Lecture 10:
Bootstrap and Kernel Methods

Bootstrap

- Measuring uncertainty of your model

Confidence interval for statistics $\hat{\theta} = g(\mathscr{D})$

- Consider a simple problem of mean estimation given access to an oracle that gives fresh samples, and we are interested in estimating mean from n samples
- We can first collect one dataset $\mathscr{D}_n = \{z_1, \ldots, z_n\} \in \mathbb{R}^n$ sampled i.i.d. from that distribution
- And compute the mean estimate using our estimator of interest, say,

$$\hat{\theta} \leftarrow g(\mathscr{D}_n) = \frac{1}{n} \sum_{i=1}^{n} z_i$$

- Furthermore, let's say we are interested in estimating how uncertain we are about this estimate. This can be done using **confidence intervals**.
- This course is not a statistics course, so we do not go into a great detail on confidence interval. We will only be using it to learn **bootstrapping**.

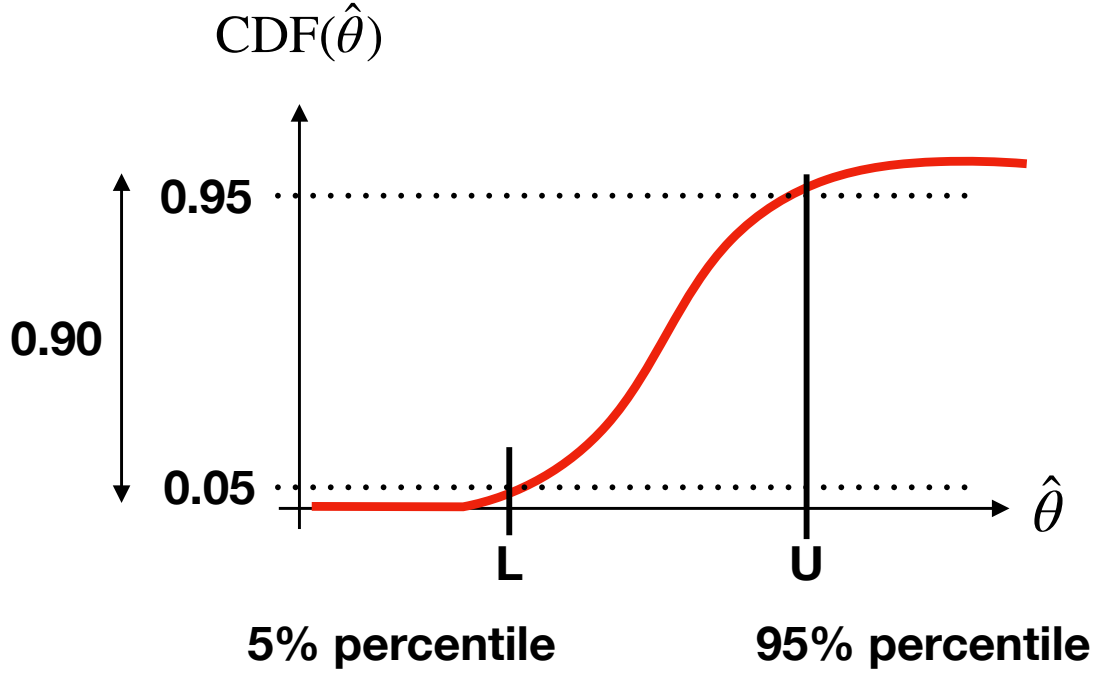
- We have a mean estimate using our estimator of interest, say,

$$\hat{\theta} \leftarrow g(\mathscr{D}_n) = \frac{1}{n}\sum_{i=1}^{n} z_i$$

  and an oracle sampler from some distribution P_Z.
- If we know the distribution of our estimate $\hat{\theta}$, then one would output a, for example, 90% confidence interval of

$$\text{CI} = \left[L, U\right]$$

- The main idea is to ensure that the actual mean, say θ^*, of the distribution P_Z falls in the CI with probability at least 90%.



Confidence interval for statistics $\hat{\theta} = g(\mathscr{D})$

- We have a mean estimate using our estimator of interest, say,

$$\hat{\theta} \leftarrow g(\mathscr{D}_n) = \frac{1}{n}\sum_{i=1}^{n} z_i$$

  and an oracle sampler from some distribution P_Z.
- If we know the distribution of our estimate $\hat{\theta}$, then one would output a, for example, 90% confidence interval of

$$\mathrm{CI} = \left[L, U\right]$$

- The main idea is to ensure that the actual mean, say θ^*, of the distribution P_Z falls in the CI with probability at least 90%.

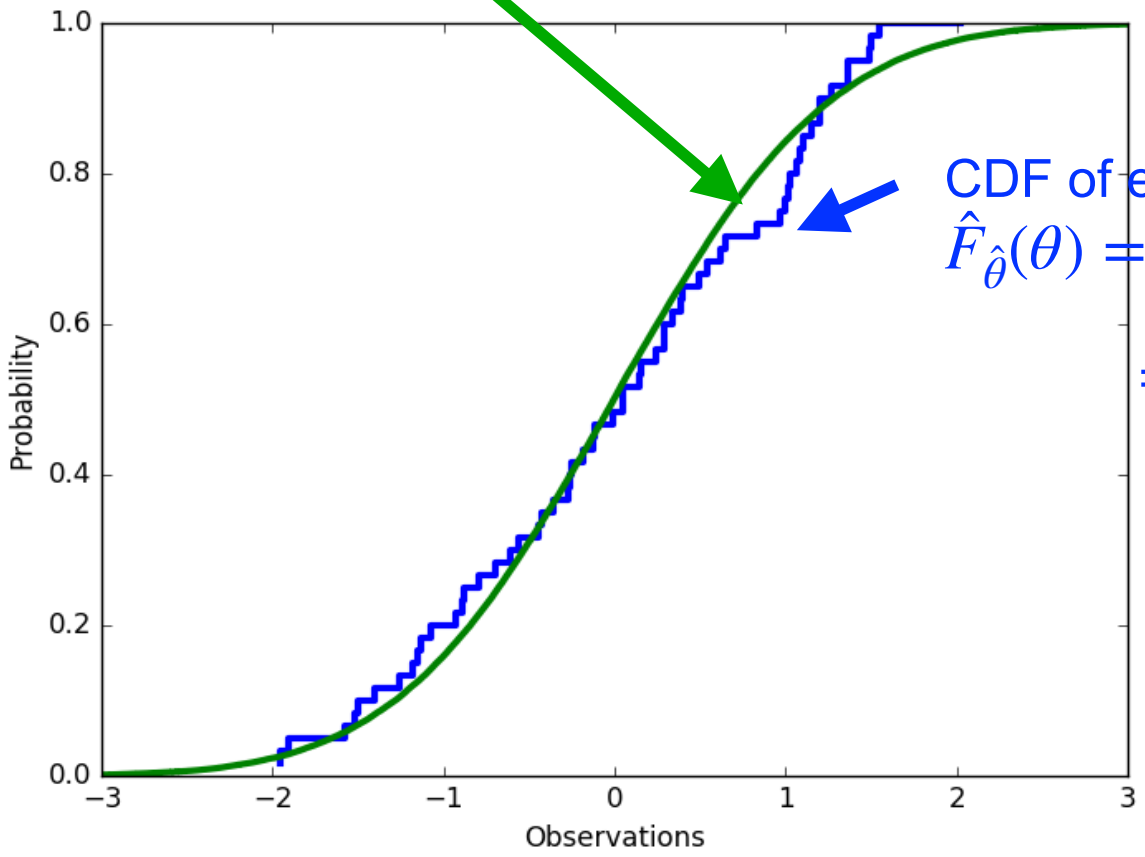
- The problem is that we typically do not know the distribution of $\hat{\theta}$, so what can we do?
  - This is easy if we have an oracle and can keep sampling new datasets $\mathscr{D}_n$ so that we can get multiple "samples" of $\hat{\theta}$ to estimate its distribution.

Confidence interval with oracle access to sampler

- "sample" many fresh datasets $\{\mathscr{D}^{(1)}, \ldots, \mathscr{D}^{(B)}\}$, each of size n
- compute estimates $\{\hat{\theta}^{(1)} = g(\mathscr{D}^{(1)}) \ \ldots, \hat{\theta}^{(B)} = g(\mathscr{D}^{(B)})\}$ for each dataset

CDF of an unknown True distribution

$$F_{\hat{\theta}}(\theta) = P_{\mathscr{D}}(g(\mathscr{D}) \leq \theta)$$



CDF of empirical observed distribution

$$\hat{F}_{\hat{\theta}}(\theta) = \hat{P}_B(g(\mathscr{D}) \leq \theta)$$

$$= \frac{1}{B} \sum_{b=1}^{B} \mathbf{I}\{\hat{\theta}^{(b)} \leq \theta\}$$

Set-up for regression/classification

We have a dataset of n samples drawn i.i.d. with (CDF) F_Z

$$\mathscr{D} = \{z_1, \ldots, z_n\} \sim F_Z$$

We compute a *statistic* of the data to get $g(\mathscr{D})$

Goal: Estimate uncertainty around $g(\mathscr{D})$.
In other words, what's the distribution of $g(\mathscr{D})$ over different random samples of $\mathscr{D}$?

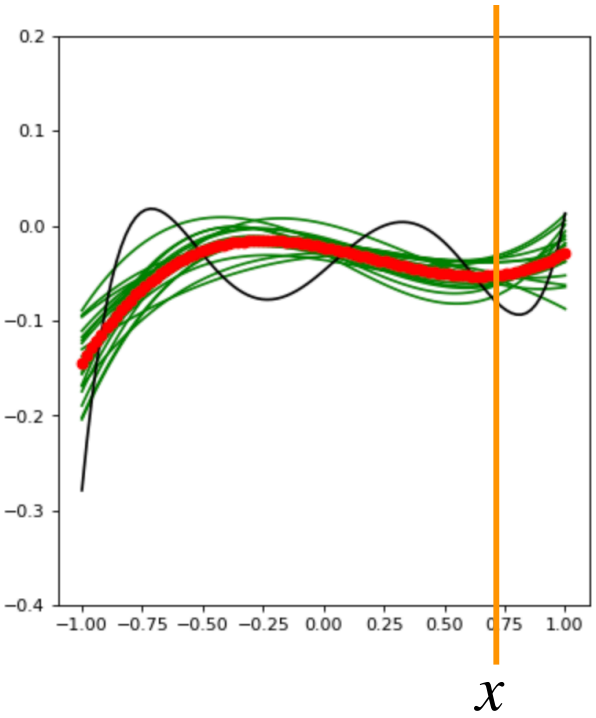
Set-up for regression/classification

We have a dataset of n samples drawn i.i.d. with (CDF) F_Z

$$\mathscr{D} = \{z_1, \ldots, z_n\} \sim F_Z$$

We compute a *statistic* of the data to get $g(\mathscr{D})$

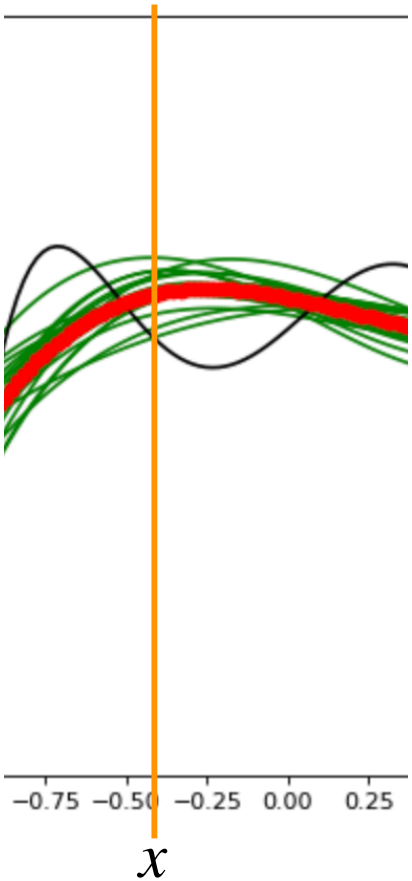
Goal: Estimate uncertainty around $g(\mathscr{D})$.
In other words, what's the distribution of $g(\mathscr{D})$ over different random samples of $\mathscr{D}$?



For example, in the example of regression, we might care about $g(\mathscr{D}) = f_{\hat{\theta}(\mathscr{D})}(x)$, where $\hat{\theta}(\mathscr{D})$ is the model parameter trained on the train-set $\mathscr{D}$, and $f_{\hat{\theta}(\mathscr{D})}(x)$ is our prediction for input x.

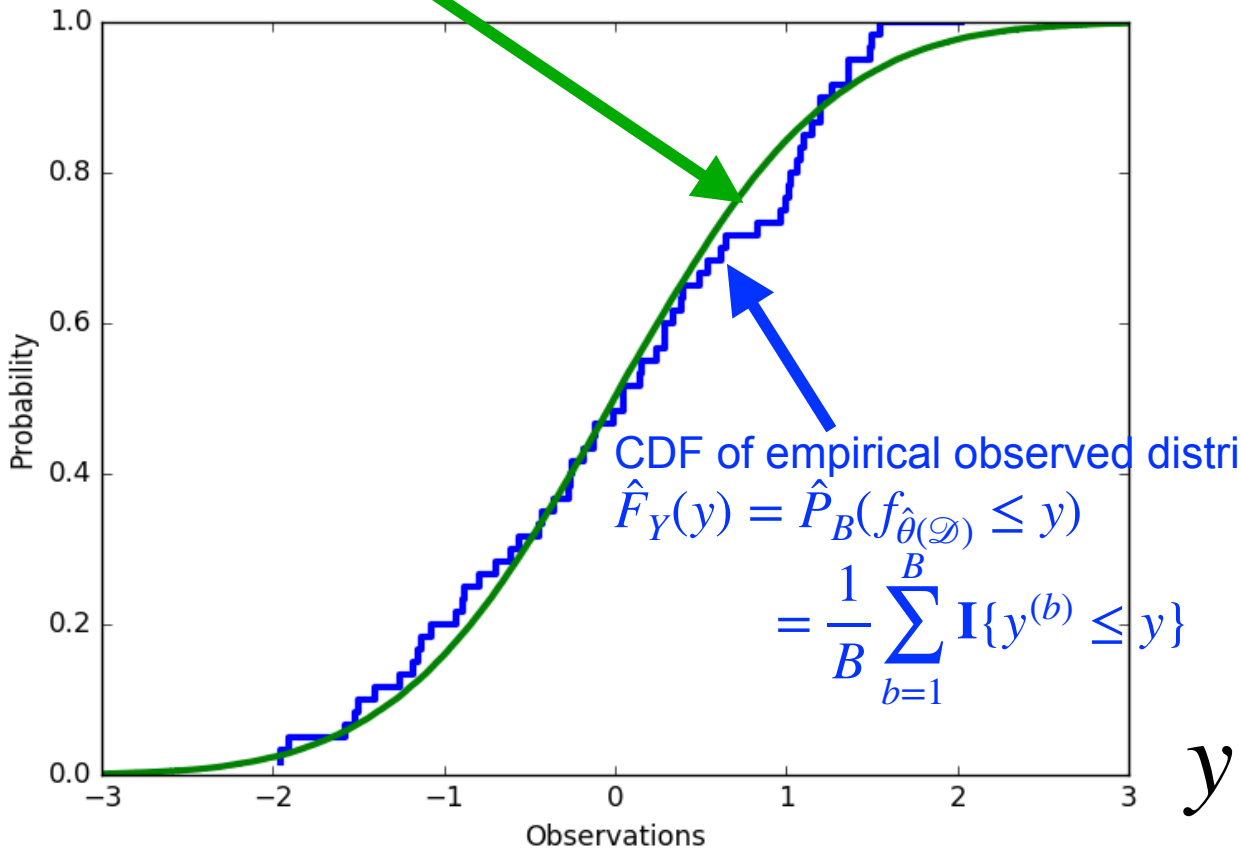
Confidence interval with oracle access to sampler

- "sample" many fresh datasets $\{\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(B)}\}$, each of size n
- train models $\{\hat{\theta}^{(1)} \ldots, \hat{\theta}^{(B)}\}$ for each dataset
- commute the prediction $\{y^{(1)} = f_{\hat{\theta}(\mathcal{D}^{(i)})}(x), \ldots, y^{(B)} = f_{\hat{\theta}(\mathcal{D}^{(i)})}(x)\}$



CDF of an unknown True distribution

$F_Y(y) = P(f_{\hat{\theta}(\mathcal{D})} \leq y)$



CDF of empirical observed distribution

$\hat{F}_Y(y) = \hat{P}_B(f_{\hat{\theta}(\mathcal{D})} \leq y)$

$= \frac{1}{B} \sum_{b=1}^{B} \mathbf{I}\{y^{(b)} \leq y\}$

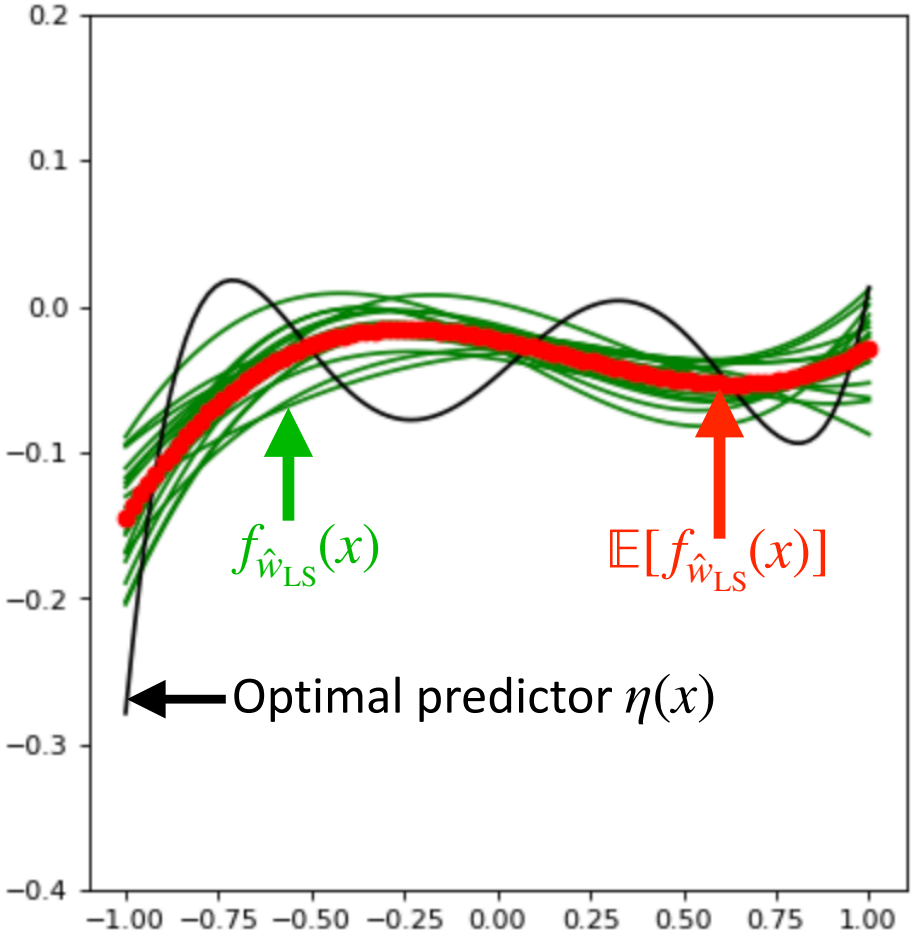
Bootstrapping: doing something seemingly impossible

- 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?




```
current train error = 0.0036791644380554187
current test error  = 0.0037962529988410953
```

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

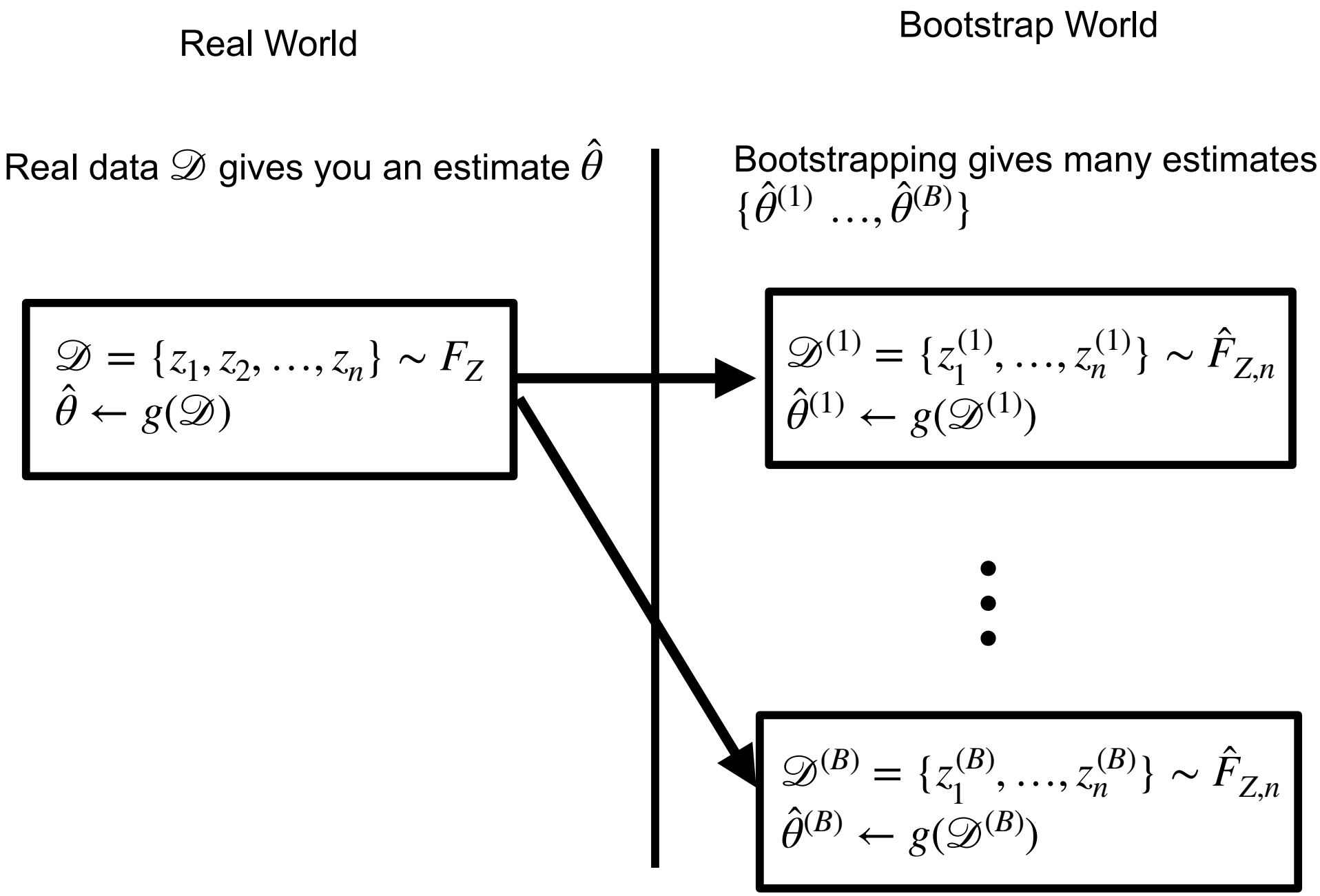
Bootstrap: Key idea

- Goal: Estimate uncertainty around an estimate $\hat{\theta} = g(\mathscr{D})$ from a dataset $\mathscr{D} = \{z_1, z_2, \ldots, z_n\} \sim F_Z$
- This would be easy if we had multiple datasets $\mathscr{D}$ drawn fresh from F_Z
- Idea: pretend that we have fresh draws from F_Z by **sampling with replacement** from $\mathscr{D}$
- Concretely,
  - for $b = 1,\ldots, B$,

Bootstrap: Key idea

- Goal: Estimate uncertainty around an estimate $\hat{\theta} = g(\mathscr{D})$ from a dataset $\mathscr{D} = \{z_1, z_2, \ldots, z_n\} \sim F_Z$
- This would be easy if we had multiple datasets $\mathscr{D}$ drawn fresh from F_Z
- Idea: pretend that we have fresh draws from F_Z by **sampling with replacement** from $\mathscr{D}$
- Concretely,
  - for $b = 1,\ldots, B$,
    - draw n samples with replacement from $\mathscr{D}$, and call them $\mathscr{D}^{(b)} = \{z_1^{(b)}, \ldots, z_n^{(b)}\} \sim \hat{F}_{Z,n}$
    - compute the estimate $\hat{\theta}^{(b)} \leftarrow g(\mathscr{D}^{(b)})$
  - use $\{\hat{\theta}^{(b)}\}_{b=1}^{B}$ to compute the confidence interval

Bootstrap: step 1: sampling with replacement

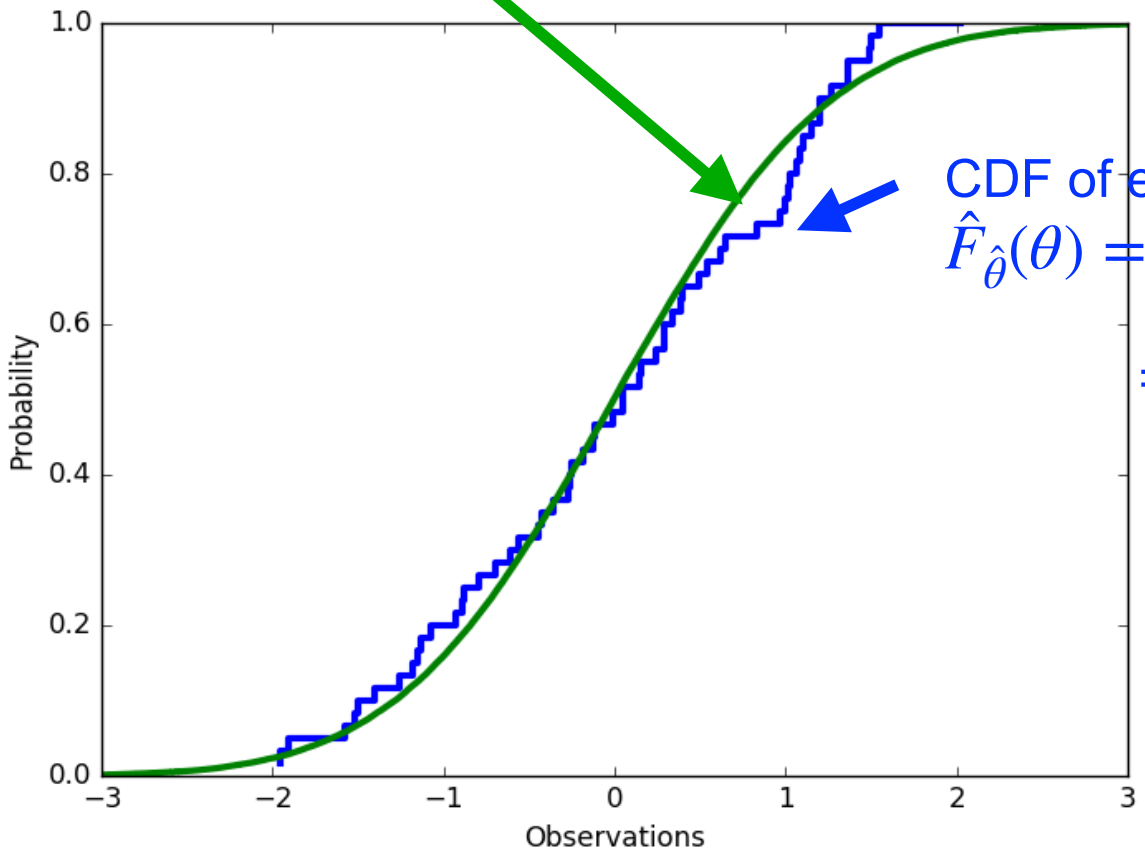


Bootstrap: step 2: confidence interval

- Bootstrap "sample" many fresh datasets $\{\mathscr{D}^{(1)}, \ldots, \mathscr{D}^{(B)}\}$, each of size n
- compute estimates $\{\hat{\theta}^{(1)} = g(\mathscr{D}^{(1)}) \ \ldots, \hat{\theta}^{(B)} = g(\mathscr{D}^{(B)})\}$ for each dataset

CDF of an unknown True distribution

$$F_{\hat{\theta}}(\theta) = P_{\mathscr{D}}(g(\mathscr{D}) \leq \theta)$$

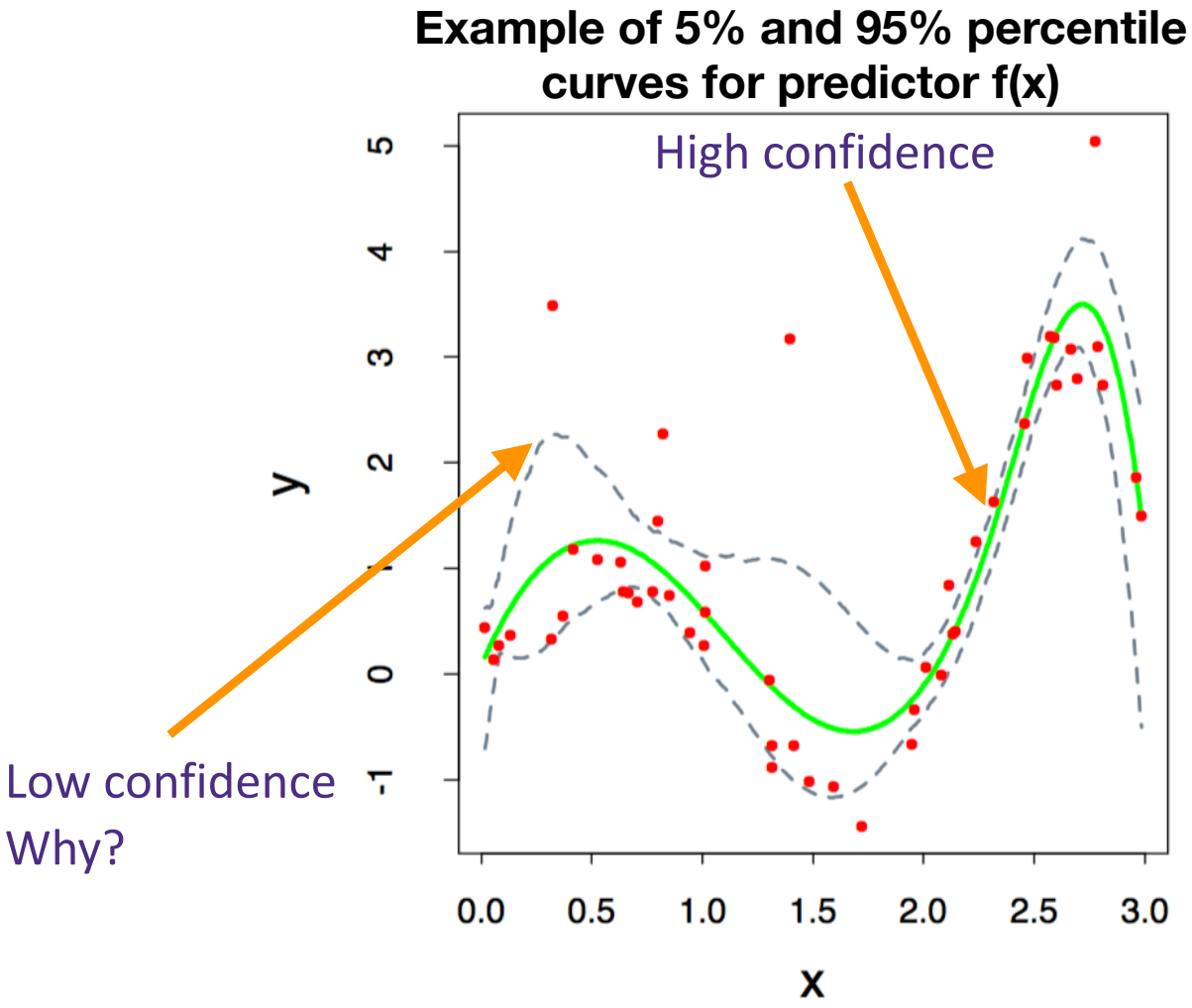


CDF of empirical observed distribution

$$\hat{F}_{\hat{\theta}}(\theta) = \hat{P}_B(g(\mathscr{D}) \leq \theta)$$

$$= \frac{1}{B}\sum_{b=1}^{B} \mathbf{I}\{\hat{\theta}^{(b)} \leq \theta\}$$

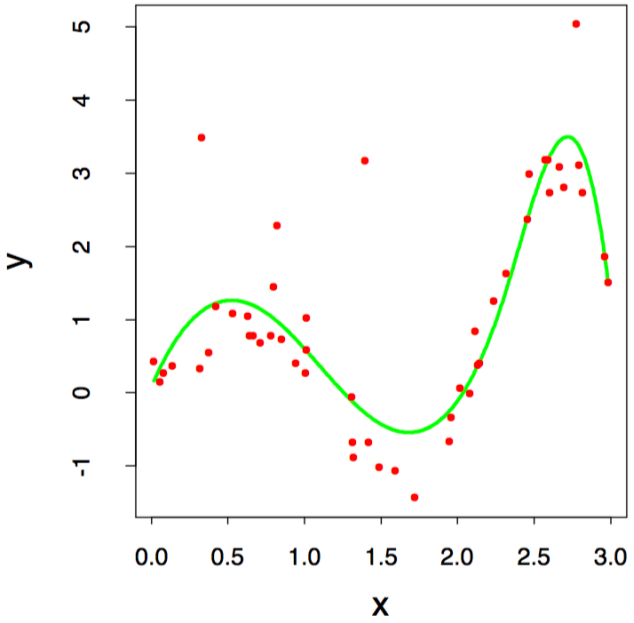
Example for 1-d regression



The green line is $f_{\mathcal{D}}(x)$, and the black dashed lines are the 5% and 95% percentiles at each x

bootstrap

training a single predictor



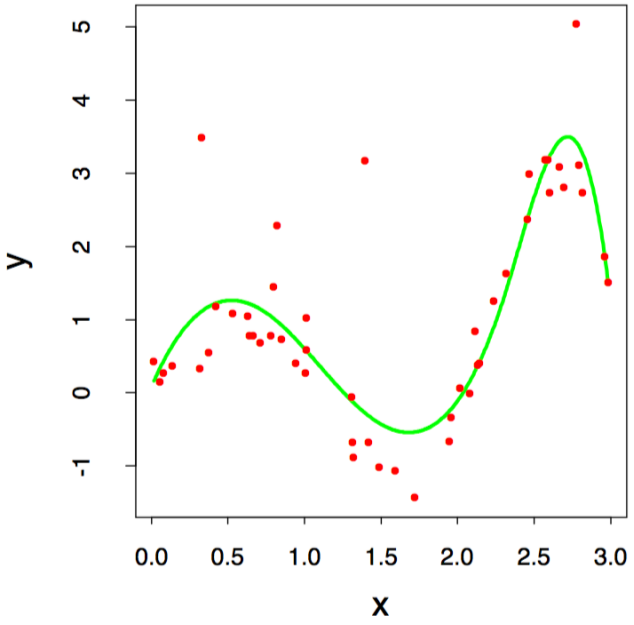
multiple bootstrapped predictors

90% confidence interval

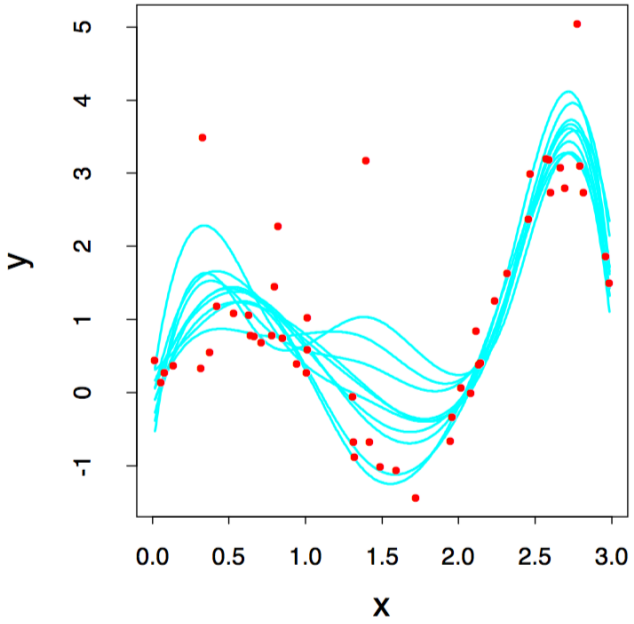
Figures from Hastie et al

bootstrap

training a single predictor



multiple bootstrapped predictors

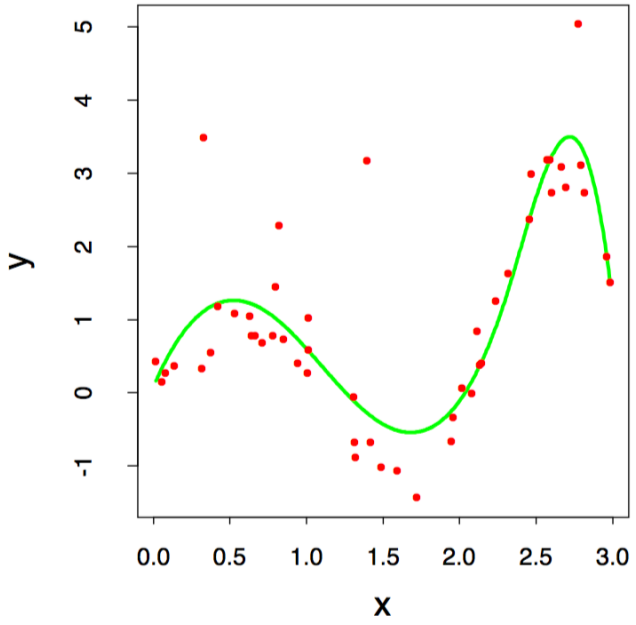


90% confidence interval

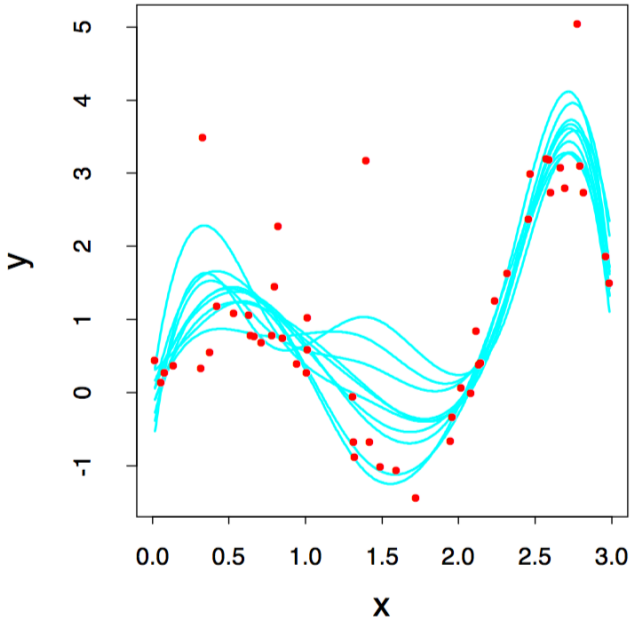
Figures from Hastie et al

bootstrap

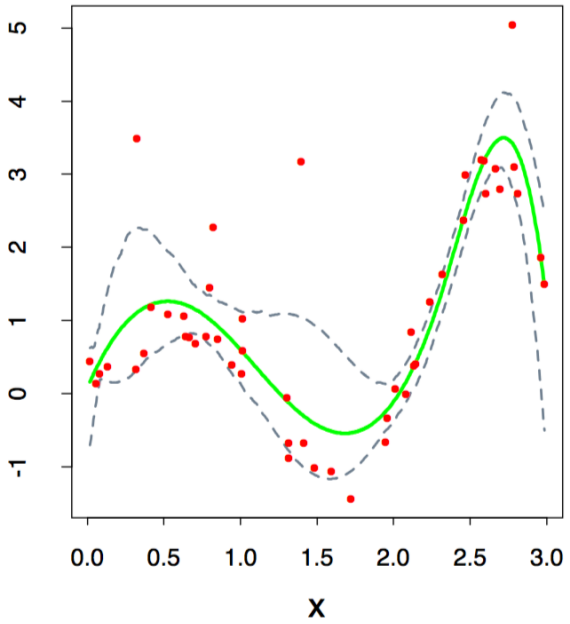
training a single predictor



multiple bootstrapped predictors



90% confidence interval



Figures from Hastie et al

Takeaways

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)

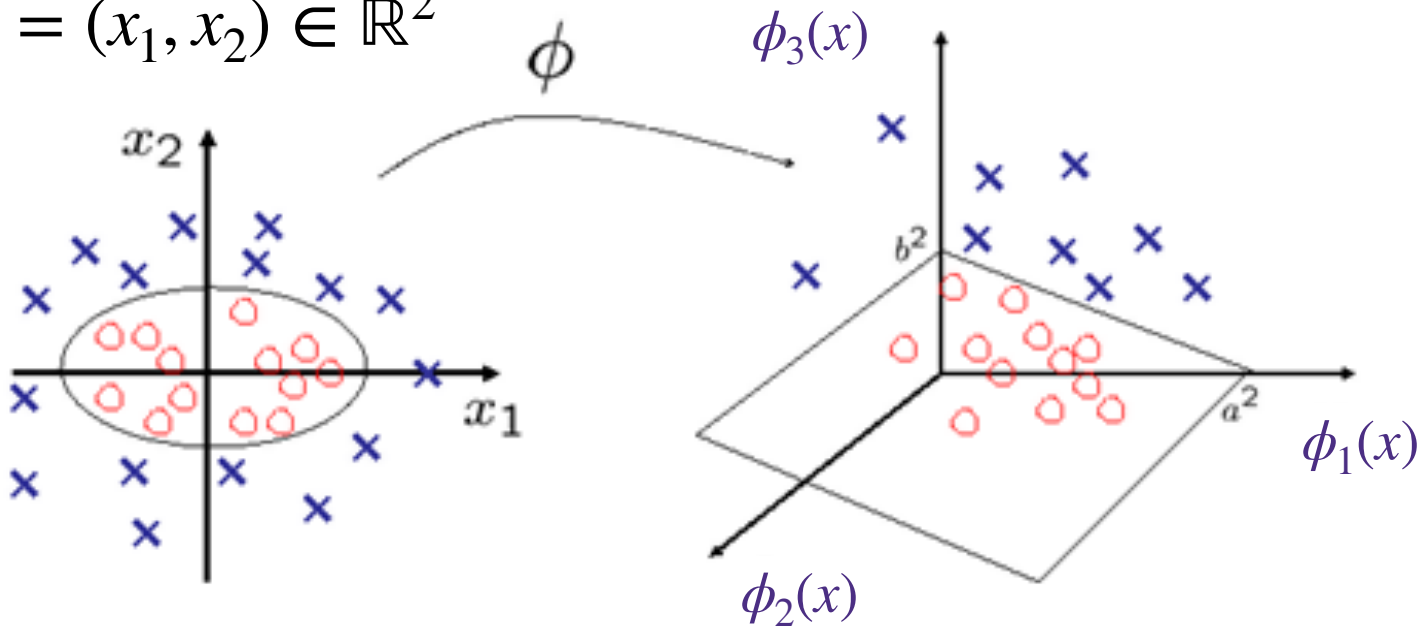
Kernel methods

-

What if the data is not linearly separable?

- Use features, for example,

$$x = (x_1, x_2) \in \mathbb{R}^2$$

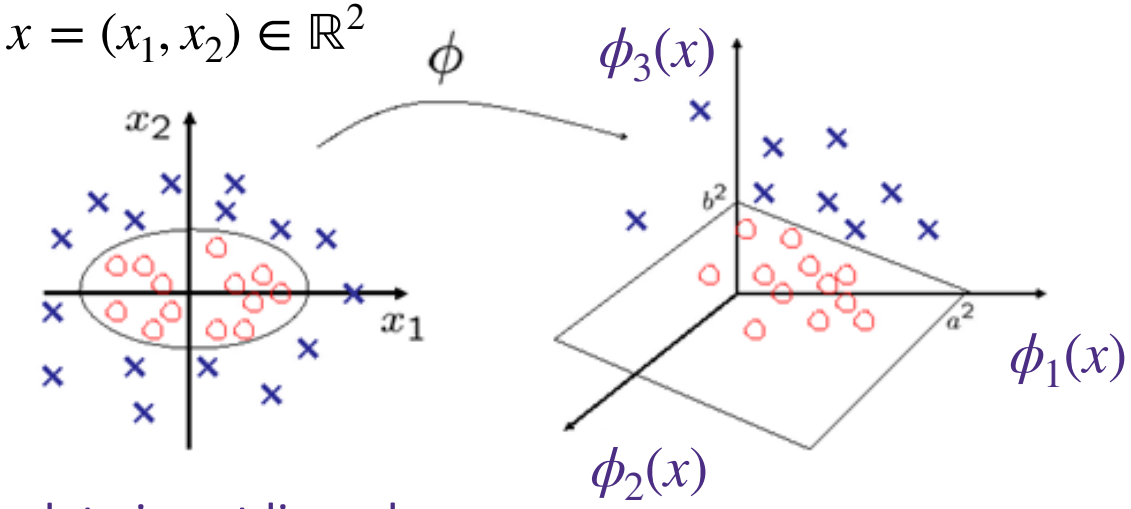


This data is not linearly separable

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?

What if the data is not linearly separable?

- Use features, for example,



This data is not linearly separable

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?

- Generally, in high dimensional feature space, it is easier to linearly separate different classes
- However, it is hard to know which feature map will work for given data
- So the rule of thumb is to use high-dimensional features and hope that the algorithm will automatically pick the right set of features
- What is wrong with this approach?

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 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:

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

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

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

- 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

A function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ 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)

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

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

A fundamental trick in ML: use kernels

A function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ 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)

- 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
$\{K(x_i, x_j)\}_{i,j \in \{1,\ldots,n\}}$

[Homework 3 Problem A2 tests this skill]

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 $K(x, x') = (x^T x')^2$

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)

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

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

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

- 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:

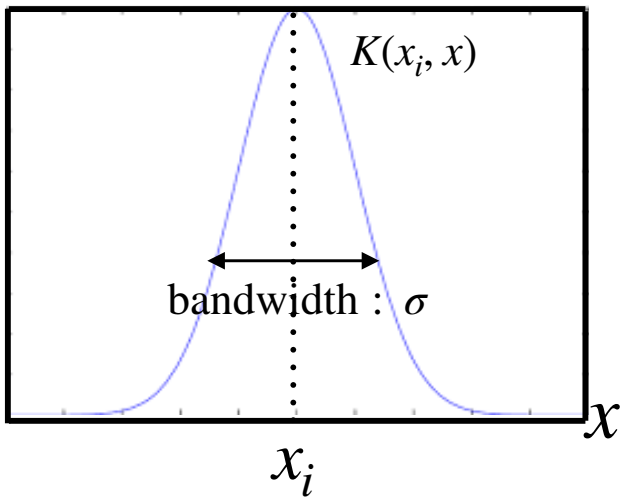
$$w = \sum_{i=1}^{n} \alpha_i \phi(x_i) \;=\; \phi(X)^T \alpha$$

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

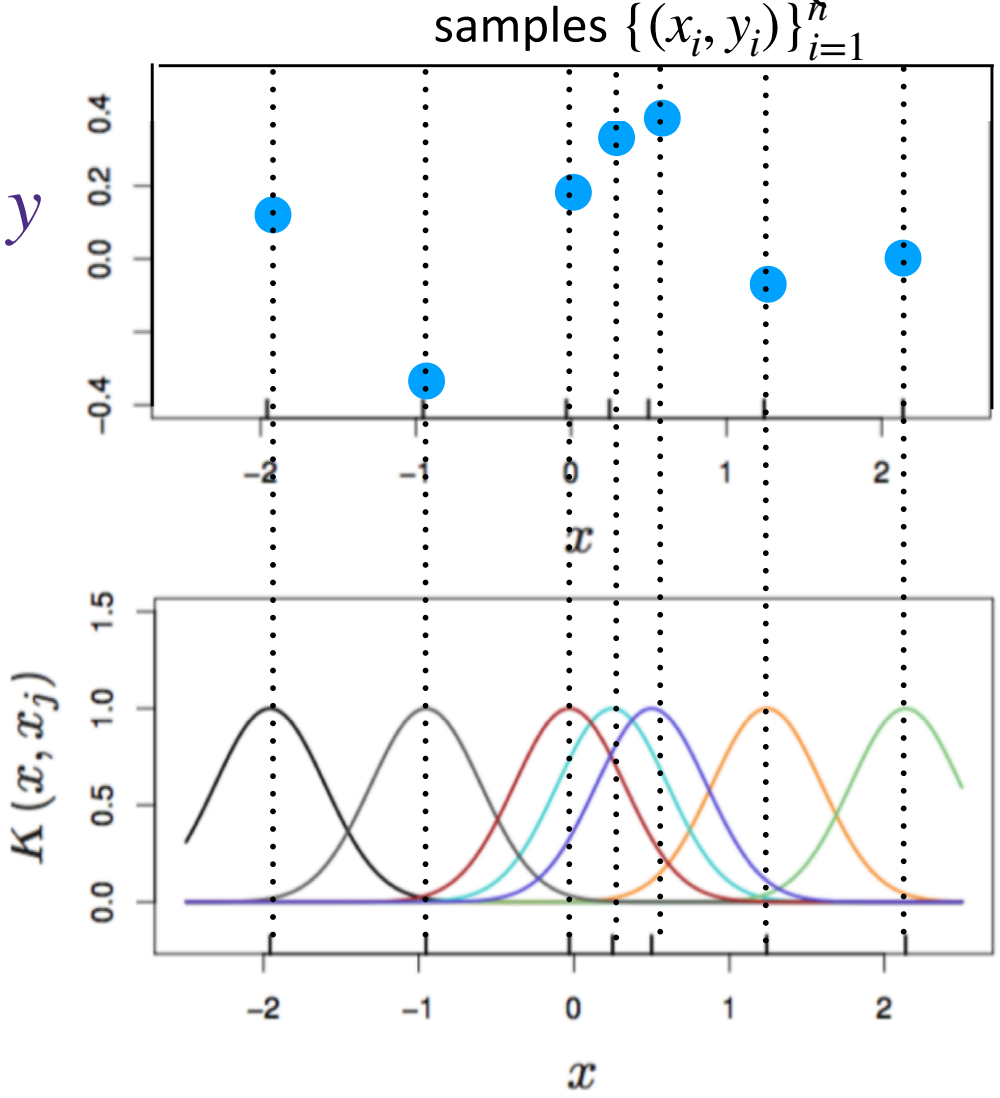
- Kernel method is not limited to linear Ridge regression, but also applies to a broad class of methods including the kernel logistic regression

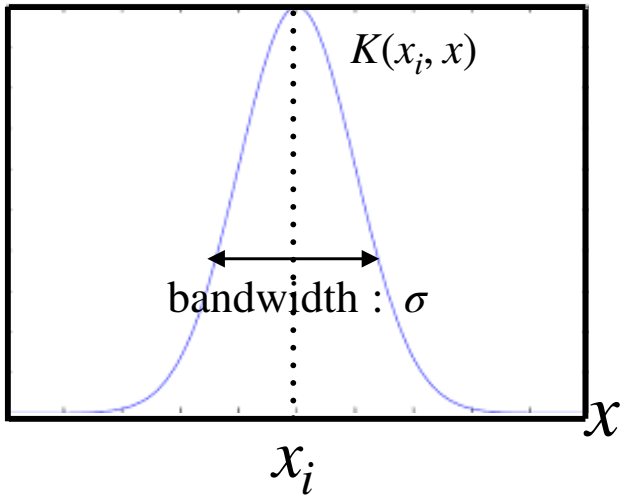
[Homework 3 Problem A3 implements kernel regression for polynomial kernel and RBF kernel]

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



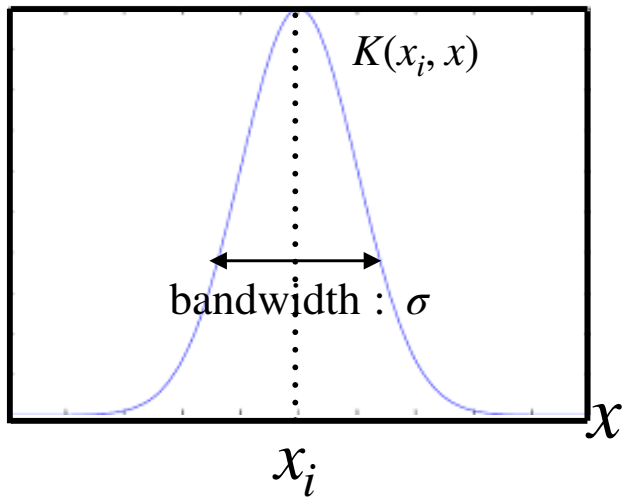
RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$

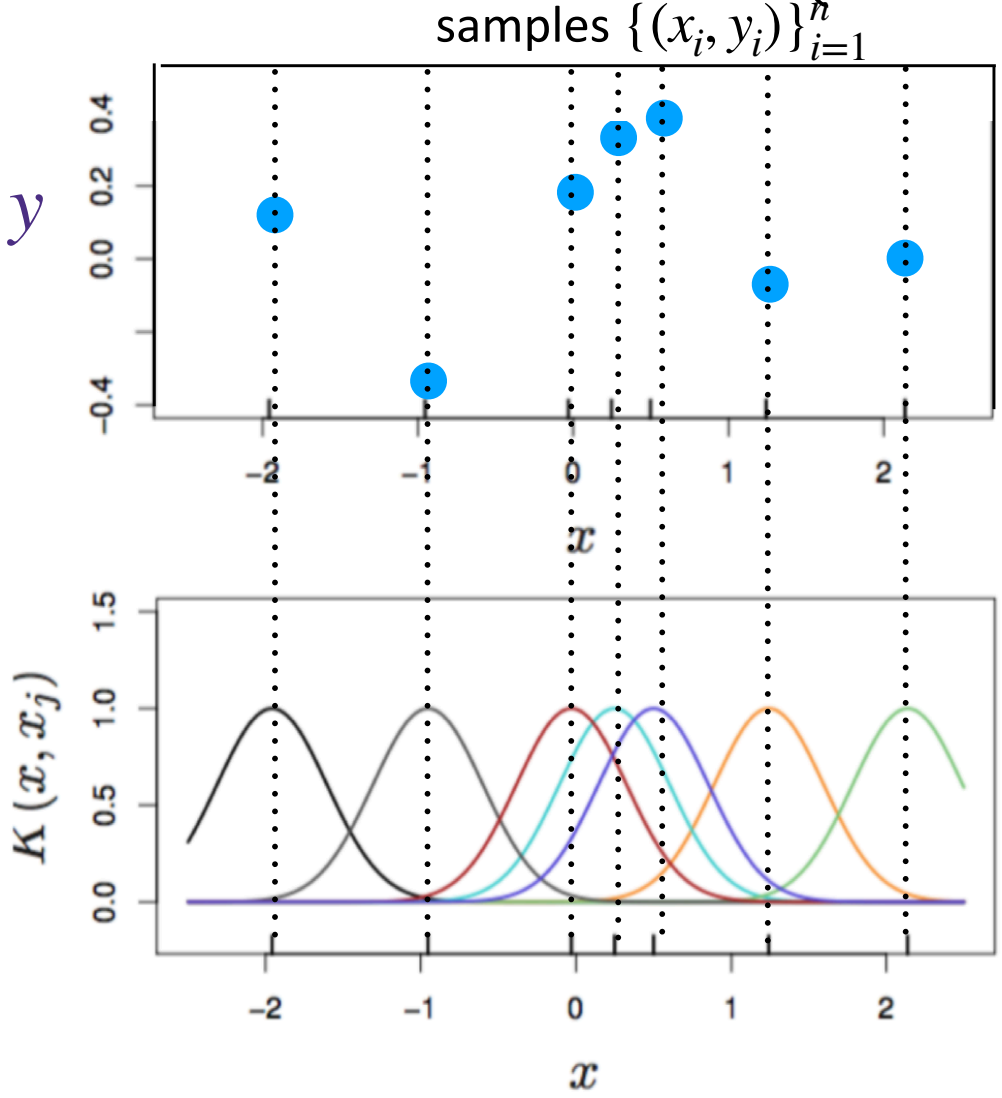


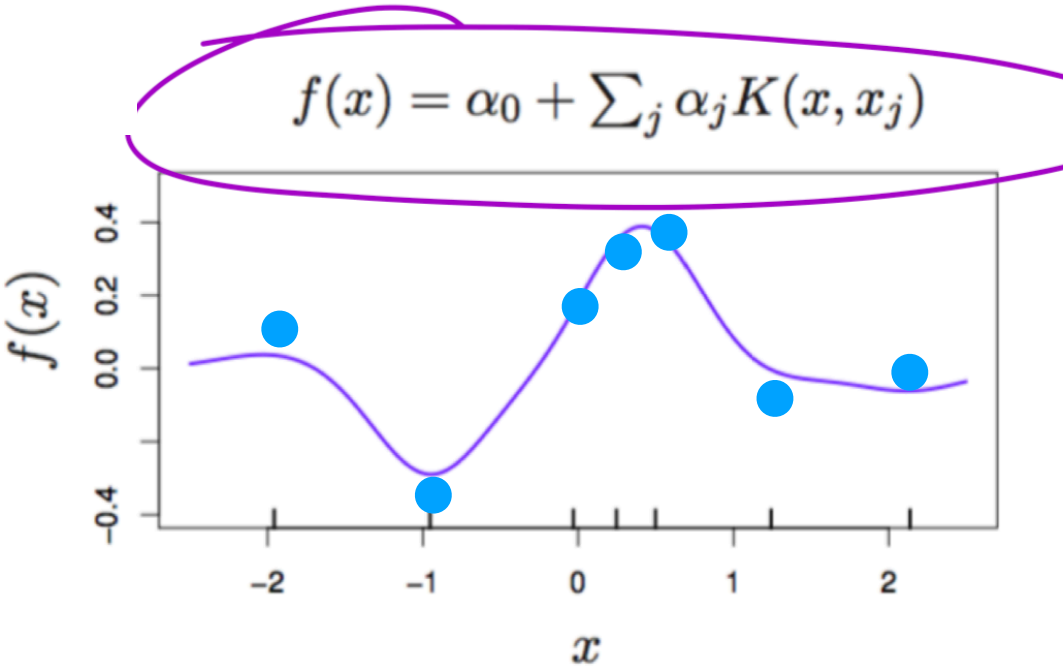


- predictor $f(x) = \sum_{i=1}^{n} \alpha_i K(x_i, x)$ is taking weighted sum of n kernel functions centered at each sample points

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







- 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

Why do we need regularization when using kernels?

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

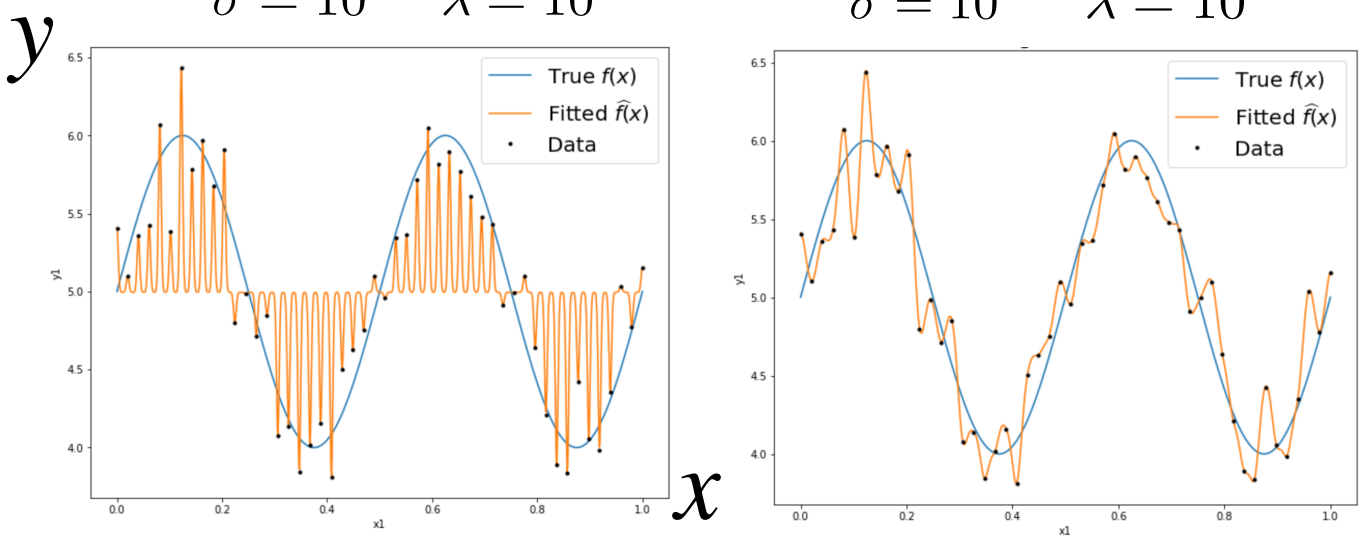
- So $\mathbf{K}$ is invertible and $\widehat{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_{n \times n})^{-1} \mathbf{y}$ is well defined.
- What if $\lambda = 0$? What goes wrong?

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

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

$\sigma = 10^{-3}$ $\lambda = 10^{-4}$ $\sigma = 10^{-2}$ $\lambda = 10^{-4}$

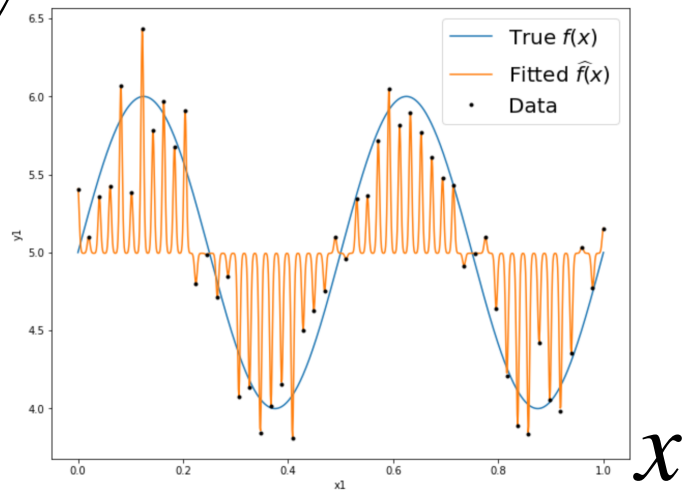


$$\widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i K(x_i, x)$$

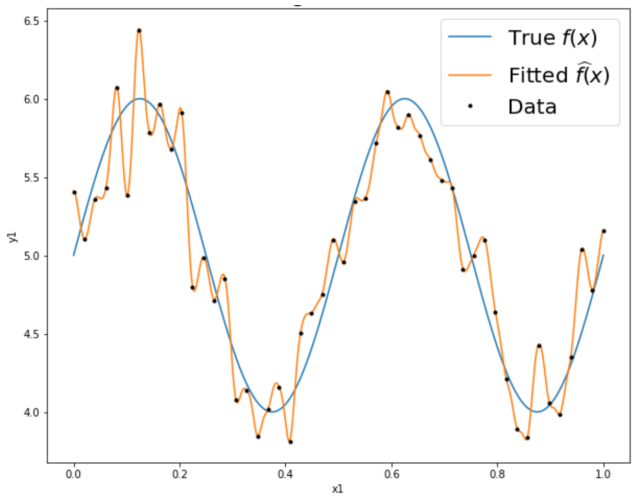
RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$

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

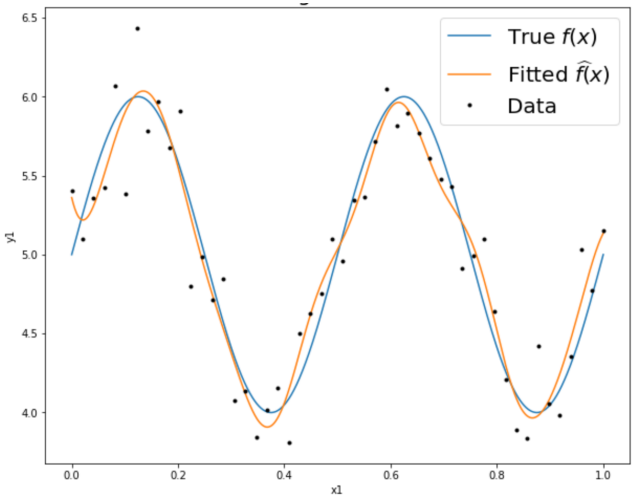
$\sigma = 10^{-3}$ $\lambda = 10^{-4}$



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

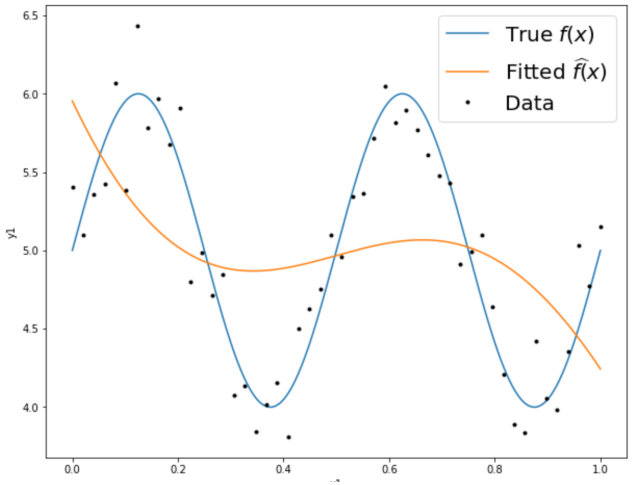


$\sigma = 10^{-1}$ $\lambda = 10^{-4}$



y x

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

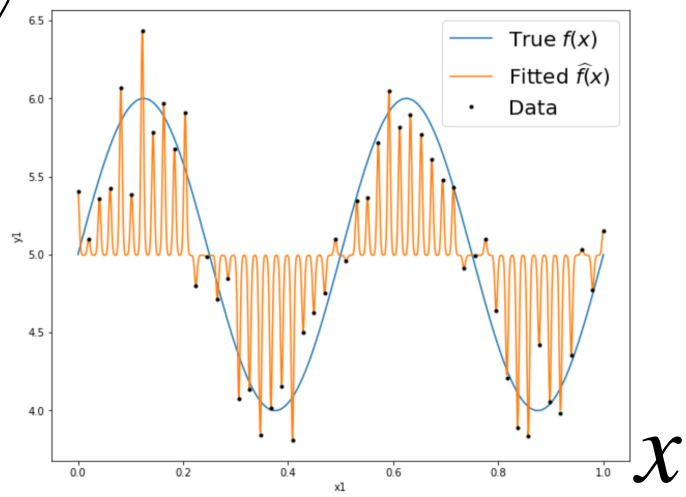


$$\widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i K(x_i, x)$$

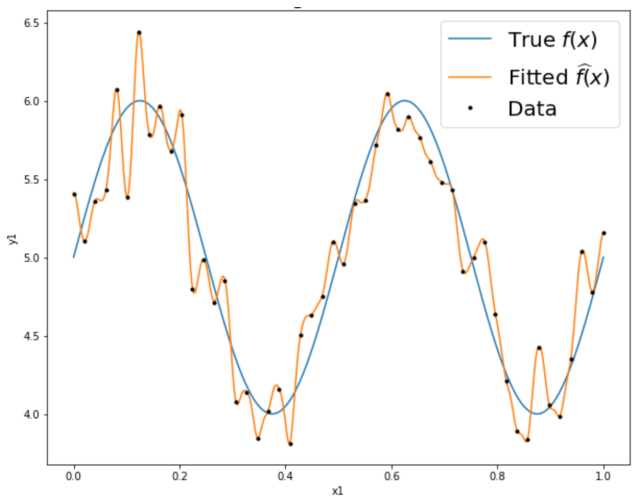
RBF kernel $k(x_i, x) = \exp\left\{ -\frac{\|x_i - x\|_2^2}{2\sigma^2} \right\}$

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

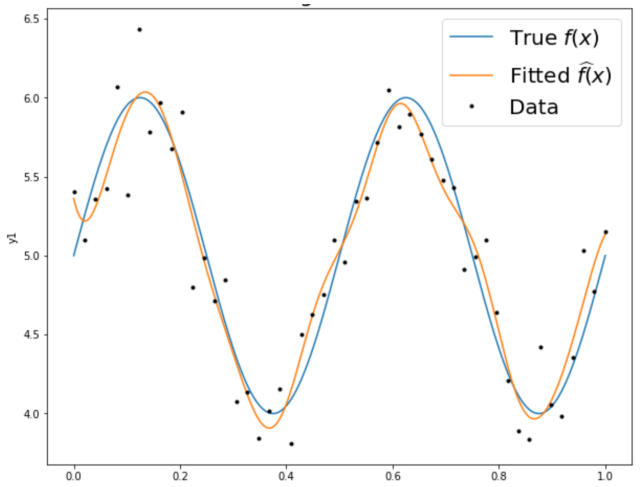
$\sigma = 10^{-3}$ $\lambda = 10^{-4}$



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

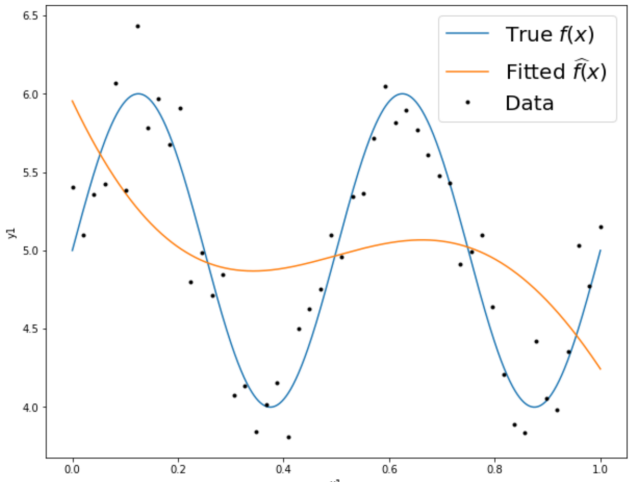


$\sigma = 10^{-1}$ $\lambda = 10^{-4}$



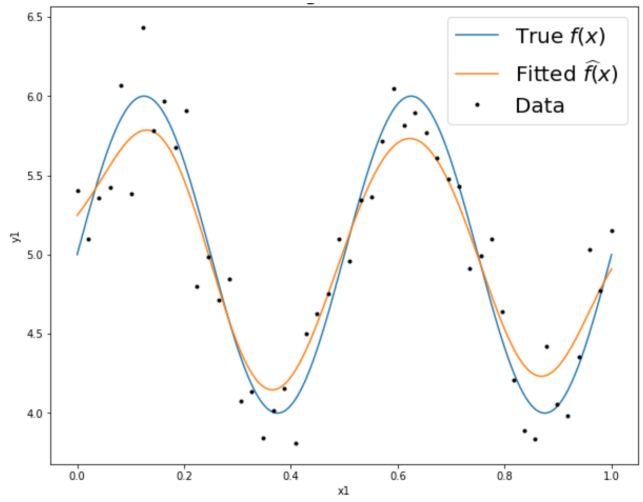
y x

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



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

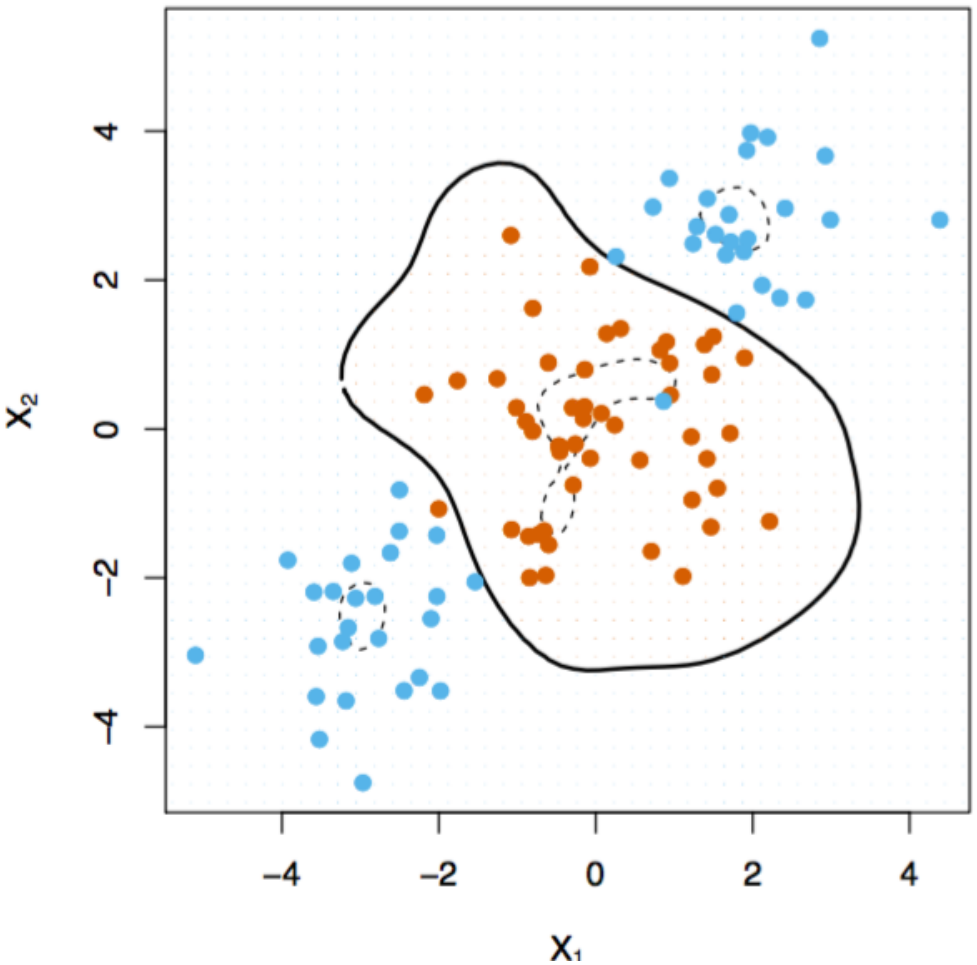
$\sigma = 10^{-1}$ $\lambda = 10^{-0}$



RBF kernel and random features

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

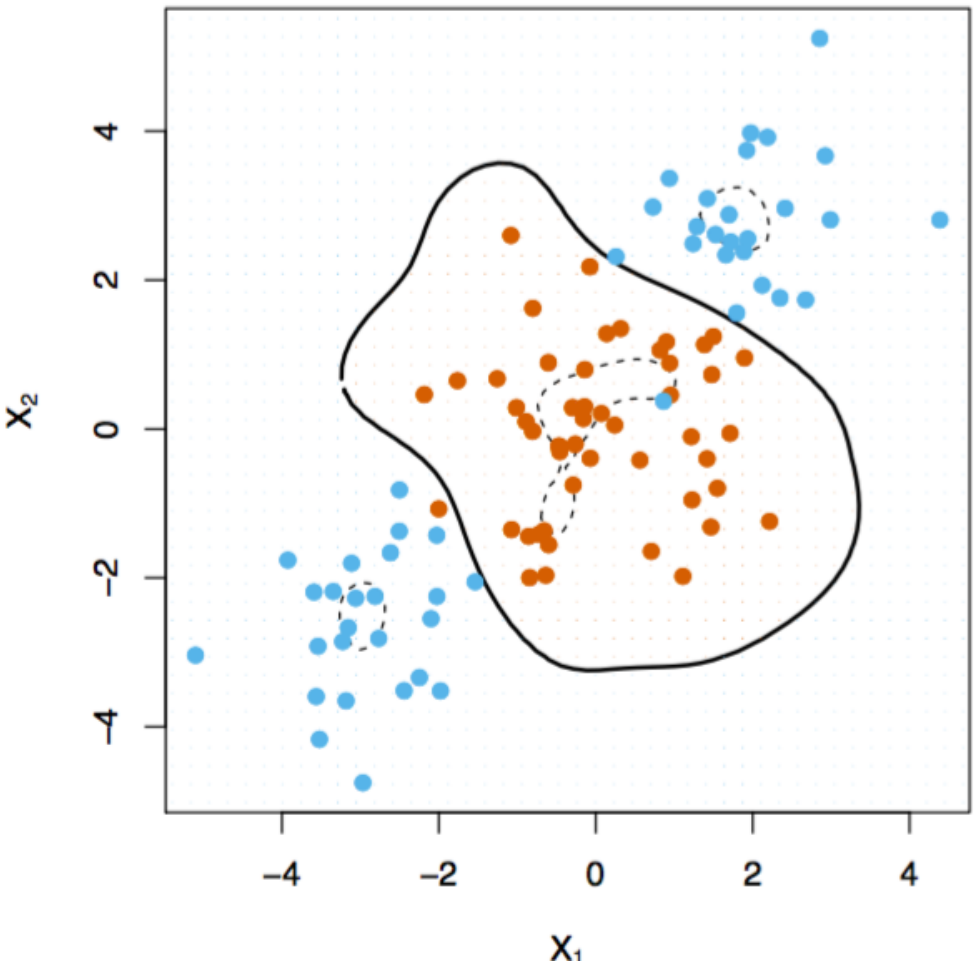
Bandwidth σ is large enough



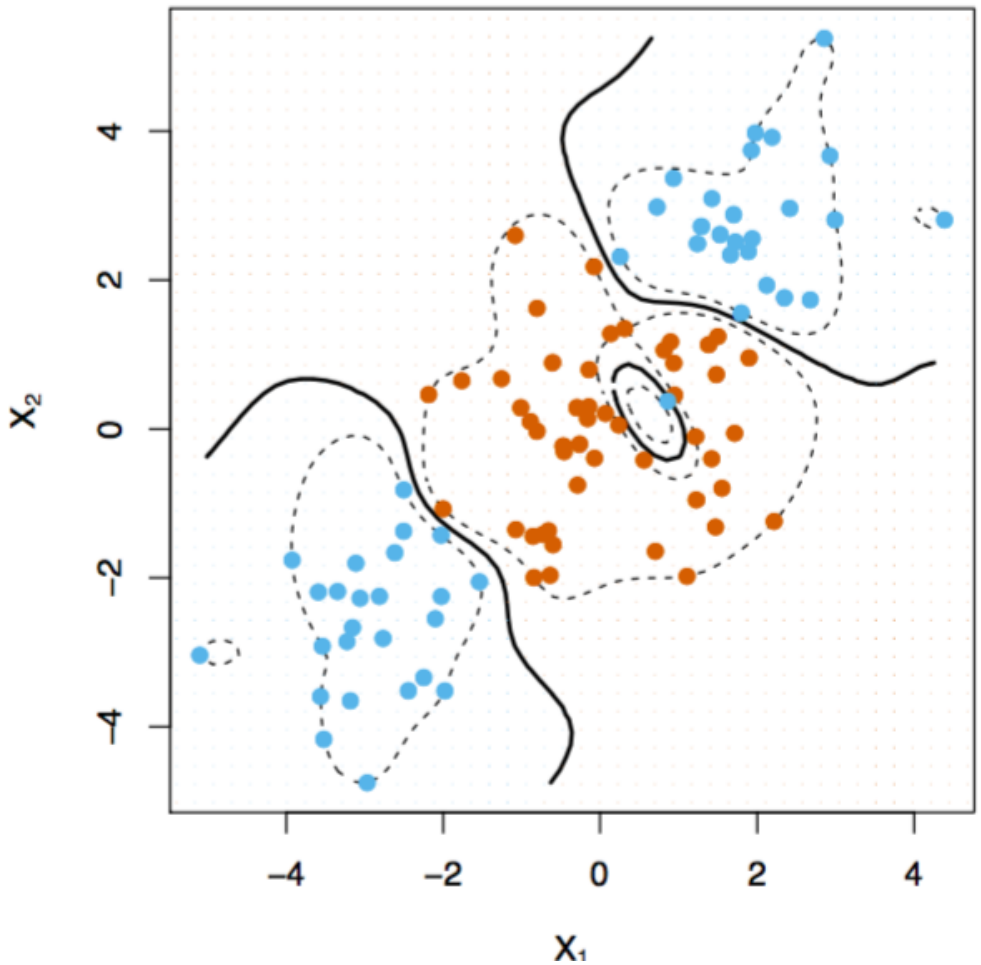
RBF kernel and random features

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

Bandwidth σ is large enough



Bandwidth σ is small



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?

Questions?