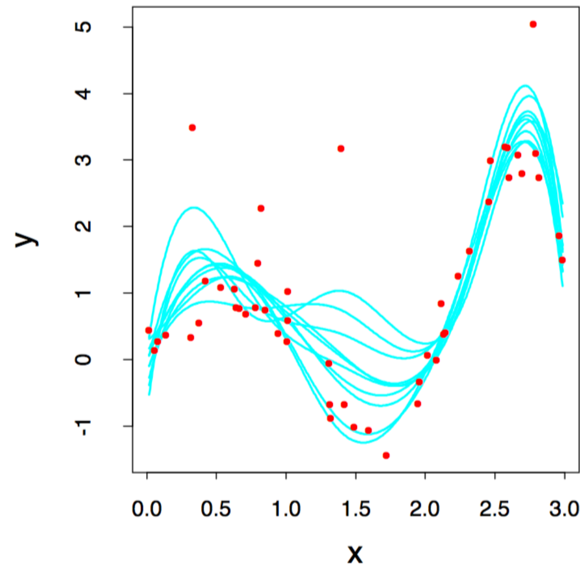


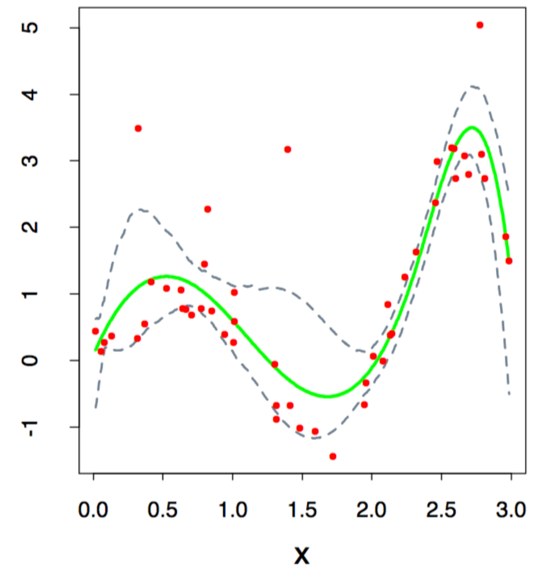
90% confidence interval

Figures from Hastie et al

bootstrap

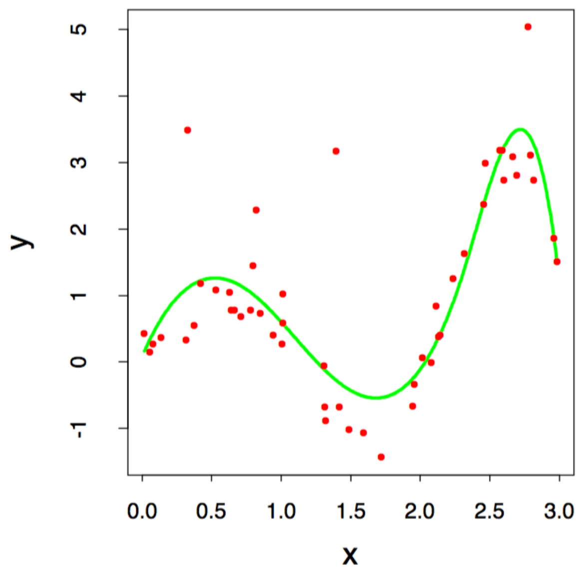




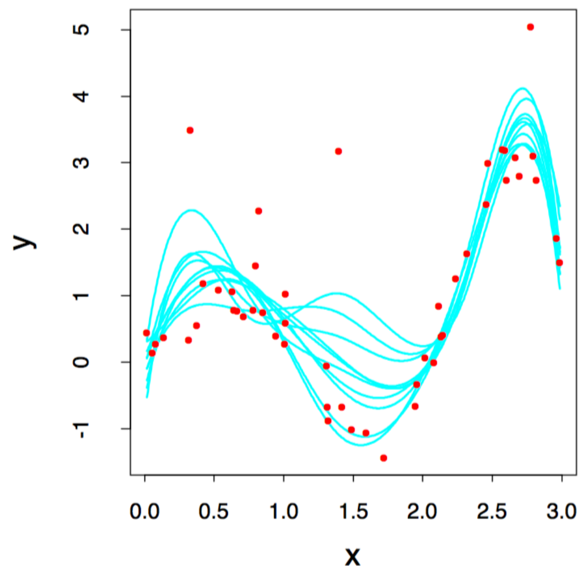


Figures from Hastie et al

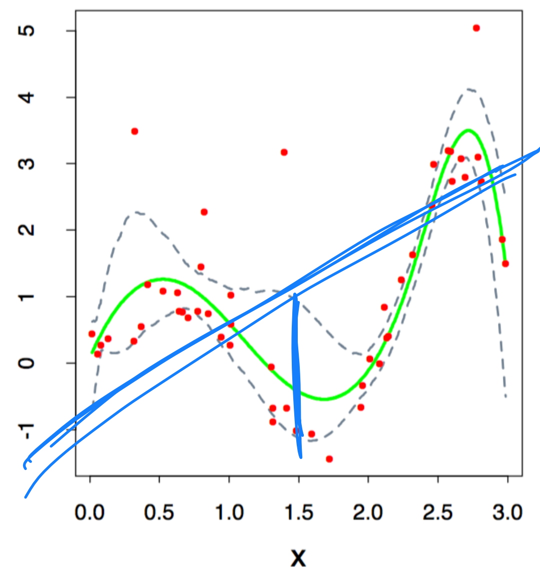
training a single predictor



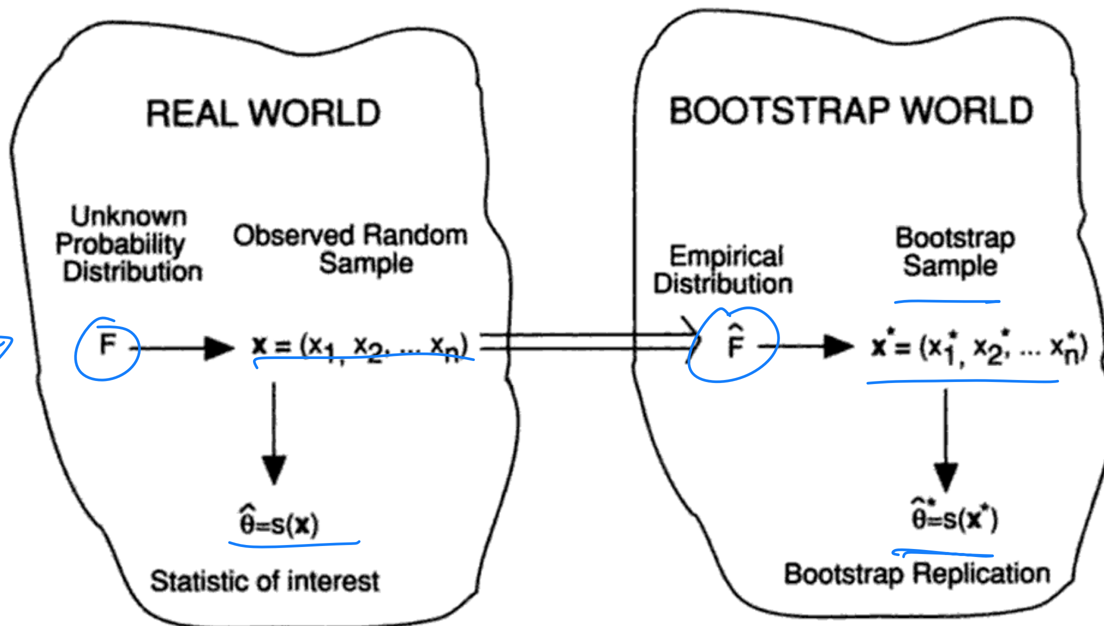
multiple bootstrapped predictors



90% confidence interval



bias / variance tradeoff



Figures from Efron et al. 1993

Takeaways

→ Bootstrap

Advantages:

- Very simple to use and generally applicable – build a confidence interval around anything
- Appears to give meaningful results even when the amount of data is very small
- Very strong asymptotic theory (as number of examples goes to infinity)

Disadvantages:

- Very few meaningful finite-sample guarantees
- Potentially computationally intensive
- Reliability hinges on test statistic and rate of convergence of empirical CDF to true CDF, which is unknown
- Poor performance on "extreme statistics" (e.g., the max)

Questions?