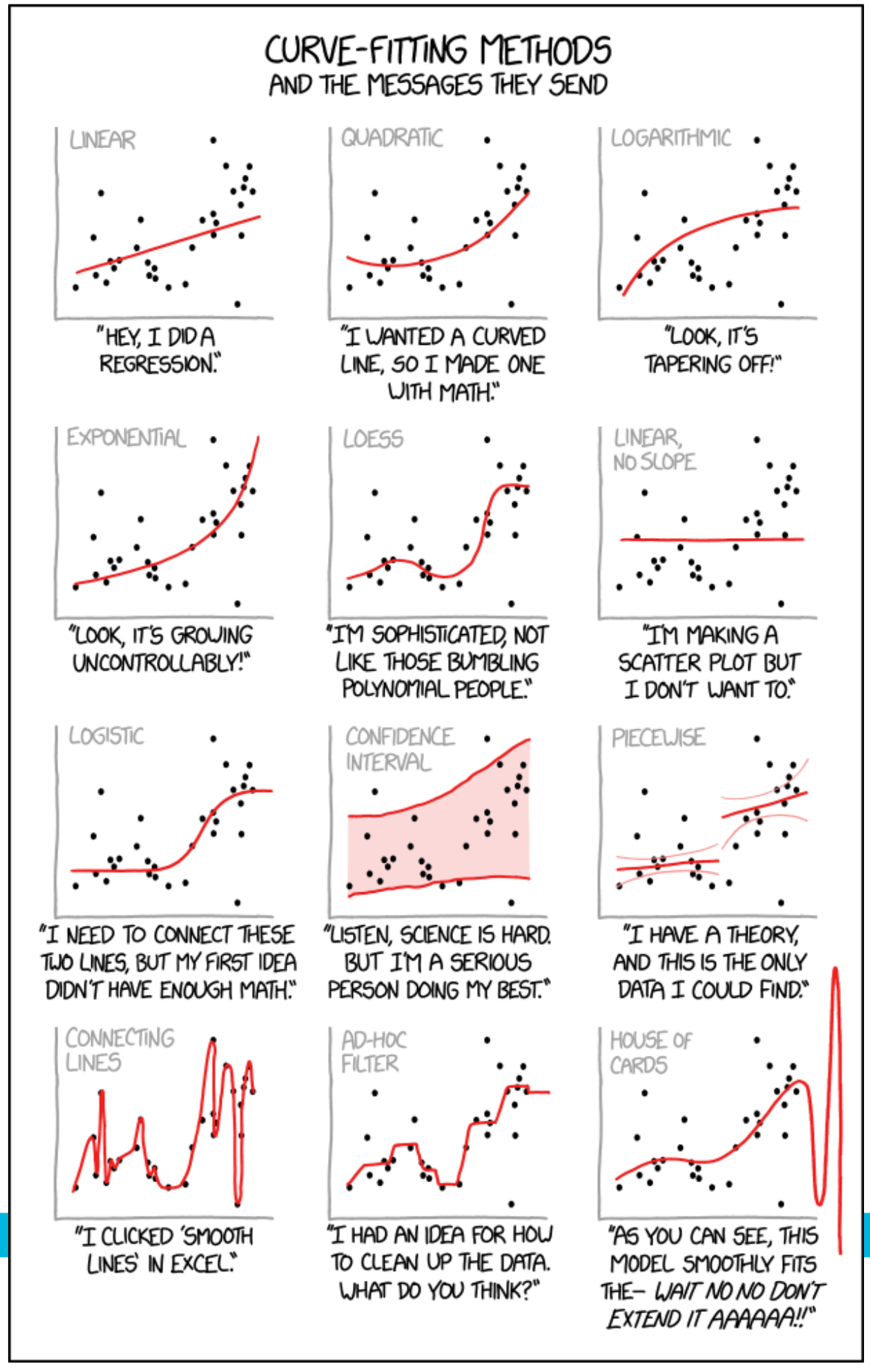
Nonlinear Regression

D.M.J. Tax

xkcd 2048



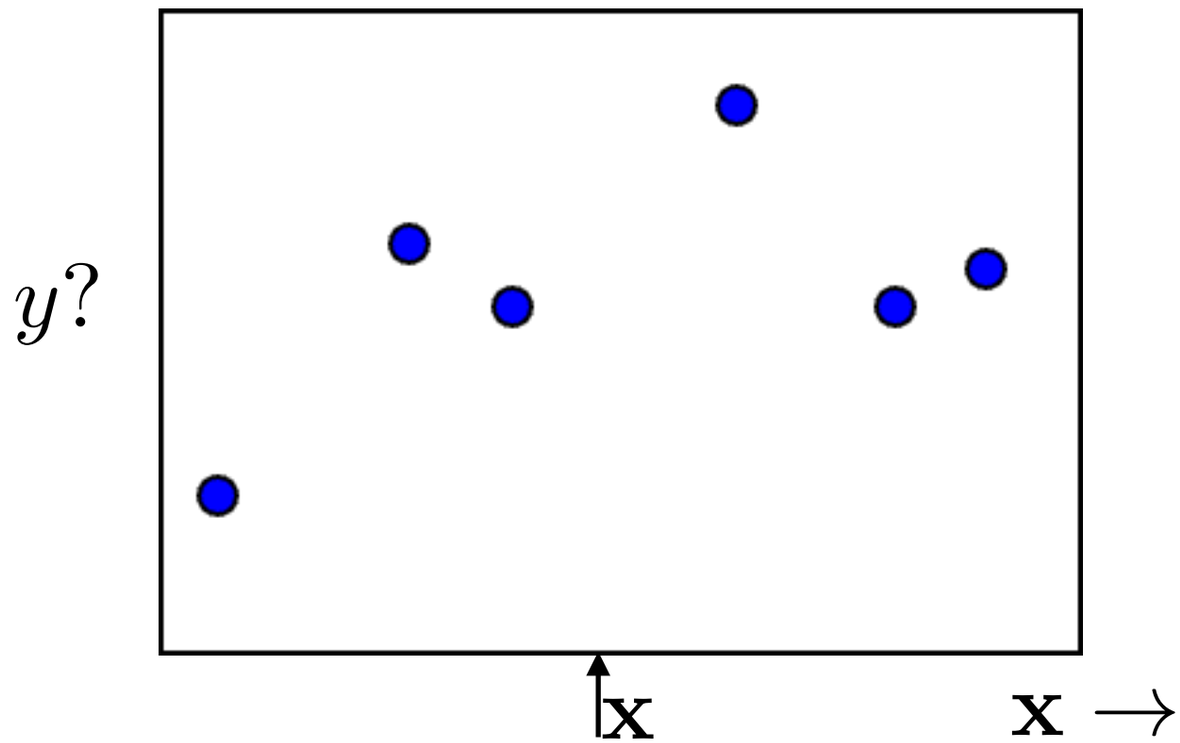
Contents

- Nonlinear 1D regression: (Kernel) Smoothers
- Local linear regression
- Kernel ridge regression
- (Gaussian Processes)
- Neural networks (short)
- Peaking and the effective number of parameters
- Conclusions

- See chapter 6 of 'Pattern Recognition and Machine Learning' by Bishop, up to section 6.4.3.

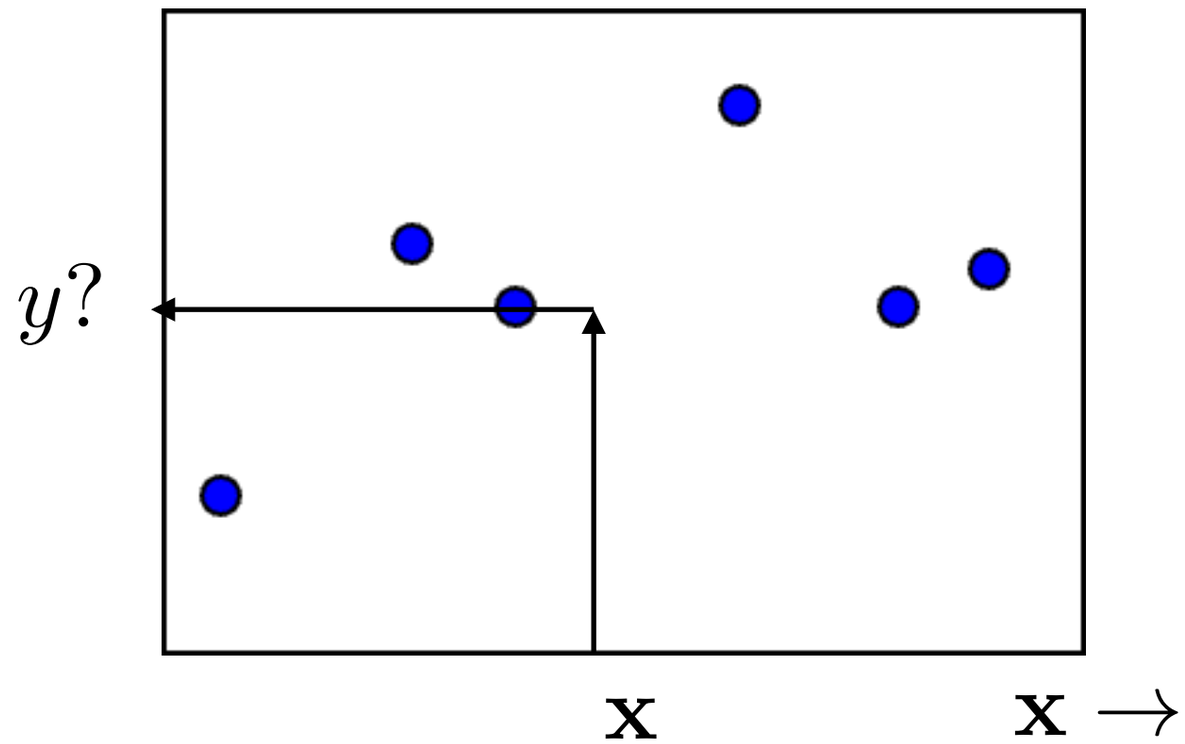
Nonlinear Regression

- Assume you don't think a linear regression is sufficient, what to do?



Nonlinear Regression

- Assume you don't think a linear regression is sufficient, what to do?



Nonlinear regression

- Given training data $(\mathbf{x}_1; y_1), ..., (\mathbf{x}_N; y_N)$ regressors estimate the curve

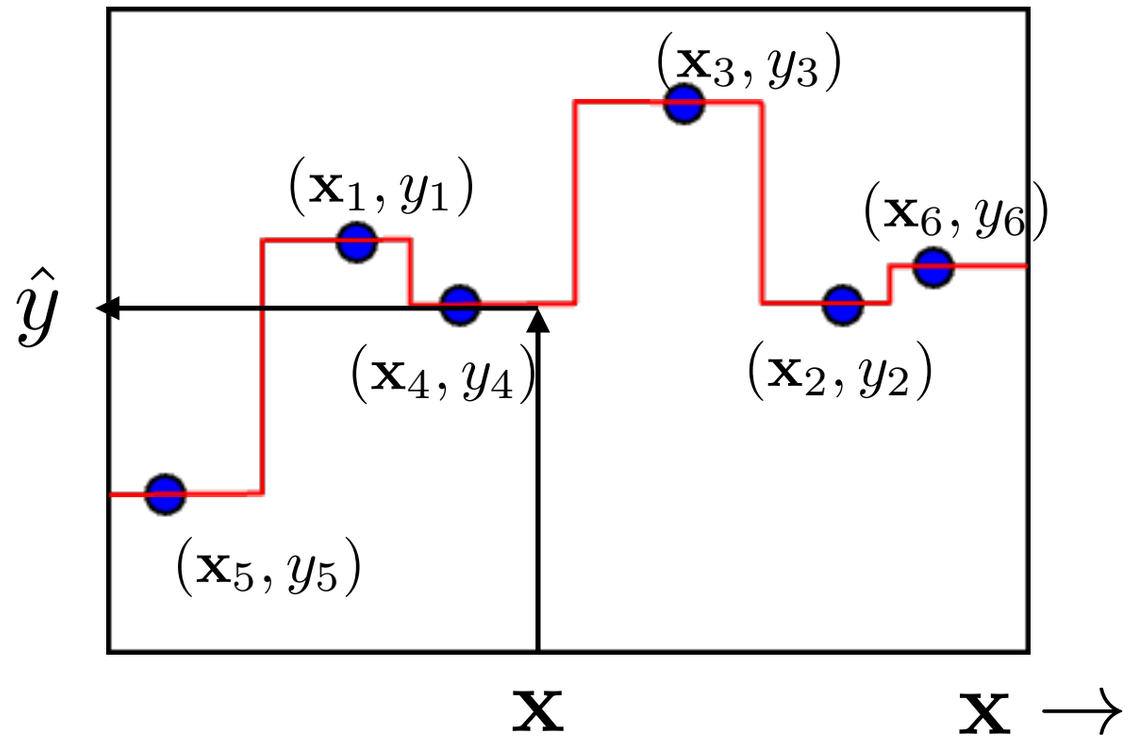
$$\hat{y}(\mathbf{x}) = E[y|\mathbf{x}]$$

- Concentrate on one-dimensional Smoothers: 'simple', 'kernel', 'local regression'

- Smoothers do Lazy learning: remember all training samples, estimate output from scratch for every new test sample

Simple smoothers

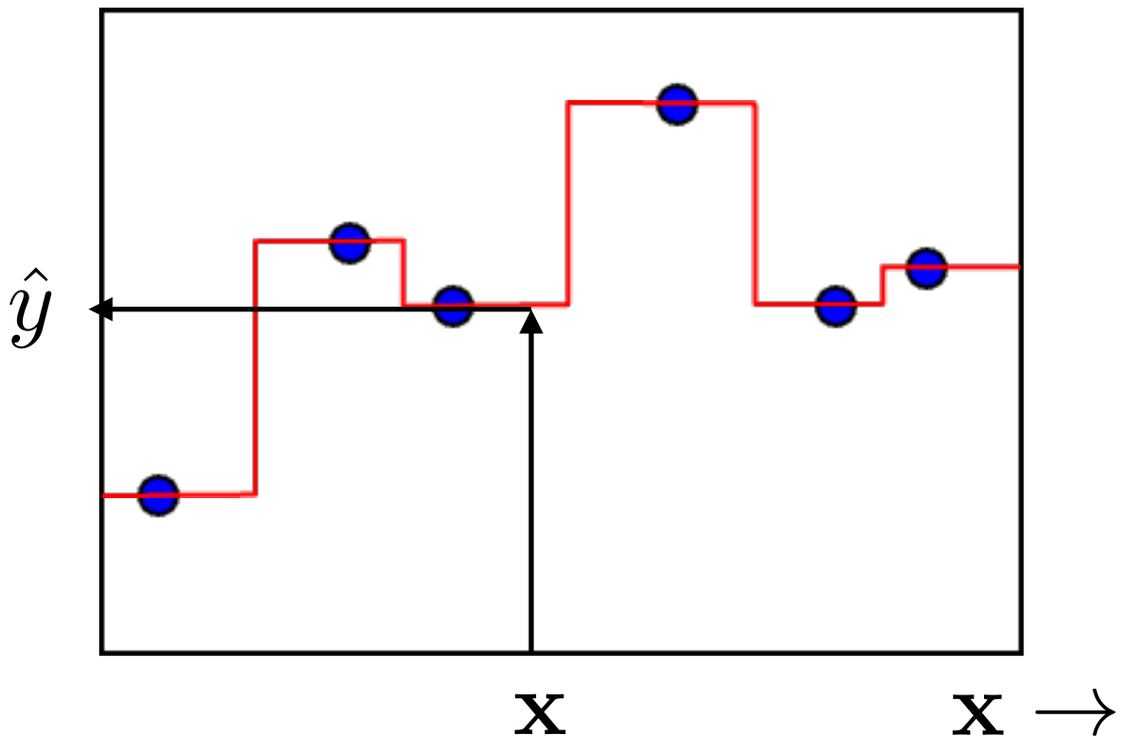
- k-nearest neighbor smoother, k=1



- How to formalize it?

Simple smoothers

- k-nearest neighbor smoother, k=1

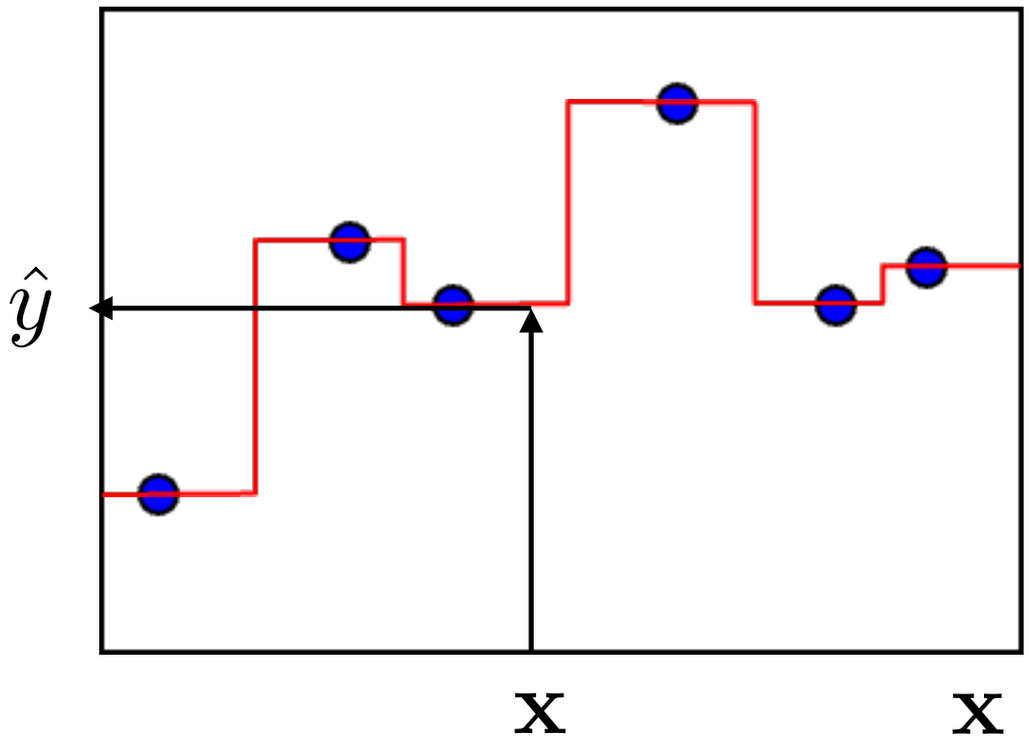


$$\hat{y}(\mathbf{x}) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x})} y_i$$

- where $\mathcal{N}_k(\mathbf{x})$ is the set of the k-nearest neighbors of **x** in the training set

Simple smoothers

- k-nearest neighbor smoother, k=1

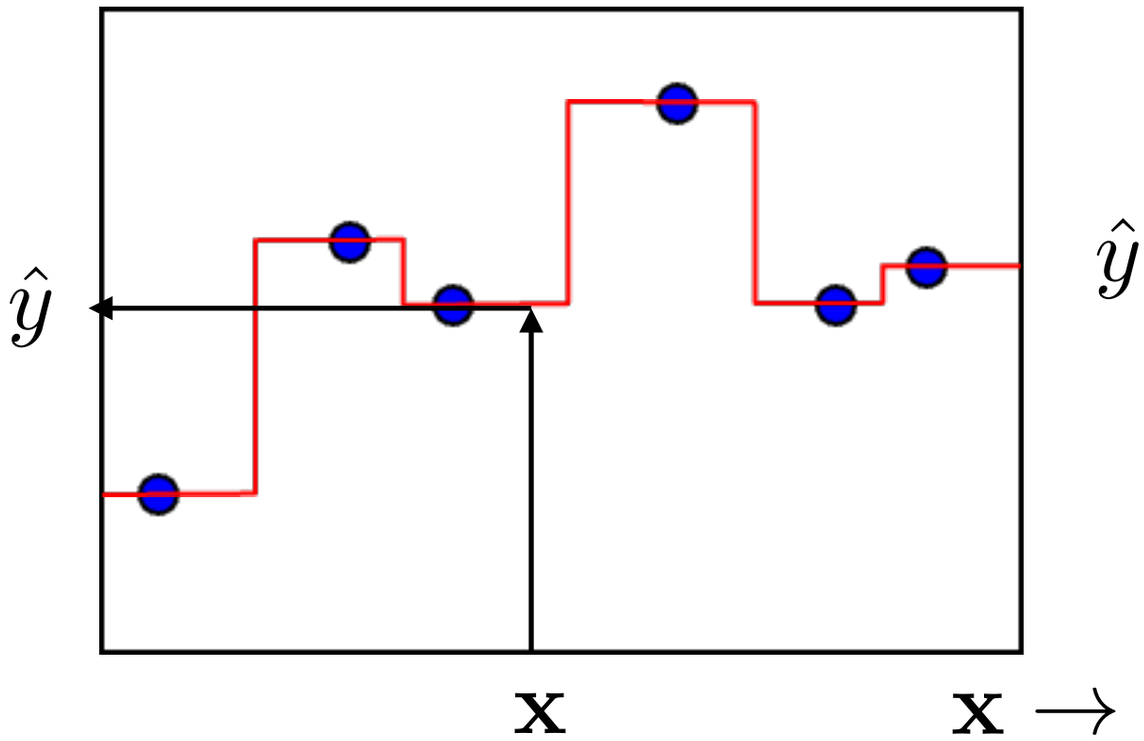


$$\hat{y}(\mathbf{x}) = \frac{1}{k} \sum_{i=1}^{N} \mathbb{I}(\mathbf{x}_i \in \mathcal{N}_k(\mathbf{x})) y_i$$

- where $\mathcal{N}_k(\mathbf{x})$ is the set of the k-nearest neighbors of **x** in the training set

Simple smoothers

- k-nearest neighbor smoother, k=1



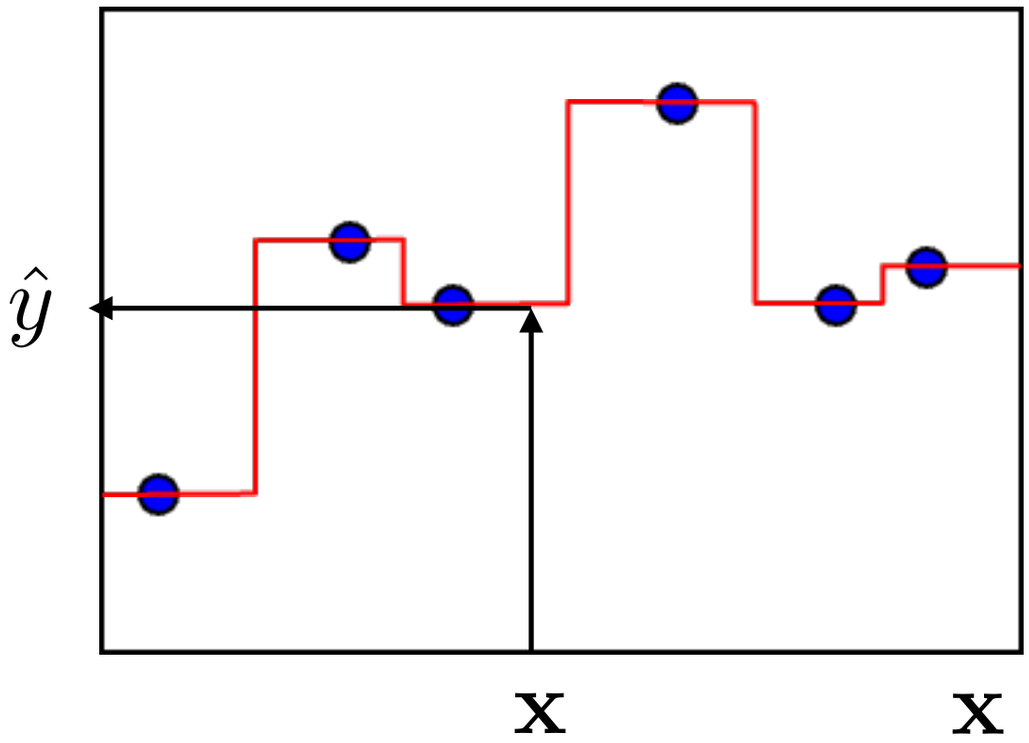
$$\hat{y}(\mathbf{x}) = \frac{1}{k}\sum_{i=1}^{N} \mathbb{I}(\mathbf{x}_i \in \mathcal{N}_k(\mathbf{x}))y_i$$

$$= \frac{\sum_{i=1}^{N} \mathbb{I}(\mathbf{x}_i \in \mathcal{N}_k(\mathbf{x}))y_i}{\sum_j \mathbb{I}(\mathbf{x}_j \in \mathcal{N}_k(\mathbf{x}))}$$

- where $\mathcal{N}_k(\mathbf{x})$ is the set of the k-nearest neighbors of **x** in the training set

Simple smoothers

- 'Nadaraya-Watson model', or 'kernel regression'



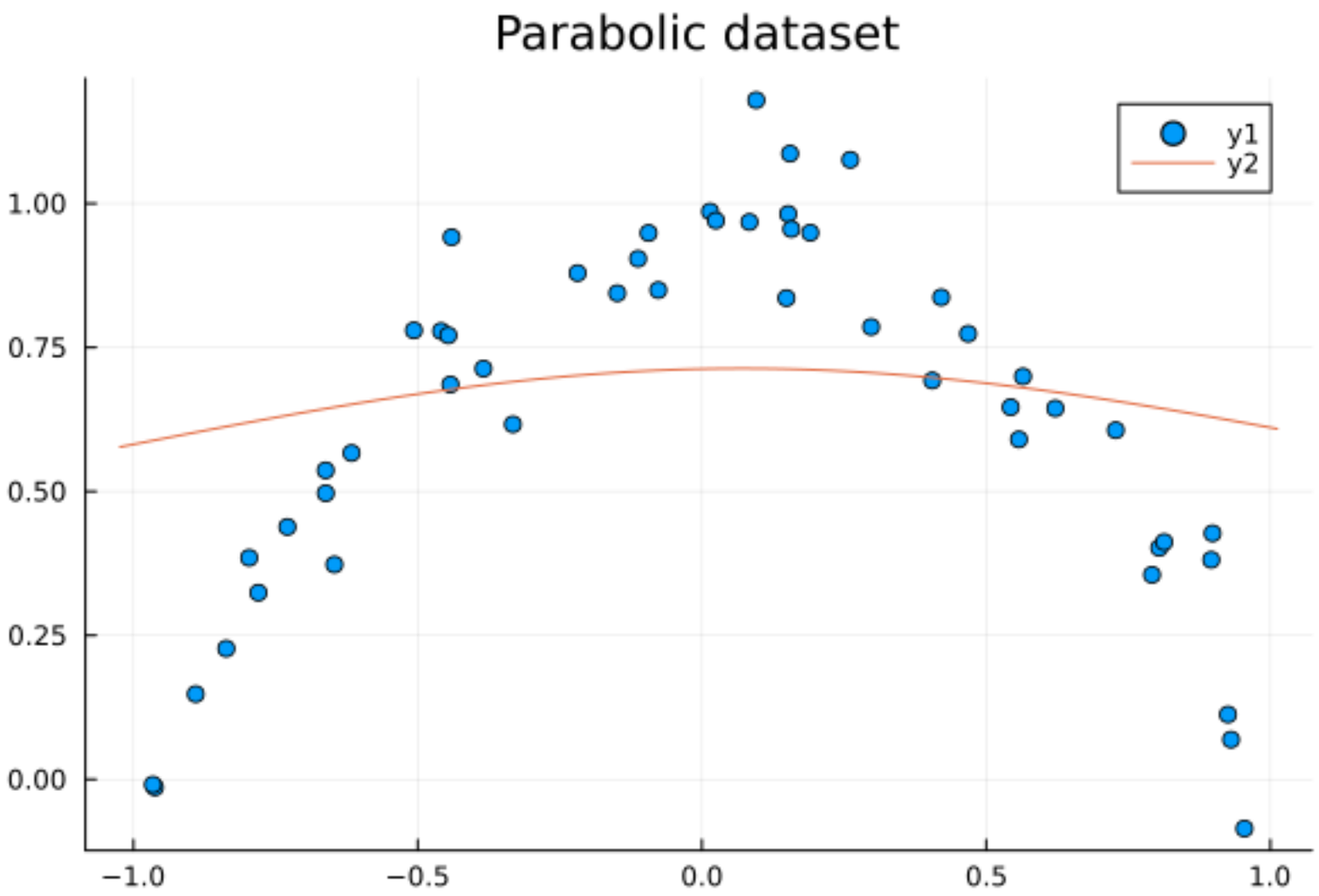
$$\hat{y}(\mathbf{x}) = \frac{\sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i) y_i}{\sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i)}$$

- where $K(\mathbf{x}, \mathbf{x}_i)$ is any kernel function (for instance, a Gaussian function)

Kernel smoother

- Use Gaussian kernel

$$K(\mathbf{x}, \mathbf{x}_i) = \frac{1}{Z} \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{2h^2}\right)$$

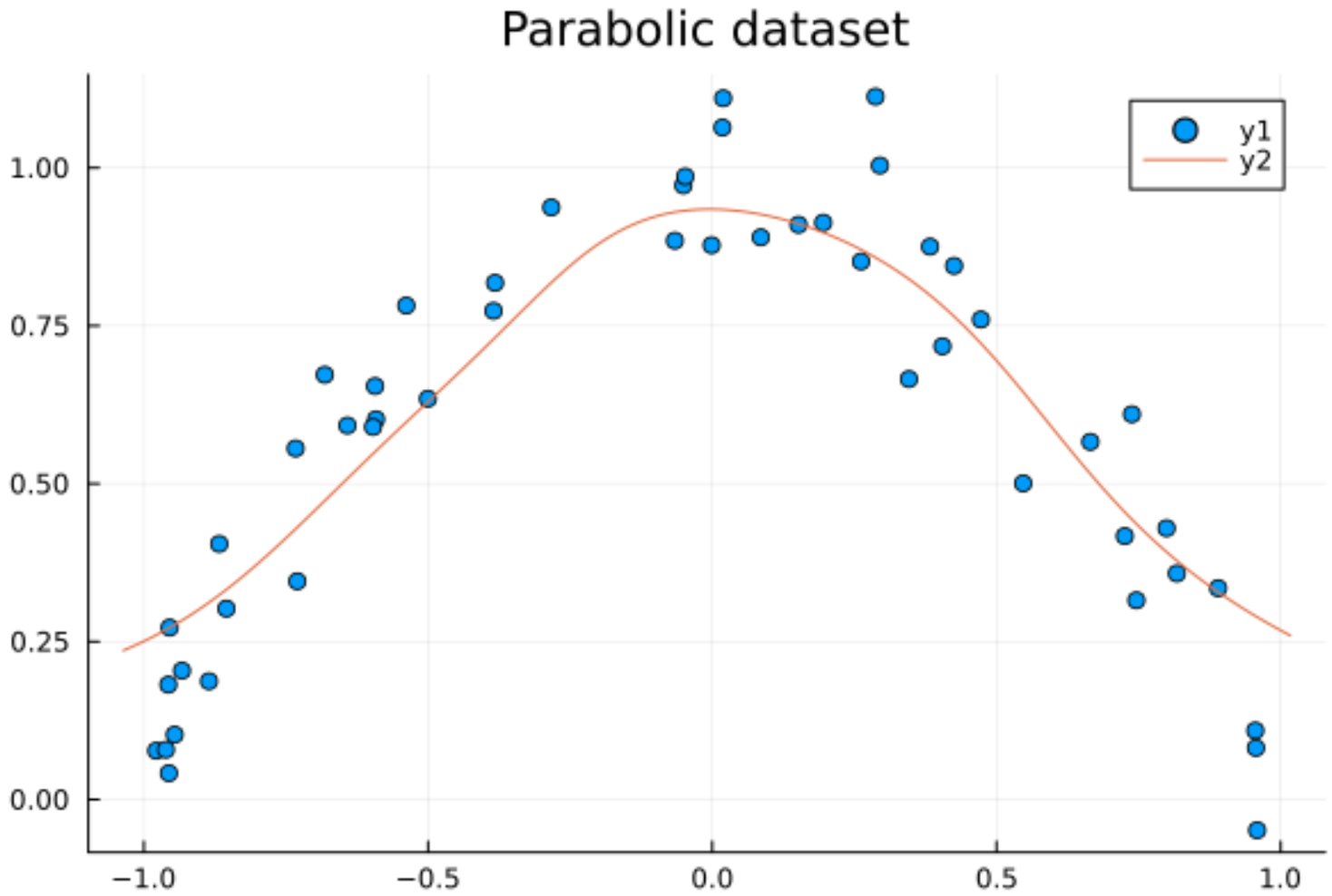


$h = 1$

Kernel smoother

- Use Gaussian kernel

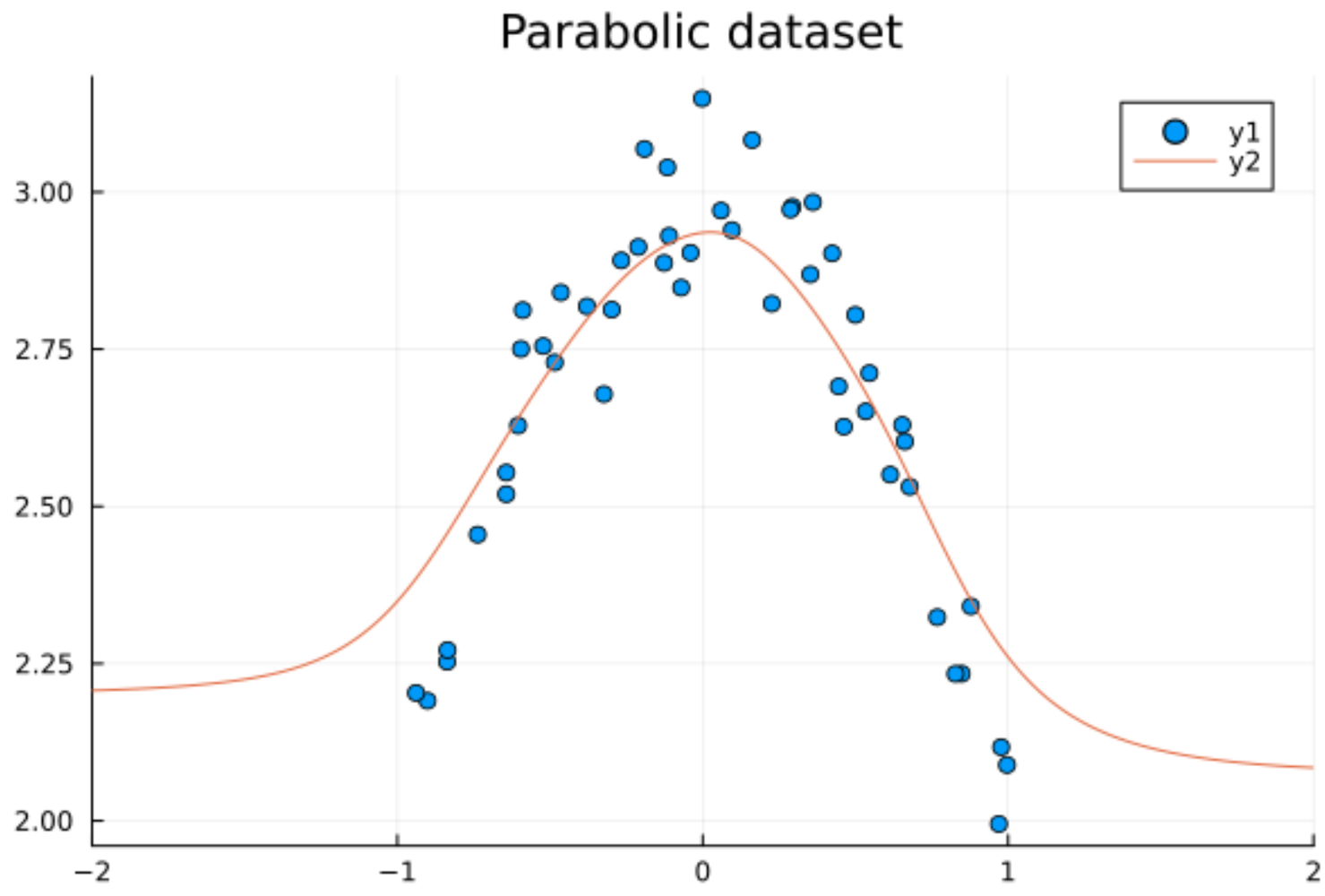
$$K(\mathbf{x}, \mathbf{x}_i) = \frac{1}{Z} \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{2h^2}\right)$$



$h = 0.3$

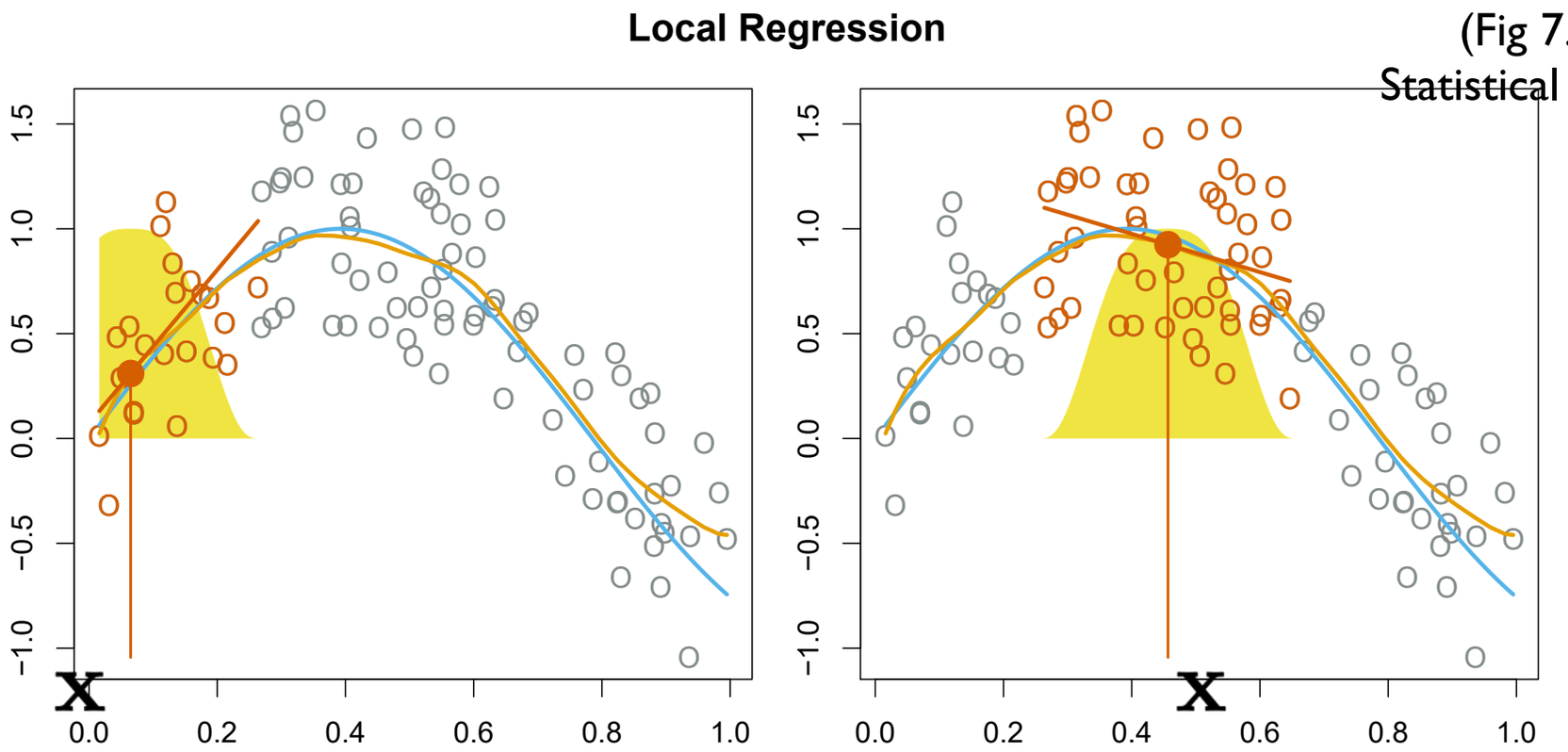
Kernel smoother

- The Gaussian kernel does extrapolate to flat line



$$h = 0.3$$

Local linear regression

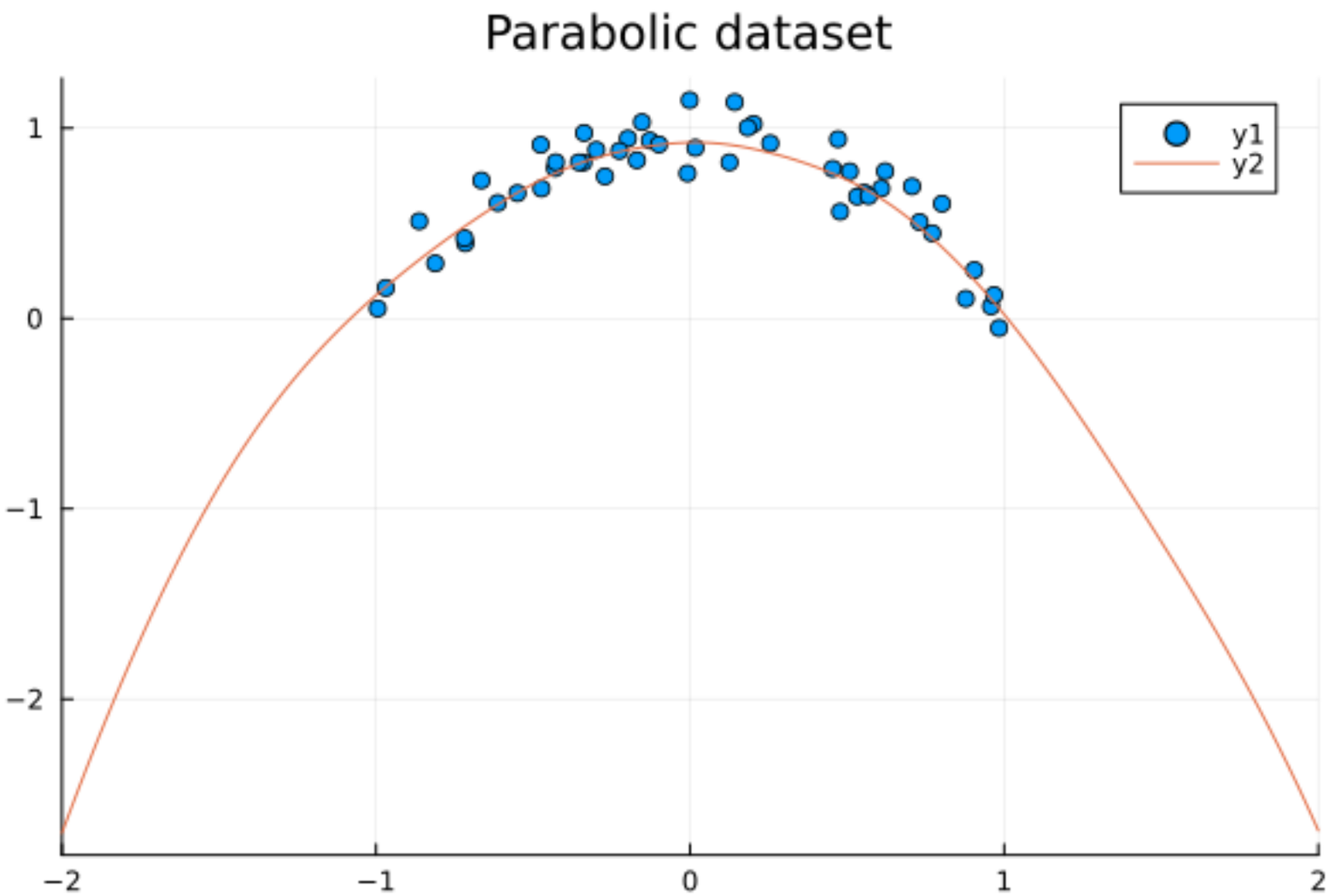


(Fig 7.9 Introduction to Statistical Learning, James et al.)

- Gather N local points around **x**
- Define a weight $w_i = K(\mathbf{x}, \mathbf{x}_i)$
- Do linear fit $L = \sum_{i=1}^{N} w_i (y_i - \beta_0 - \beta^T \mathbf{x}_i)^2$
- Predict $\hat{f}(\mathbf{x}) = \hat{\beta}_0 + \hat{\beta}^T \mathbf{x}$

Local linear regression

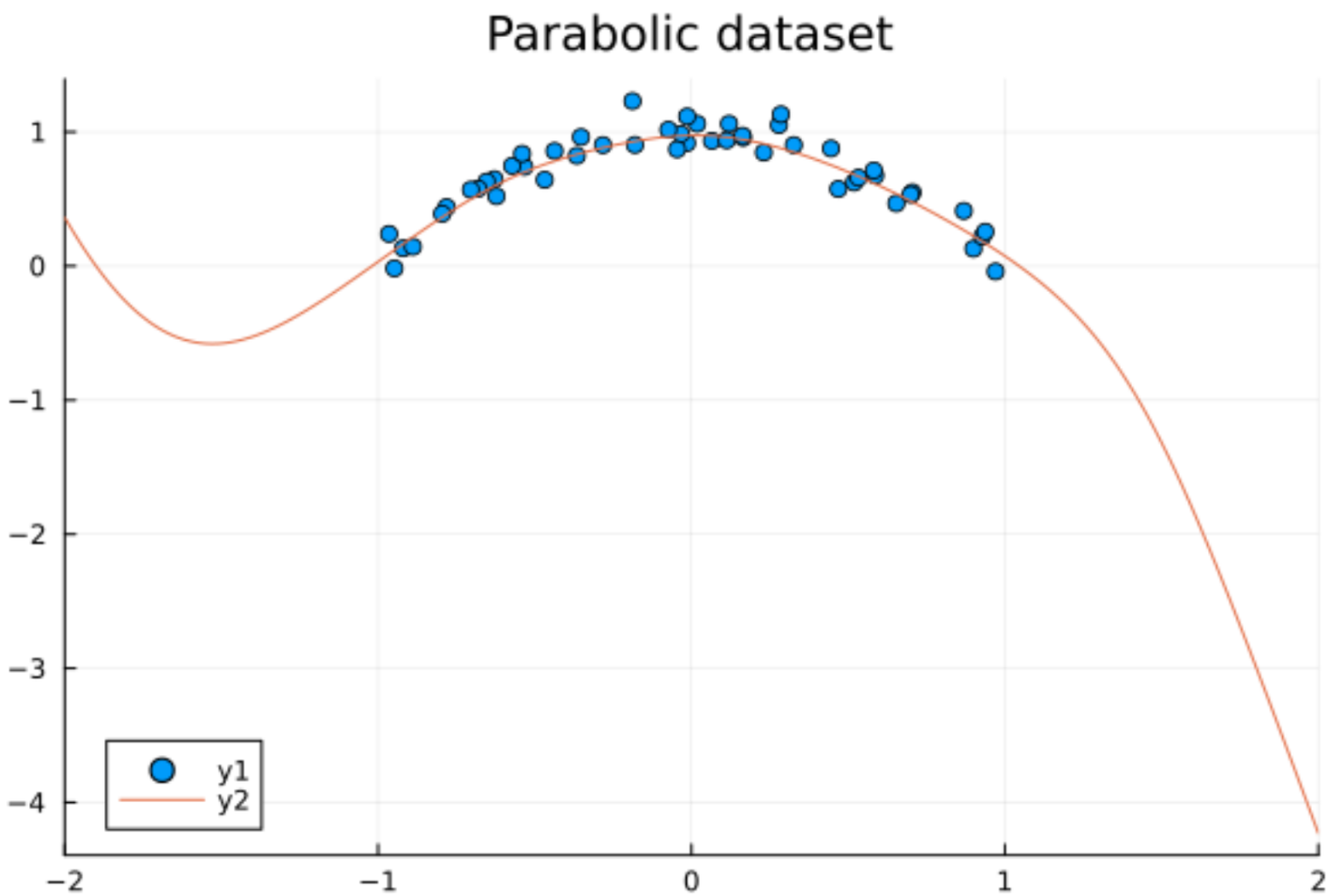
- Fit a line on the local neighborhood



$h = 0.3$

Local linear regression

- What is happening here?



$h = 0.3$

(Dis-)Advantages?

- + Relatively simple to implement
- + Hardly any training (lazy learning)
- + Relatively easy interpretation of hyperparameters
- (+ Some mathematical analysis is possible)

- - Very expensive during evaluation
- - Need to optimize the hyperparameters

Ridge regression

- Start with the classical linear regression, with some regularization:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(\mathbf{w}^T\mathbf{x}_i - y_i)^2 + \lambda\|\mathbf{w}\|^2$$

- What was the solution again?

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

- Fine. Now what if we want to make it nonlinear?

Kernelize? Nonlinear features

- Linear classifier

$$\hat{y} = \mathbf{w}^T \mathbf{x} + w_0 = w_2 x_2 + w_1 x_1 + w_0$$

- Why not add quadratic terms?

$$\hat{y} = w_5 x_2^2 + w_4 x_1^2 + w_3 x_1 x_2 + w_2 x_2 + w_1 x_1 + w_0$$

- Or higher order terms?
- Or other types of functions?

$$\hat{y} = w_4 \sin(x_2) + w_3 \cos(x_1) + w_2 x_2 + w_1 x_1 + w_0$$

- Advantages/disadvantages?

Kernelize

- Adding features is dangerous:
  - we can suffer from the curse of dimensionality,
  - the computational effort becomes large
- There is a situation where we can avoid large computations: if the model only contains inner products between feature vectors:

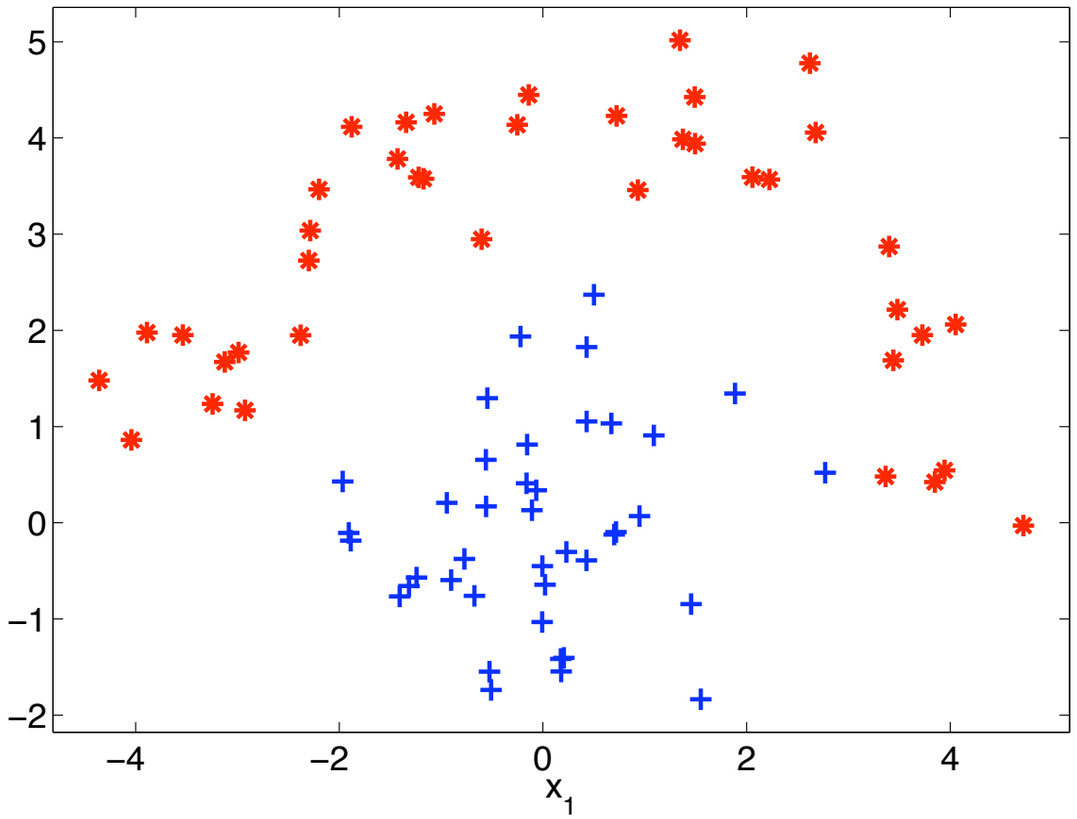
$$\mathbf{x}^T\mathbf{x}$$

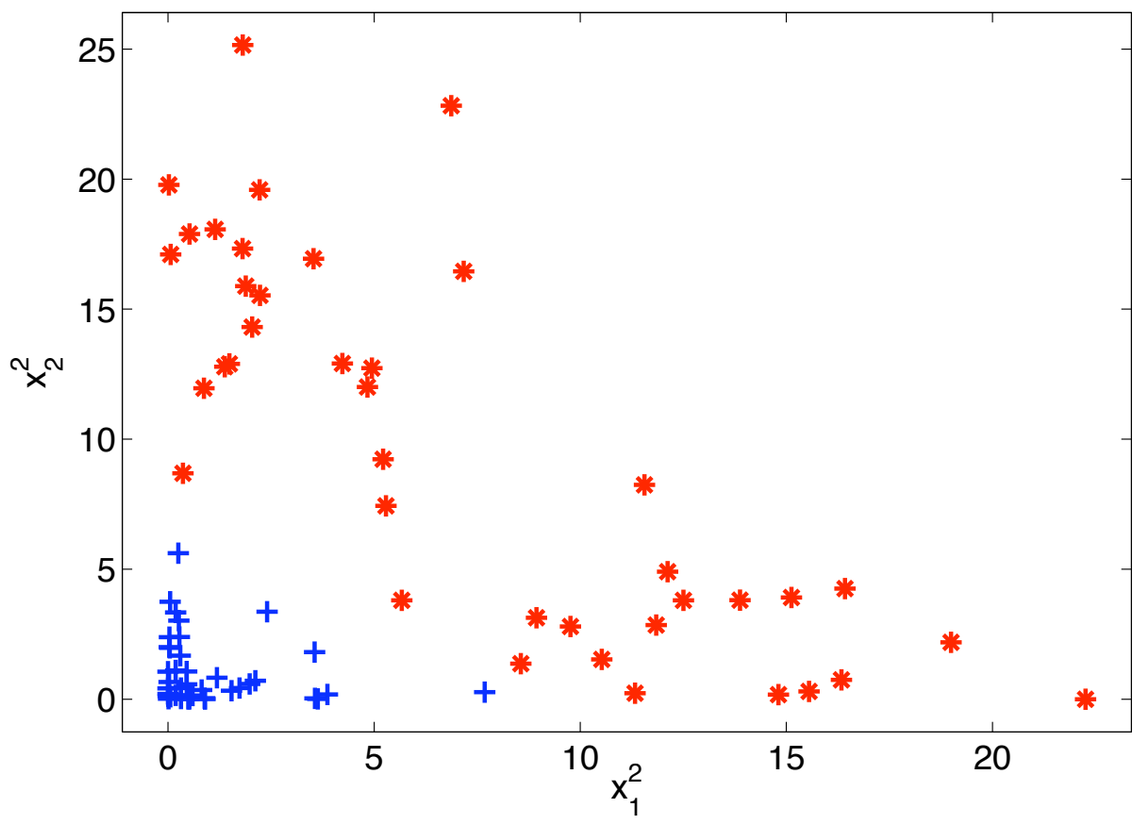
- Now replace the feature vector with mapped versions:

$$\mathbf{x}^T\mathbf{x} \rightarrow \mathbf{\Phi}(\mathbf{x})^T\mathbf{\Phi}(\mathbf{x})$$

- Nice when the inner product between the mapped feature vectors is 'easy'

Example transformation





- Original data: $\mathbf{x} = (x_1, x_2)$
- Mapped data: $\mathbf{\Phi}(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$

TU Delft

Polynomial kernel

- When we have two vectors

$$\mathbf{\Phi}(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

$$\mathbf{\Phi}(\mathbf{y}) = (y_1^2, y_2^2, \sqrt{2}y_1y_2)$$

$$\begin{aligned}\mathbf{\Phi}(\mathbf{x})^T\mathbf{\Phi}(\mathbf{y}) &= x_1^2y_1^2 + x_2^2y_2^2 + 2x_1x_2y_1y_2 \\ &= \left((x_1, x_2)(y_1, y_2)^T\right)^2 \\ &= \left(\mathbf{x}^T\mathbf{y}\right)^2\end{aligned}$$

it becomes very cheap to compute the inner product.

Polynomial kernel

- When we have two vectors

$$\mathbf{\Phi}(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

$$\mathbf{\Phi}(\mathbf{y}) = (y_1^2, y_2^2, \sqrt{2}y_1y_2)$$

$$\begin{aligned}\mathbf{\Phi}(\mathbf{x})^T\mathbf{\Phi}(\mathbf{y}) &= x_1^2y_1^2 + x_2^2y_2^2 + 2x_1x_2y_1y_2 \\ &= \left((x_1, x_2)(y_1, y_2)^T\right)^2 \\ &= \left(\mathbf{x}^T\mathbf{y}\right)^2 \quad = K(\mathbf{x}, \mathbf{y})\end{aligned}$$

kernel function

it becomes very cheap to compute the inner product.

Ridge regression

- Start with the classical linear regression, with some regularization:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(\mathbf{w}^T\mathbf{x}_i - y_i)^2 + \lambda\|\mathbf{w}\|^2$$

- Now kernelize, or assume a feature mapping

$$\tilde{\mathbf{x}} = \Phi(\mathbf{x})$$

- So minimizing

$$\mathcal{L}(\tilde{\mathbf{w}}) = \frac{1}{2}\sum_{i=1}^{N}(\tilde{\mathbf{w}}^T\Phi(\mathbf{x}_i) - y_i)^2 + \lambda\|\tilde{\mathbf{w}}\|^2$$

gives

$$\hat{\mathbf{w}} = (\Phi(\mathbf{X})^T\Phi(\mathbf{X}) + \lambda\mathbf{I})^{-1}\Phi(\mathbf{X})^T\mathbf{y}$$

Kernel Ridge regression

- Applying this linear regression to a test point **z**:

$$\hat{y} = \Phi(\mathbf{z})^T \hat{\mathbf{w}} = \Phi(\mathbf{z})^T (\Phi(\mathbf{X})^T \Phi(\mathbf{X}) + \lambda \mathbf{I})^{-1} \Phi(\mathbf{X})^T \mathbf{y}$$

$(D \times 1)$ $(1 \times D)$ $(D \times N)$ $(N \times D)$ $(D \times D)$ $(D \times N)$ $(N \times 1)$

- In order to do the kernel trick, we need

$$K(\mathbf{X}, \mathbf{X}) = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \dots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & & \\ k(\mathbf{x}_N, \mathbf{x}_1) & \dots & k(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix} = \Phi(\mathbf{X})\Phi(\mathbf{X})^T$$

$(N \times N)$

Need a trick

- How to convert $(\mathbf{AB}+\mathbf{I})^{-1} \rightarrow (\mathbf{BA}+\mathbf{I})^{-1}$??????

- It appears:

$$(\mathbf{AB}+\mathbf{I}_D)^{-1}\mathbf{A} = \mathbf{A}(\mathbf{BA}+\mathbf{I}_N)^{-1}$$

- (Proof:

$$\begin{aligned}(\mathbf{AB}+\mathbf{I}_D)^{-1}\mathbf{A} &= (\mathbf{AB}+\mathbf{I}_D)^{-1}\mathbf{A}(\mathbf{BA}+\mathbf{I}_N)(\mathbf{BA}+\mathbf{I}_N)^{-1} \\ &= (\mathbf{AB}+\mathbf{I}_D)^{-1}(\mathbf{ABA}+\mathbf{A})(\mathbf{BA}+\mathbf{I}_N)^{-1} \\ &= (\mathbf{AB}+\mathbf{I}_D)^{-1}(\mathbf{AB}+\mathbf{I}_D)\mathbf{A}(\mathbf{BA}+\mathbf{I}_N)^{-1} \\ &= \mathbf{A}(\mathbf{BA}+\mathbf{I}_N)^{-1} \qquad)\end{aligned}$$

Kernel Ridge regression

- Applying this linear regression to a test point **z**:

$$\hat{y} = \Phi(\mathbf{z})^T \hat{\mathbf{w}} = \Phi(\mathbf{z})^T (\underbrace{\Phi(\mathbf{X})^T}_{\mathbf{A}} \underbrace{\Phi(\mathbf{X})}_{\mathbf{B}} + \lambda \mathbf{I})^{-1} \underbrace{\Phi(\mathbf{X})^T}_{\mathbf{A}} \mathbf{y}$$

$$= \Phi(\mathbf{z})^T \Phi(\mathbf{X})^T (\Phi(\mathbf{X}) \Phi(\mathbf{X})^T + \lambda \mathbf{I})^{-1} \mathbf{y}$$

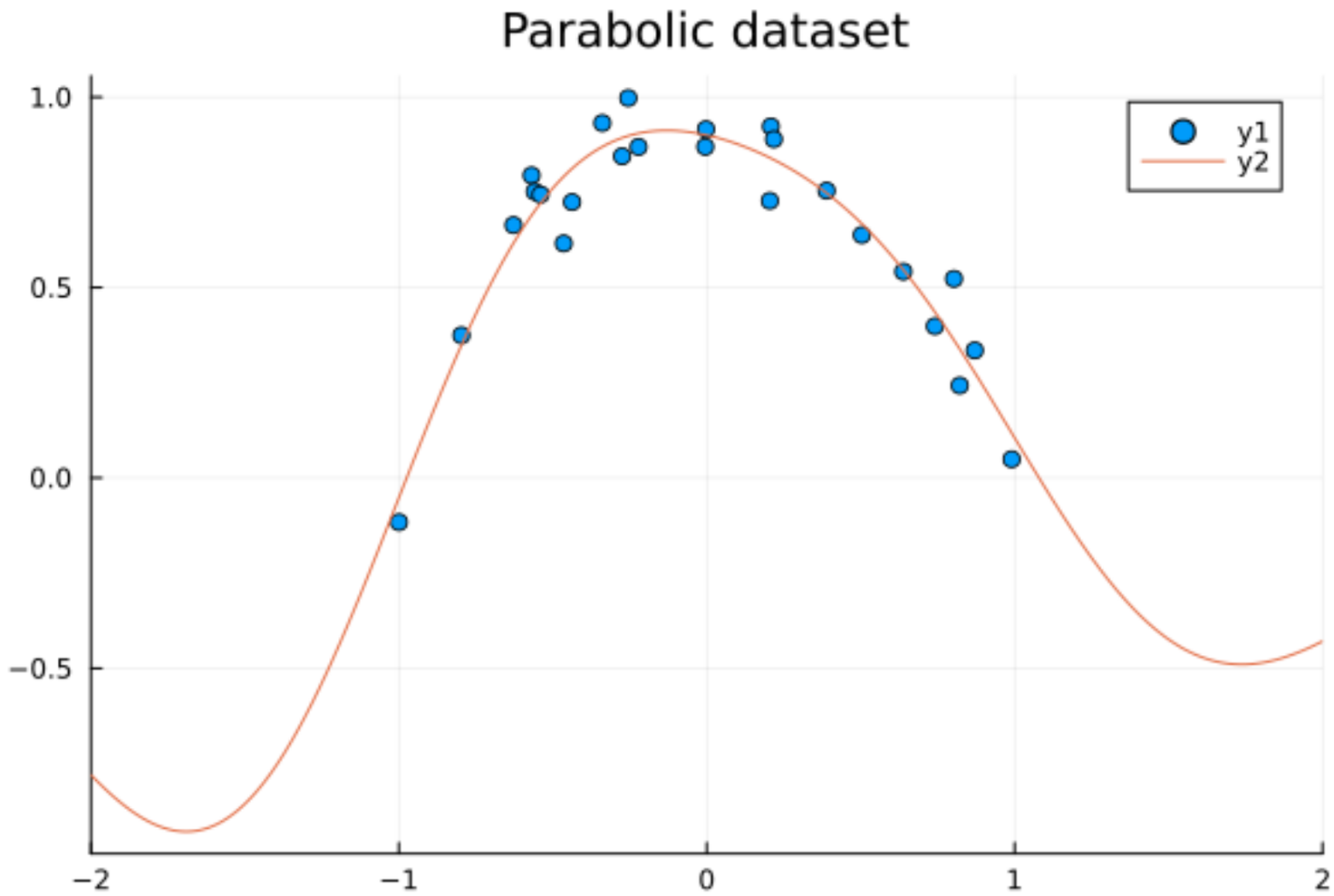
$$= K(\mathbf{z}, \mathbf{X})^T (K(\mathbf{X}, \mathbf{X}) + \lambda \mathbf{I})^{-1} \mathbf{y}$$

$$= \mathbf{y}^T (K(\mathbf{X}, \mathbf{X}) + \lambda \mathbf{I})^{-1} \mathbf{k}(\mathbf{z})$$

- where:

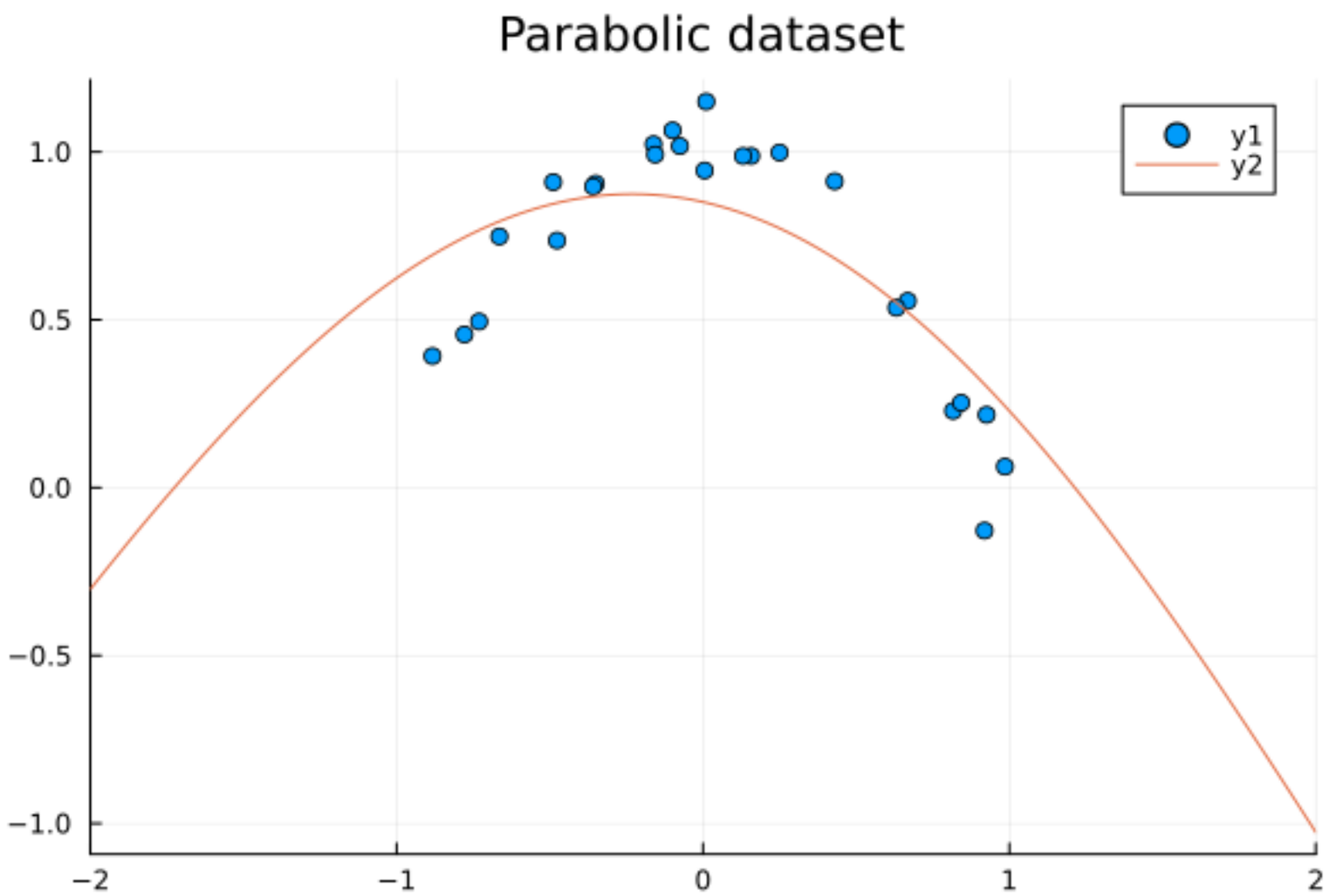
$$\mathbf{k}(\mathbf{z}) = [K(\mathbf{z}, \mathbf{x}_1), ..., K(\mathbf{z}, \mathbf{x}_N)]^T$$

Kernel Ridge regression



- With standard Gaussian kernel $\sigma = 1,\ \lambda = 0.01$

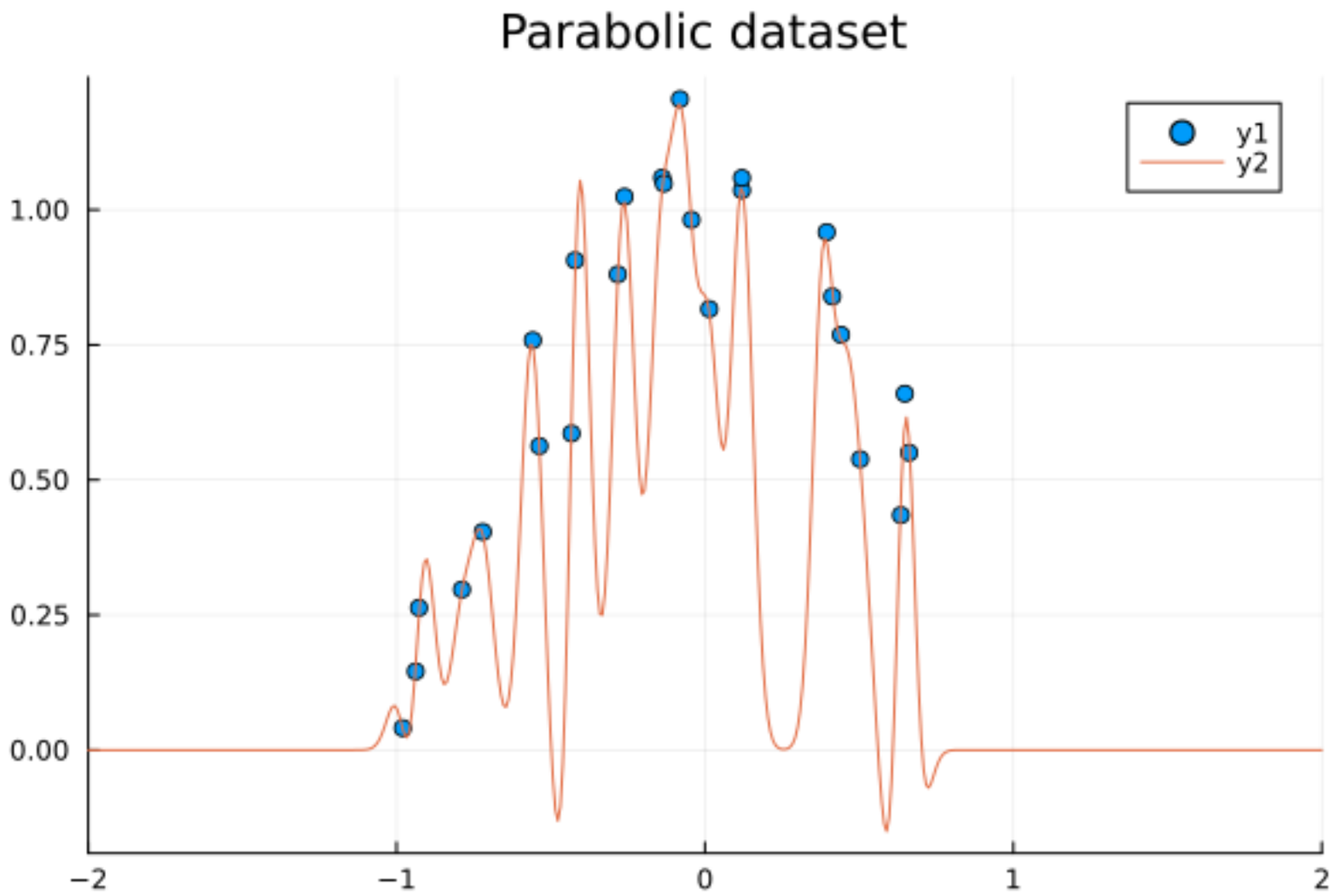
Kernel Ridge regression



much smoother function

- With standard Gaussian kernel $\sigma = 5,\ \lambda = 0.01$

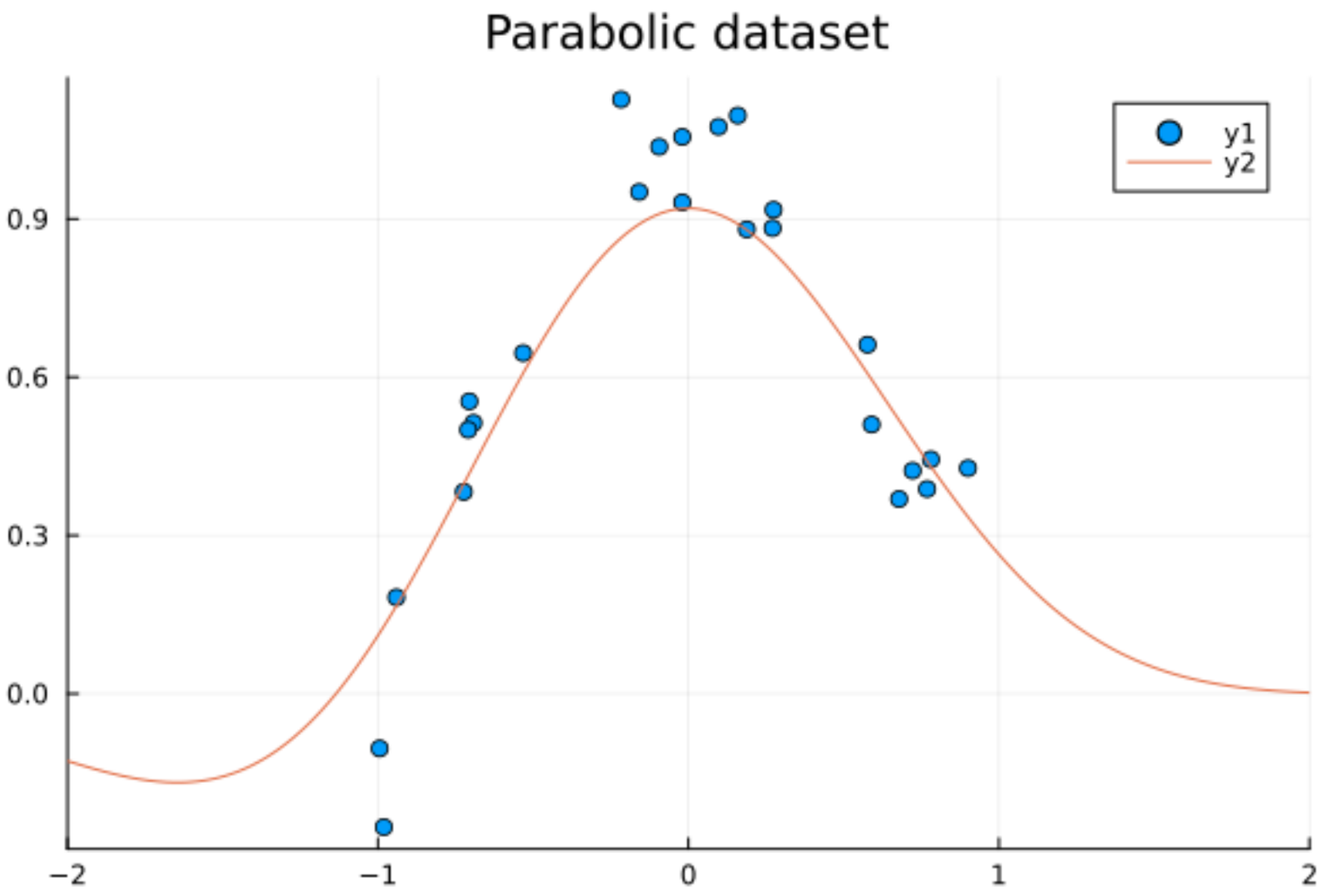
Kernel Ridge regression



much rougher function

- With standard Gaussian kernel $\sigma = 0.05,\ \lambda = 0.01$

Kernel Ridge regression



function stays closer to 0

- With standard Gaussian kernel $\sigma = 1.0,\ \lambda = 0.9$

(Dis-)Advantages

- + Relative easy to implement
- + Only need to store one (Nx1) weight vector
- + Relatively easy interpretation of hyperparameters

- - Need to invert the (NxN) kernel matrix
- - Need to optimize the hyperparameters

- BONUS: it is possible to get an additional estimate of the uncertainty: Gaussian Processes

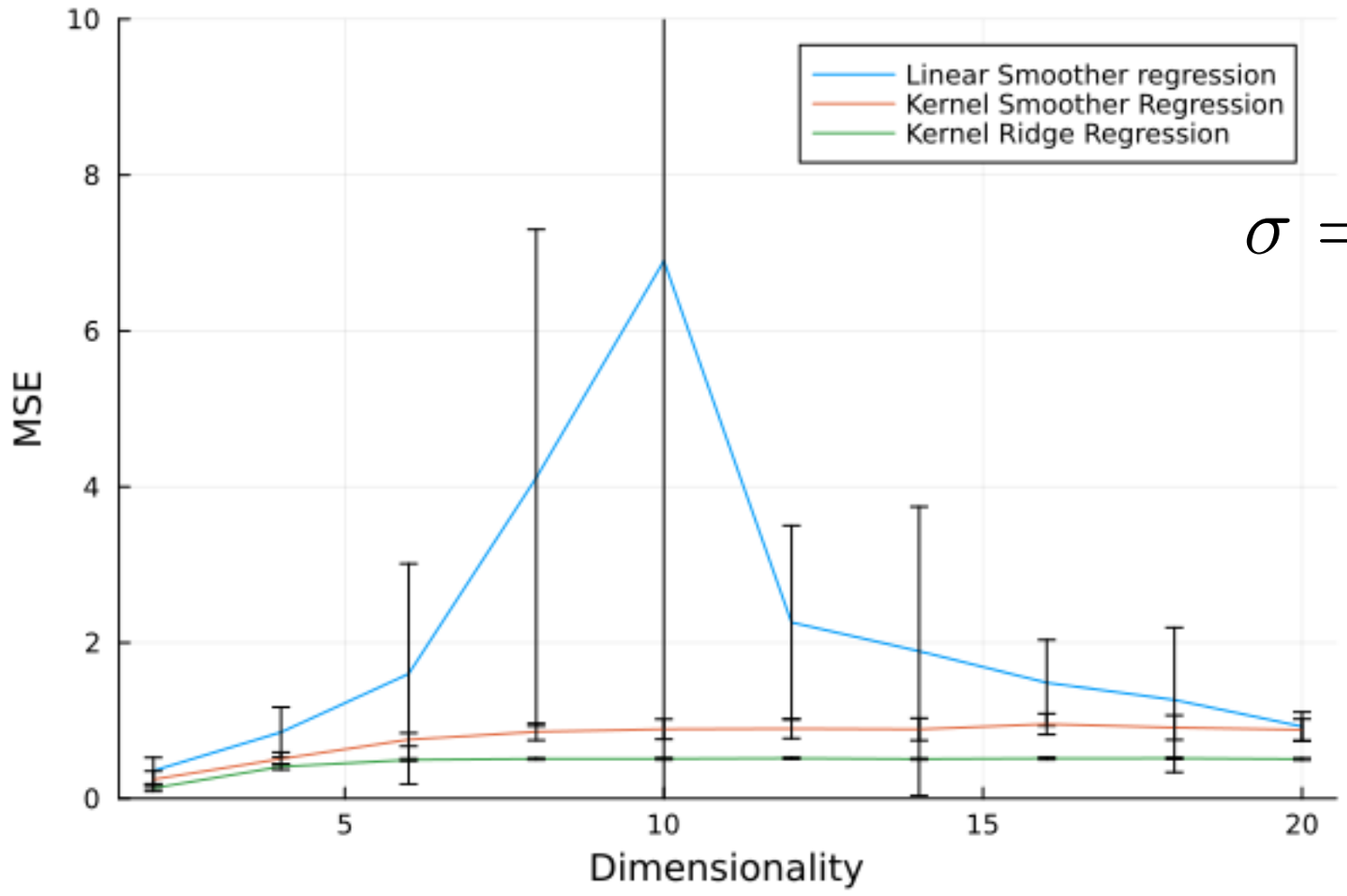
Neural networks

- The default 'solution' nowadays are (deep) neural networks, often with MSE

$$\mathcal{L}(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N}(f_{W_L}(...f_{W_1}(\mathbf{x}_i)...) - y_i)^2$$

- No need for (inverse) kernel matrices, or storing the training set
- But training involves many tricks, hyperparameters and tuning, and typically requires large training sets

Feature curve: peaking



$\sigma = 1,\ \lambda = 0.01$

- Nonlinear regression data, N=10 training obj.
- Linear regression shows the peaking phenomenon (needs inverse of $\mathbf{X}^T\mathbf{X}$)

Back to smoothers

- Note that for Gaussian processes, the mean

$$\hat{y}(\mathbf{x}_{N+1}) = m(\mathbf{x}_{N+1}) = \mathbf{k}^T \mathbf{C}_N^{-1} \mathbf{y}$$

also is of the form of

$$\hat{y}(\mathbf{x}) = \mathbf{s}^T(\mathbf{x})\mathbf{y}$$

smoothing function

all the labels of the training data

- Compare with kernel regressor:

$$\hat{y}(\mathbf{x}) = \frac{\sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i) y_i}{\sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i)}$$

Effective nr of parameters

- Assume we have a smoother:

$$\hat{y}(\mathbf{x}) = \mathbf{s}^T(\mathbf{x})\mathbf{y}$$

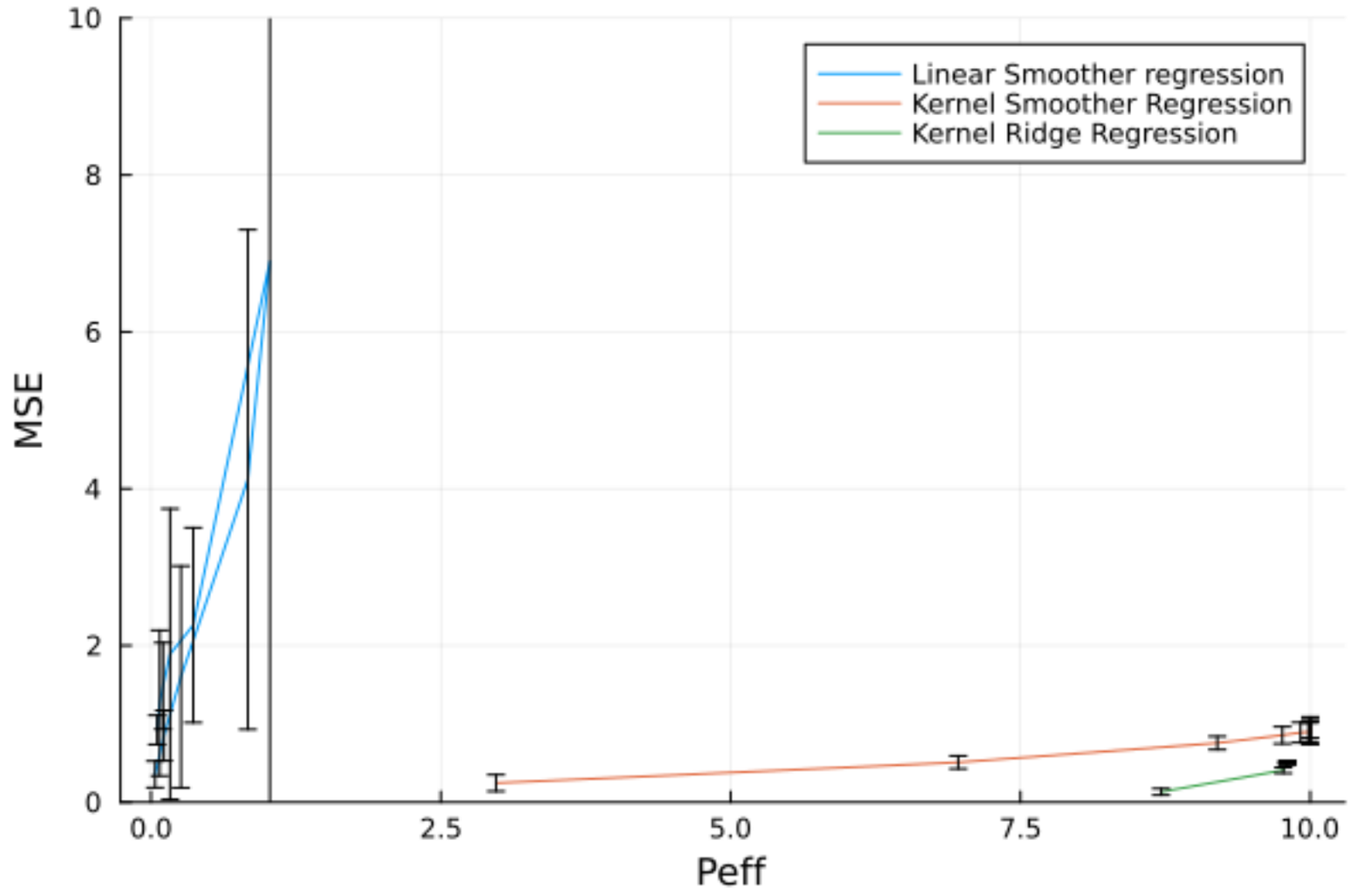
- So, for the training set holds

$$\hat{\mathbf{y}} = \mathbf{S}\mathbf{y}$$

- One definition for the effective nr of parameters is:

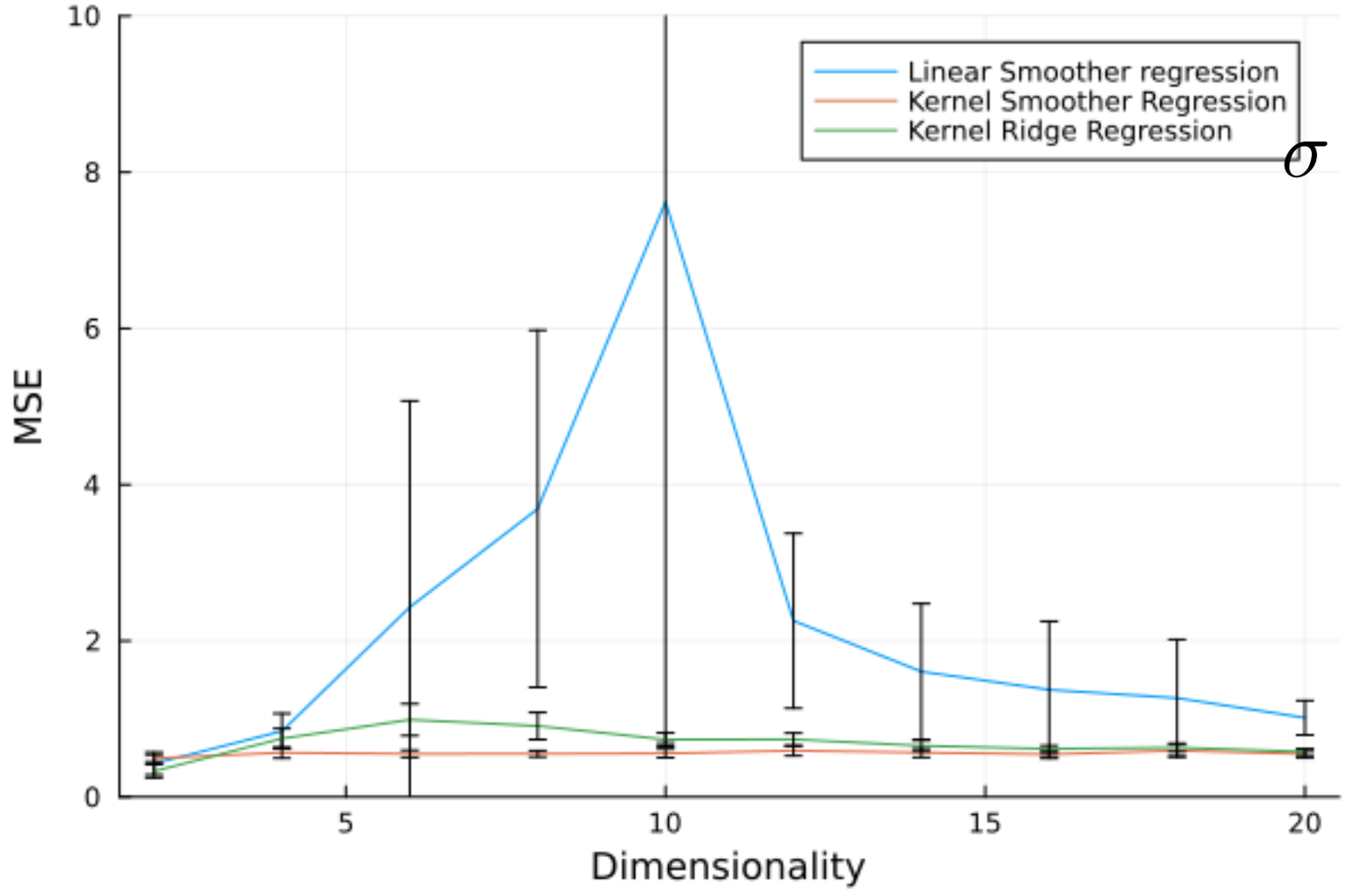
$$p_{\text{eff}} = \text{trace}(\mathbf{S})$$

Peff as complexity



- Kernel methods have higher effective number of parameters (up to 10)
- Linear regression first increases, then decreases Peff

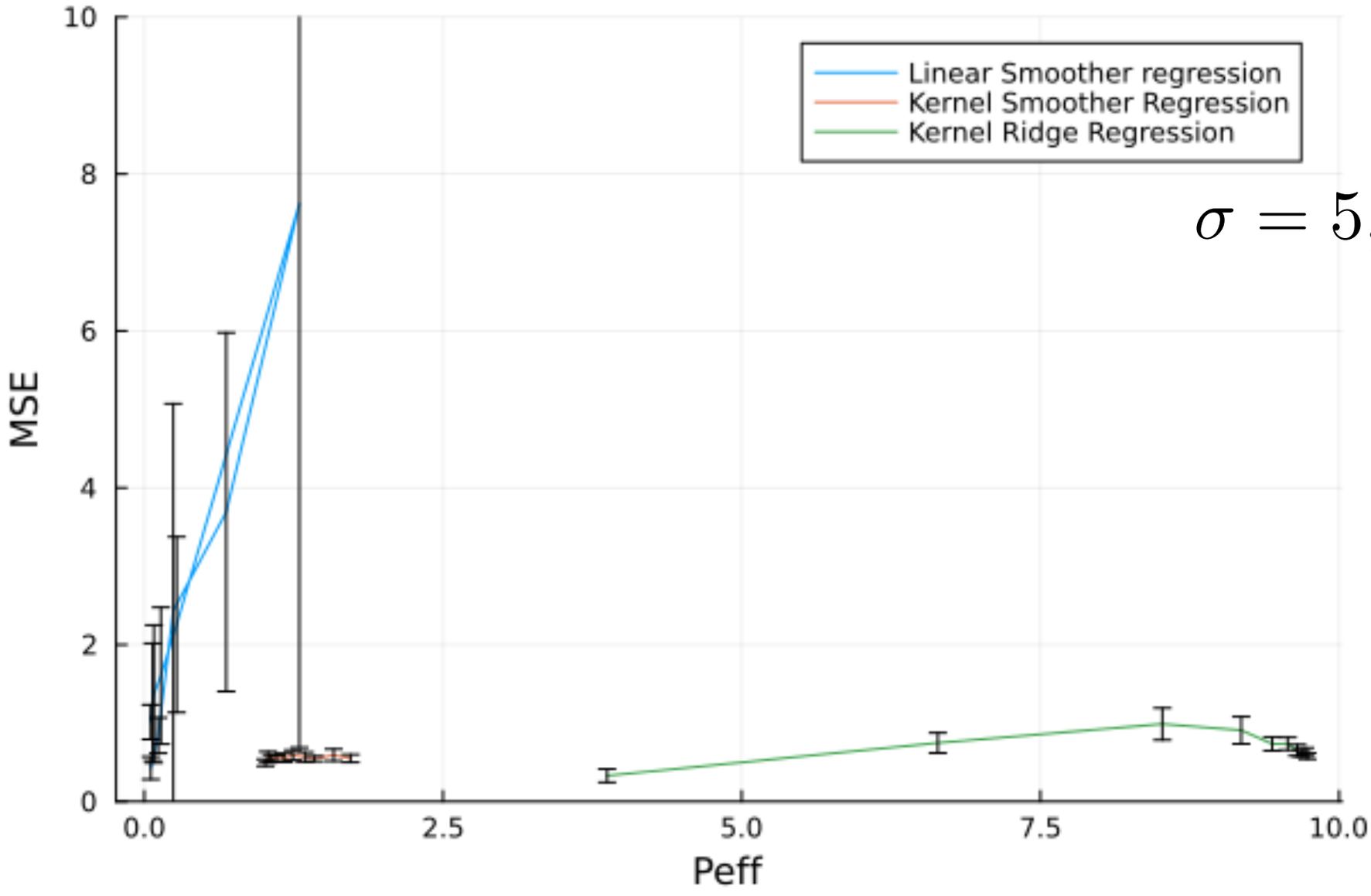
Feature curve



$\sigma = 5,\ \lambda = 0.01$

- Changing the scale influences the performances of kernel methods

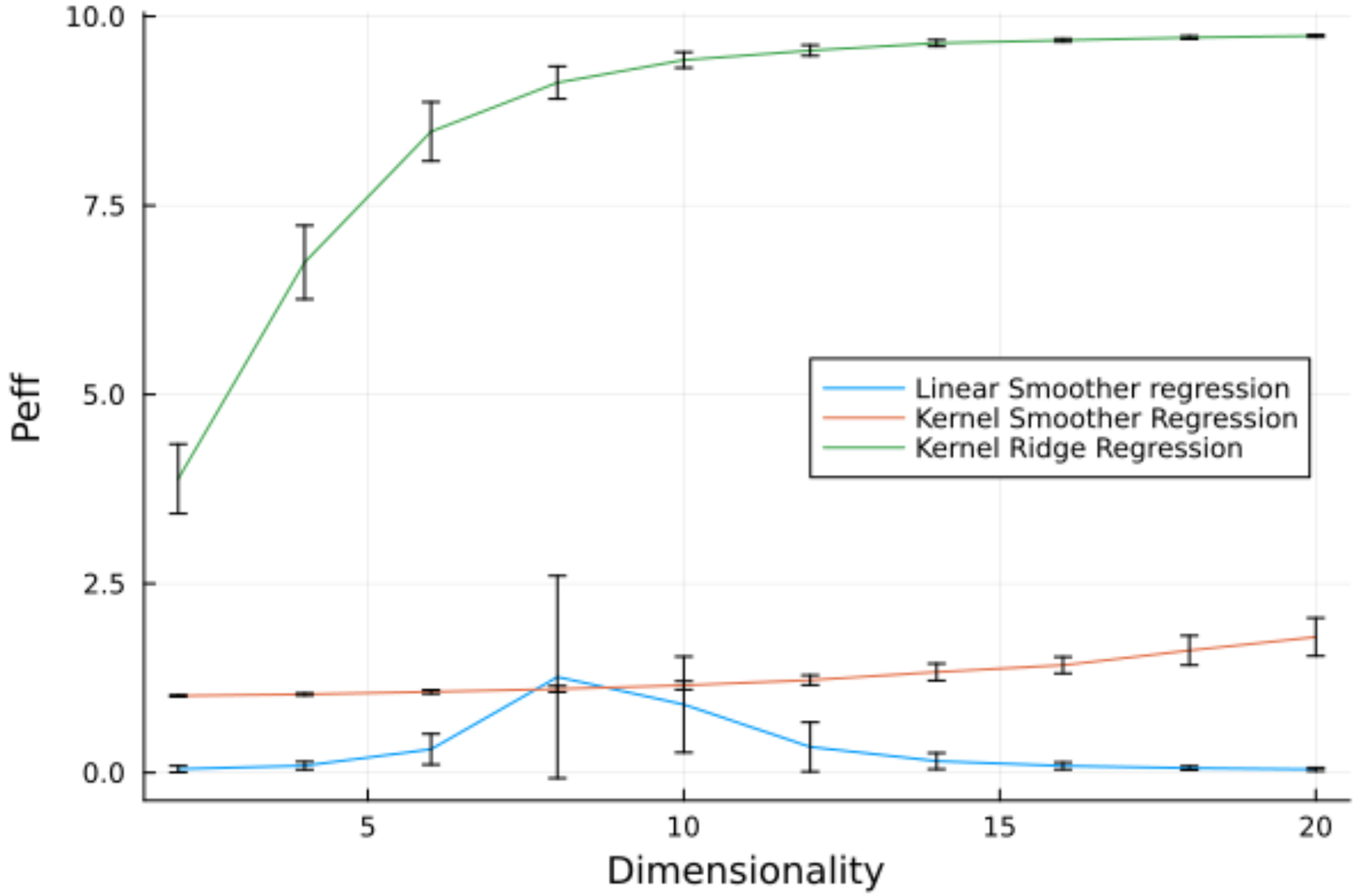
Complexity



$\sigma = 5,\ \lambda = 0.01$

- By increasing sigma, the complexity goes down

Peff vs dimensionality



- Note how Peff increases and decreases for linear regression

Conclusions

- Alternative to Deep Learning, nonlinear Smoothers can be used:
  - Less hyperparameters (typically a scale and a regularization parameter)
  - often interpretable
  - often well performing on smaller datasets
- Drawbacks
  - (Typically) all training data has to be stored
  - Inference time is long or inverse kernel matrix is needed