Introduction to Machine Learning II

Over/under-fitting, regularization, bias/variance trade-off

A Bayesian view to curve fitting

Dimosthenis Karatzas (dimos@cvc.uab.es)

REVISITING LINEAR REGRESSION: UNDERFITTING AND OVERFITTING

Linear Regression: Current formulation

Our current formulation is as follows:

$$\begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \approx \begin{bmatrix} \mathbf{x}^{(1)^T} \\ \vdots \\ \mathbf{x}^{(m)^T} \end{bmatrix} \mathbf{w}$$

Equivalently: $\mathbf{y} \approx \mathbf{X}^T \mathbf{w}$

$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})$, where

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \sum_{i=1}^{m} L\left(\mathbf{x}^{(i)^T} \mathbf{w}, y^{(i)}\right)$$

Generating new features

We mentioned that given any set of features, we could generate new ones.

E.g. in order to do polynomial regression, we generated from a set of features $x_1, x_2, \dots, x_n$ new features like $x_1^2, x_2^2, x_3^2, \dots, x_n^2$

Let's define a function $\phi(\mathbf{x})$ that operates on a set of features, and generates a new set of features

Input: original list of features

$$\begin{tikzcd} {[x]} \arrow[r, bend left, "\phi(\mathbf{x})"] & \begin{matrix} [1] \\ [1, x] \\ [1, x, x^2] \\ [1, x, \sqrt{x}] \\ \dots \end{matrix} \end{tikzcd}$$

Output: a new list of features

A more generic formulation

We introduce a parametric family of functions, and a loss function:

$$\begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \approx \begin{bmatrix} \phi\left(\mathbf{x}^{(1)}\right)^T \\ \vdots \\ \phi\left(\mathbf{x}^{(m)}\right)^T \end{bmatrix} \mathbf{w}$$

Equivalently: $\mathbf{y} \approx \phi(\mathbf{X})^T \mathbf{w}$

$\widehat{\mathbf{w}} = \arg \min_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}, \phi)$, where

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}, \phi) = \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^T \mathbf{w}, y^{(i)}\right)$$

With this notation, and the right selection of ϕ, we can recover also polynomial regression as we have already seen

Example - linear

Let's say we set:

$\phi(x) = [1, x]^T$ (i.e. the linear basis)

$L(a, b) = \frac{1}{2m}(a - b)^2$ (i.e. a quadratic loss)

Then, we recover basic linear regression:

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}, \phi) = \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^T \mathbf{w}, y^{(i)}\right)$$

$$= \frac{1}{2m} \sum_{i=1}^{m} \left([1, x^{(i)}]\mathbf{w} - y^{(i)}\right)^2$$

Example - polynomial

ϕ projects, or embeds, the original data into a new basis

Let’s say we set:

$\phi(x) = [1, x, x^2]^T$ (i.e. degree two polynomial basis)

$L(a, b) = \frac{1}{2m}(a - b)^2$ (i.e. a quadratic loss)

Then, we recover (second degree) polynomial regression:

$$J(\mathbf{w}; \mathbf{X}, \mathbf{y}, \phi) = \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^T \mathbf{w}, y^{(i)}\right)$$

$$= \frac{1}{2m}\sum_{i=1}^{m}\left(\left[1, x^{(i)}, \left(x^{(i)}\right)^2\right]\mathbf{w} - y^{(i)}\right)^2$$

w are the coefficients (weights) of the linear (in basis ϕ) model we want to estimate

Problem Instances

It can be useful to think of the data in a **generative** way

We assume there is an underlying "true" function f_{true} that we want to estimate

All we have to use for estimation are corrupted samples of this ideal function f_{true}

$$y^{(i)} = f_{true}\left(\mathbf{x}^{(i)}\right) + N(0, \sigma)$$

where $N(0, \sigma)$ is a zero-mean Gaussian noise term with variance σ^2:

$$N(0, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

Important: we observe only corrupted samples y^i, we never observe the function f_{true} directly

The pair $(\mathbf{X}, \mathbf{y})$ is often called an **instance** of the problem

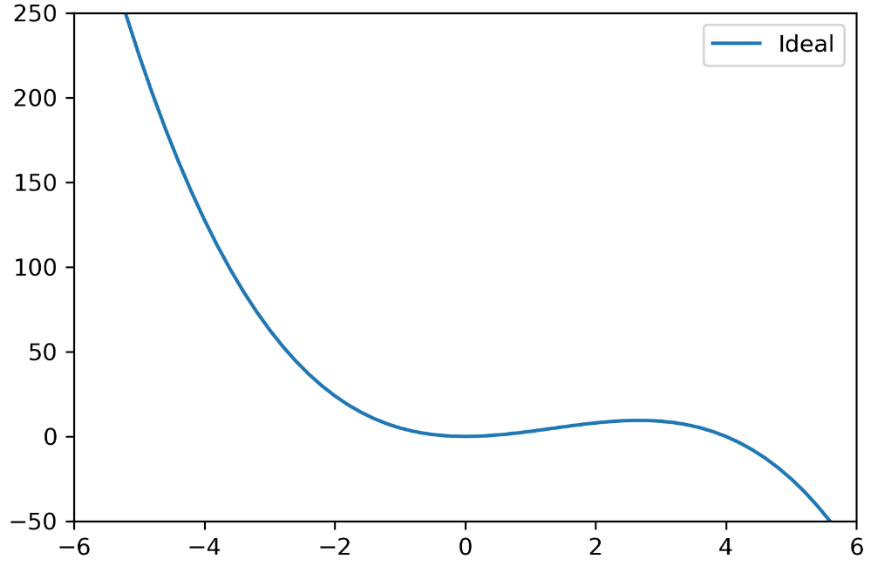
Running Example: a noisy cubic

Suppose our data model is:

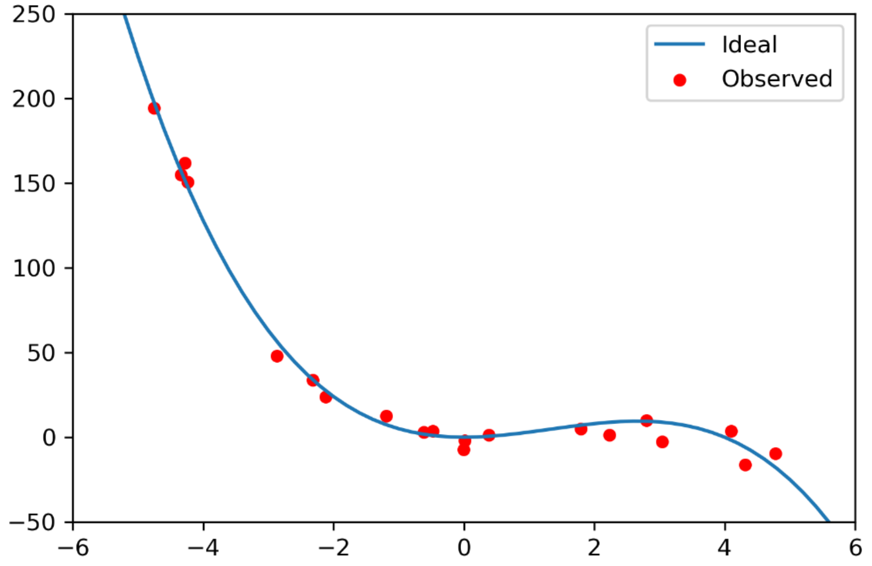
$$y^i = f\left(x^i\right) + N(0, \sigma)$$

$$f_{true}(x) = 4x^2 - x^3, \qquad \sigma = 5$$

Behind the scenes



What we observe



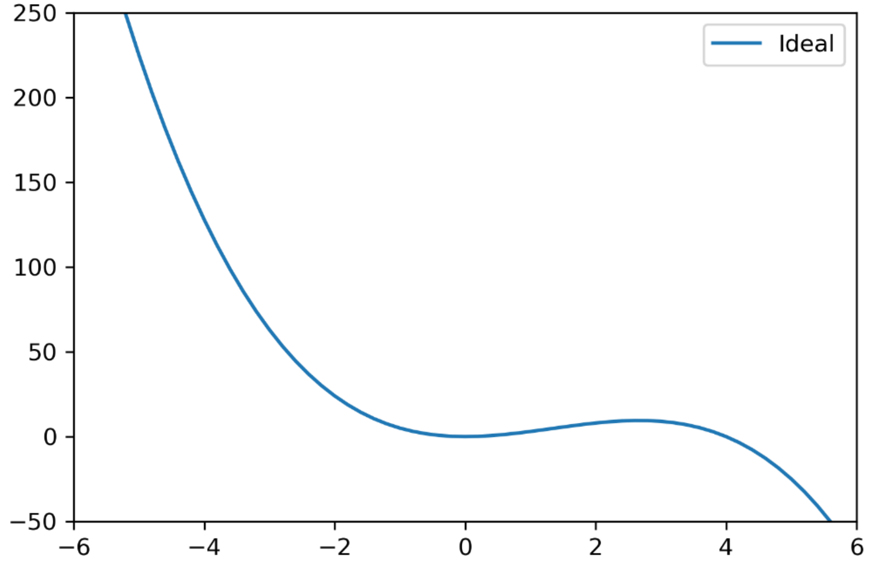
Running Example: a noisy cubic

Suppose our data model is:

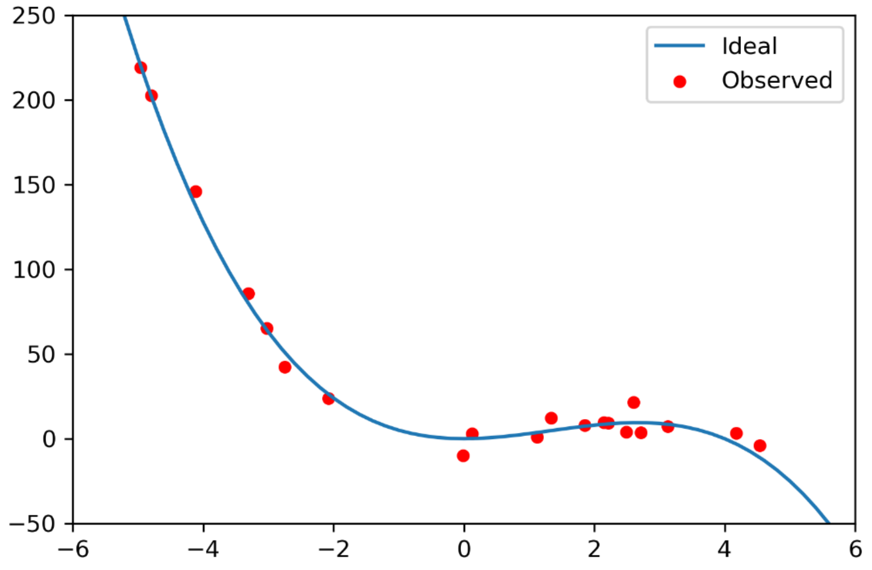
$$y^i = f(x^i) + N(0, \sigma)$$

$$f_{true}(x) = 4x^2 - x^3, \qquad \sigma = 5$$

Behind the scenes



What we observe



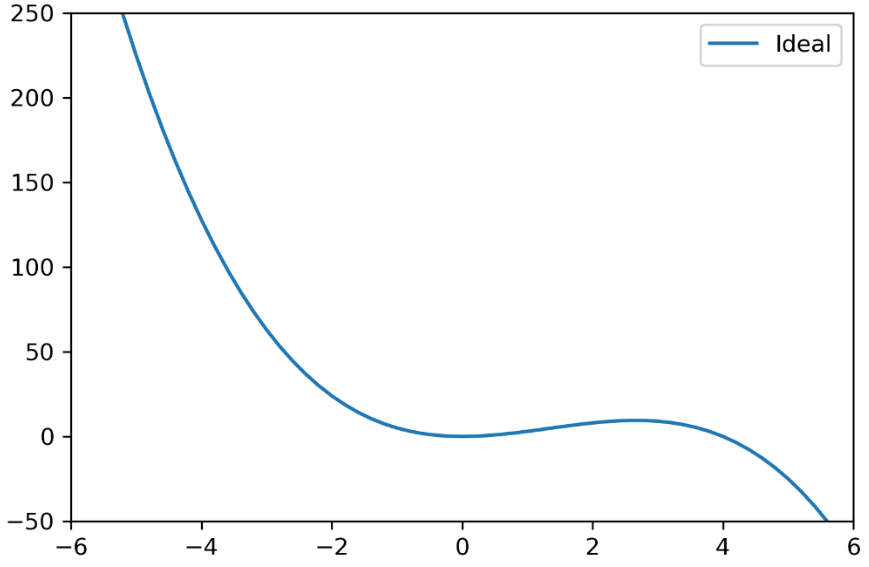
Running Example: a noisy cubic

Suppose our data model is:

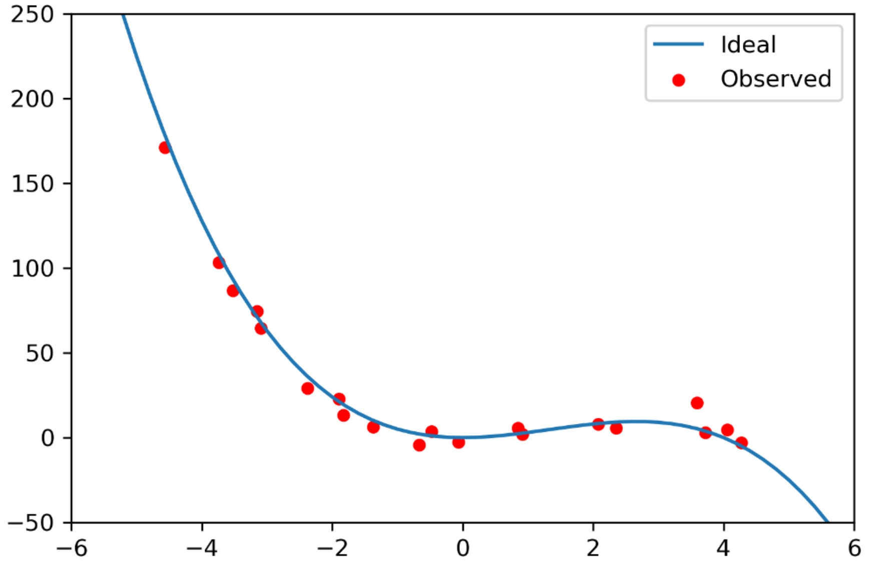
$$y^i = f\left(x^i\right) + N(0, \sigma)$$

$$f_{true}(x) = 4x^2 - x^3, \qquad \sigma = 5$$

Behind the scenes



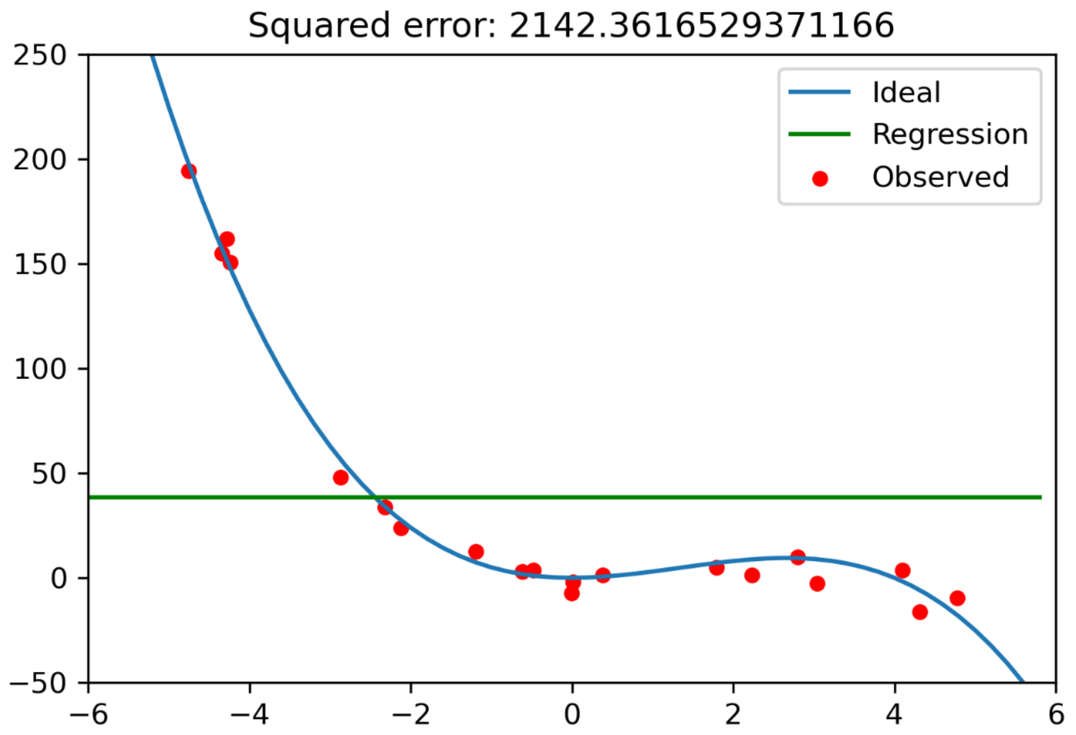
What we observe



A first approximation: the constant model

Assume we have a problem instance $(\mathbf{X}, \mathbf{y})$

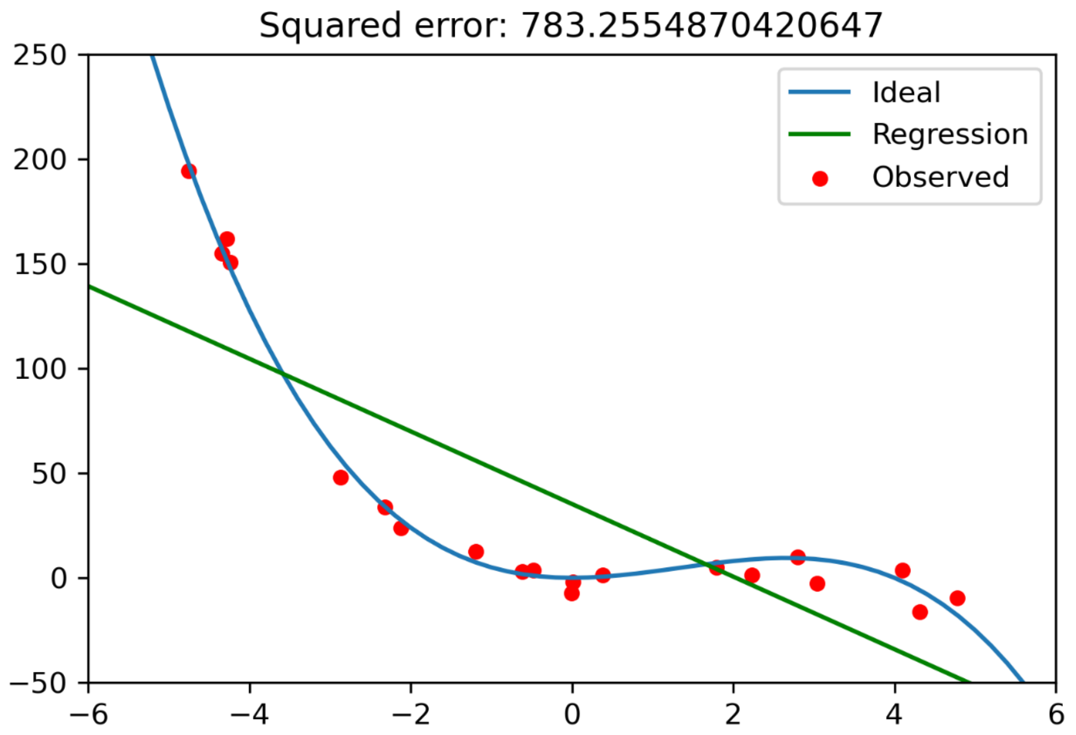
We start with the simplest model imaginable: $\phi(x) = [1]$



Increasing model complexity: a linear model

On the same problem instance $(\mathbf{X}, \mathbf{y})$

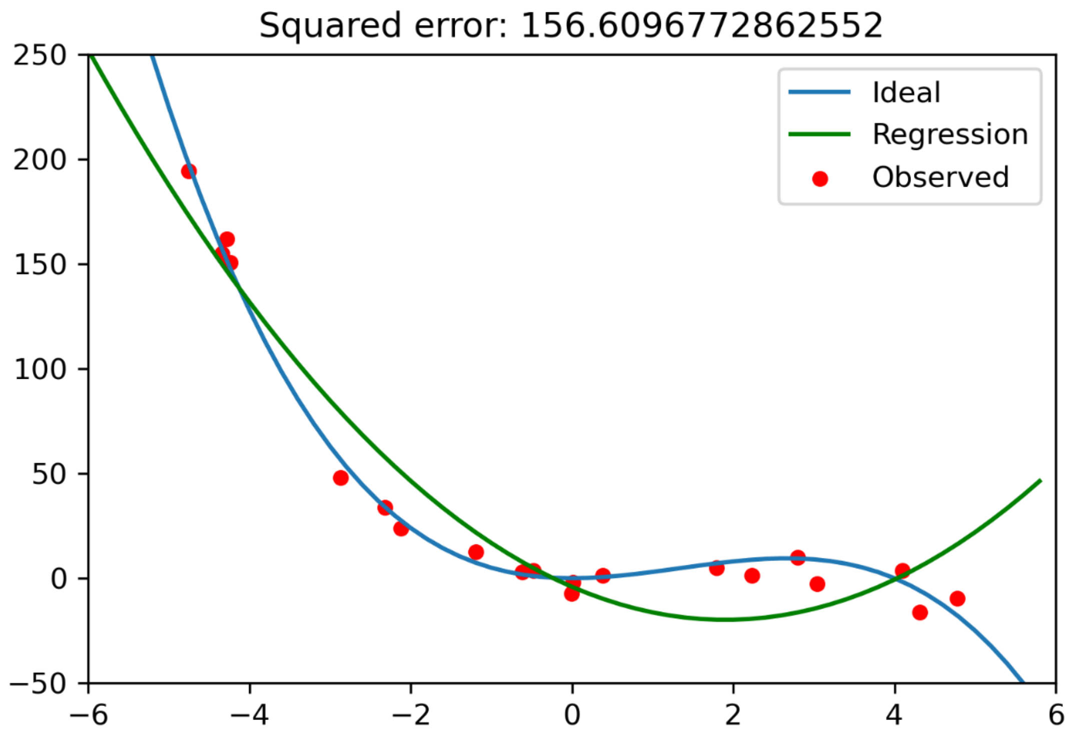
Linear design matrix: $\phi(x) = [1, x]$



Increasing model complexity: a two-degree polynomial model

On the same problem instance $(\mathbf{X}, \mathbf{y})$

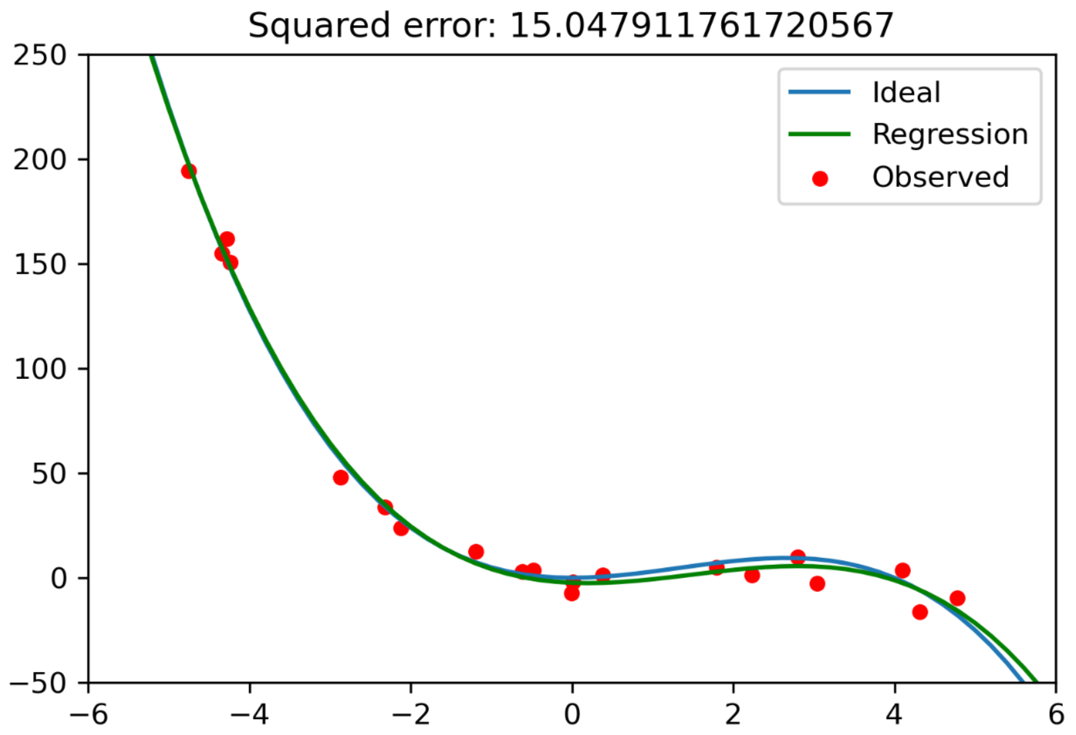
Quadratic design matrix: $\phi(x) = [1, x, x^2]$



Increasing model complexity: a three-degree polynomial model

On the same problem instance $(\mathbf{X}, \mathbf{y})$

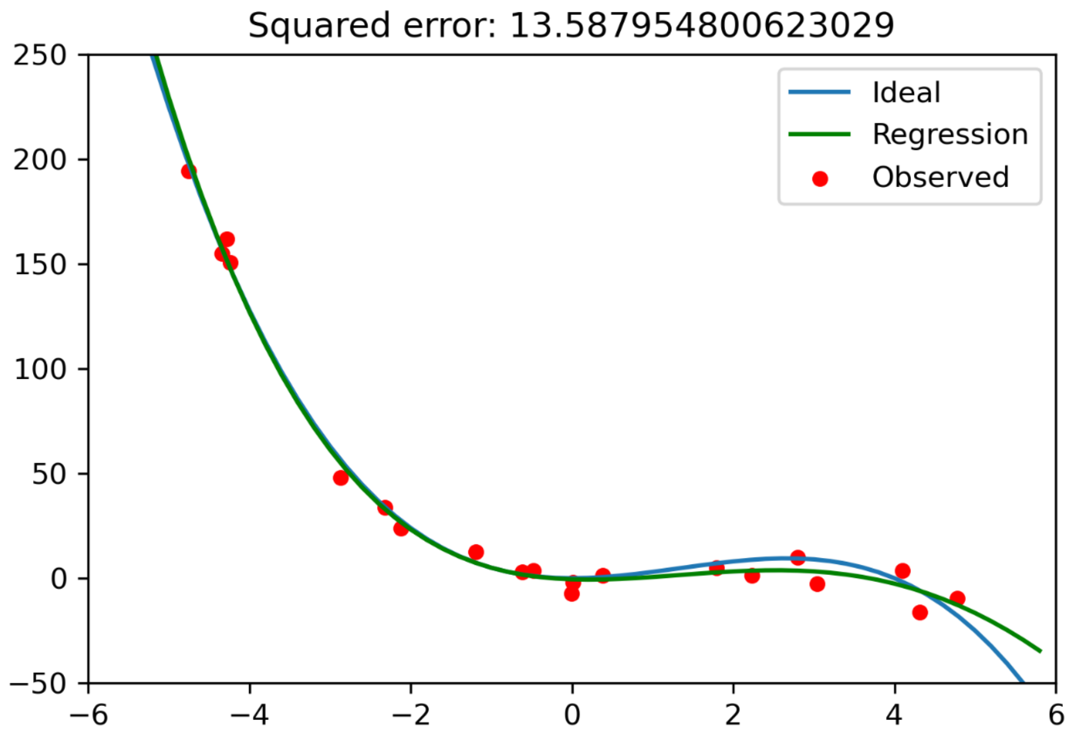
Cubic design matrix: $\phi(x) = [1, x, x^2, x^3]$



Increasing model complexity: a four-degree polynomial model

On the same problem instance $(\mathbf{X}, \mathbf{y})$

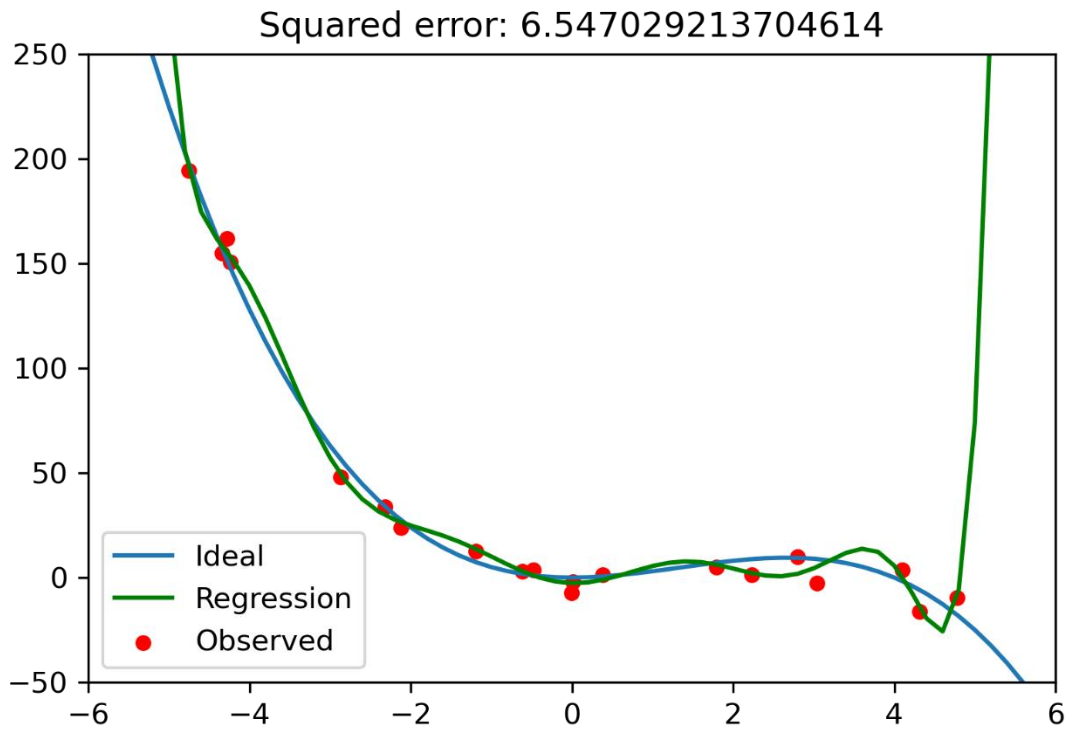
Fourth-degree design matrix: $\phi(x) = [1, x, x^2, x^3, x^4]$



Increasing model complexity: a ten-degree polynomial model

On the same problem instance $(\mathbf{X}, \mathbf{y})$

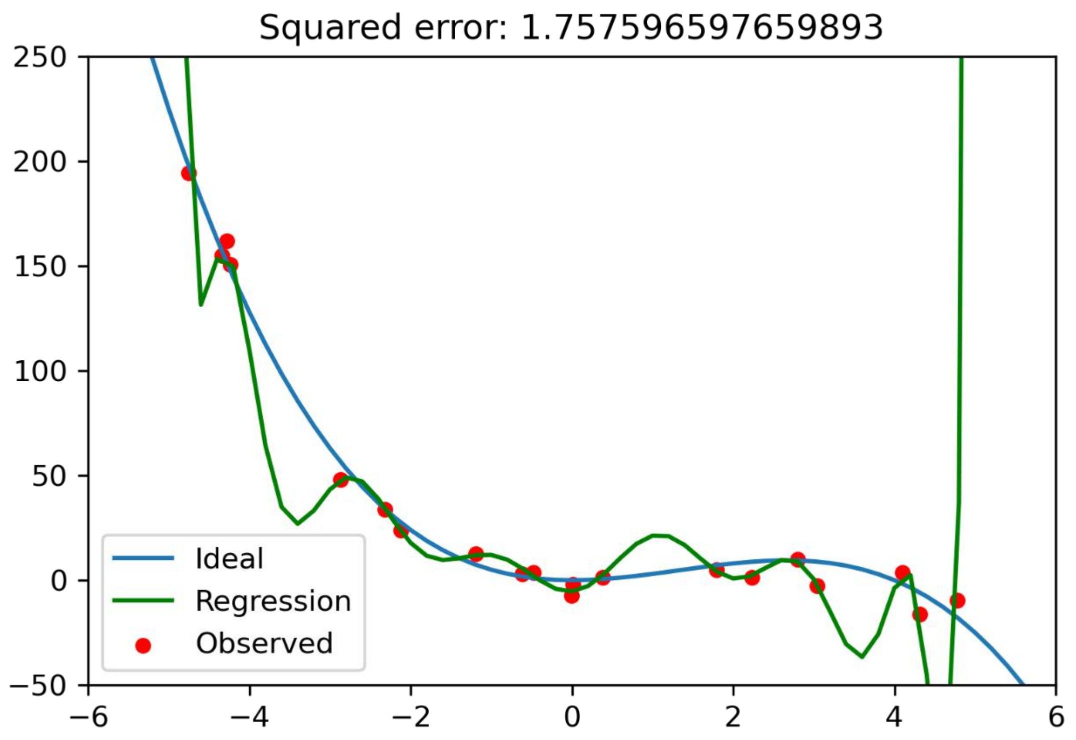
Tenth degree design matrix: $\phi(x) = [1, x, x^2, \ldots, x^{10}]$



Increasing model complexity: a 15-degree polynomial model

On the same problem instance $(\mathbf{X}, \mathbf{y})$

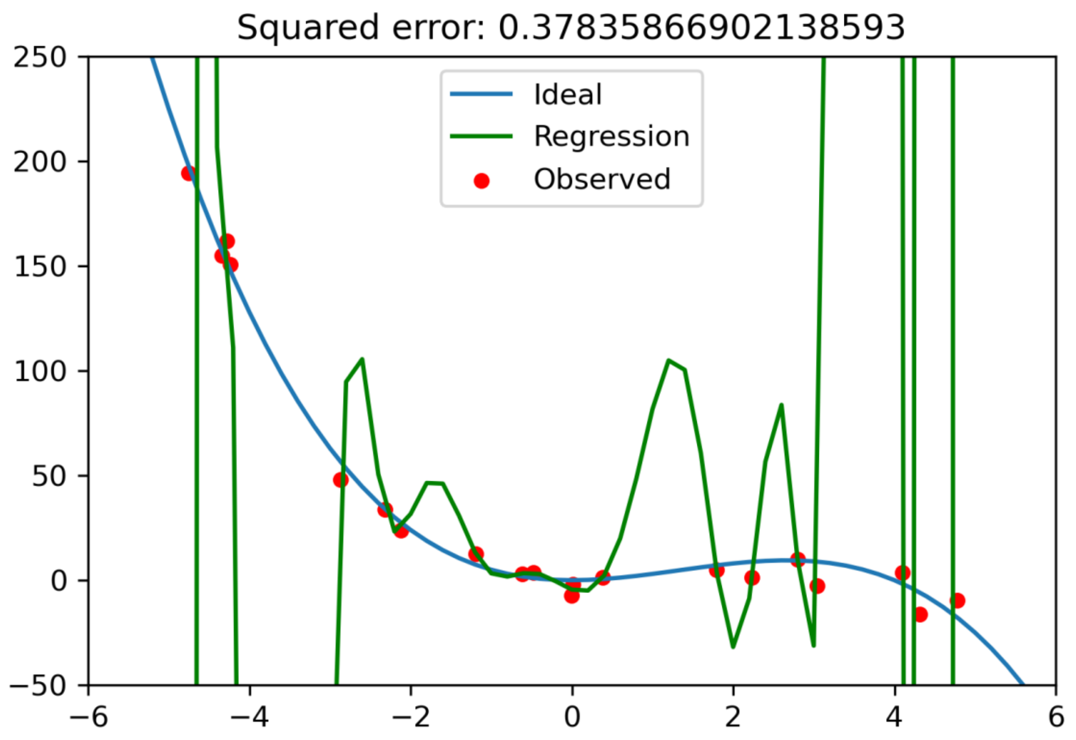
Fifteenth degree design matrix: $\phi(x) = [1, x, x^2, \ldots, x^{15}]$



Increasing model complexity: a 20-degree polynomial model

On the same problem instance $(\mathbf{X}, \mathbf{y})$

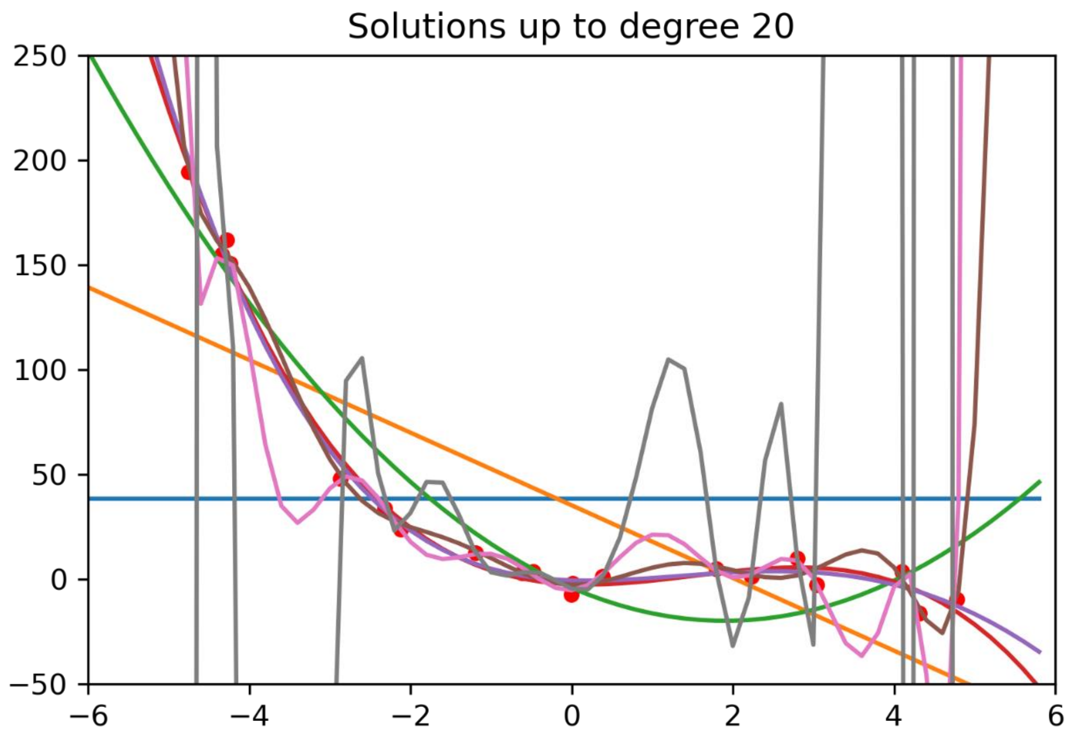
Twentieth degree design matrix: $\phi(x) = [1, x, x^2, \ldots, x^{20}]$



Increasing model complexity: when should we stop?

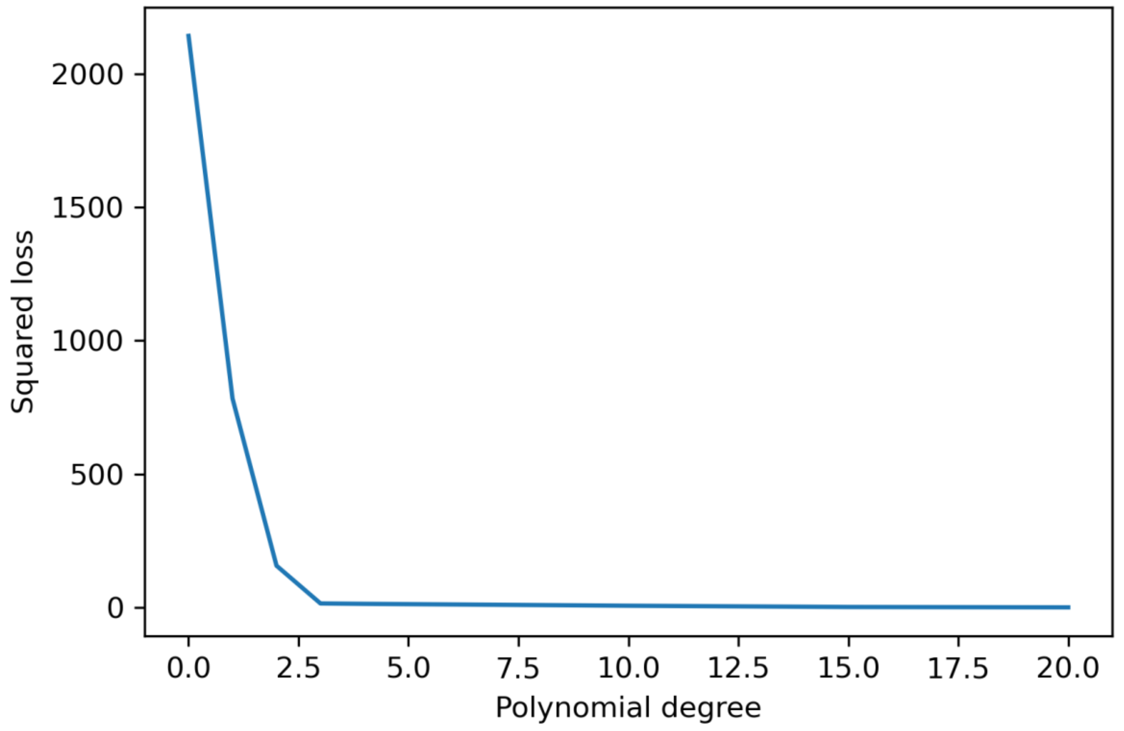
On the same problem instance $(\mathbf{X}, \mathbf{y})$

High degree design matrix: $\phi(x) = [1, x, x^2, \ldots, x^N]$



A look at the fitting cost

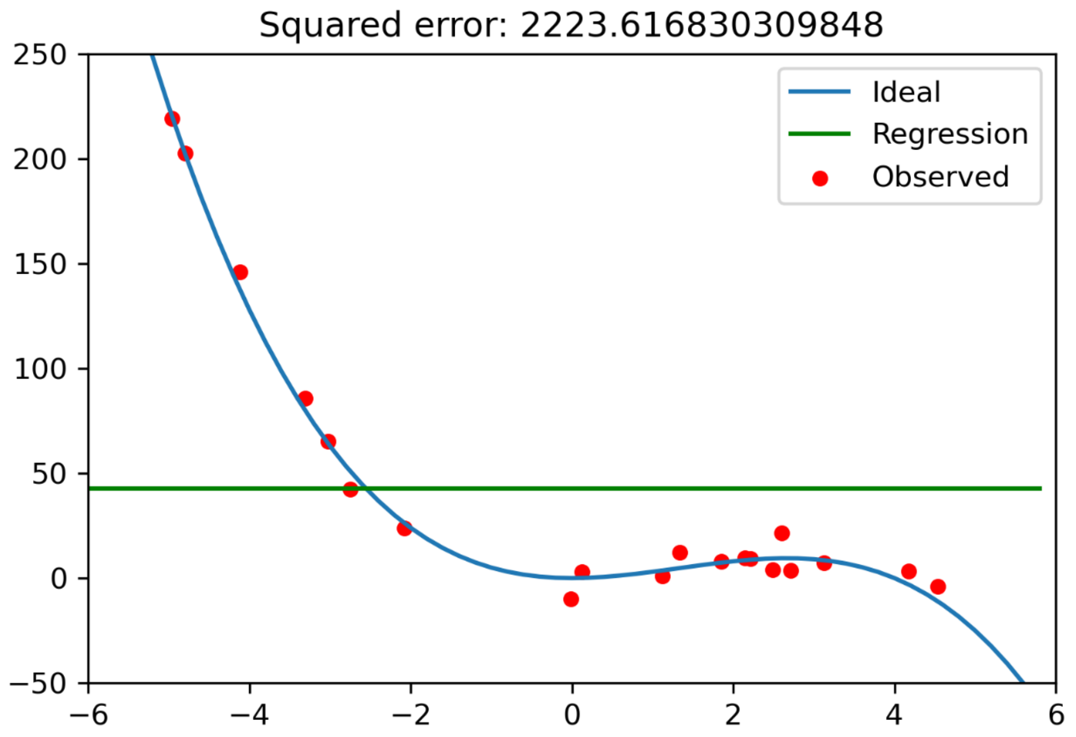
If we analyse squared loss as a function of polynomial degree, we see that error quickly drops, and then stabilizes:



The fitting spectrum: simple models

“Simpler" models have high bias and low variance, and tend to **underfit**.

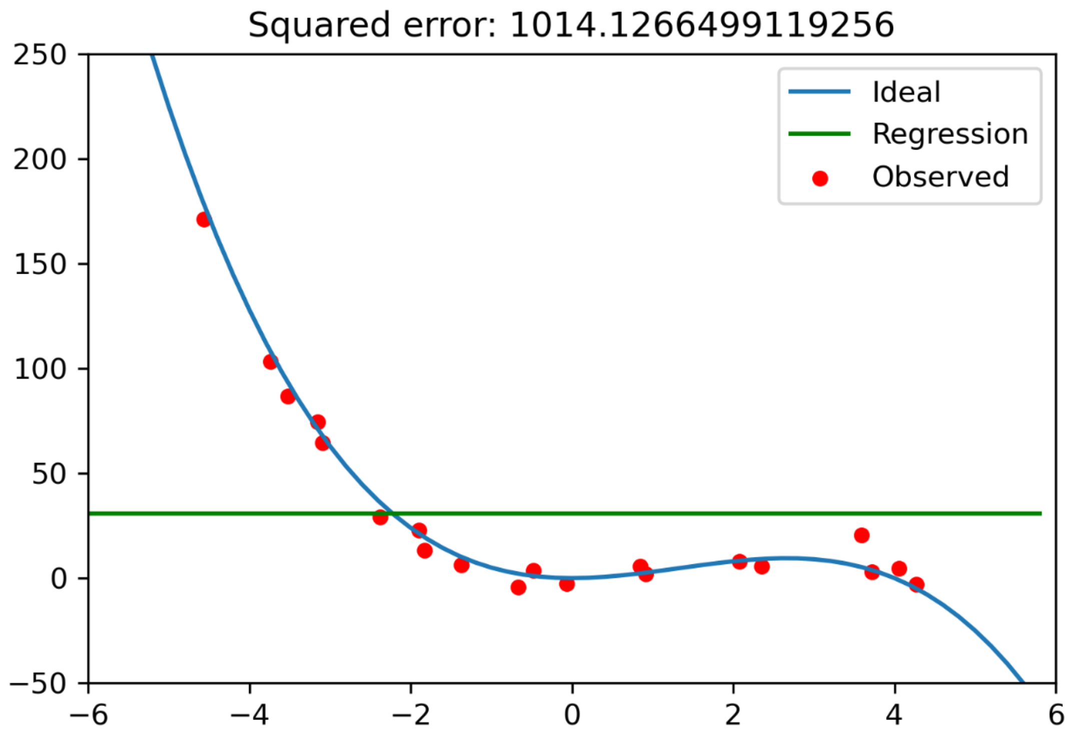
This means that if you repeat fitting over multiple problem instances, all solutions will be biased towards a particular solution:



The fitting spectrum: simple models

“Simpler" models have high bias and low variance, and tend to **underfit**.

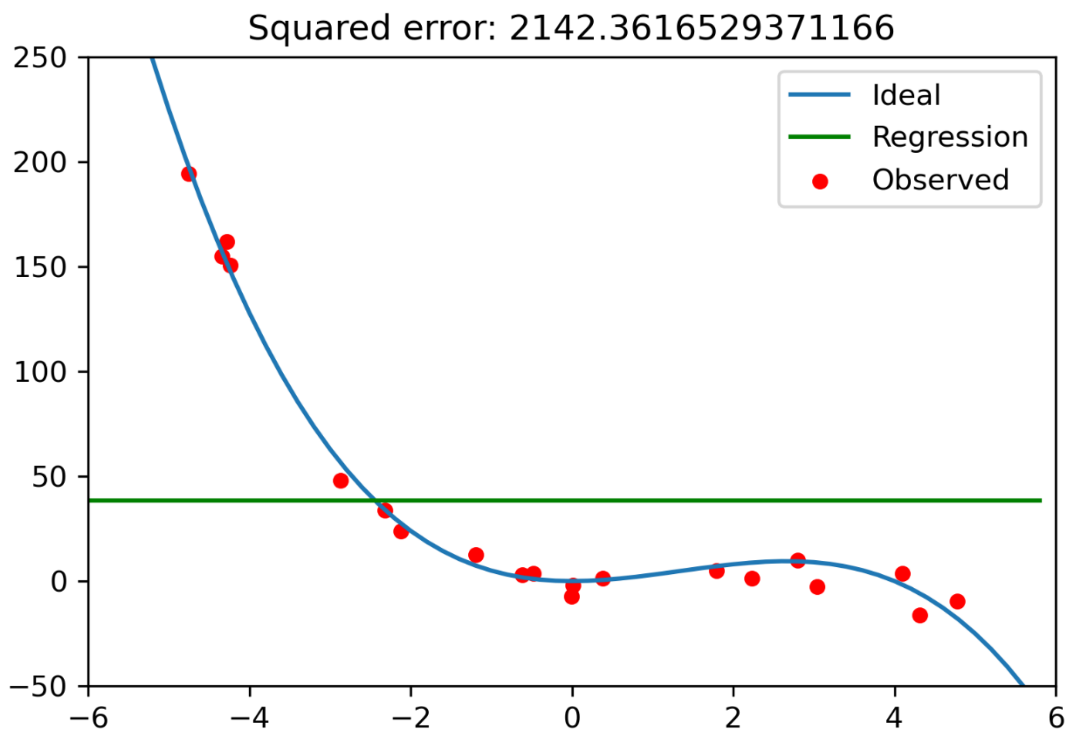
This means that if you repeat fitting over multiple problem instances, all solutions will be biased towards a particular solution:



The fitting spectrum: simple models

“Simpler" models have high bias and low variance, and tend to **underfit**.

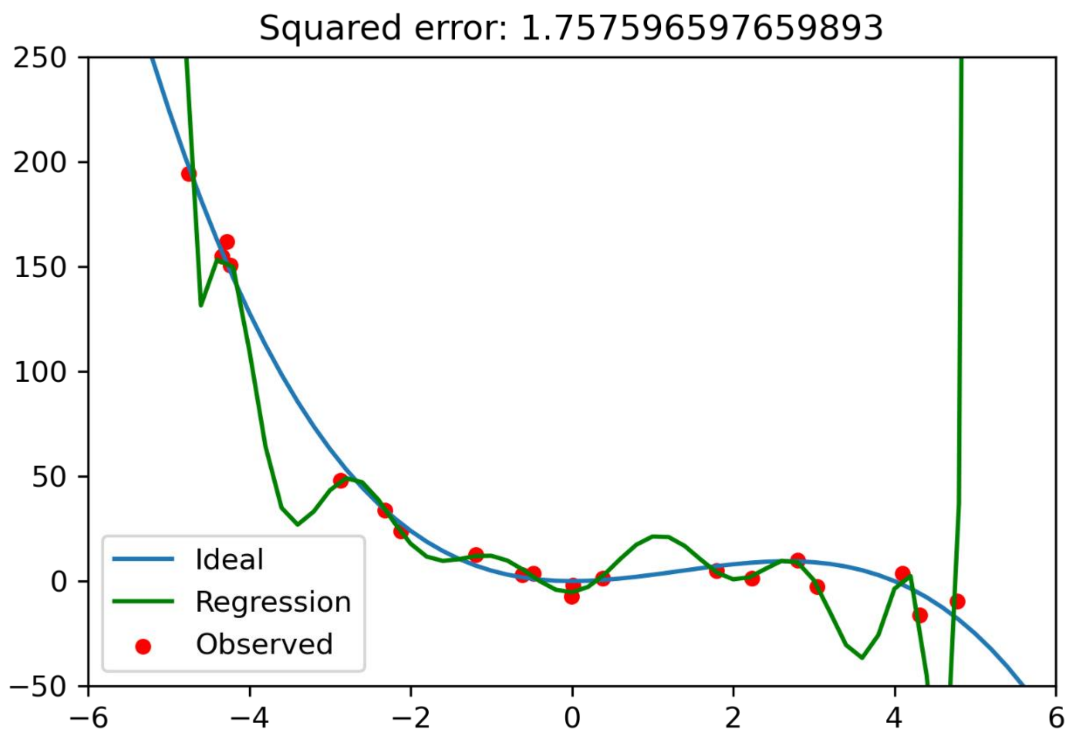
This means that if you repeat fitting over multiple problem instances, all solutions will be biased towards a particular solution:



The fitting spectrum: complex models

“Complex" models have low bias and high variance, and tend to **overfit**.

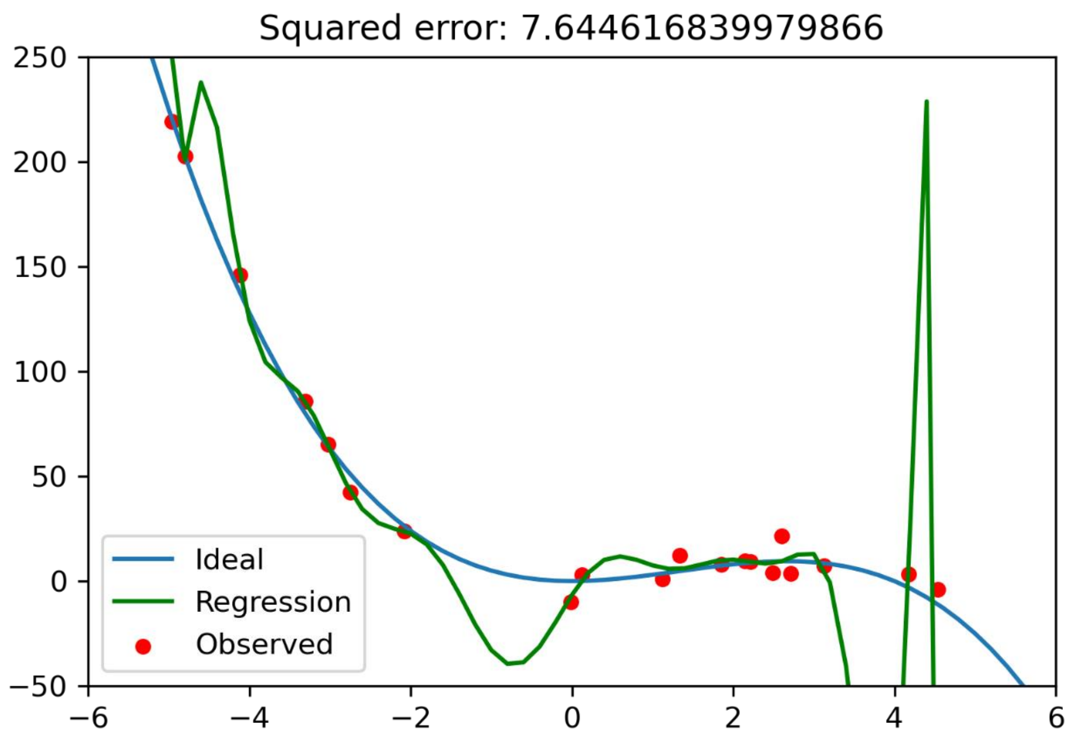
This means that if you repeat fitting over multiple problem instances, all solutions will be all over the place:



The fitting spectrum: complex models

“Complex" models have low bias and high variance, and tend to **overfit**.

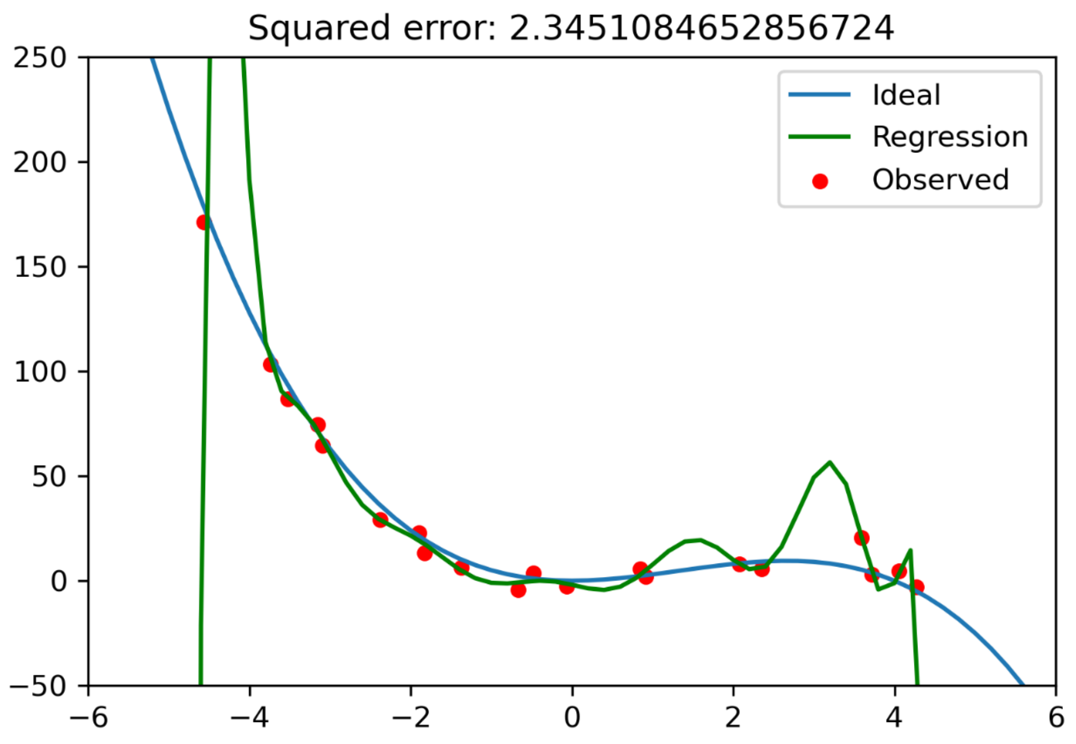
This means that if you repeat fitting over multiple problem instances, all solutions will be all over the place:



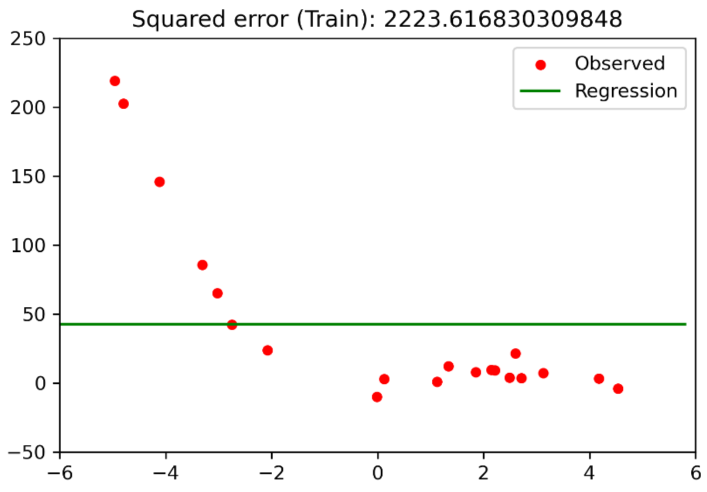
The fitting spectrum: complex models

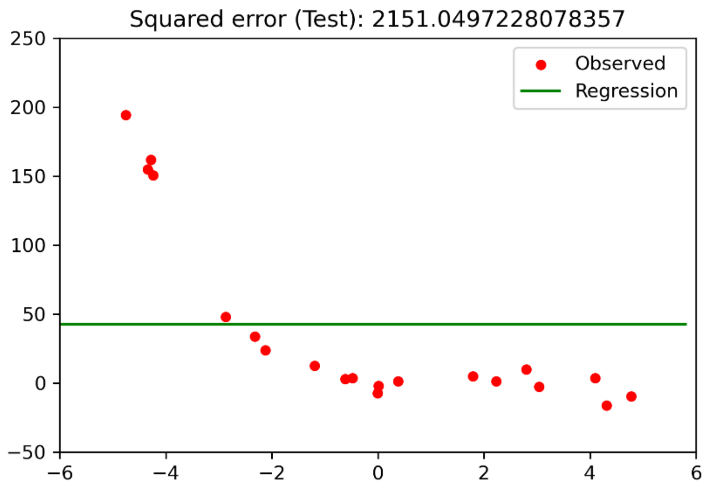
“Complex" models have low bias and high variance, and tend to **overfit**.

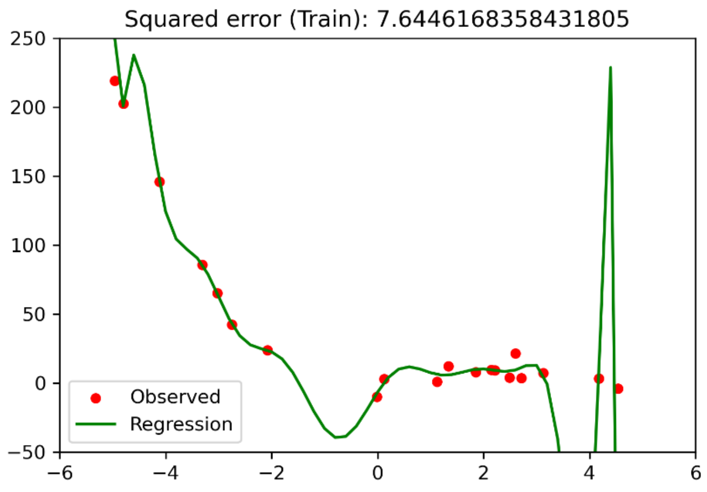
This means that if you repeat fitting over multiple problem instances, all solutions will be all over the place:

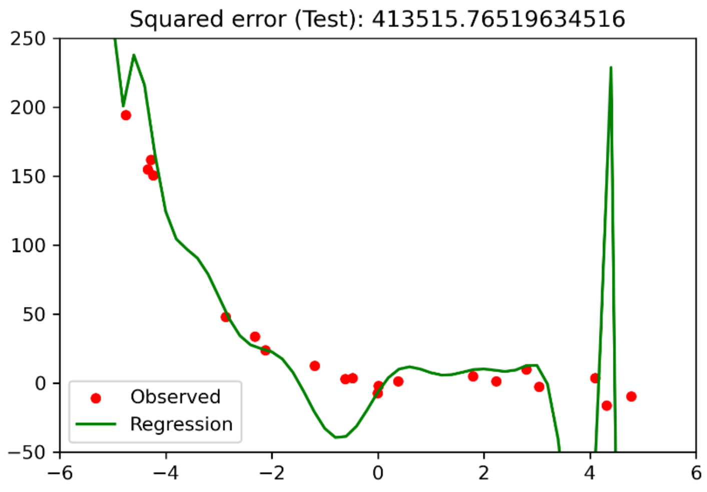


Generalisation









The Bias/Variance trade-off

What we are witnessing here is a simple instance of a much studied principle in statistics, pattern recognition and machine learning, known as the **bias/variance trade-off** (or sometimes bias/variance decomposition)

It is a formal way of decomposing the errors in estimation

- The **bias** in an estimator is the error resulting from bad assumptions made in your model; learning can miss relevant information in the data due to bias
- The **variance** in an estimator is error due to sensitivity to fluctuations in the data; estimators with high variance fit noise and can miss the underlying ideal function

Though rarely practical to compute precisely, the trade-off between bias and variance in an estimator is a fundamental concept that greatly aids understanding of how models behave

The Bias/Variance trade-off

The bias/variance trade-off is usually described like this:

$$error = Bias(\hat{f}; k)^2 + Var(\hat{f}; k) + \sigma^2$$

where k is the “complexity” of your model $\hat{f}$ and σ^2 is the irreducible error (noise)

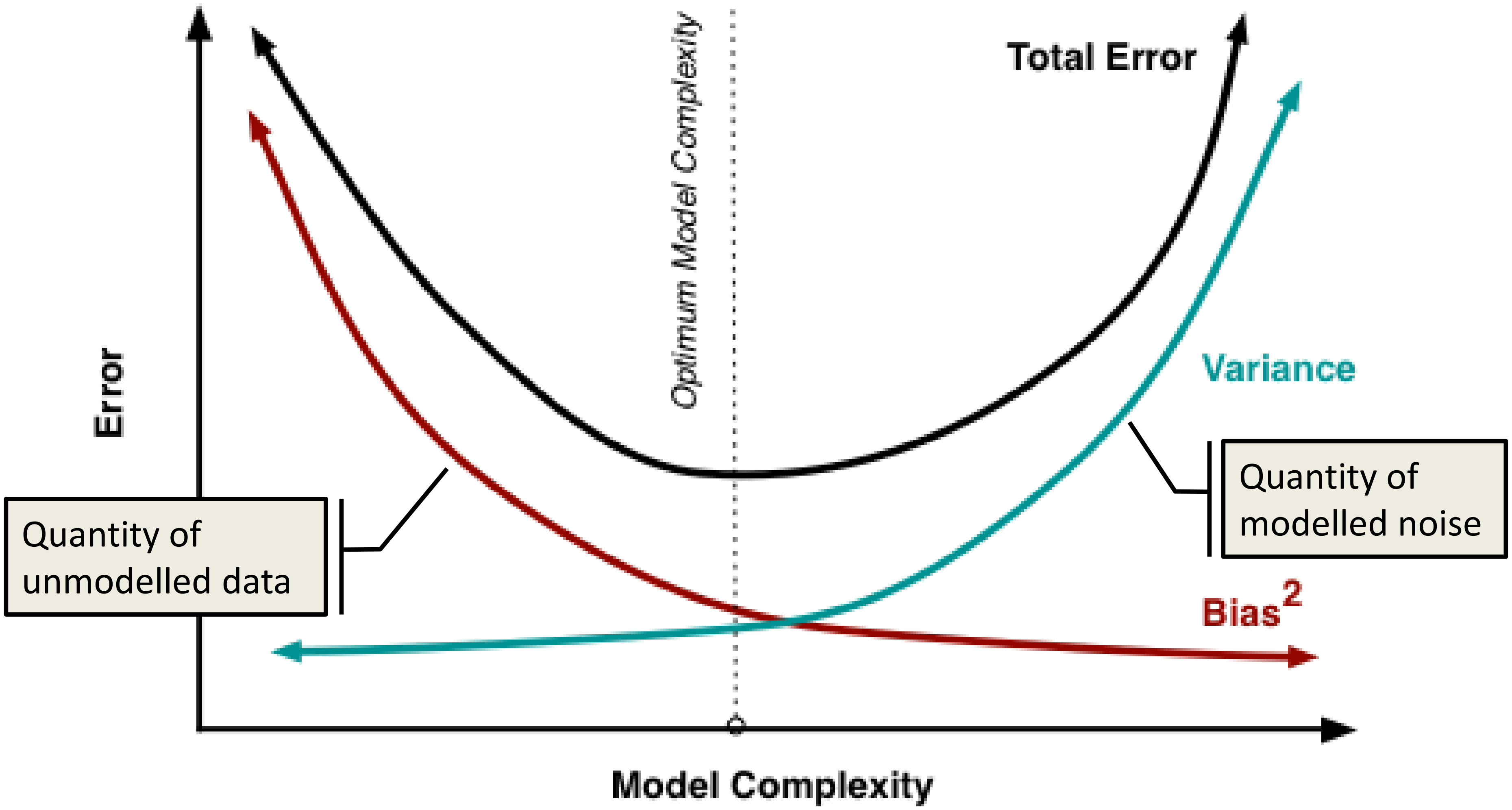
The irreducible error σ^2 is the unavoidable error any estimator will have due to noise in the observation model

Explicitly calculating the bias and variance of an estimator can be challenging. It depends

- on the underlying statistics of the phenomena you want to estimate
- on the loss function
- on the rather vague notion of how “complex” your model is

In order to control our fit, we need a way of characterizing this complexity

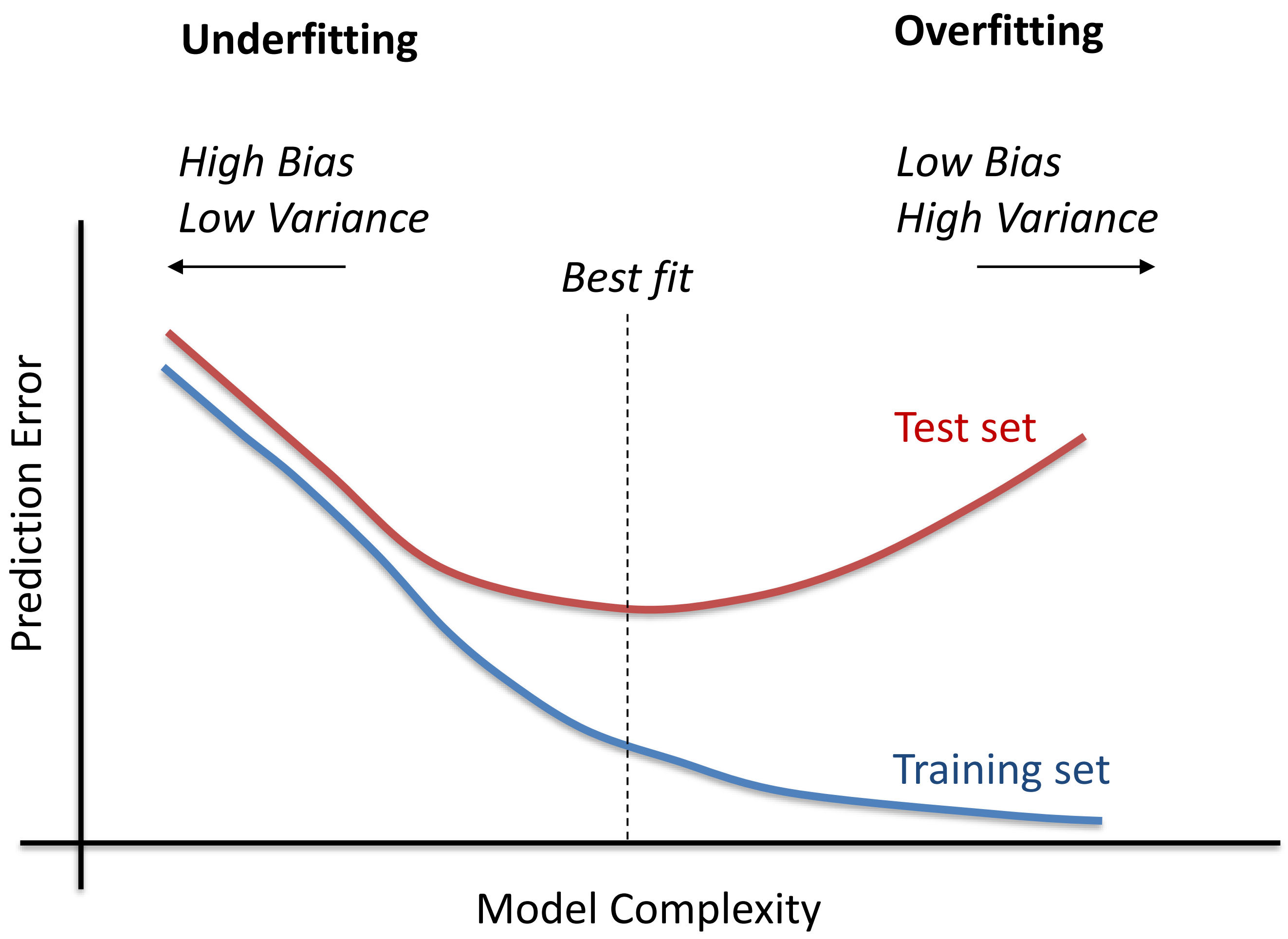
The Bias/Variance trade-off



Quantity of unmodelled data

Quantity of modelled noise

Generalisation



REGULARIZATION

Our cumbersome measure of complexity

In our simple examples, we had the awkward complexity parameter N, the degree of polynomial to fit: $\phi(x) = [1, x, x^2, \ldots, x^N]$

Sometimes we don't even have such an explicit complexity parameter

For data of higher dimensionality, we could use a product term embedding:

$$\phi(x) = [x_0 x_0, x_0 x_1, \ldots, x_0 x_n, x_1 x_0, \ldots, x_n x_n]$$

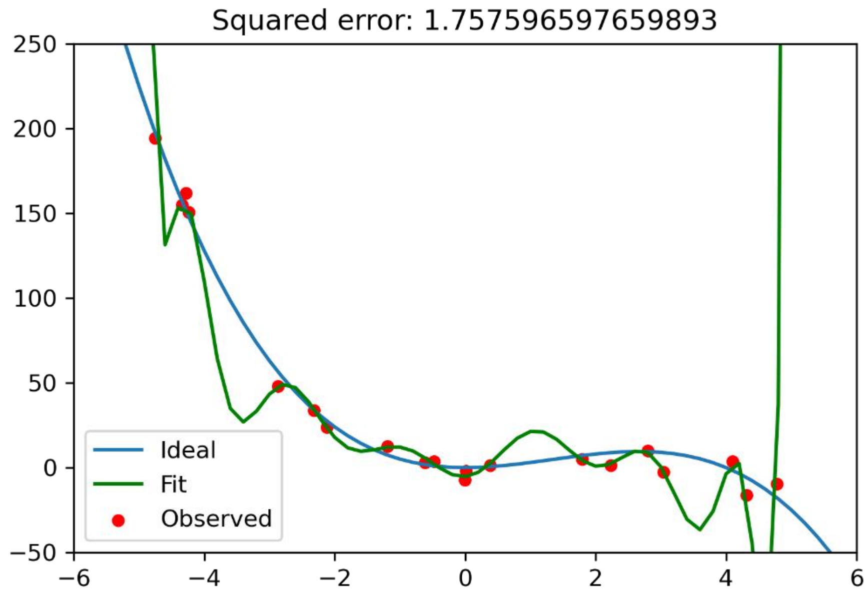
For such an embedding it is unclear how to parametrize complexity without being completely arbitrary

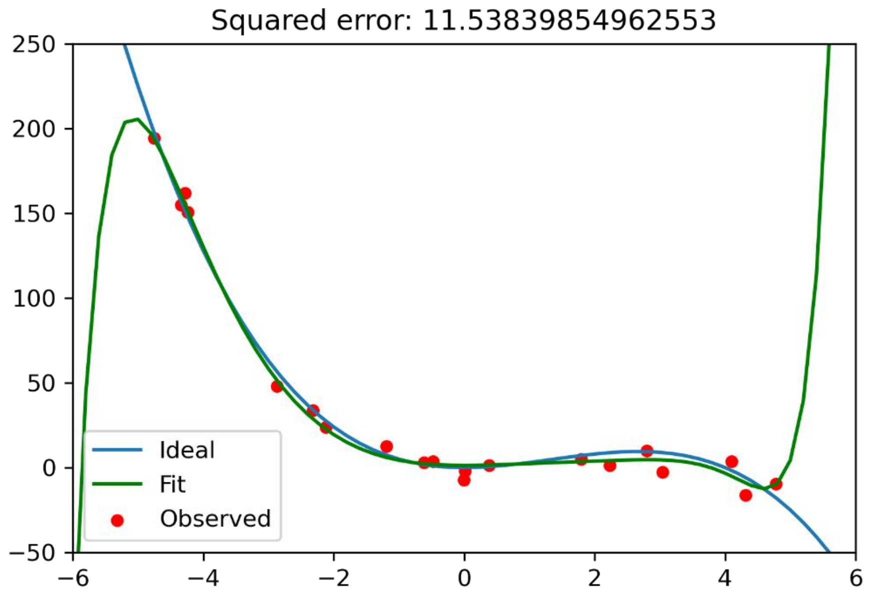
Sometimes data is just high-dimensional (hundreds of thousands of dimensions, at times) - in the case of high-dimensional data we do not (yet) have any way of controlling the complexity of representation

Instead of explicitly controlling the complexity of representation, we will control the complexity of the fit

Same model, different complexity

These two fits are both 15-degree polynomials



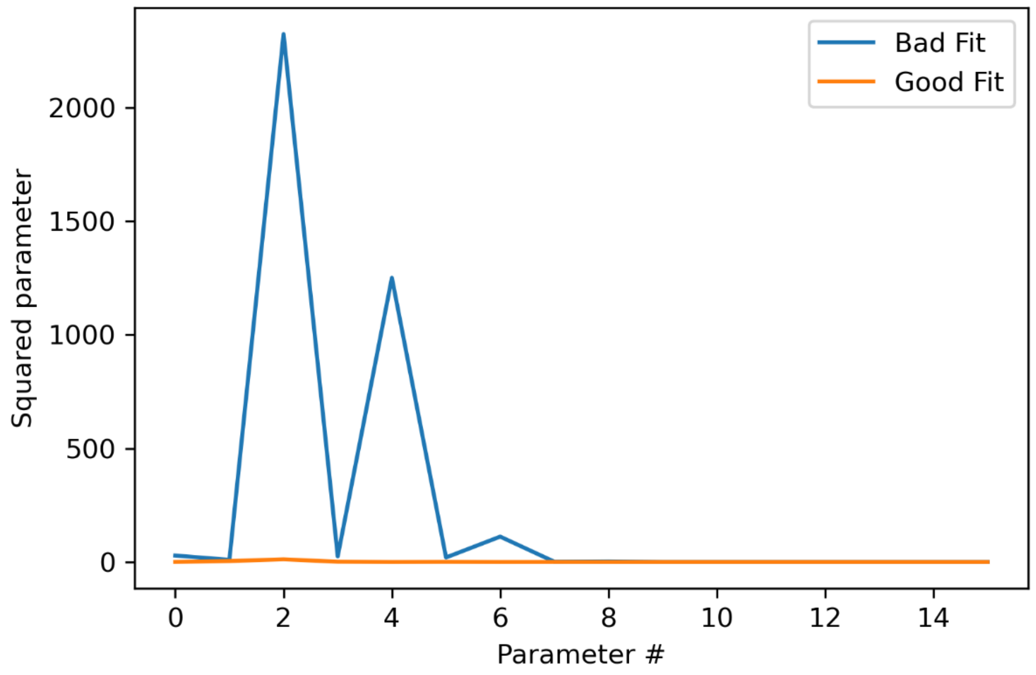


Which one is “better”?

What is the difference?

A look in the parameters

A look into the squared parameters (w_i^2) reveals:



Parameter Magnitude

It turns out that parameter magnitude is not a bad measure of fit complexity
What we do is *penalize* high parameter values by adding a *regularization term* to the loss:

$$
\begin{aligned}
J(\mathbf{w}; \mathbf{X}, \mathbf{y}, \phi) &= \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^{T}\mathbf{w}, y^{(i)}\right) + \lambda(\mathbf{w}^{\boldsymbol{T}}\mathbf{w}) \\
&= \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^{T}\mathbf{w}, y^{(i)}\right) + \lambda \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} w_i^2
\end{aligned}
$$

Parameter Magnitude

$$J(\mathbf{w};\mathbf{X},\mathbf{y},\phi) = \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^{T}\mathbf{w}, y^{(i)}\right) + \lambda(\mathbf{w}^{\boldsymbol{T}}\mathbf{w})$$

$$= \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^{T}\mathbf{w}, y^{(i)}\right) + \lambda\sum_{\boldsymbol{k}=\mathbf{1}}^{\boldsymbol{n}} w_k^2$$

The new parameter λ is called the **regularization coefficient**
It controls the trade-off between fitting the data (small λ, high variance) and minimizing model complexity (high λ, low variance)
This form (quadratic loss, quadratic regularizer) is variously called *ridge regression, Tikhonov regularization, or* ℓ_2 *−regularized linear regression*

Gradient

We minimize, as usual, by taking the gradient and setting to zero

$$\frac{\partial}{\partial w_i} J(\mathbf{w}; \mathbf{X}, \mathbf{y}, \phi, \lambda) = \frac{\partial}{\partial w_i} \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^T \mathbf{w}, y^{(i)}\right) + \frac{\partial}{\partial w_i} (\lambda \mathbf{w}^T \mathbf{w})$$

$$= \frac{\partial}{\partial w_i} \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^{(i)}\right)^T \mathbf{w}, y^{(i)}\right) + \lambda \frac{\partial}{\partial w_i} (\mathbf{w}^T \mathbf{w})$$

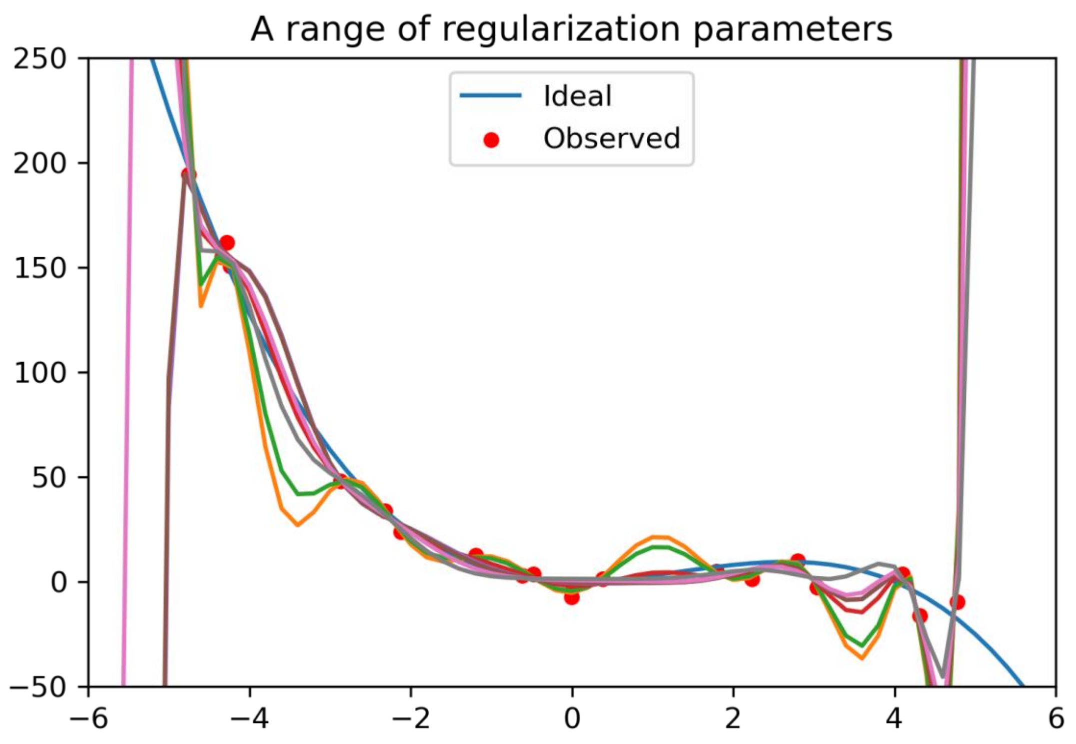
$$= \text{least squares gradient} + 2\lambda \mathbf{w}$$

So, gradient descent for regularized linear regression is identical to that of normal linear regression with an extra term for partial derivatives of the regularization term

A parametrised family of solutions

The parameter λ now gives us a way to control the complexity of a fit

Note that it is independent of the basis used; it is completely generic



Ridge Regression

Quadratic regularization is a powerful tool. However, we have been dancing around an important issue: we usually have no knowledge of the ideal function we wish to estimate

This means that, without making assumptions, it can be difficult to estimate the bias and variance of our estimators

Consequently, it may not be clear which λ to choose

Also, quadratic regularizers yield solutions with small coefficients, but with energy evenly spread out (i.e. they are *dense*).

Addressing (some of) these problems:

- If you have lots of data, split it into multiple folds and use cross validation to estimate your model (and bias and variance)
- If you need a sparse solution, change the regularizer (and pay the price)

The Lasso – ℓ_1 loss

The Lasso loss function, is formed by substituting the quadratic penalty with a penalty on the absolute value of the coefficients:

$$J(\mathbf{w};\mathbf{X},\mathbf{y},\phi)=\sum_{i=1}^{m}L\left(\phi\left(\mathbf{x}^{i}\right)^{T}\mathbf{w},y^{i}\right)+\lambda\sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{n}}|w_i|$$

The resulting estimator is called the Lasso for "Least Absolute Shrinkage and Selection Operator"

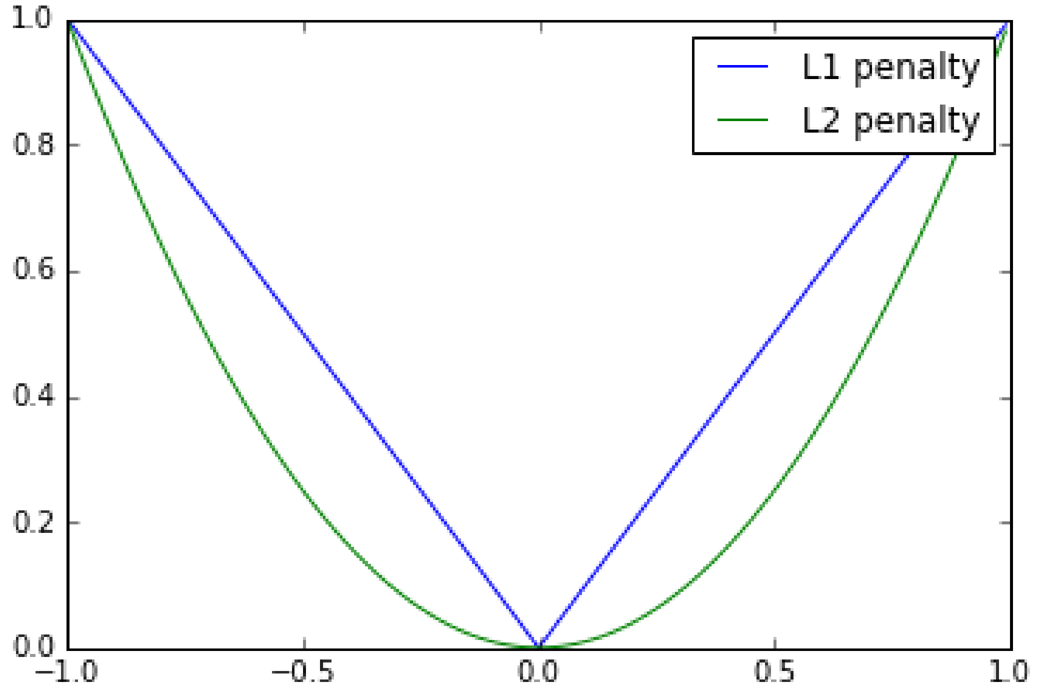
It is called "Lasso" because it has the effect of selecting a few (number depending on λ) coefficients with non-zero magnitude

ℓ_1 (lasso) vs ℓ_2 (ridge) loss

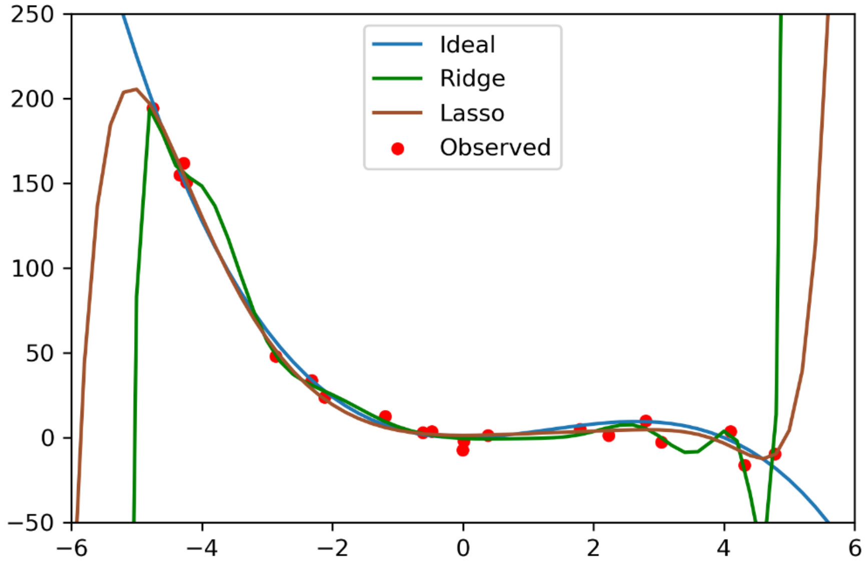
The small variation between the ℓ_1 (lasso) and the ℓ_2 (ridge) loss has a profound effect on the solution

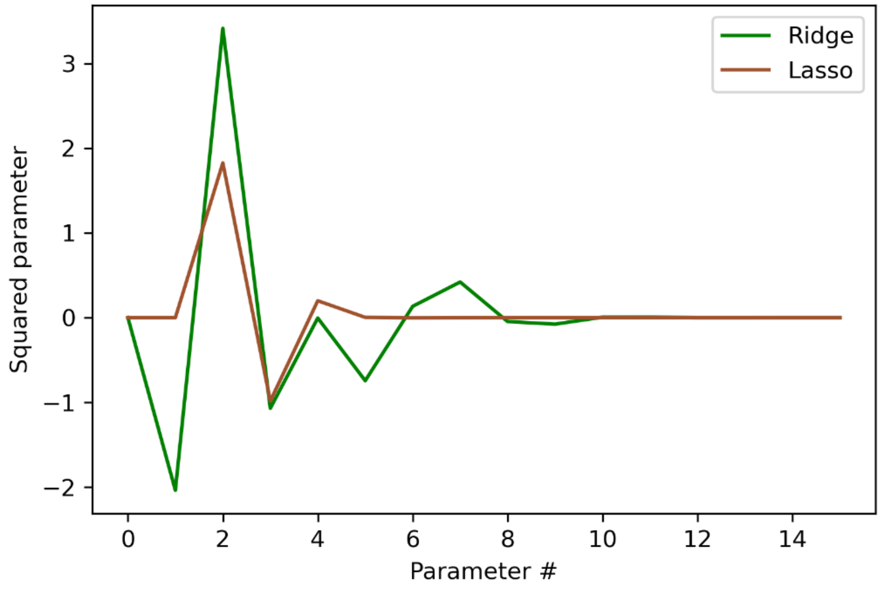
To see this imagine how the addition of a quadratic versus absolute penalty “encourages” solutions to evolve:

- The quadratic (ℓ_2) penalty discourages big coefficients (but lots of small ones are OK)
- The linear (ℓ_1) penalty never “discounts" already-small coefficients



A comparison of ridge and lasso





The Lasso comes pretty close to capturing the original ideal

The ridge regressor can't help but "wiggle" its way through the data

The Lasso is performing a type of model selection, while the ridge regressor is a dense linear solution

Problem: we can no longer use the normal equation or simple gradient descent to solve

Note: Double Descent in Deep Nets

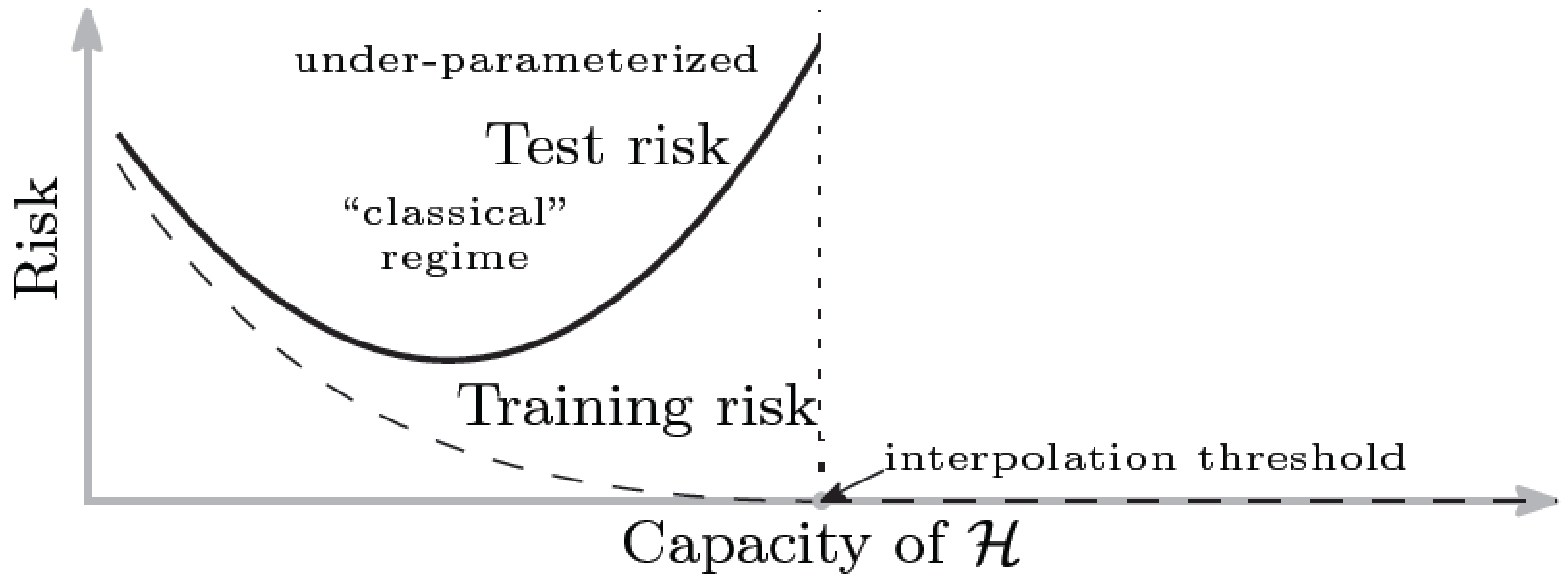


Image from:
"Reconciling modern machine learning practice and the bias-variance trade-off", https://arxiv.org/abs/1812.11118

Note: Double Descent in Deep Nets

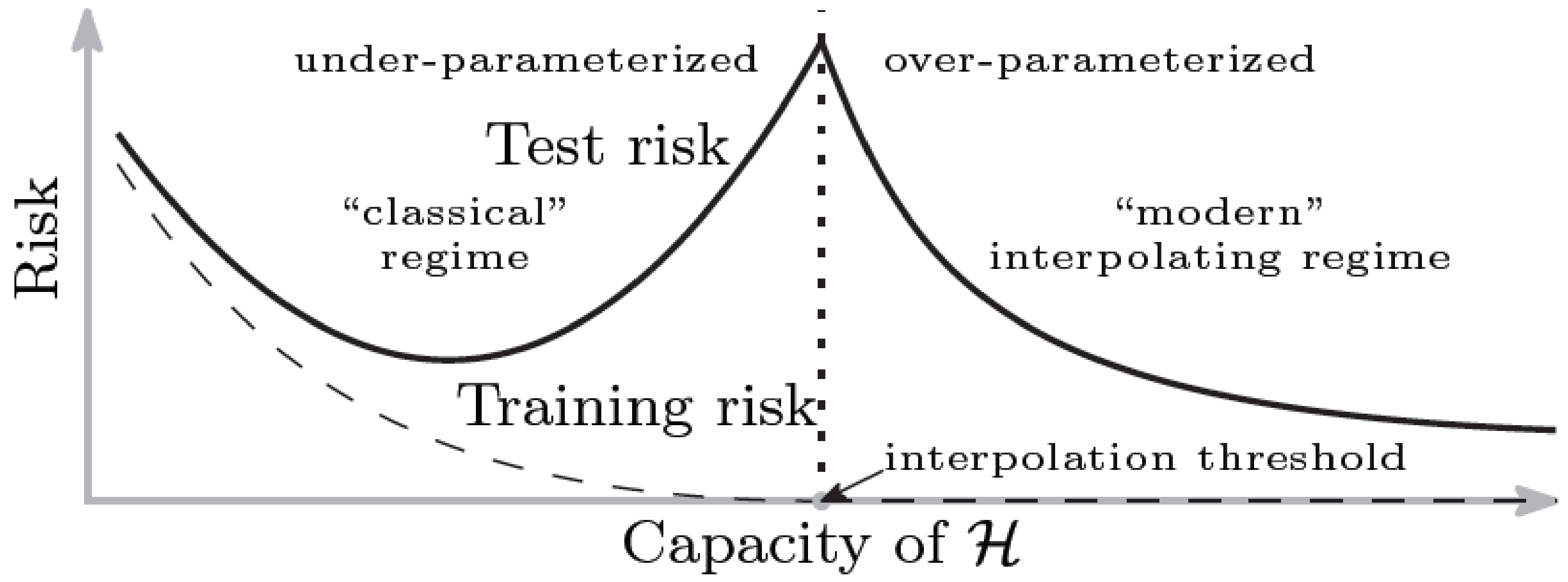


Image from:
"Reconciling modern machine learning practice and the bias-variance trade-off", https://arxiv.org/abs/1812.11118

Note: Double Descent in Deep Nets

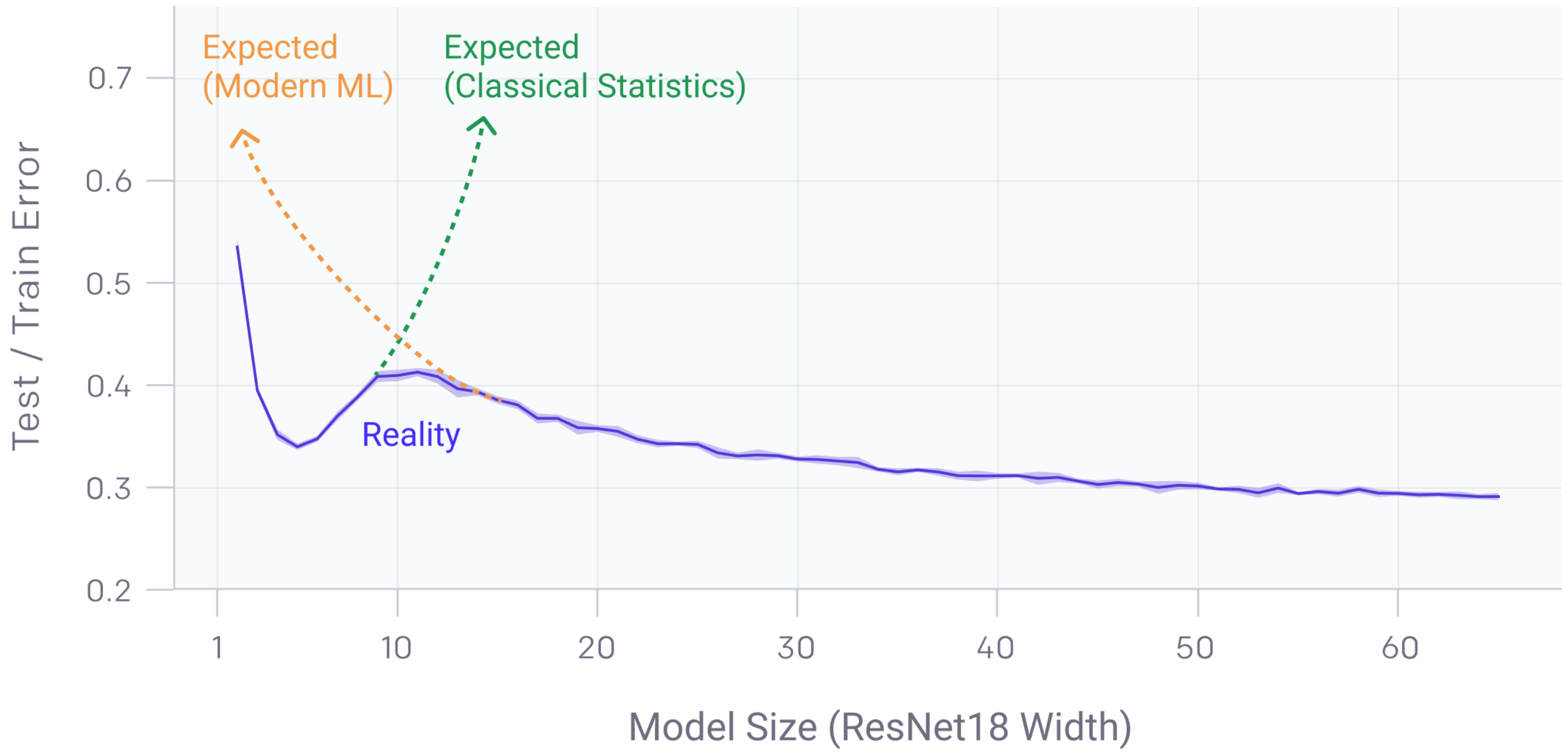


Image from:
"Deep Double Descent: Where Bigger Models and More Data Hurt", https://arxiv.org/abs/1912.02292
https://openai.com/blog/deep-double-descent/

A Bayesian view to curve fitting

PARAMETER ESTIMATION

The Rules of Probability

Sum Rule $$p(X) = \sum_{Y} p(X, Y)$$

Product Rule $$p(X, Y) = p(Y \mid X)\, p(X)$$

Bayes' Theorem

$$p(Y|X) = \frac{p(X|Y)\,p(Y)}{p(X)}$$

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$

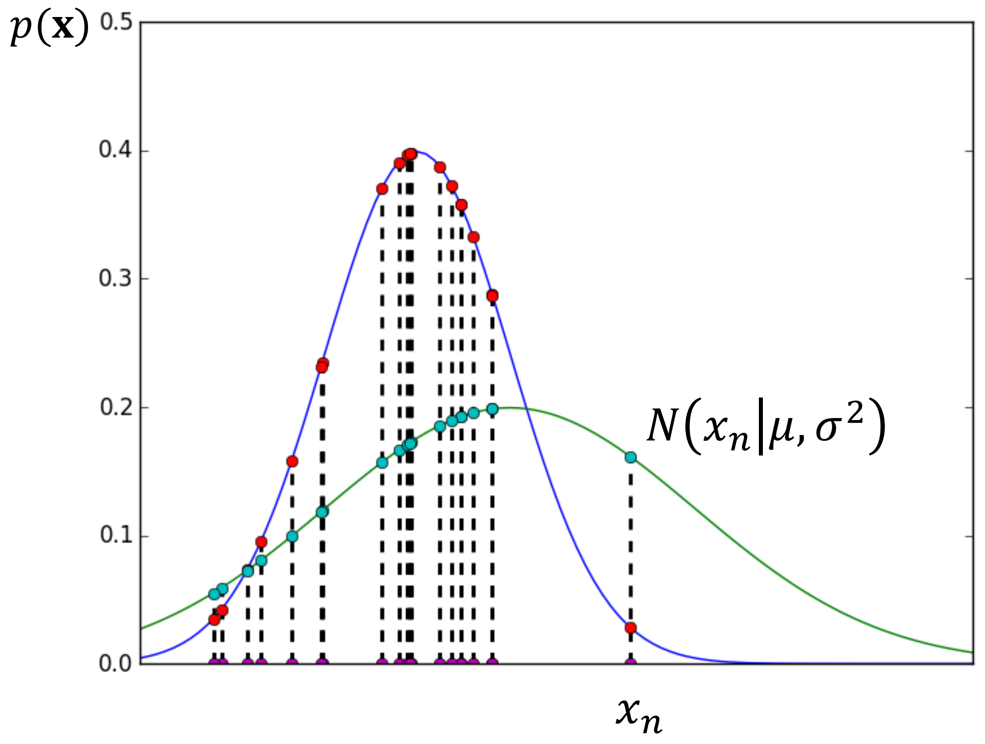
Gaussian parameter estimation

Imagine a set of samples that are drawn independently from the same distribution. The samples are **independent and identically distributed**

Suppose the underlying distribution is a Gaussian. We want to estimate the parameters of the Gaussian *(μ, σ^2)* from the samples we have

Intuition: View parameters as fixed unknown quantities. Maximise the probability of obtaining the samples

$$\underbrace{p(\mathbf{x}|\mu,\sigma^2)}_{\text{Likelihood function}} = \prod_{n=1}^{N} N(x_n|\mu,\sigma^2)$$



Maximum (Log) Likelihood

$$p(\mathbf{x}|\mu,\sigma^2) = \prod_{n=1}^{N} N(x_n|\mu,\sigma^2)$$

Easier to maximise the log of this function (deal with sums instead of products)

$$\ln p\left(\mathbf{x}|\mu,\sigma^2\right) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(2\pi)$$

$$\mu_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N} x_n \qquad \sigma^2_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}\left(x_n-\mu_{\mathrm{ML}}\right)^2$$

A Bayesian view to curve fitting

CURVE FITTING RE-VISITED

Curve Fitting – The Bayesian Way

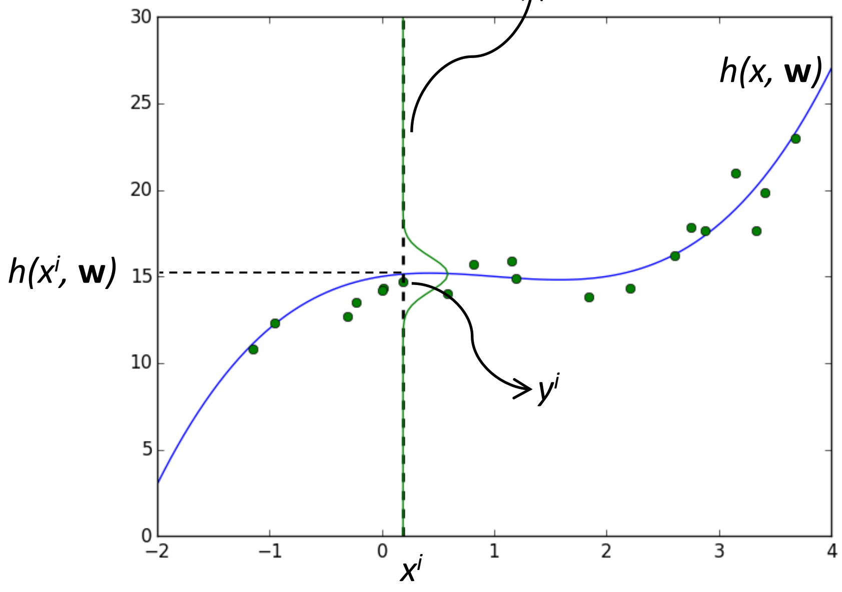
Goal: To be able to make predictions for the target variable *y* given some new value of the input variable *x*, on the basis of a training dataset comprising N input values $\mathbf{x} = (x_1, x_2, \ldots, x_N)^T$ and their corresponding target values $\mathbf{y} = (y_1, y_2, \ldots, y_N)^T$

Intuition: Express our uncertainty over the value of the target variable using a (Gaussian) probability distribution.

The mean of this Gaussian distribution would be the target variable itself *h(x, w)* and it would have some (unknown) precision, say β

Remember $\beta = 1/\sigma^2$

$$p(y|x, \mathbf{w}, \beta) = N(y|h(x, \mathbf{w}), \sigma^2)$$



Hint: remember the generative view of data, underlying "true" function and observed samples corrupted with Gaussian noise?
$y = h(x, \mathbf{w}) + N(0, \sigma)$

Maximum (Log) Likelihood

$$p(\mathbf{y}|\mathbf{x},\mathbf{w},\sigma^2) = \prod_{n=1}^{N} N(y_n|h(x_n,\mathbf{w}),\sigma^2)$$

Easier to maximise the log of this function (deal with sums instead of products)

To determine the coefficients **w** – maximise in respect to **w**

$$\ln p\,(\mathbf{y}|\mathbf{x},\mathbf{w},\sigma^2) = -\underbrace{\frac{1}{2\sigma^2}\sum_{n-1}^{N}\{h(x_n,\mathbf{w}) - y_n\}^2}_{\frac{1}{\sigma^2}J(\mathbf{w})} - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(\,2\pi)$$

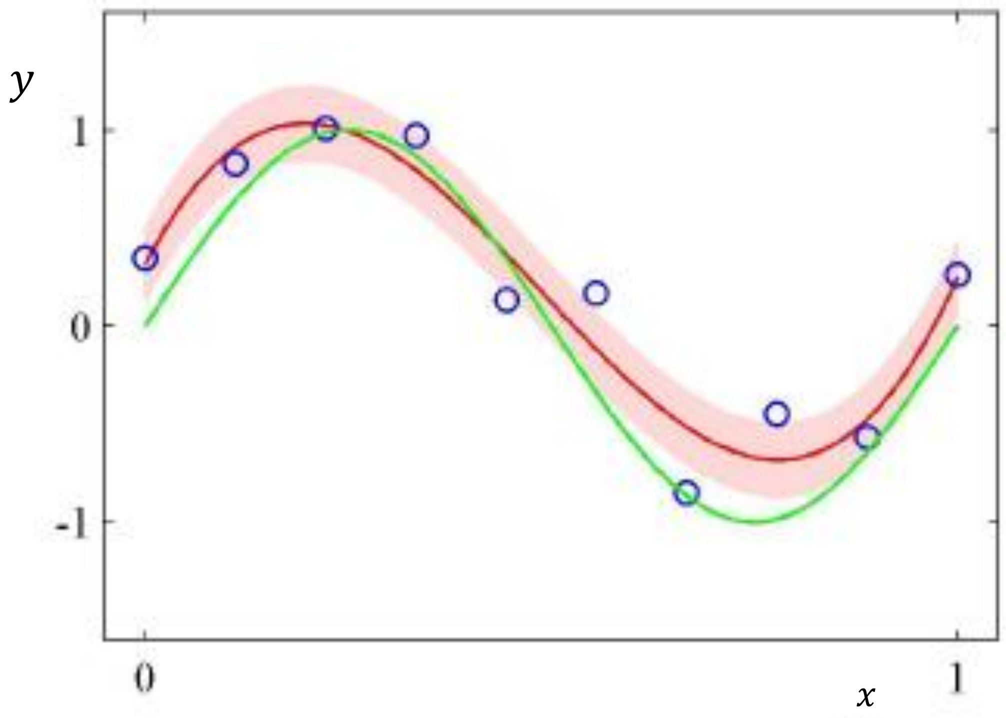
Omit the last two terms as they do not depend on **w**. Ignore σ^2. Turns out it is equivalent to minimising the sum-of-squares error function!

To determine the standard deviation minimise with respect to σ^2, given $\mathbf{w}_{\mathrm{ML}}$

$$\frac{1}{\beta_{ML}} = \sigma_{ML}^2 = \frac{1}{N}\sum_{n=1}^{N}\{h(x_n,\mathbf{w}_{ML}) - y_n\}^2$$

Predictive Distribution

$$p(y|x, \mathbf{w}_{ML}, \beta_{ML}) = N(y|h(x, \mathbf{w}_{ML}), \beta_{ML}^{-1})$$



Our predictions are now expressed in terms of the *predictive distribution* of the target value, which gives the probability distribution over *y* given an input value *x*, than simply a point estimate.

MAP: A step towards Bayes

Intuition: What if we had some idea about the right parameters in advance… the Bayesian approach gives us the mechanism to take it into account.

M parameters plus the bias

$$p(\mathbf{w}) = N(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}) = \left(\frac{\alpha}{2\pi}\right)^{(M+1)/2} \exp\left\{-\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\right\}$$

α is an "hyperparameter" – controls the distribution of model parameters (**w**)

Using Bayes' theorem, we get the posterior. Determine **w** by finding the most probable value of **w** given the data **x**: maximise posterior probability

$$p(\mathbf{w}|\mathbf{x},\mathbf{y}) \propto p(\mathbf{y}|\mathbf{x},\mathbf{w})p(\mathbf{w})$$

Maximising the posterior turns out to be equivalent to minimising the regularized sum of squared error with a regularisation parameter of $\lambda = \alpha/\beta$

$$\beta\tilde{J}(\mathbf{w}) = \frac{\beta}{2}\sum_{n=1}^{N}\{h(x_n,\mathbf{w}) - y_n\}^2 + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

Bayesian Curve Fitting

We have found the most probable values for **w**, but this is still far from a true Bayesian treatment.

Intuition: any values are possible for **w**, with some associated probability. If we consistently apply the sum and product rules, we end up integrating over all possible values, instead of using a point estimate of **w** for our predictions.

$$p(y|x,\mathbf{x},\mathbf{y}) = \int p(y|x,\mathbf{w})p(\mathbf{w}|\mathbf{x},\mathbf{y})d\mathbf{w}$$

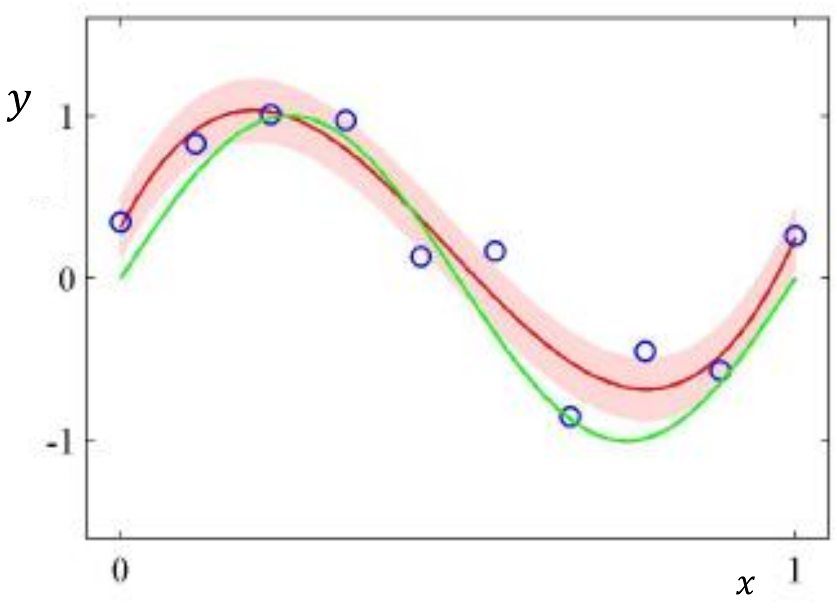
This can be calculated analytically: $p(y|x,\mathbf{x},\mathbf{y}) = N(y|m(x),\boxed{s^2(x)})$

$$s^2(x) = \underbrace{\beta^{-1}}_{\text{Uncertainty due to noise on target variables}} + \underbrace{\varphi(x)^T\mathbf{S}\varphi(x)}_{\text{Uncertainty in the parameters } \mathbf{w}}$$

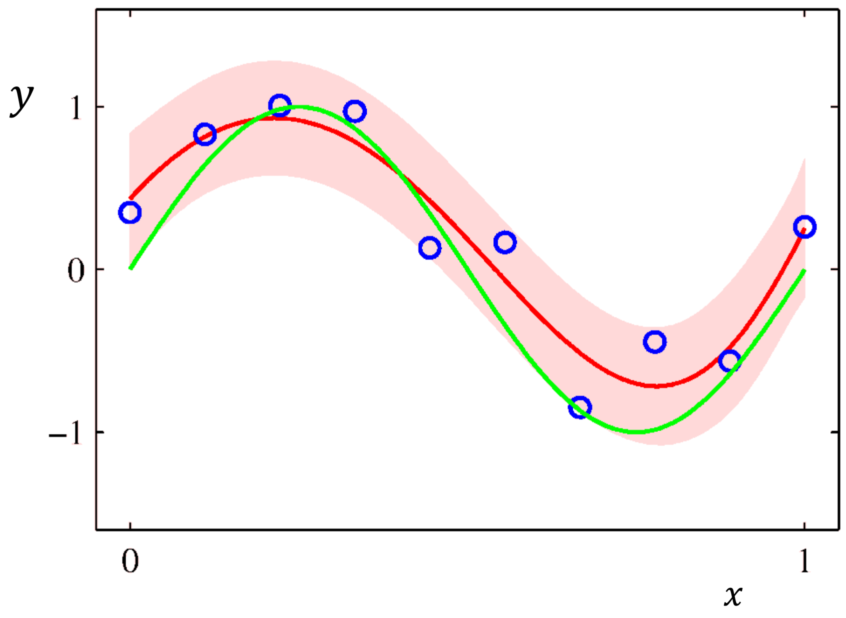
Bayesian Predictive Distribution

$p(y|x, \mathbf{w}_{ML}, \beta_{ML}) = N(y|h(x, \mathbf{w}_{ML}), \beta_{ML}^{-1})$

$p(y|x, \mathbf{x}, \mathbf{y}) = N(y|m(x), s^2(x))$



Predictive distribution



Bayesian Predictive distribution

Literate Models for Computer Vision: Combining vision, language and reading

WHEN: Monday 20 December 2021 from 10:00 to 17.00 CET
WHERE: Online
HOW TO REGISTER: https://www.i-aida.org/course/vision-and-language-reading-systems-and-multi-modal-representations/

10:00 – 11:00 Introduction and Scene Text Understanding Overview
11:00 – 12:00 Common blocks for multi-modal systems
12:00 – 13:00 Scene text for Fine Grained Image Classification and Cross-modal retrieval
13:00 – 15:00 Lunch break
15:00 – 16:00 Scene text for Captioning and VQA
16:00 – 17:00 Demo session

Resources (I)

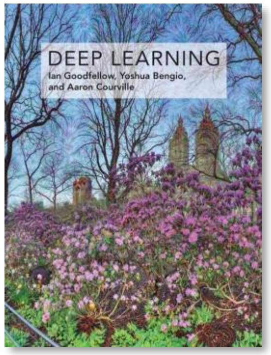
I. Goodfellow, Y. Bengio, A. Courville, "Deep Learning", MIT Press, 2016
http://www.deeplearningbook.org/

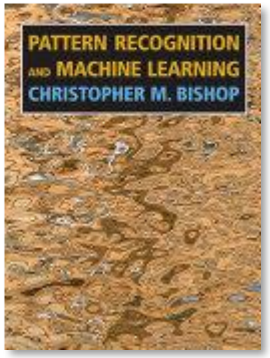
C. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006
http://research.microsoft.com/en-us/um/people/cmbishop/prml/index.htm

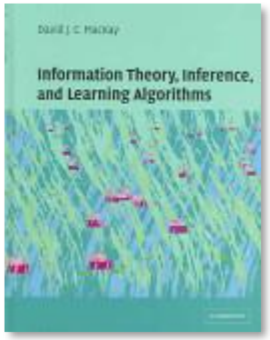
D. MacKay, "Information Theory, Inference and Learning Algorithms", Cambridge University Press, 2003
http://www.inference.phy.cam.ac.uk/mackay/

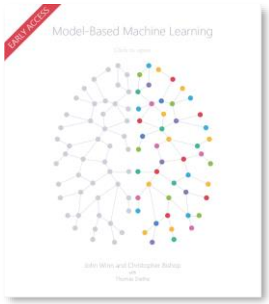
R.O. Duda, P.E. Hart, D.G. Stork, "Pattern Classification", Wiley & Sons, 2000
http://books.google.com/books/about/Pattern_Classification.html?id=Br33IRC3PkQC

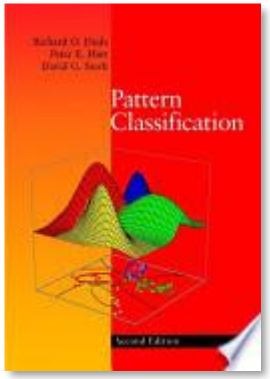
J. Winn, C. Bishop, "Model-Based Machine Learning", early access
http://mbmlbook.com/











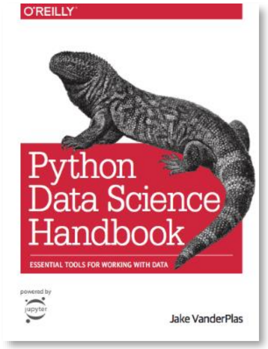
Resources (II)

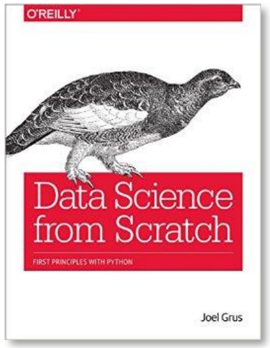
"Python Data Science Handbook: Essential tools for working with Data", Jake VanderPlas, O'Reilly Media, 2016, 1st Ed.
https://jakevdp.github.io/PythonDataScienceHandbook/

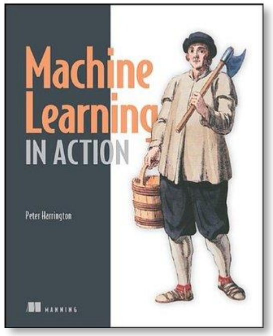
"Data Science from Scratch: First Principles with Python", Joel Grus, O'Reilly Media, 2015, 1st Ed.
https://github.com/joelgrus/data-science-from-scratch

"Machine Learning in Action", P. Harrington, Manning, 2012

"The Foundations of Data Science", Ani Adhikari and John DeNero, [Online]
https://www.inferentialthinking.com/chapters/intro


O'REILLY
Python Data Science Handbook
ESSENTIAL TOOLS FOR WORKING WITH DATA
powered by jupyter
Jake VanderPlas





Further Info

- Many of the slides of these lectures have been adapted from various highly recommended online lectures and courses:
  - Andrew Ng's *Machine Learning Course*, Coursera
    https://www.coursera.org/course/ml
  - Andrew Ng's *Deep Learning Specialization,* Coursera
    https://www.coursera.org/specializations/deep-learning
  - Victor Lavrenko's *Machine Learning* Course
    https://www.youtube.com/channel/UCs7alOMRnxhzfKAJ4JjZ7Wg
  - Fei Fei Li and Andrej Karpathy's *Convolutional Neural Networks for Visual Recognition*
    http://cs231n.stanford.edu/
  - Geoff Hinton's *Neural Networks for Machine Learning, (*ex Coursera)
    https://www.youtube.com/playlist?list=PLiPvV5TNogxKKwvKb1RKwkq2hm7ZvpHz0
  - Luis Serrano's introductory videos
    https://www.youtube.com/channel/UCgBncpylJ1kiVaPyP-PZauQ
  - Michael Nielsen's *Neural Networks and Deep Learning*
    http://neuralnetworksanddeeplearning.com/