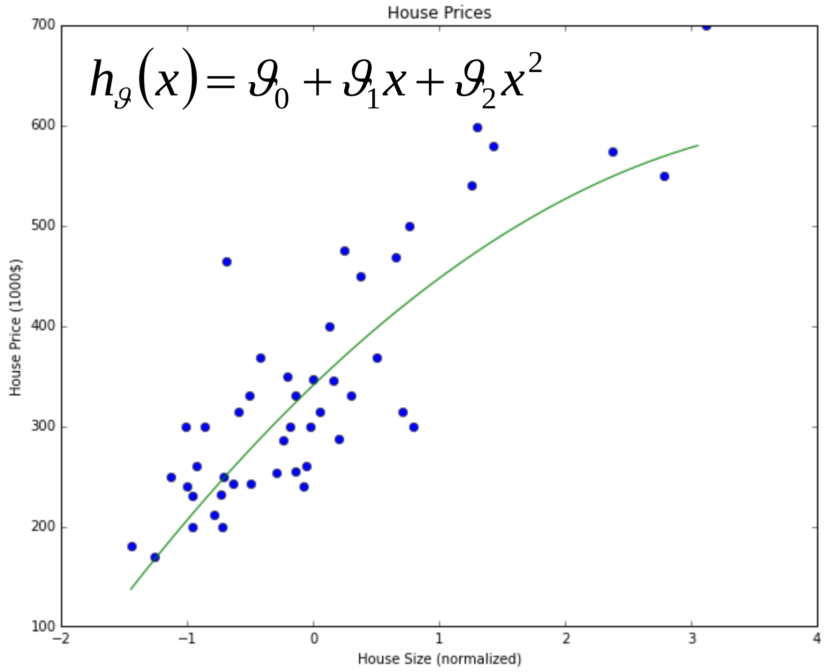
Pattern Analysis and Recognition

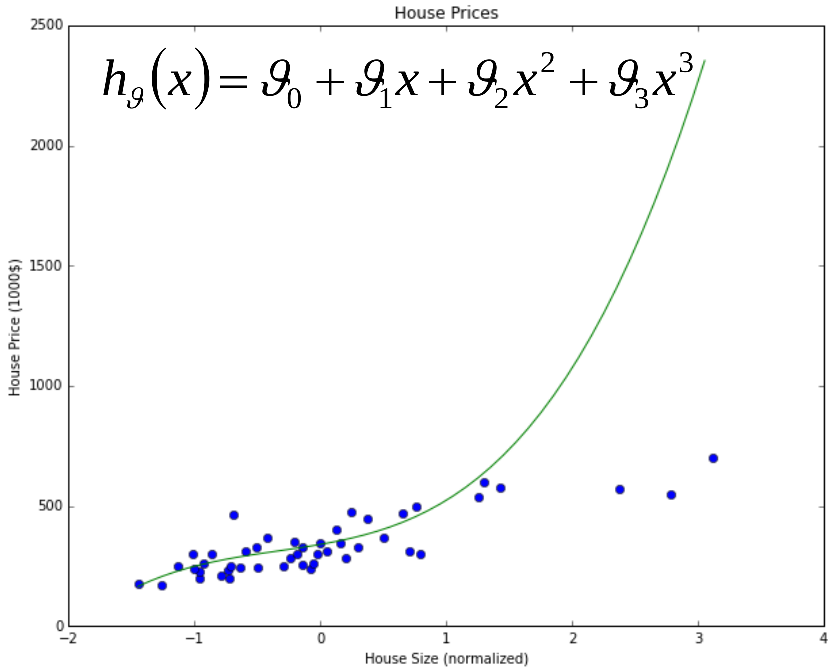
Lecture 3: Over/under-fitting, regularization, bias/variance trade-off

Last time on Pattern Analysis and Recognition

RECAP

Polynomial models for house prices





$$h_\vartheta(x) = \vartheta_0 + \vartheta_1 (size) + \vartheta_2 (size^2) + \vartheta_3 (size^3)$$

$$h_\vartheta(x) = \vartheta_0 + \vartheta_1 x_1 + \vartheta_2 x_2 + \vartheta_3 x_3 \qquad , \begin{aligned} x_1 &= size \\ x_2 &= size^2 \\ x_3 &= size^3 \end{aligned}$$

Feature scaling – mean normalisation

Aim: get every feature x_j into approximately a $-1 \leq x_j \leq 1$ range

Mean normalization:

- Subtract from each feature x_j the feature mean (μ_j) to make features have approximately zero mean
- Divide by the feature range or the standard deviation (s_j)
- Do not apply to x_0 !!!

$$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{s_j}$$

$$\mu_j = \overline{x_j} = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$s_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(x_j^{(i)} - \mu_j\right)^2}$$

Normal Equation

Inverse of Matrix $X^T X$

$$\vartheta = \left(X^T X\right)^{-1} X^T y$$

What if $X^T X$ is non-invertible?

- Redundant features (linearly dependent).
- Too many features (e.g. $m \leq n$).

Solution: delete some features, or use regularization

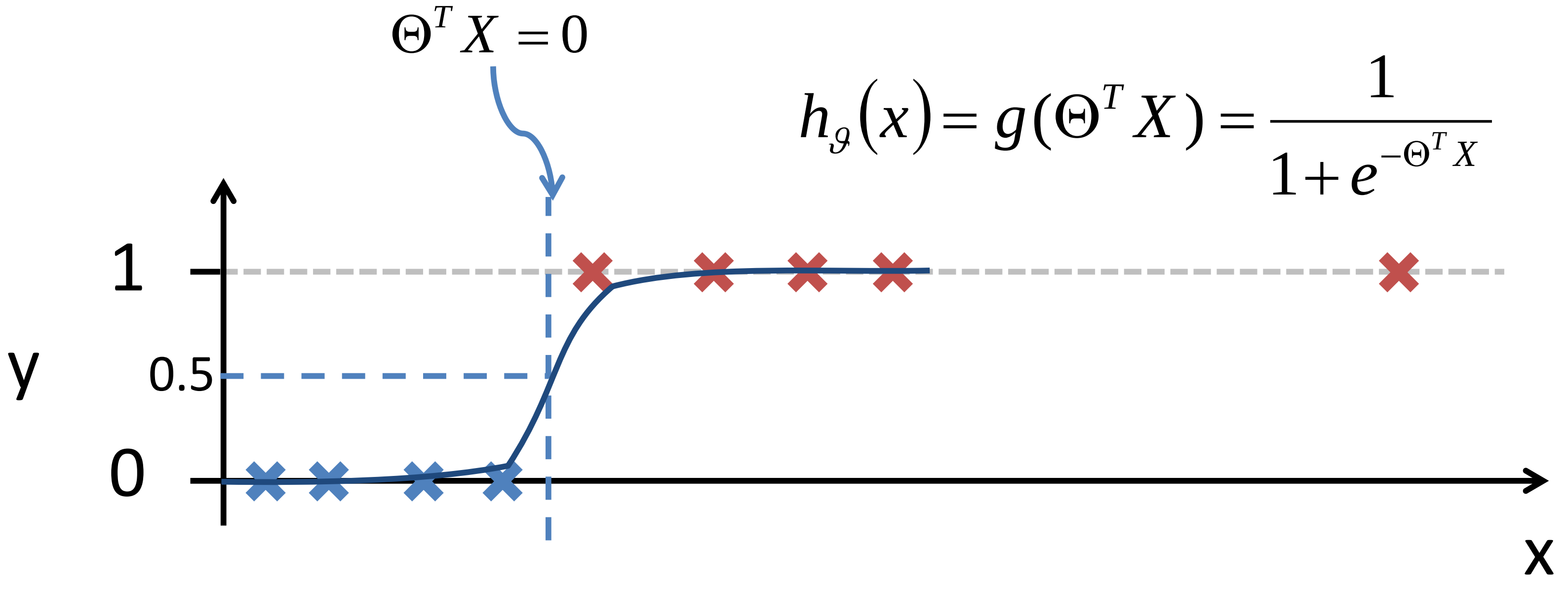
You can still calculate the pseudo-inverse matrix $(X^T X)^+$

```
numpy.linalg.pinv()
```

Our first Classifier: Logistic Regression

Suppose we predict "$y=1$" if $h_\vartheta(x) \geq 0.5$

predict "$y=0$" if $h_\vartheta(x) < 0.5$



Slides adapted from A. Bagdanov (2015)

REVISITING LINEAR REGRESSION: UNDERFITTING AND OVERFITTING

The story thus far…

In previous lectures we saw how to fit parametric, linear models to discrete instances of data.

These “fits" were driven by a simple function, a cost or loss function:

$$h_\vartheta(x) = \sum_{i=1}^{n} \vartheta_i x_i \qquad \text{Assume } x_0 = 1$$

$$J(\vartheta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\vartheta\left(x^i\right) - y^i\right)^2$$

Today we will take a closer look at the limitations of using a single loss function to quantify the quality of the fit

In the process we will touch on some deep and pervasive issues in pattern recognition and machine learning

Linear Regression: An Equivalent Formulation

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

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

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

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

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

With this notation, we recover the linear regressions we have already seen

Example 1D

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:

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

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

Example 2D

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:

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

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

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

Problems and Problem Instances

Regression is a problem of *function estimation* or *function approximation*

We are given an observed instance of data and targets (**X**; **y**)

From this, we should estimate function that *best* fits this observed data.

Written like this, it is the very definition of an underconstrained problem.

What is the *best* function? Is minimizing the loss function L enough?

How do we define *function* for the purposes of minimization? Linear? Polynomial? What degree?

Problem Instances

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

We assume there is an underlying "true" function *f* that we want to estimate

All we have to use for estimation are corrupted samples of this ideal function *f*

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

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* directly

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

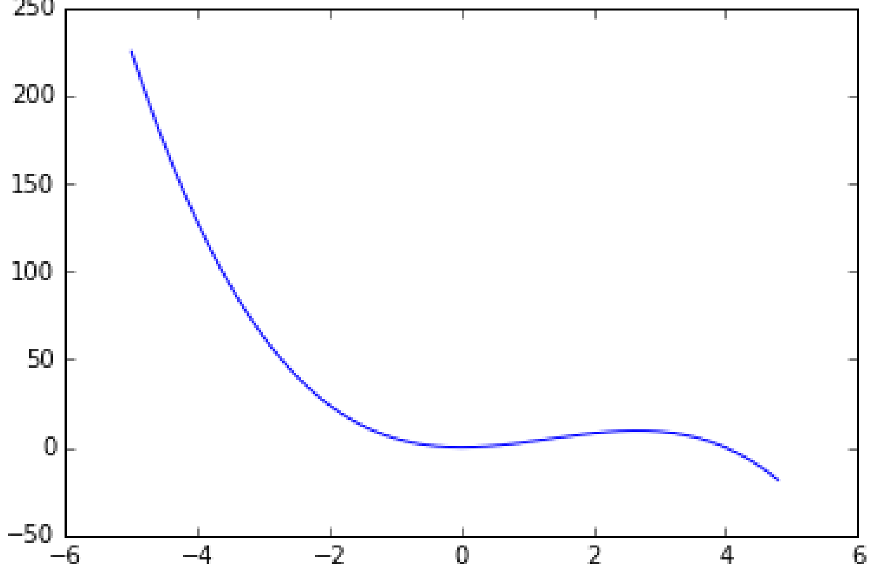
Running Example: a noisy cubic

Suppose our data model is:

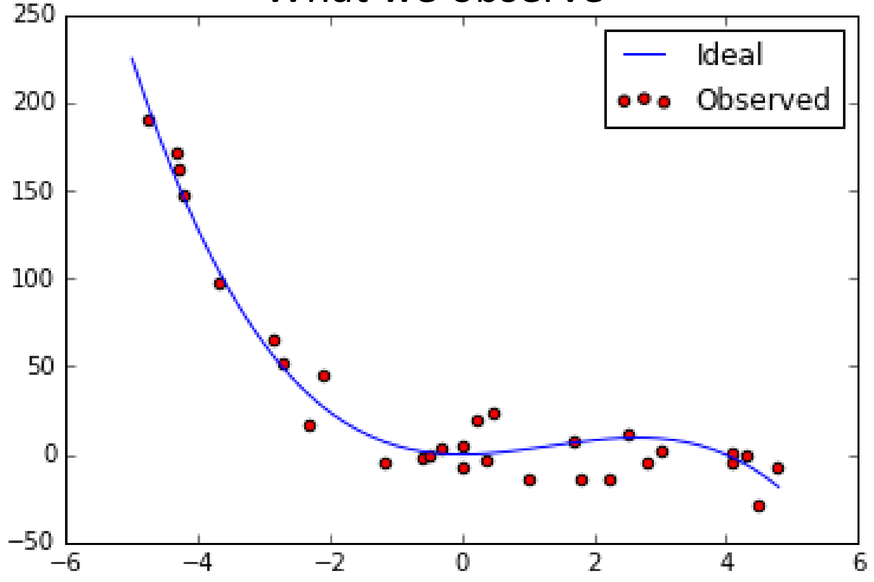
$$y^i = f(x^i) + N(0, \sigma)$$

$$f(x) = 4x^2 - x^3, \qquad \sigma = 5$$

Behind the scenes



What we observe



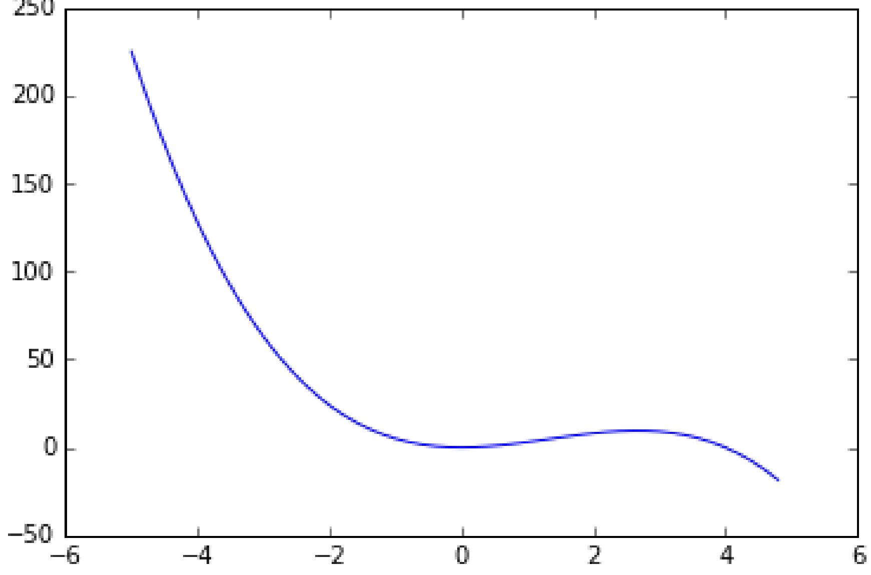
Running Example: a noisy cubic

Suppose our data model is:

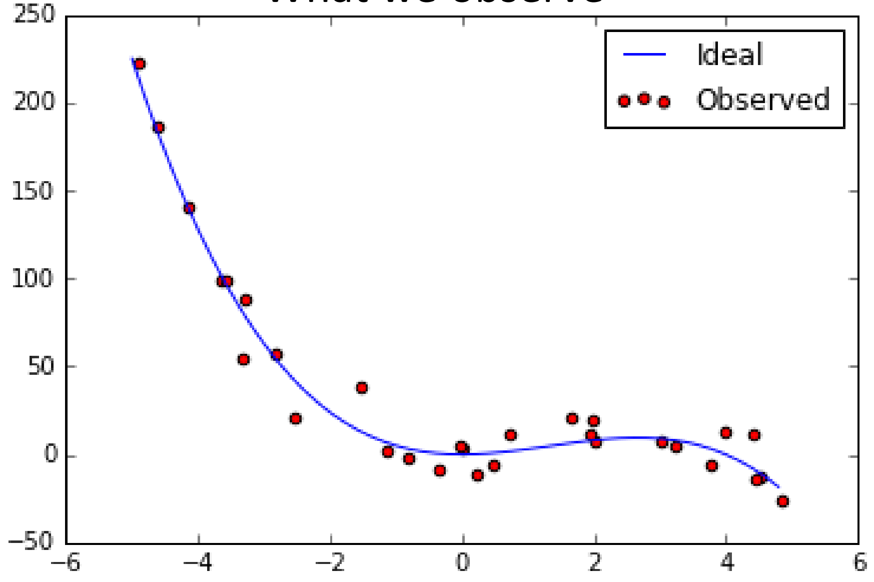
$$y^i = f(x^i) + N(0, \sigma)$$

$$f(x) = 4x^2 - x^3, \qquad \sigma = 5$$

Behind the scenes



What we observe



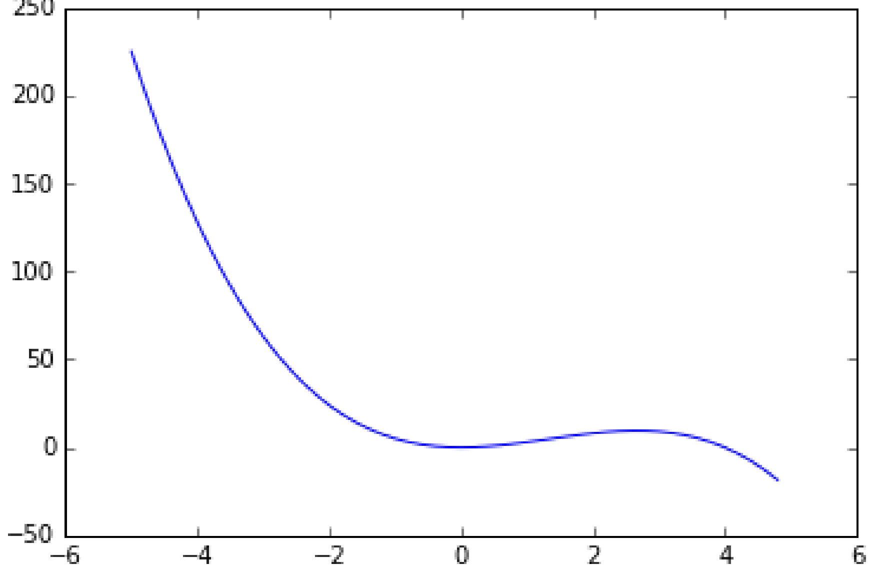
Running Example: a noisy cubic

Suppose our data model is:

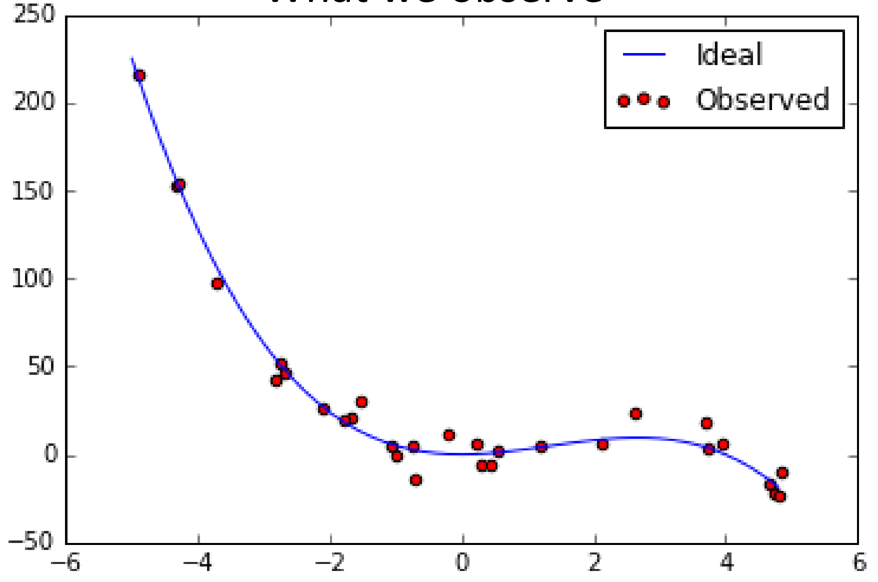
$$y^i = f\left(x^i\right) + N(0, \sigma)$$

$$f(x) = 4x^2 - x^3, \qquad \sigma = 5$$

Behind the scenes



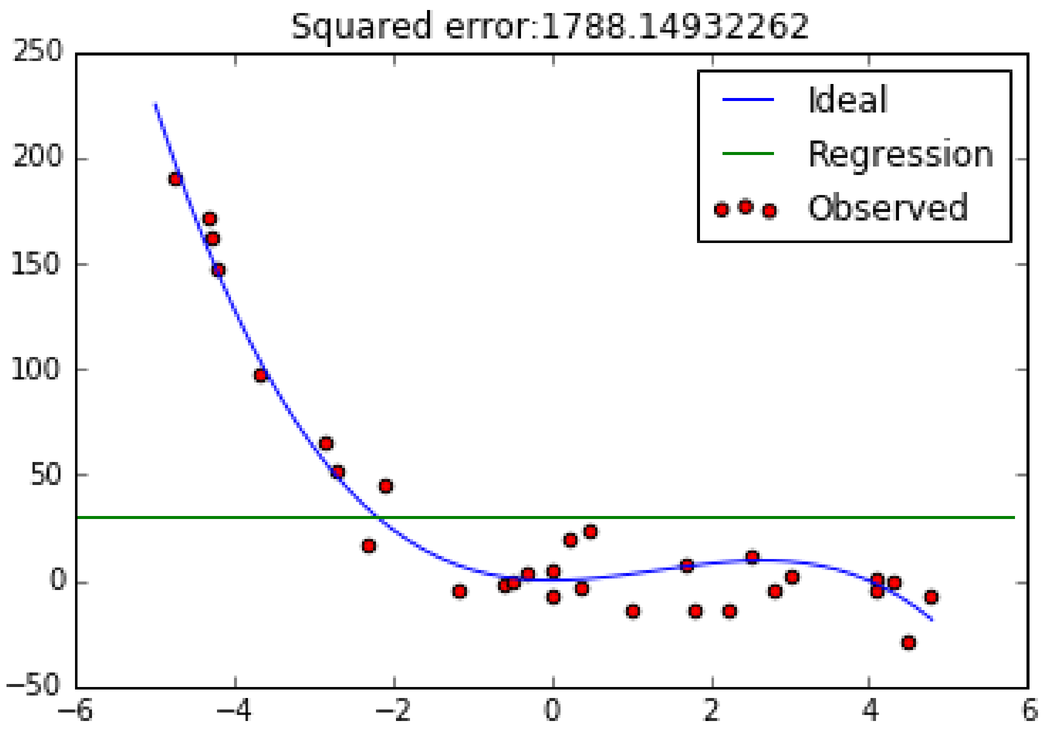
What we observe



A first approximation: the constant model

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

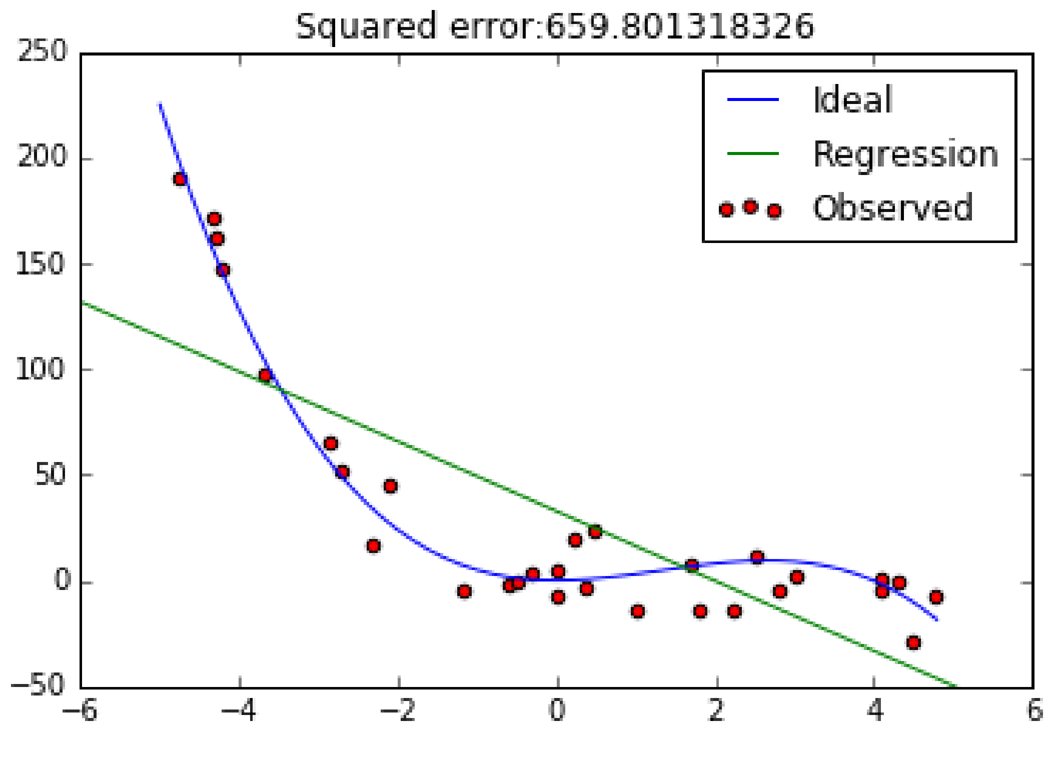
We start with the simplest model imaginable: $\phi(x) = [1]$



Increasing model complexity: a linear model

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

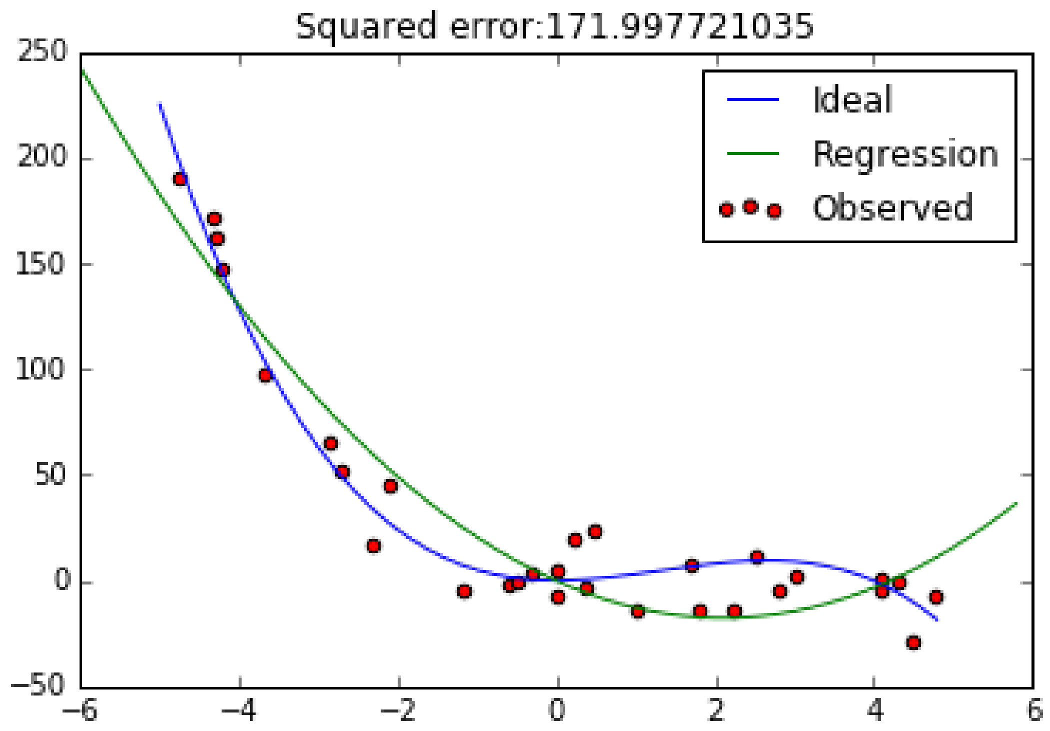
Linear design matrix: $\phi(x) = [1, x]$



Increasing model complexity: a two-degree polynomial model

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

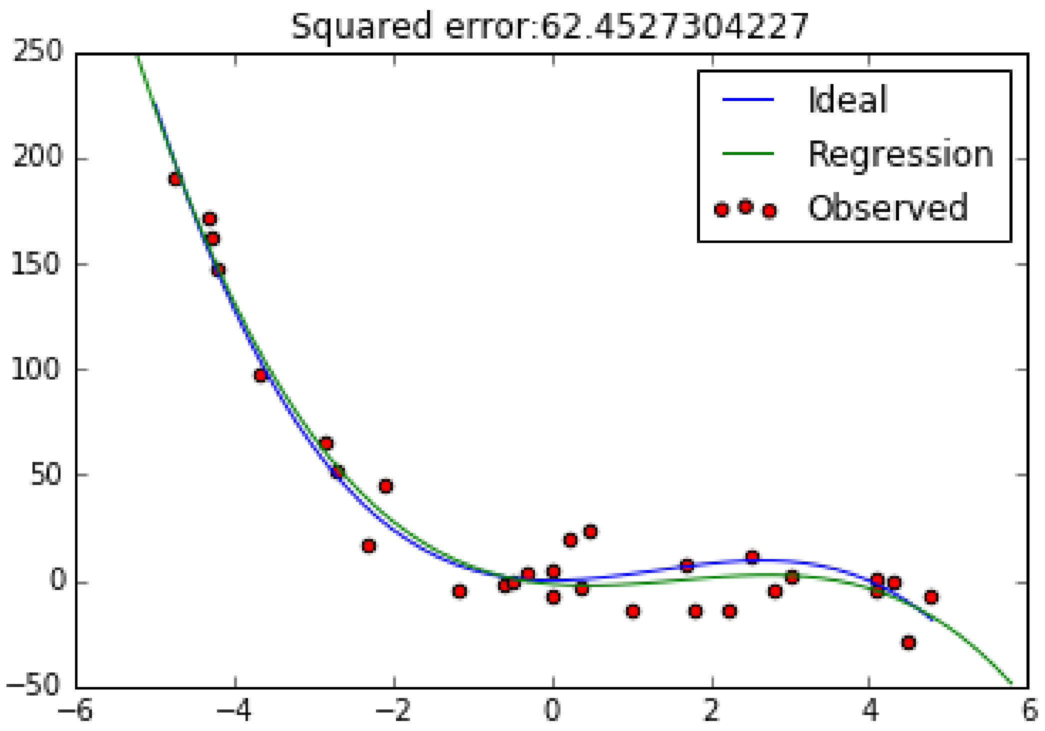
Quadratic design matrix: $\phi(x) = [1, x, x^2]$



Increasing model complexity: a three-degree polynomial model

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

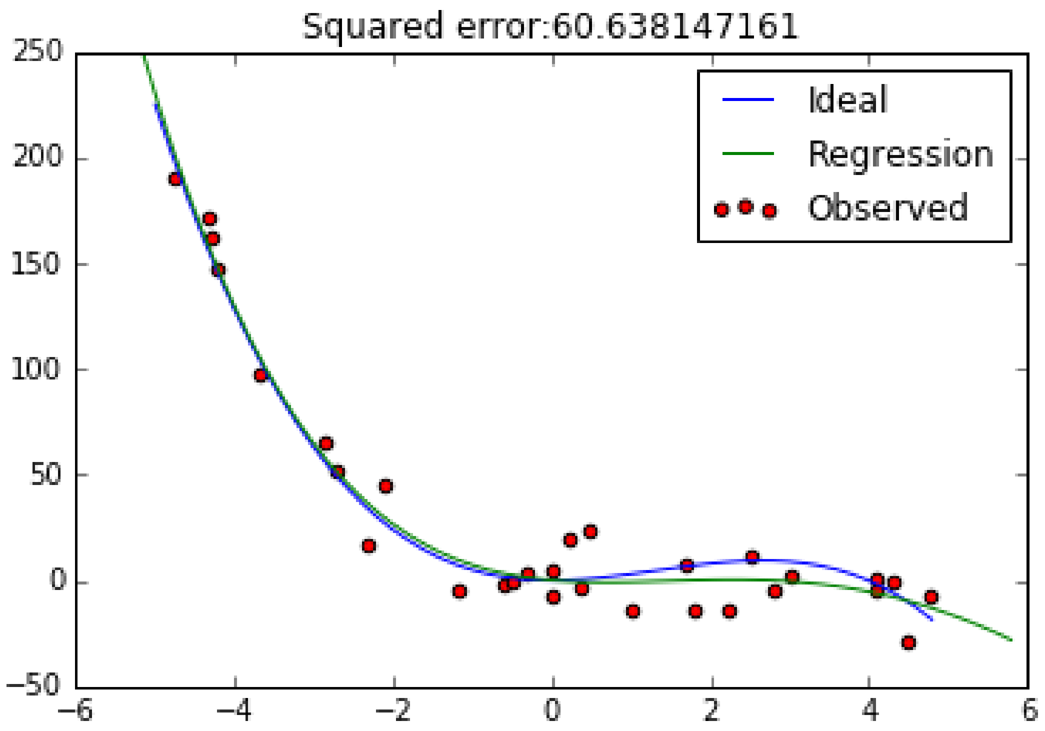
Cubic design matrix: $\phi(x) = [1, x, x^2, x^3]$



Increasing model complexity: a four-degree polynomial model

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

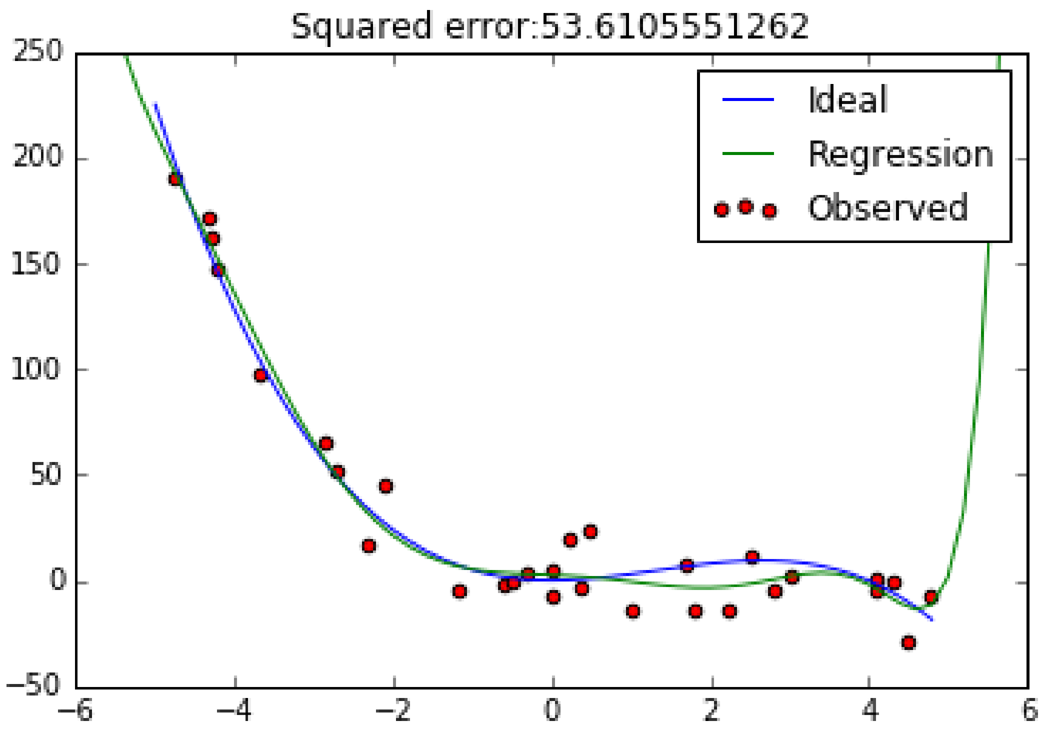
Fourth-degree design matrix: $\phi(x) = [1, x, x^2, x^3, x^4]$



Increasing model complexity: a ten-degree polynomial model

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

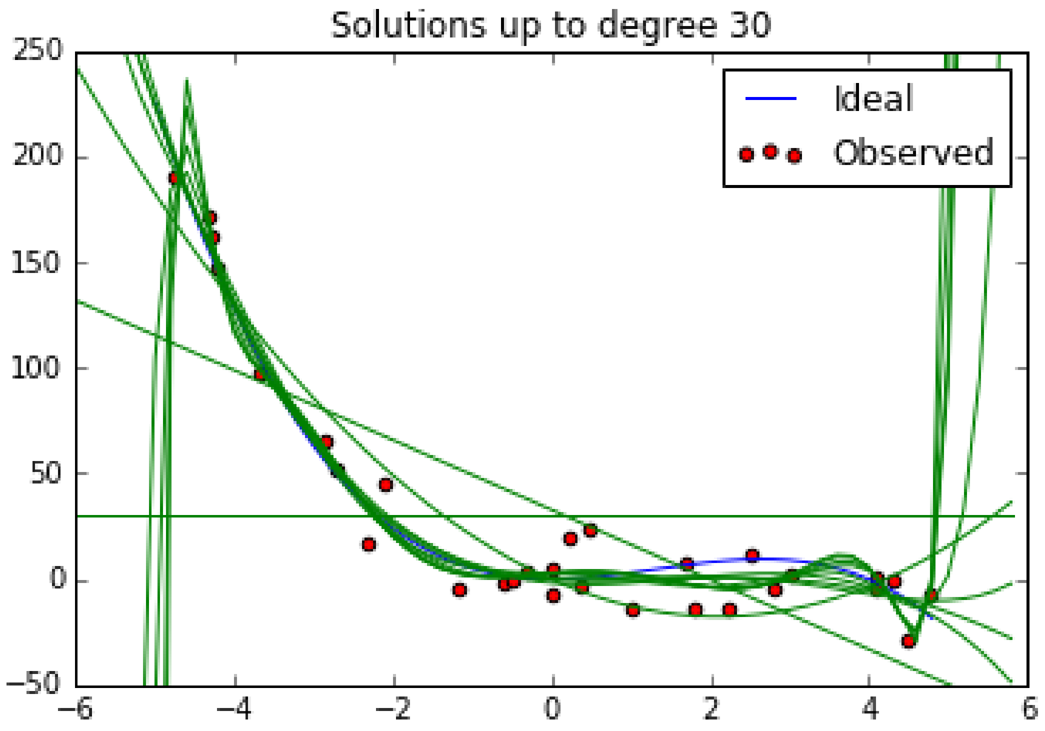
Tenth degree design matrix: $\phi(x) = [1, x, x^2, \ldots, x^{10}]$



Increasing model complexity: when should we stop?

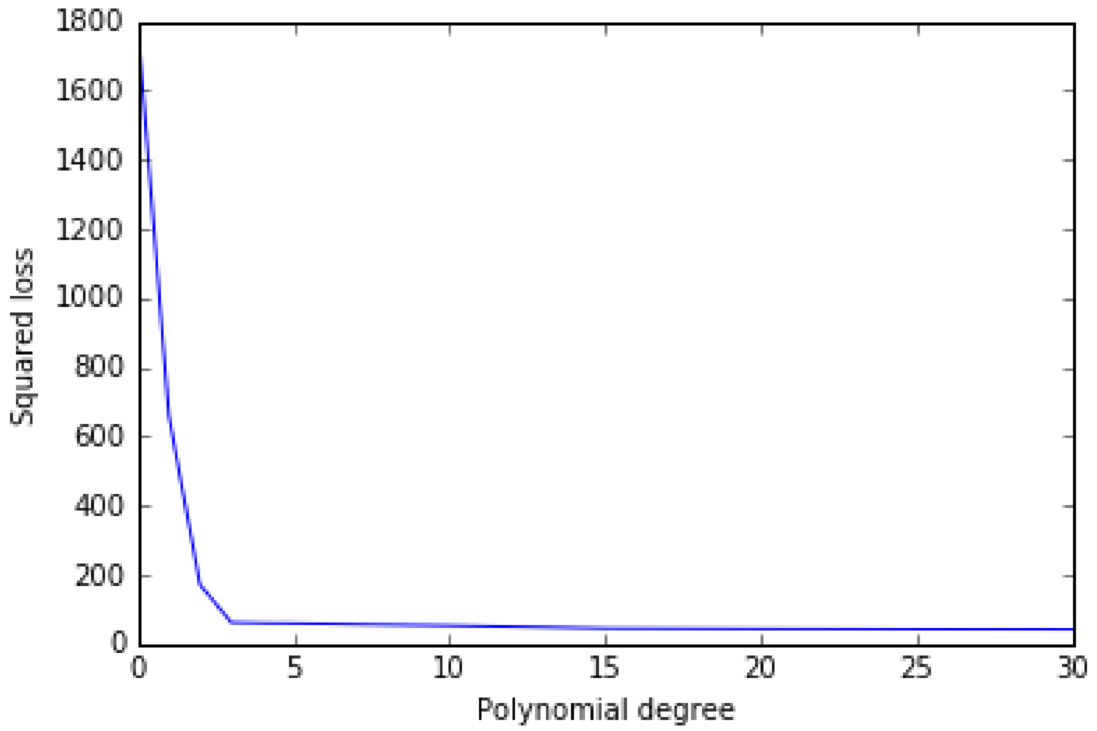
On the same problem instance $(\mathbf{X}, \mathbf{y})$

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



A look at the fitting cost

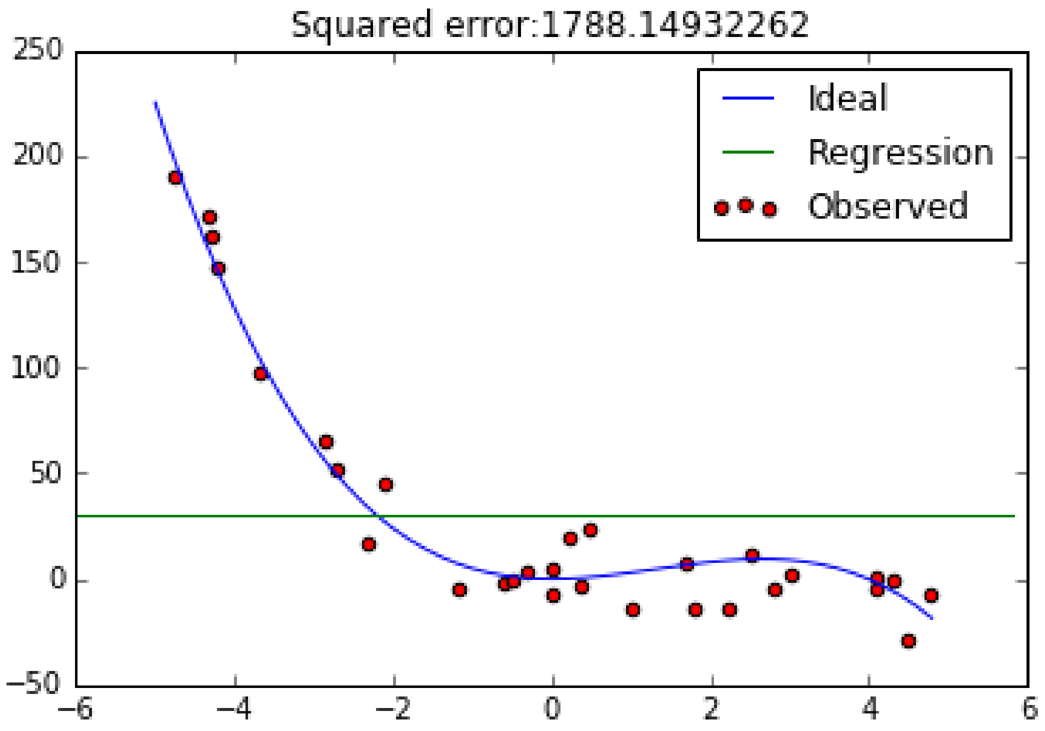
If we analyse squared loss as a function of polynomial degree, we see that error quickly drops, and then stabilizes:



The fitting spectrum: simple models

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

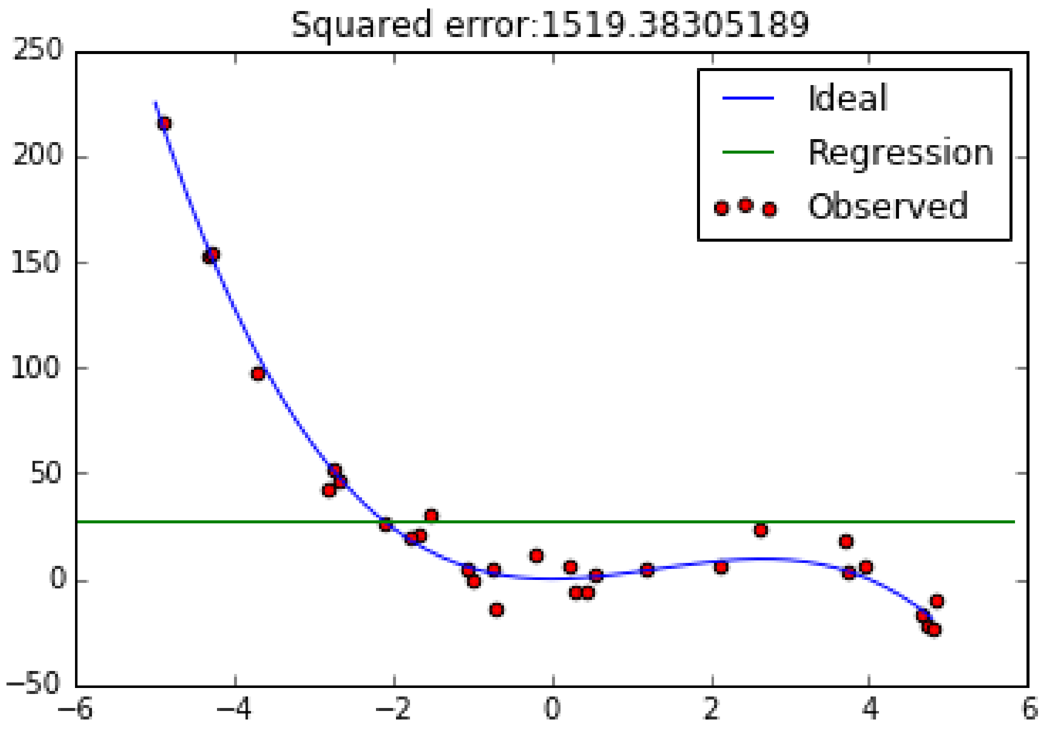
This means that if you repeat fitting over multiple problem instances, all solutions will be biased towards a particular solution:



The fitting spectrum: simple models

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

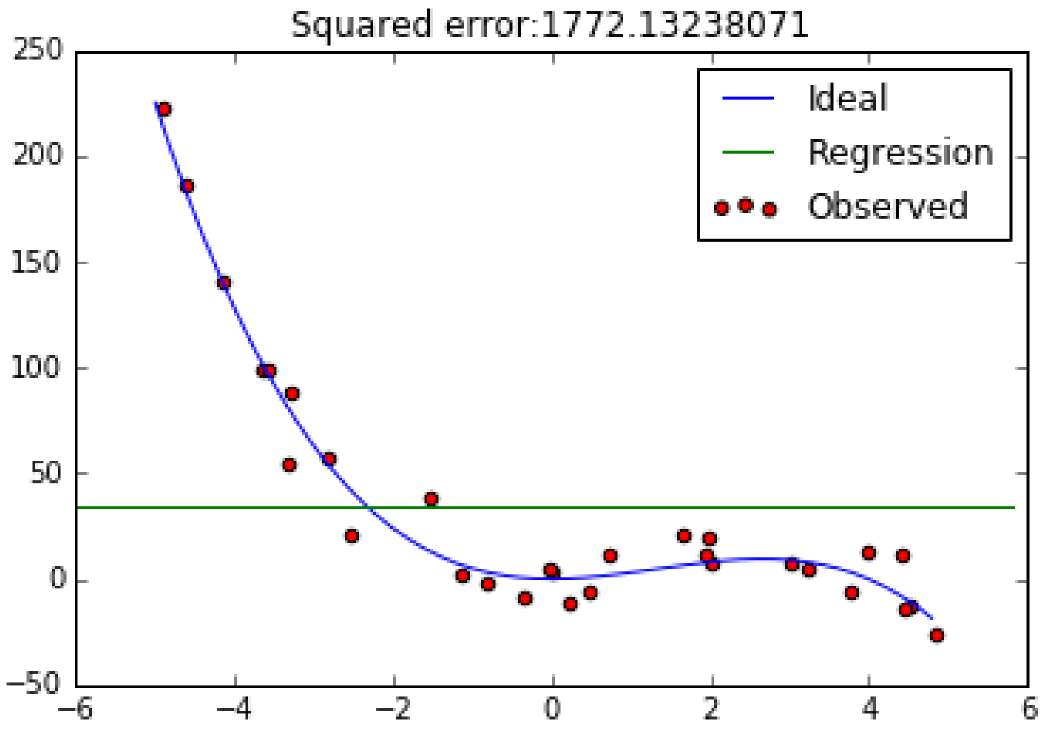
This means that if you repeat fitting over multiple problem instances, all solutions will be biased towards a particular solution:



The fitting spectrum: simple models

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

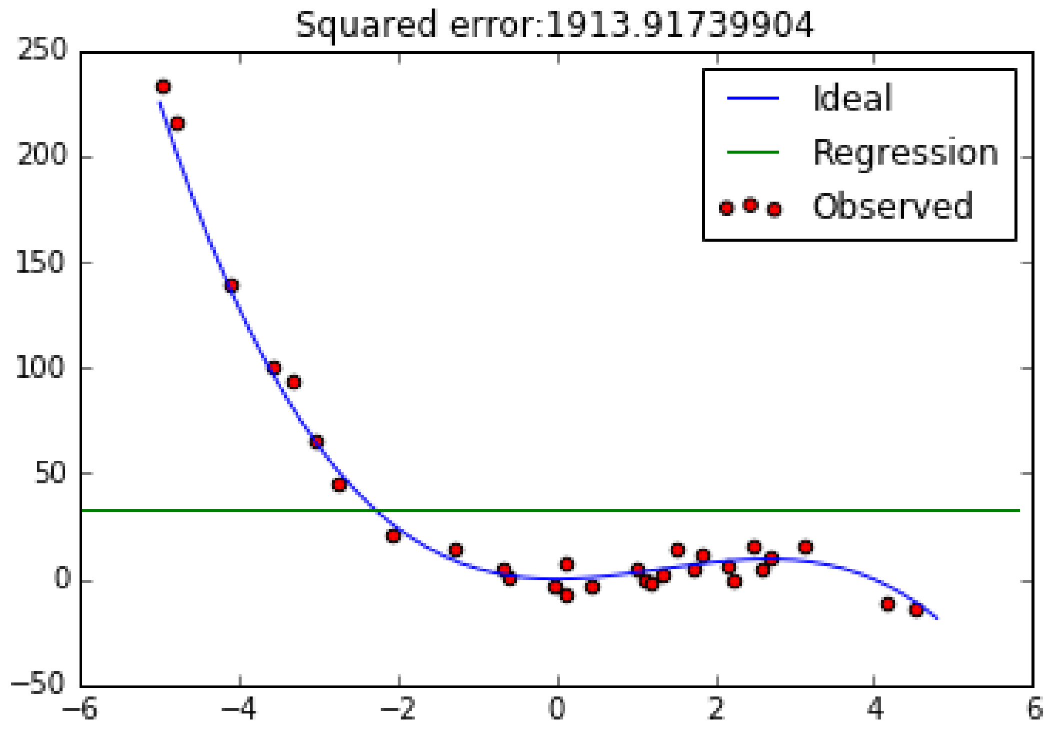
This means that if you repeat fitting over multiple problem instances, all solutions will be biased towards a particular solution:



The fitting spectrum: 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:



Underfitting

Using a constant or linear model leads to biased solutions

The constant model is biased towards the mean of the input data

These simple models are very stable

But, we already saw that they don't necessarily fit the data well

The fit depends, of course, on the underlying ideal function *f* from which the data is derived

Advantages

Simple models are usually low-dimensional and easy to fit

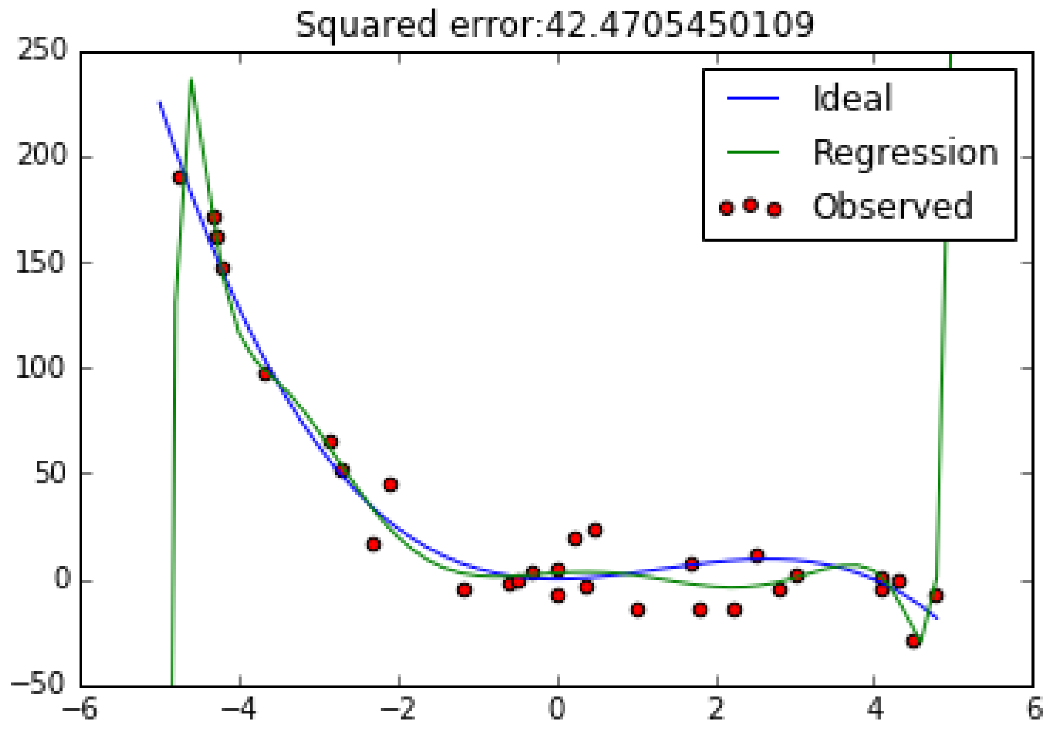
It is easy to predict their behaviour (since they are biased)

They usually do not have the expressive power to get lost in noise

The fitting spectrum: complex models

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

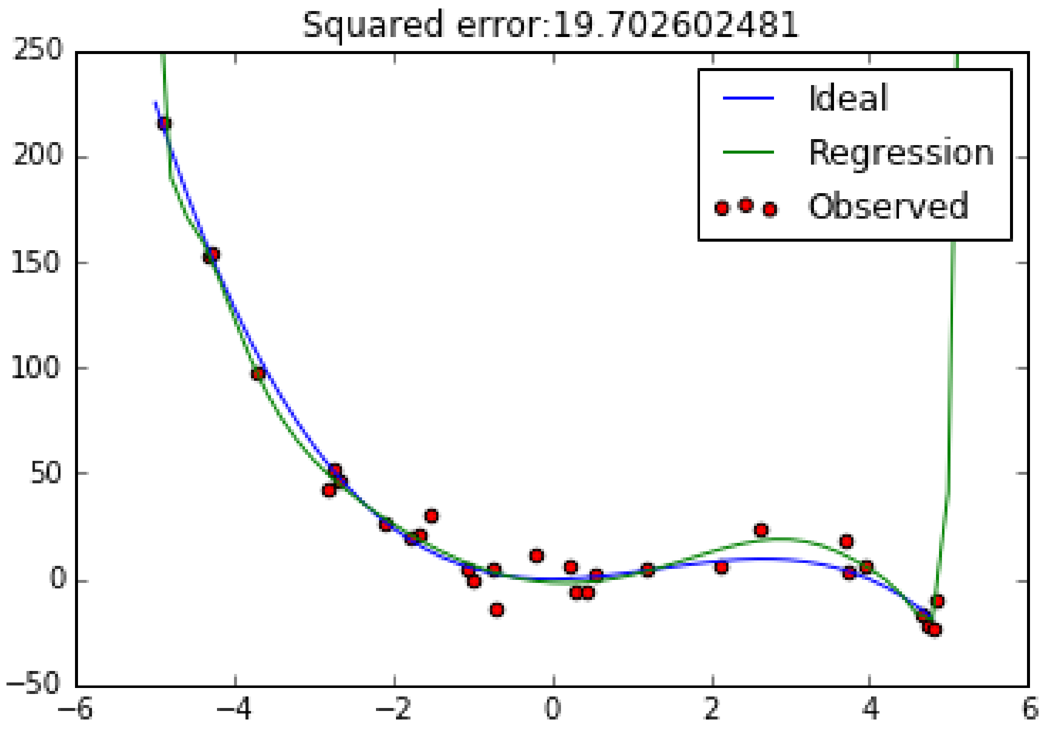
This means that if you repeat fitting over multiple problem instances, all solutions will be all over the place:



The fitting spectrum: complex models

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

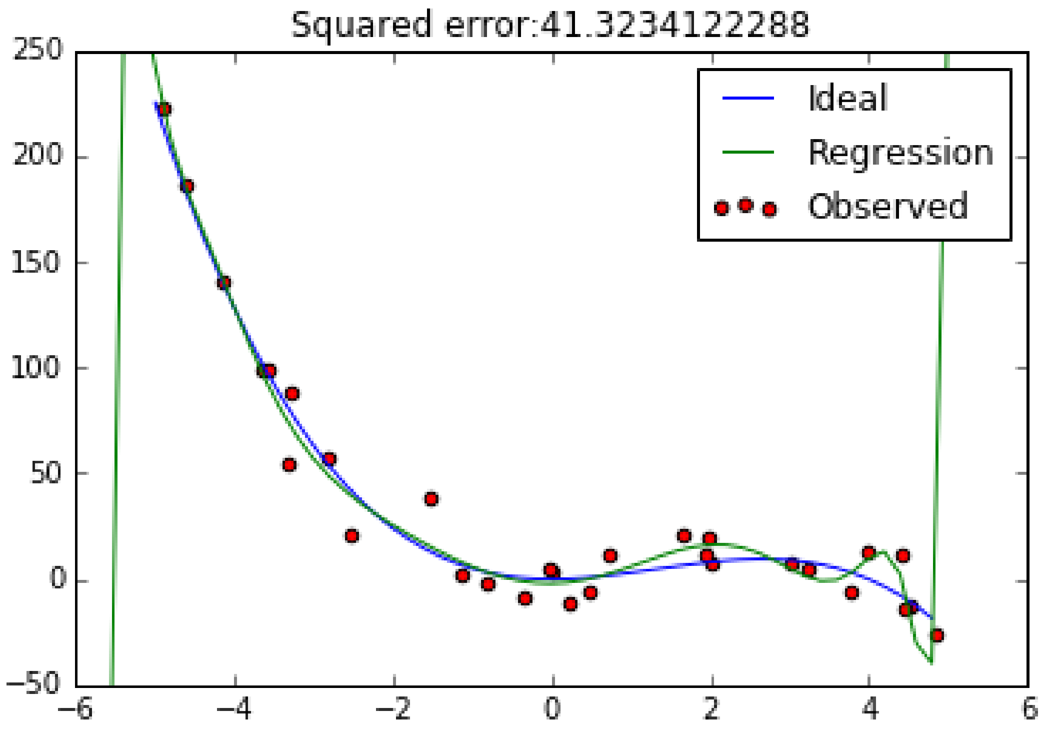
This means that if you repeat fitting over multiple problem instances, all solutions will be all over the place:



The fitting spectrum: complex models

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

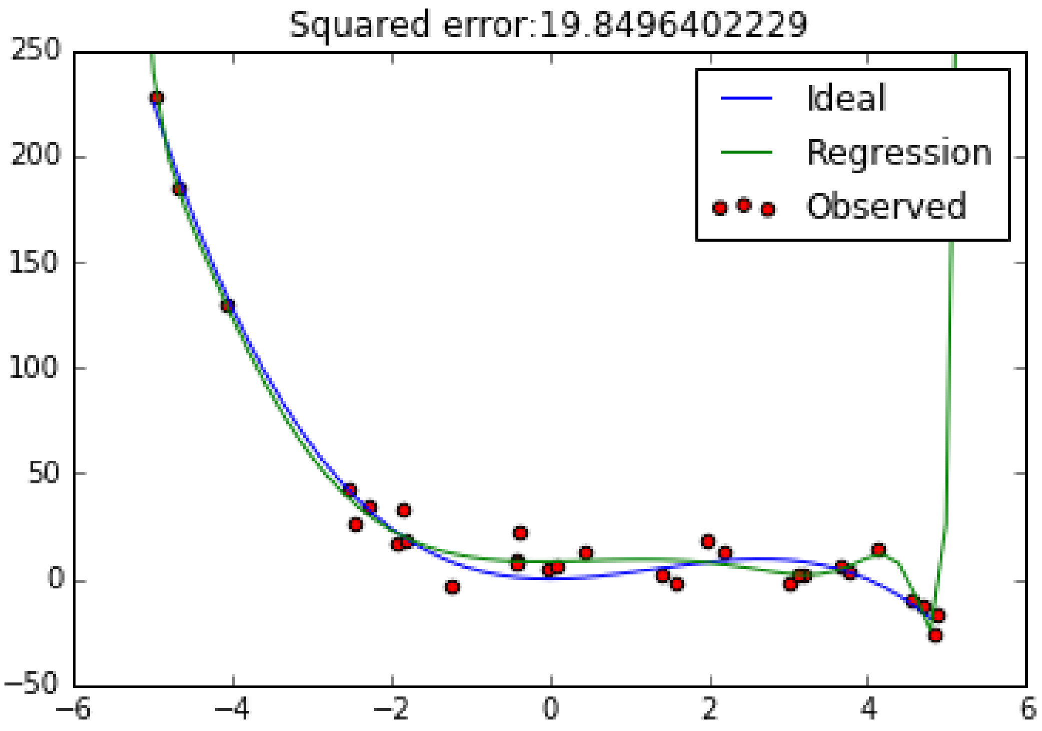
This means that if you repeat fitting over multiple problem instances, all solutions will be all over the place:



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:



Overfitting

Using a high-order polynomial model leads to solutions with high variance (low bias)

The model can fit the available data very well (in terms of squared error)

But the solution may be very unstable

This means that it does not generalize well to new data

The fit and generalization ability depends, of course, on the underlying ideal function f from which the data is derived

Advantages

You can usually count on very low average loss

Very expressive models can be fit with relatively little data

Can be useful if you are sure the ideal function needs a complex explanation (and that it's not just noise)

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)

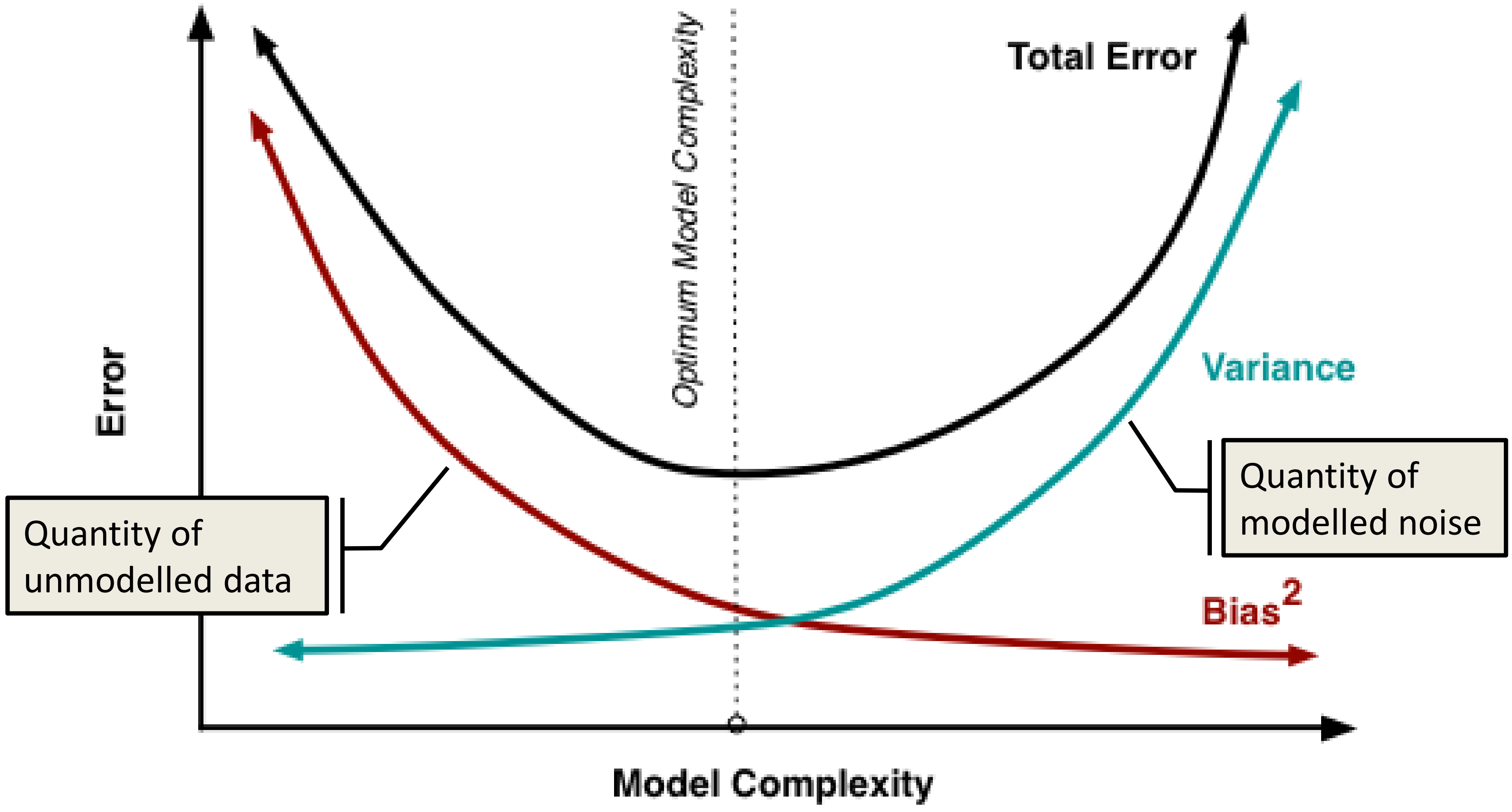
The irreducible error σ^2 is the unavoidable error any estimator will have due to noise in the observation model

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 $\mathcal{L}$.
- 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



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:

$$\phi(x) = [x_0x_0, x_0x_1, \ldots, x_0x_n, x_1x_0, \ldots, x_nx_n]$$

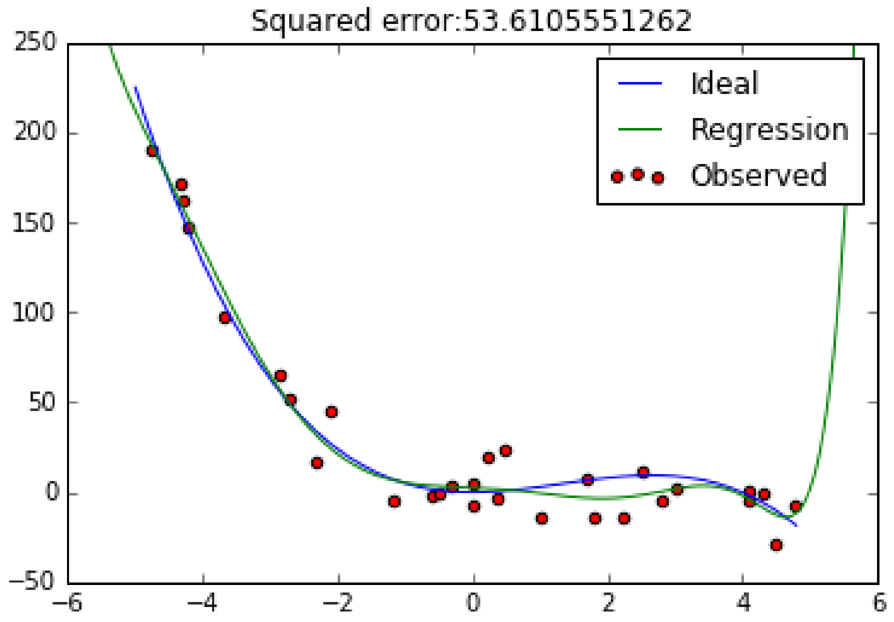
For such an embedding it is unclear how to parametrize complexity without being completely arbitrary

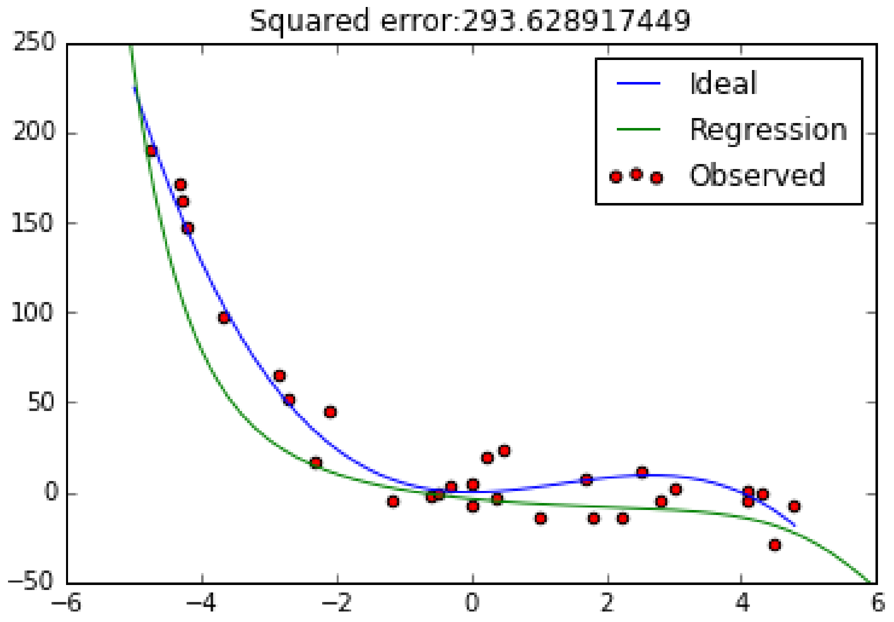
Sometimes data is just high-dimensional (hundreds of thousands of dimensions, at times) - in the case of high-dimensional data we do not (yet) have any way of controlling the complexity of representation

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 10-degree polynomials



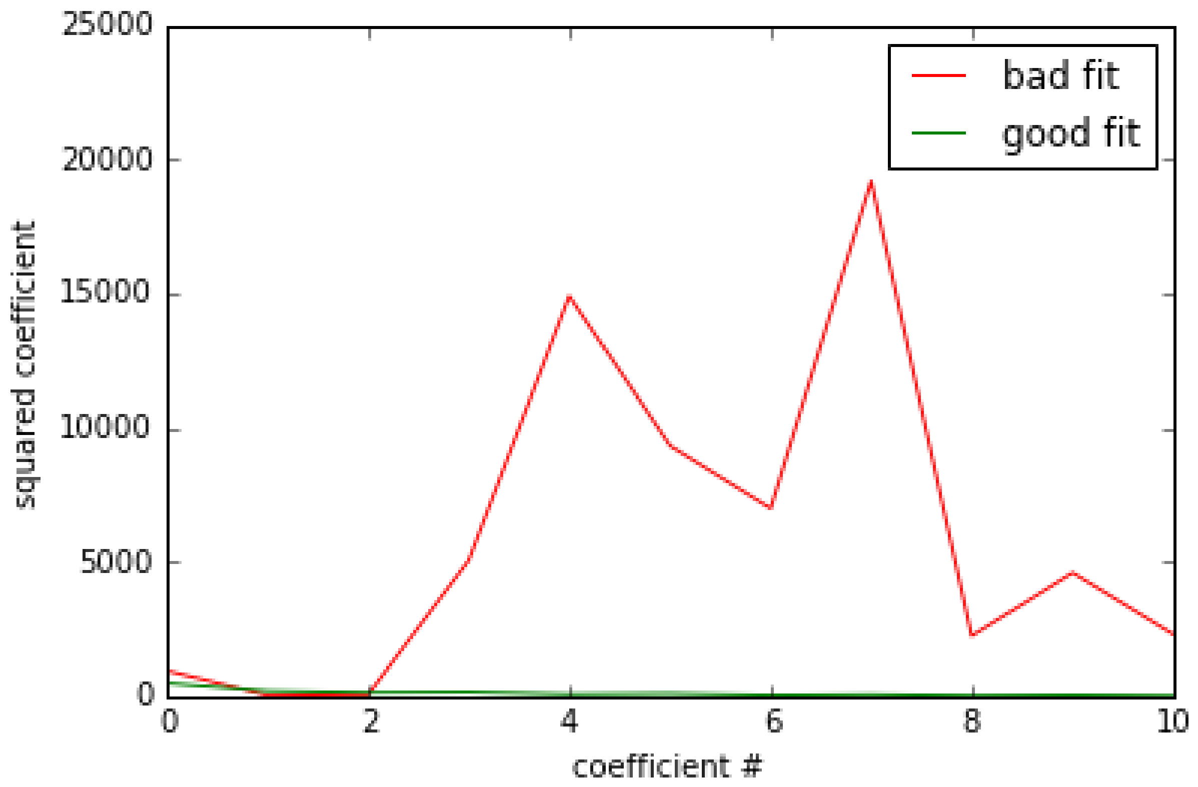


Which one is “better”?

What is the difference?

A look in the coefficients

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



Coefficient Magnitude

It turns out that coefficient magnitude is not a bad measure of fit complexity

What we do is *penalize* high coefficients by adding a *regularization term* to the loss:

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

Coefficient Magnitude

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

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

$$\begin{aligned}
\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}, \phi, \lambda) &= \nabla_{\mathbf{w}} \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^i\right)^T \mathbf{w}, y^i\right) + \nabla_{\mathbf{w}}(\lambda \mathbf{w}^T \mathbf{w}) \\
&= \nabla_{\mathbf{w}} \sum_{i=1}^{m} L\left(\phi\left(\mathbf{x}^i\right)^T \mathbf{w}, y^i\right) + \lambda \nabla_{\mathbf{w}}(\mathbf{w}^T \mathbf{w}) \\
&= \text{least squares gradient} + 2\lambda \mathbf{w}
\end{aligned}$$

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

Normal Equation

For ridge regression there is an analytic, normal equation solution:

$$\hat{\mathbf{w}} = (\phi(\mathbf{X})^T \phi(\mathbf{X}) + \lambda \mathbf{I_0})^{-1} \phi(\mathbf{X})^T \mathbf{y}$$

Where $\mathbf{I_0}$ is the $(n+1) \times (n+1)$ identity matrix with a 0 in the first position (no bias penalty):

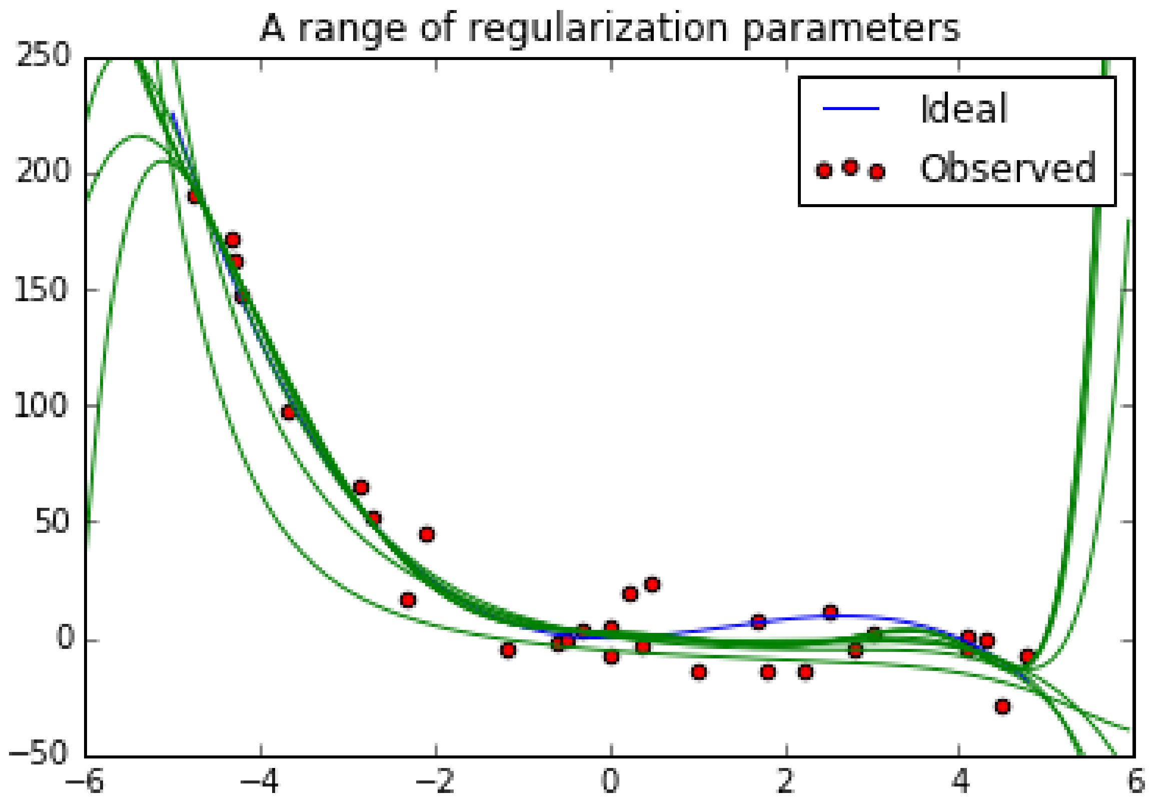
$$\mathbf{I_0} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

Bonus: adding this diagonal “ridge” to $\phi(\mathbf{X})^T \phi(\mathbf{X})$ also guarantees invertibility

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:

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

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

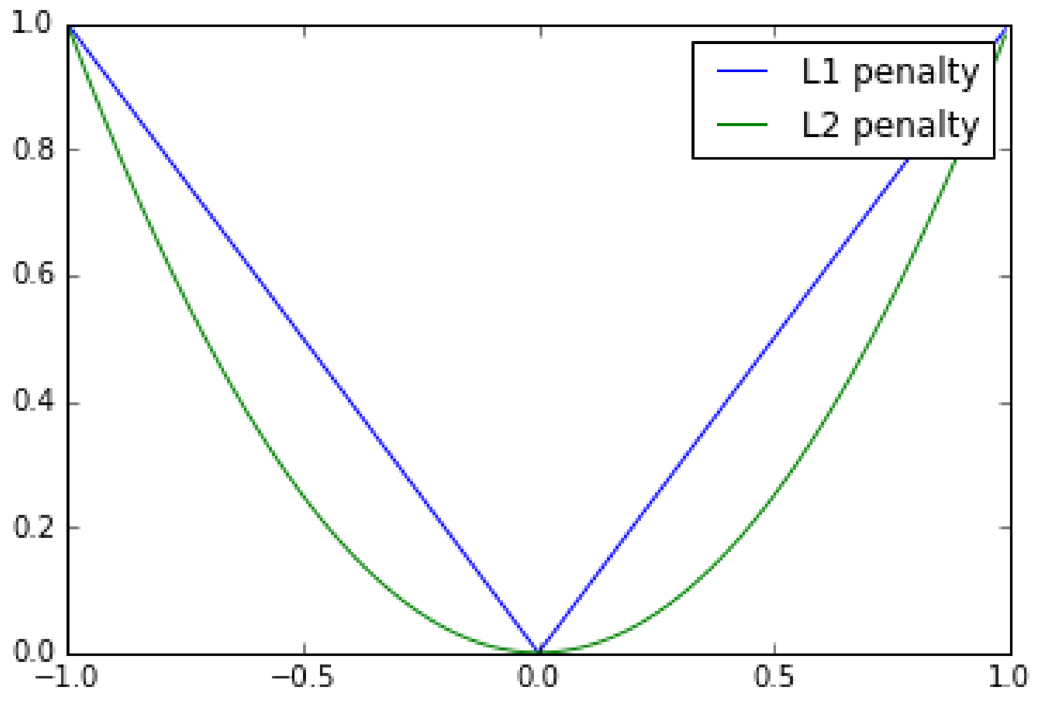
It is called "Lasso" because it has the effect of selecting a few (number depending on λ) coefficients with non-zero magnitude

ℓ_1 (lasso) vs ℓ_2 (ridge) loss

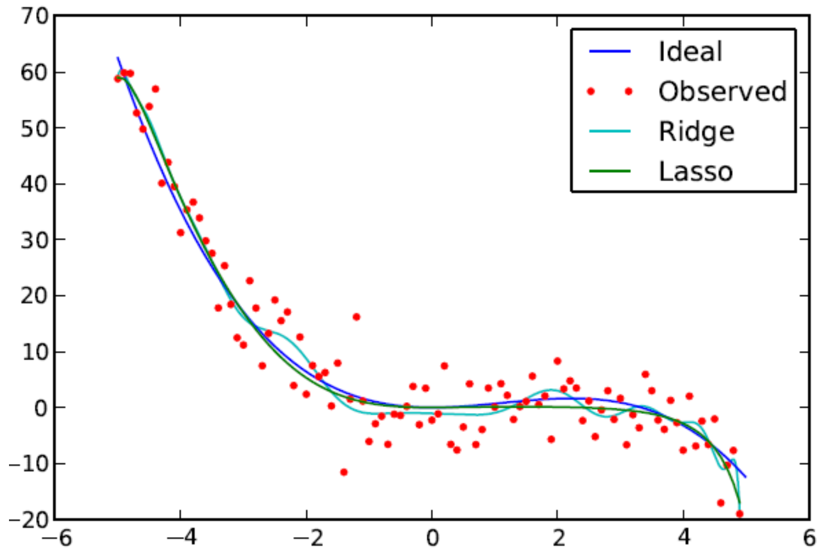
The small variation between the ℓ_1 (lasso) and the ℓ_2 (ridge) loss has a profound effect on the solution

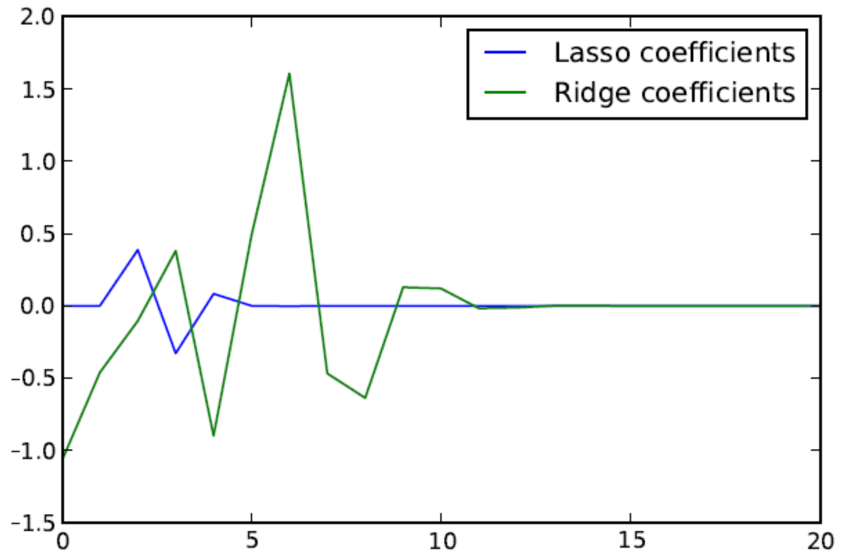
To see this imagine how the addition of a quadratic versus absolute penalty “encourages” solutions to evolve:

- 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

Summary

Regularization is important in most learning methods - Often, without regularization you simply can't learn anything meaningful from data at hand

The risk of underfitting or overfitting is very real

When interpreting regularization parameters, remember the bias/variance trade-off:

- More regularization (big λ) → high bias, low variance
- Less regularization (small λ) → low bias, high variance

Never forget about irreducible error (noise), and make sure you're not fitting your models to it

Never blindly trust squared error

When in doubt, cross validate!

What's Next

		Mondays	**Tuesdays**				
		16:00 - 18:00	**15:00 - 17:00**				
Practical Sessions		**M**	**T**	**W**	**T**	**F**	**Lectures**
	Feb	8	9	10	11	12	Introduction and Linear Regression
P0. Introduction to Python, Linear Regression		15	16	17	18	19	Logistic Regression, Normalization
P1. Text non-text classification (Logistic Regression)		22	23	24	25	26	Regularization, Bias-variance decomposition
	Mar	29	1	2	3	4	Normalization and subspace methods (dimensionality reduction)
		7	8	9	10	11	Probabilities, Bayesian inference
Discussion of intermediate deliverables / project presentations		14	15	16	17	18	Parameter Estimation, Bayesian Classification
		21	22	23	24	25	Easter Week
	Apr	28	29	30	31	1	Clustering, Gausian Mixture Models, Expectation Maximisation
P2. Feature learning (k-means clustering, NN, bag of words)		4	5	6	7	8	Nearest Neighbour Classification
		11	12	13	14	15	
		18	19	20	21	22	Kernel methods
Discussion of intermediate deliverables / project presentations		25	26	27	28	29	Support Vector Machines, Support Vector Regression
P3. Text recognition (multi-class classification using SVMs)	**May**	2	3	4	5	6	Neural Networks
		9	10	11	12	13	Advanced Topics: Metric Learning, Preference Learning
		16	17	18	19	20	Advanced Topics: Deep Nets
Final Project Presentations		23	24	25	26	27	Advanced Topics: Structural Pattern Recognition
	Jun	30	31	1	2	3	Revision

LEGEND	
🟦	Project Follow Up
🟪	Project presentations
🟩	Lectures
🟪	Project Deliverable due date
🟥	Vacation / No Class