Linear & affine transformations

- Vector $\mathbf{y} \in \mathbb{R}^m$ is a *linear transformation of* $\mathbf{x} \in \mathbb{R}^n$ iff it can be written as $\mathbf{y} = \mathbf{A}\mathbf{x}$ for some $\mathbf{A} \in \mathbb{R}^{m \times n}$ where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

- Vector $\mathbf{y} \in \mathbb{R}^m$ is an *affine transformation of* $\mathbf{x} \in \mathbb{R}^n$ iff it can be written as $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}$ for some $\mathbf{b} \in \mathbb{R}^m$.
  - Affine transformation = linear transformation + translation
- A *linear model* assumes that the outputs $\mathbf{y}$ are a linear or affine transformation of the inputs $\mathbf{x}$.

Affine transformation as linear

- An affine transformation can be represented as a linear transformation of an 'augmented' input in $\mathbb{R}^{n+1}$

$$\mathbf{y} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \mathbf{A}\mathbf{x} + \mathbf{b}$$

$$\begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{m1} & \cdots & a_{mn} & b_m \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{bmatrix}$$

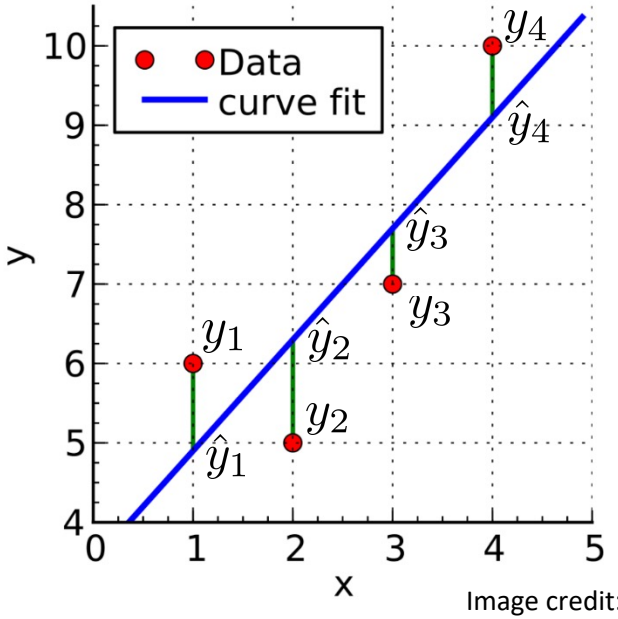
- Most 'linear models' are actually *affine*, as above

An example of *linear least squares*

- Regression model for a single output:

  Assumes $\hat{y}$ a linear function of inputs $\mathbf{x} = \begin{bmatrix} 1 & x_1 & \cdots & x_D \end{bmatrix}^T$

- <u>Supervised</u>: minimizes *squared error* w.r.t. *targets* y_i



Example: 1D line (D=1) with slope *a* and intercept *b*

$$\hat{y} = ax + b$$

Image credit: Krishnavedala - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15462765

An example of *linear least squares*

x → $ax + b$ → $\hat{y}$ (prediction) → y (target)

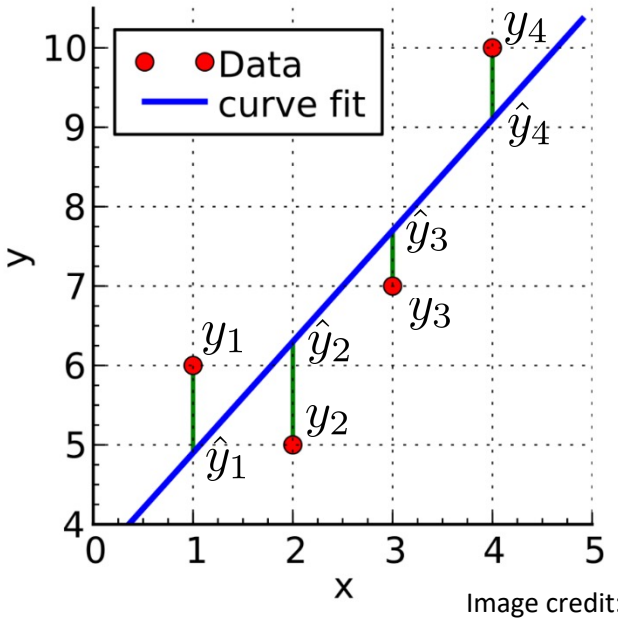


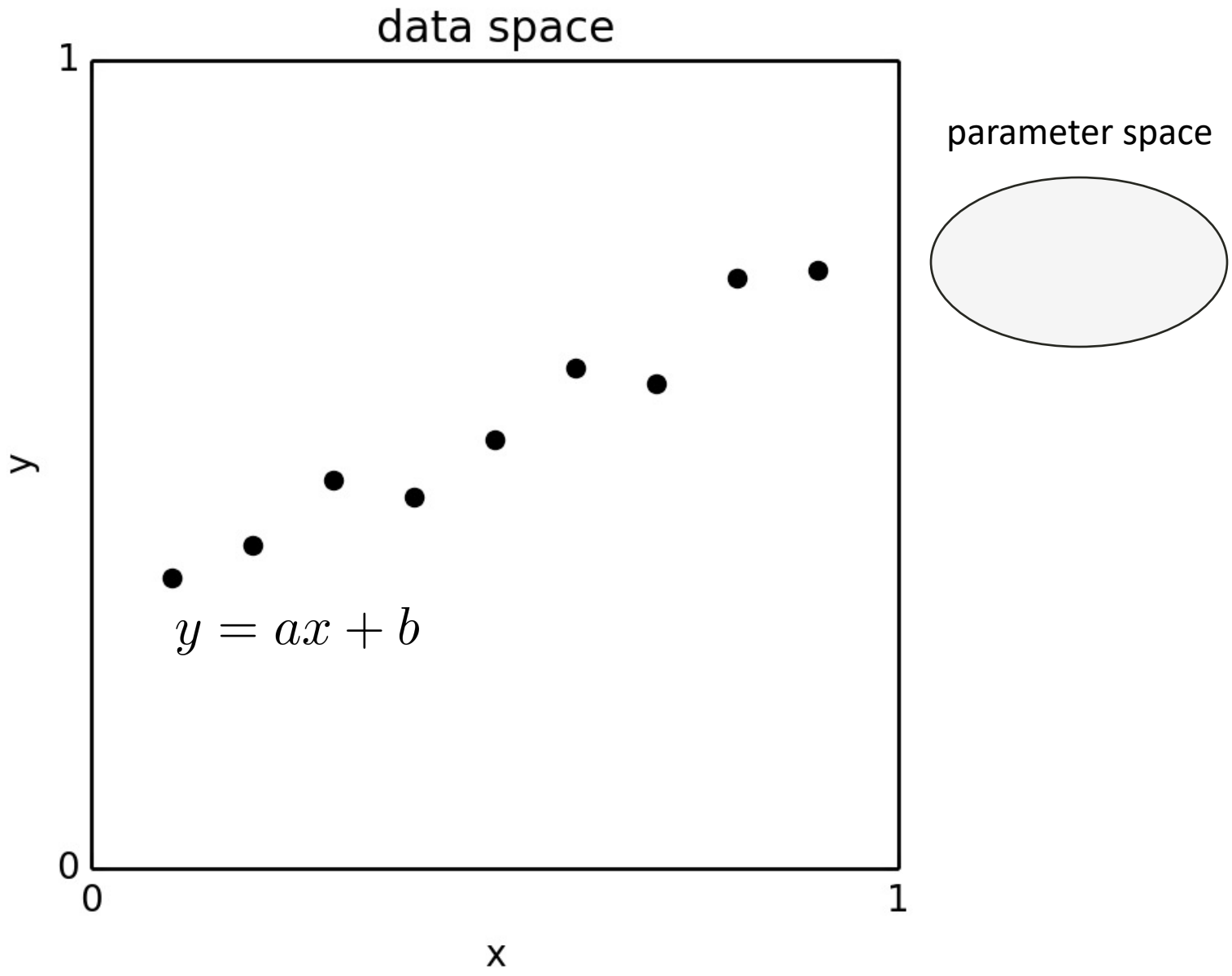
Image credit: Krishnavedala - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15462765

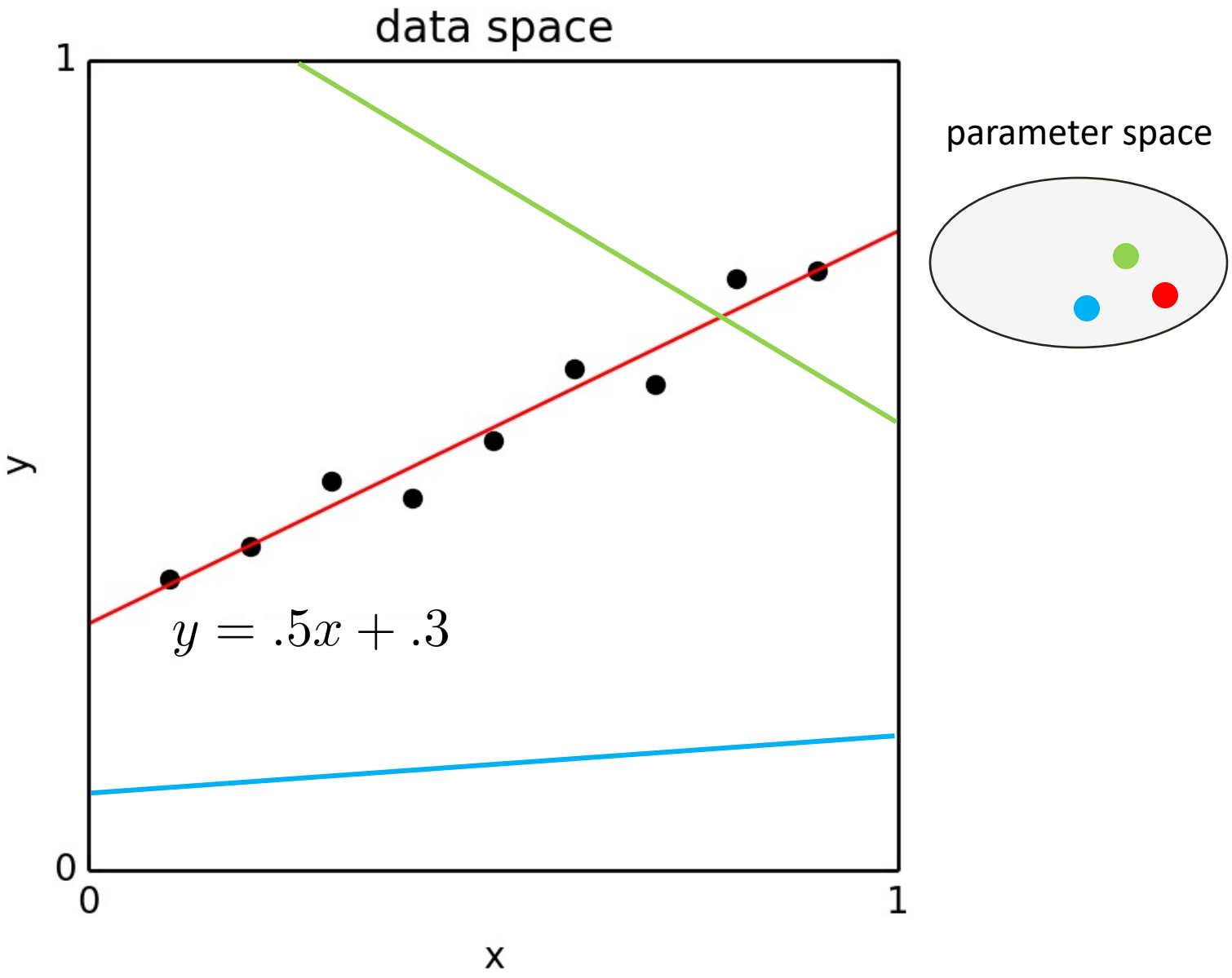
minimize squared difference, summed over training samples

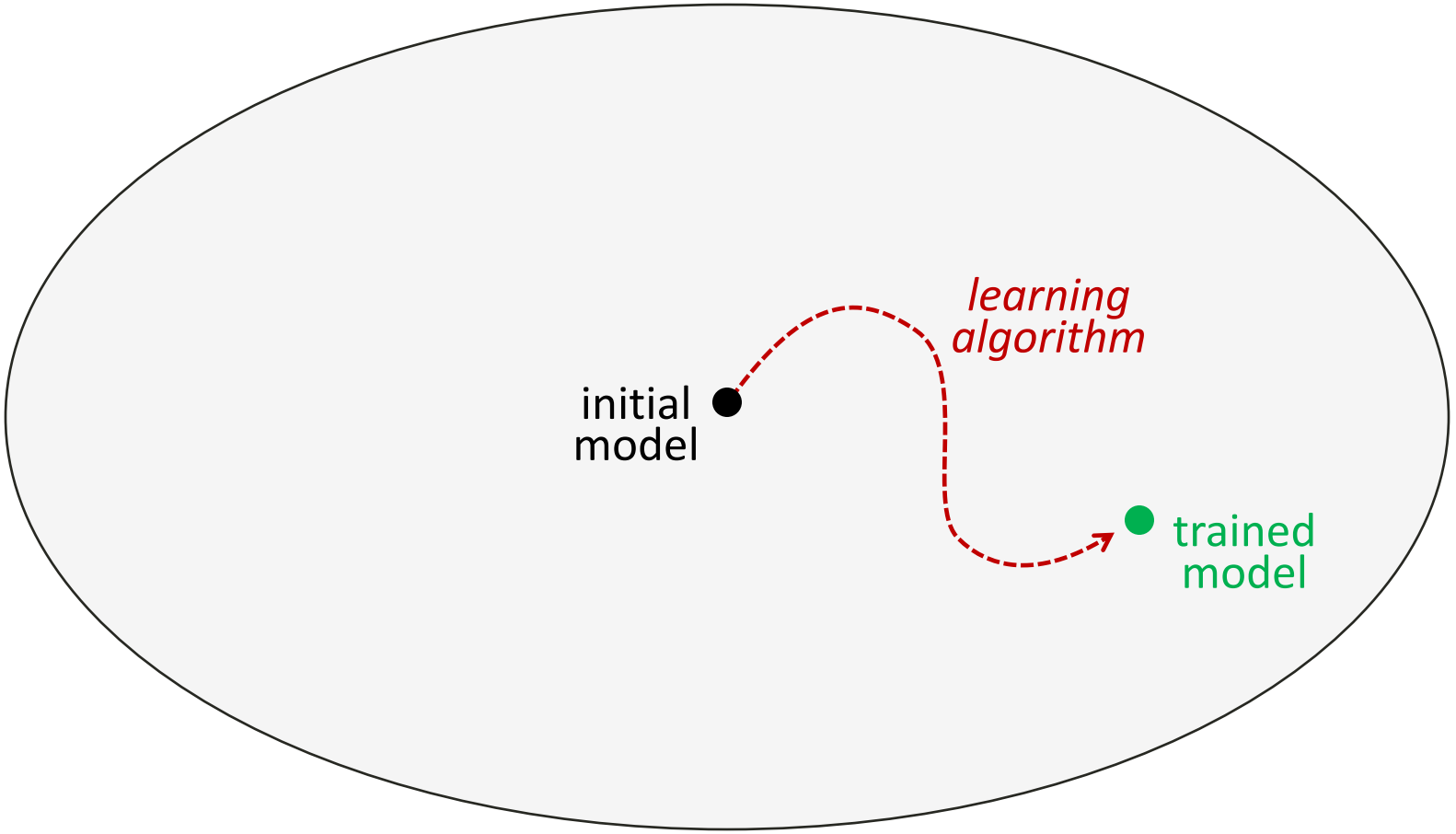
$$\ell(a,b) = \sum_{i=1}^{4} \frac{1}{2}(y_i - \hat{y}_i)^2$$

remember, this is a function of *a* and *b*!

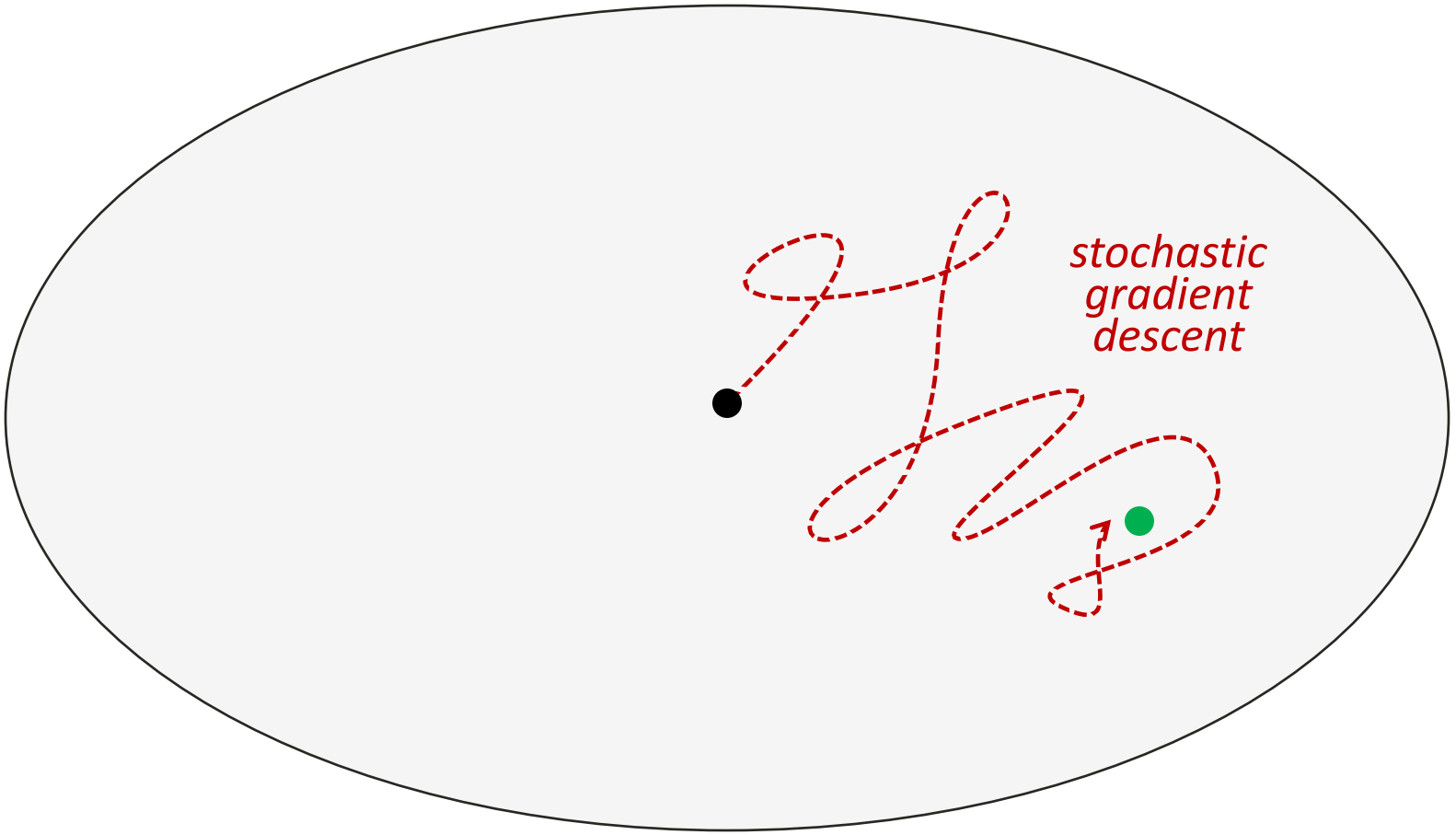
$$\hat{y}_i = ax_i + b$$


data space
1
0
0
1
y
x
$y = ax + b$
parameter space


data space
1
0
0
1
x
y
$y = .5x + .3$
parameter space



But what “learning algorithm” are we talking about?



Demo of “stochastic gradient descent” learning algorithm

Minimizing with gradients

How does the derivative help minimize a function of a single variable? Say, $f(w) = \frac{1}{2}(3-w)^2$ where

$$\frac{\mathrm{d}f}{\mathrm{d}w}(w) = w - 3$$

Direct solve method: set $\frac{\mathrm{d}f}{\mathrm{d}w} = 0$ and solve for *w*

$$0 = w - 3 \quad \Rightarrow \quad w^* = 3$$

Or, **descent method**: take small steps in direction of $-\frac{\mathrm{d}f}{\mathrm{d}w}$

$$w_{\mathrm{new}} = w - \eta \frac{\mathrm{d}f}{\mathrm{d}w}(w) \qquad \eta = 0.05 \text{ (for example)}$$

Minimizing with gradients

How does the gradient help minimize a function $f(a, b)$ of two variables?

$$\nabla f(a,b) = \begin{bmatrix} \frac{\partial f}{\partial a}(a,b) \\ \frac{\partial f}{\partial b}(a,b) \end{bmatrix}$$

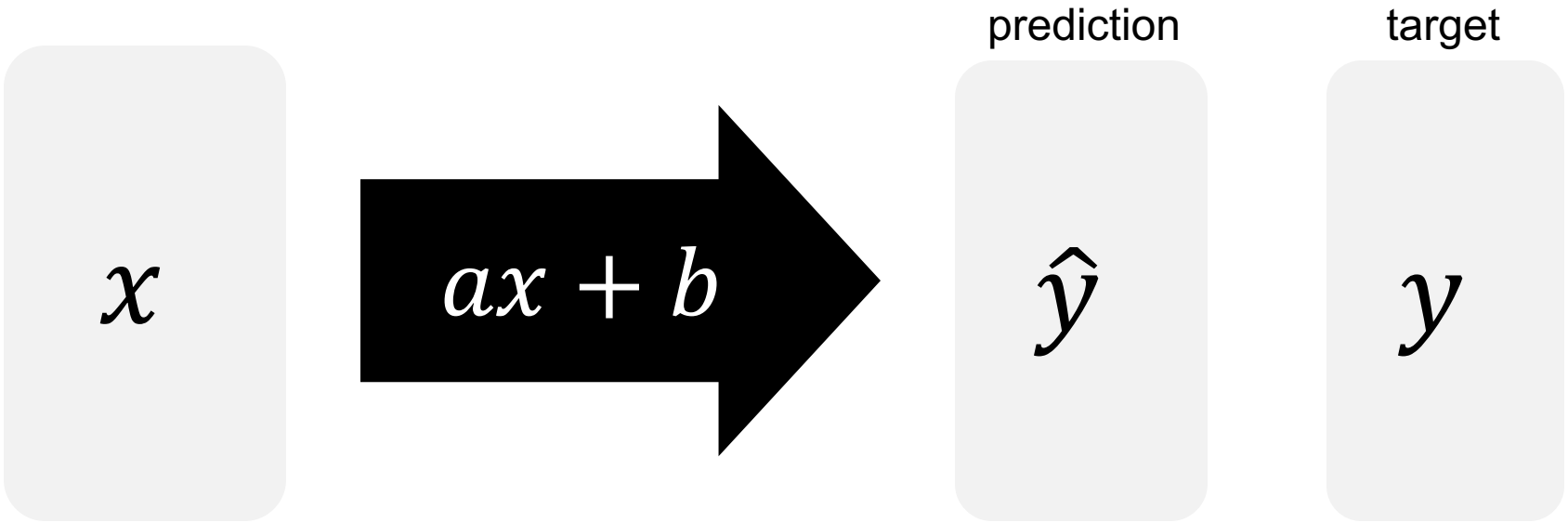
Direct solve: set $\nabla f = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and solve system of equalities for *a, b*

Or, **descent**: take small steps in direction of $-\nabla f$

$$a_{\text{new}} = a - \eta \frac{\partial f}{\partial a}(a,b)$$
$$b_{\text{new}} = b - \eta \frac{\partial f}{\partial b}(a,b)$$

$$\eta = 0.05 \text{ (for example)}$$

Gradient-based learning example



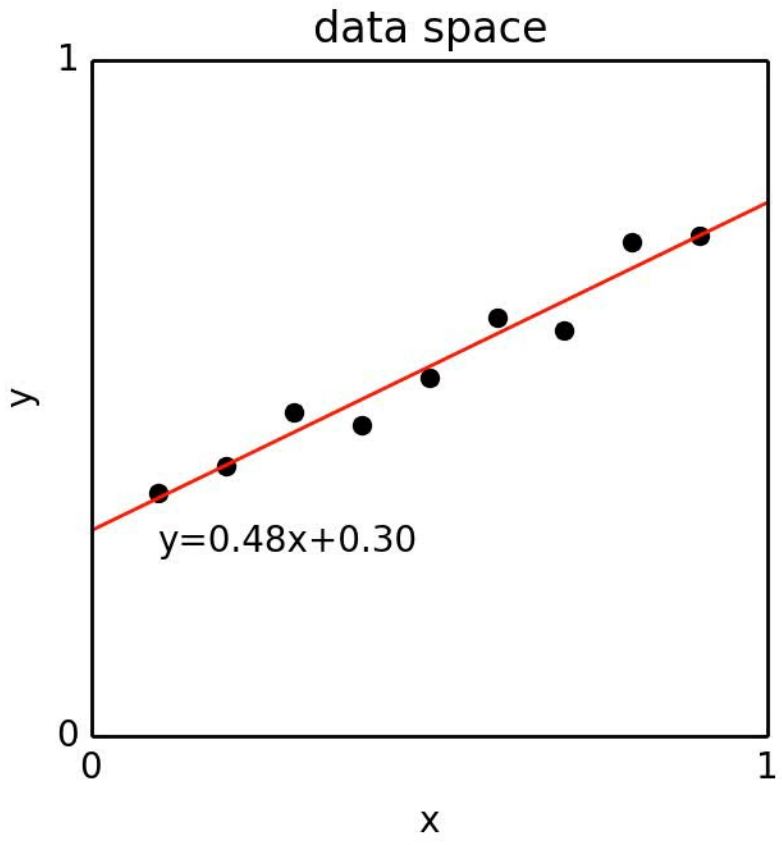
$$\ell(a, b) = \frac{1}{2}(y - \hat{y})^2$$
$$= \frac{1}{2}(y - ax - b)^2$$

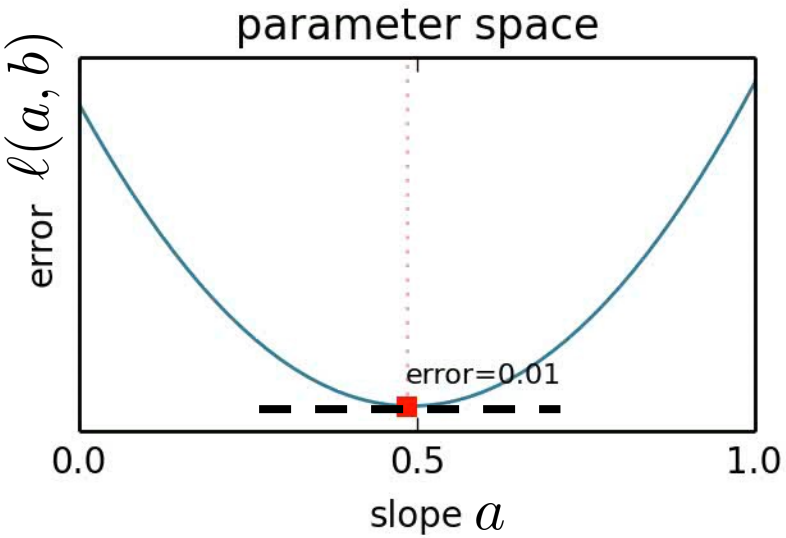
Gradient-based learning example

$$\ell(a,b) = \frac{1}{2}(y-\hat{y})^2$$
$$= \frac{1}{2}(y-ax-b)^2$$

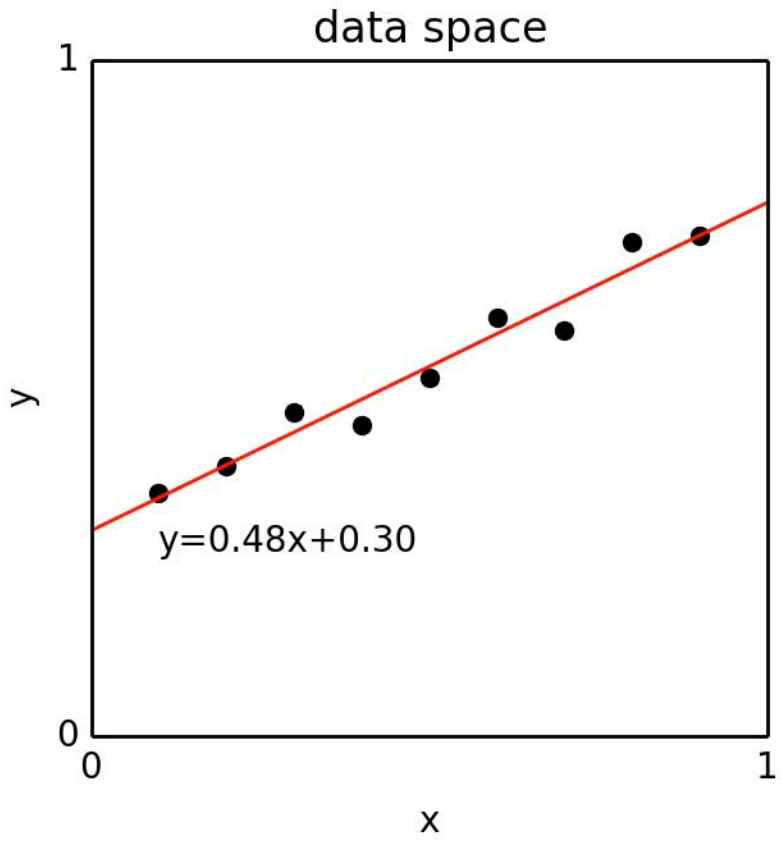
gradient for single training point (x,y)

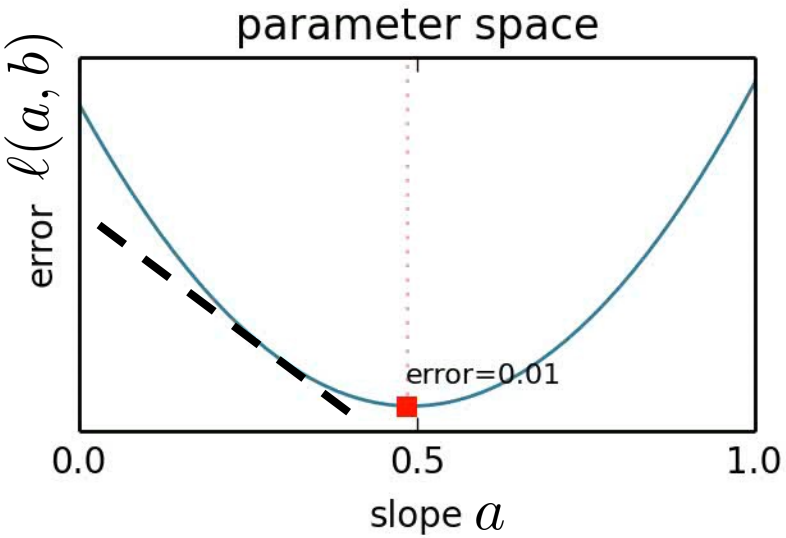
$$\begin{cases} \frac{\partial \ell}{\partial a} = (\hat{y}-y)x \\ \frac{\partial \ell}{\partial b} = (\hat{y}-y) \end{cases}$$



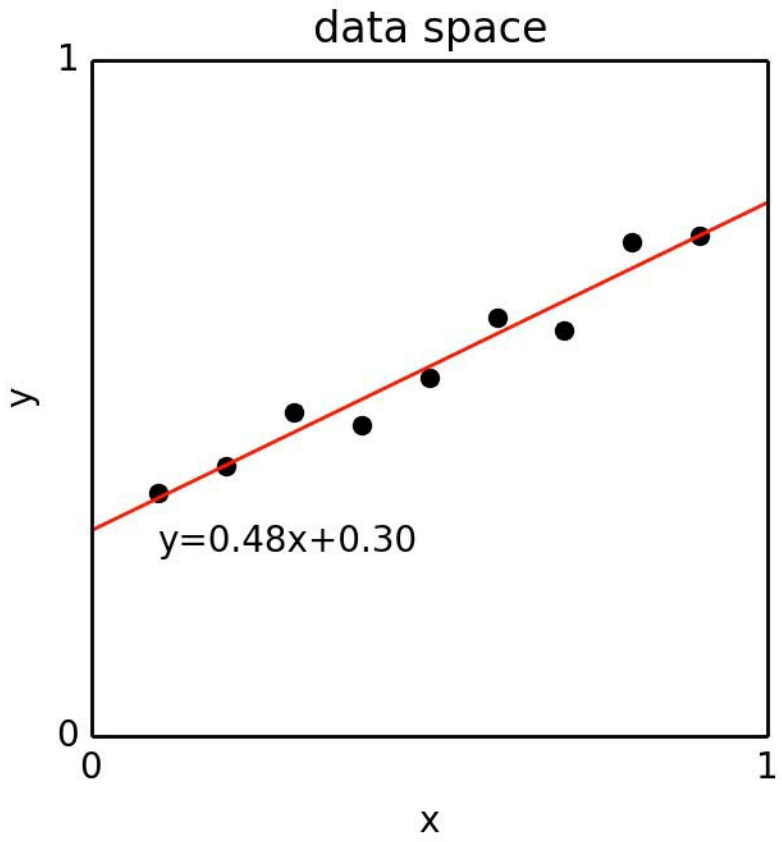


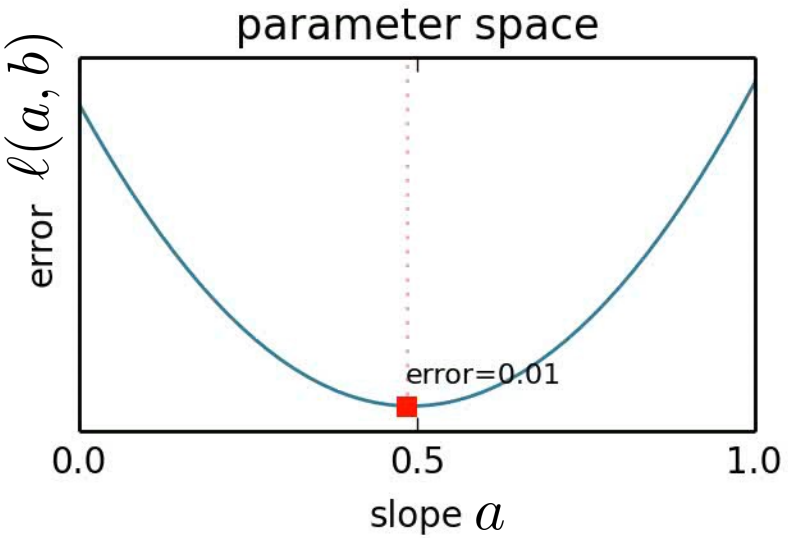
$$\frac{\partial f}{\partial a}(0.48, 0.30) \approx 0$$



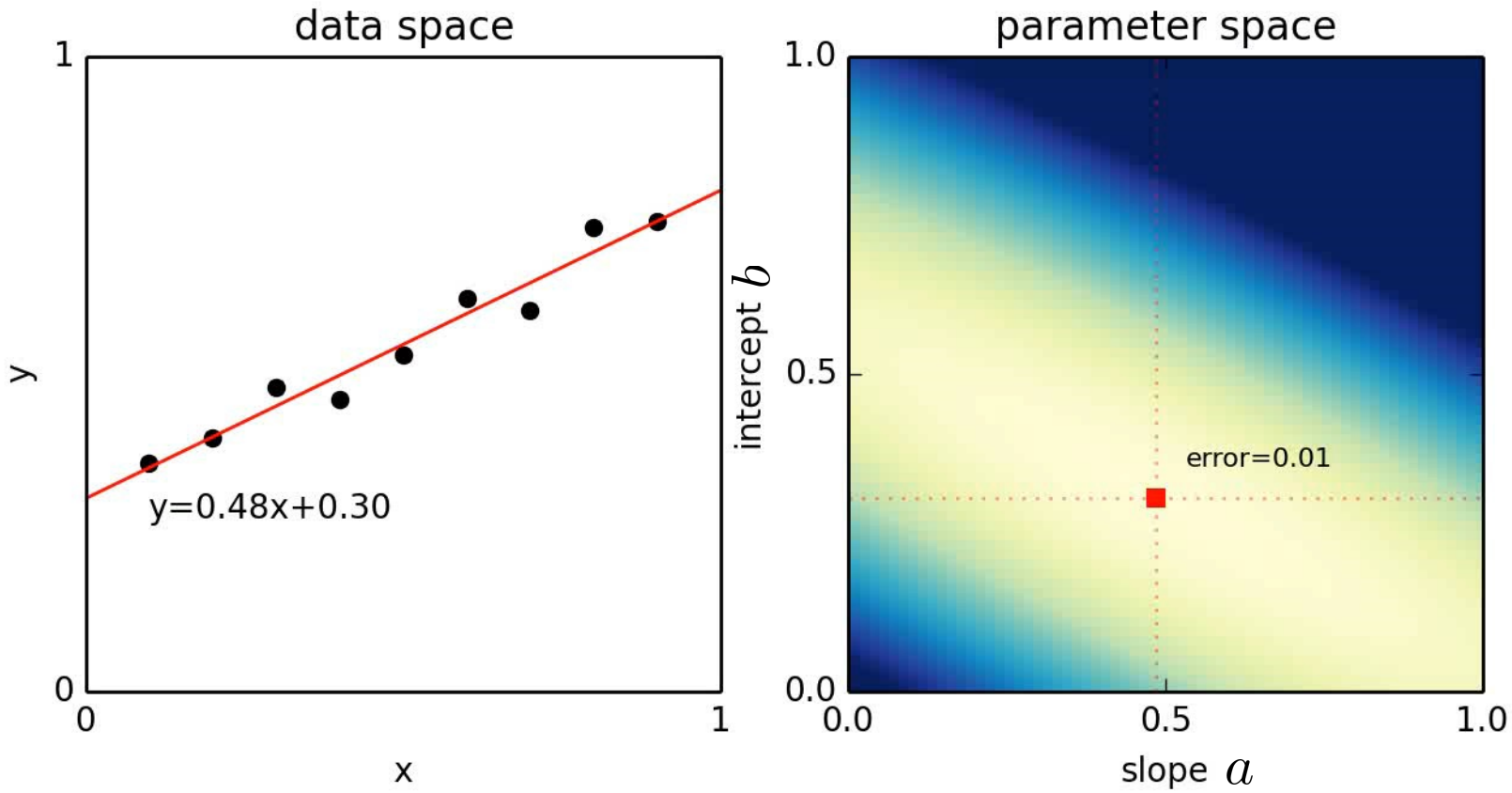


$$\frac{\partial f}{\partial a}(0.25, 0.30) < 0$$





Error lowest when *a* is near 0.5

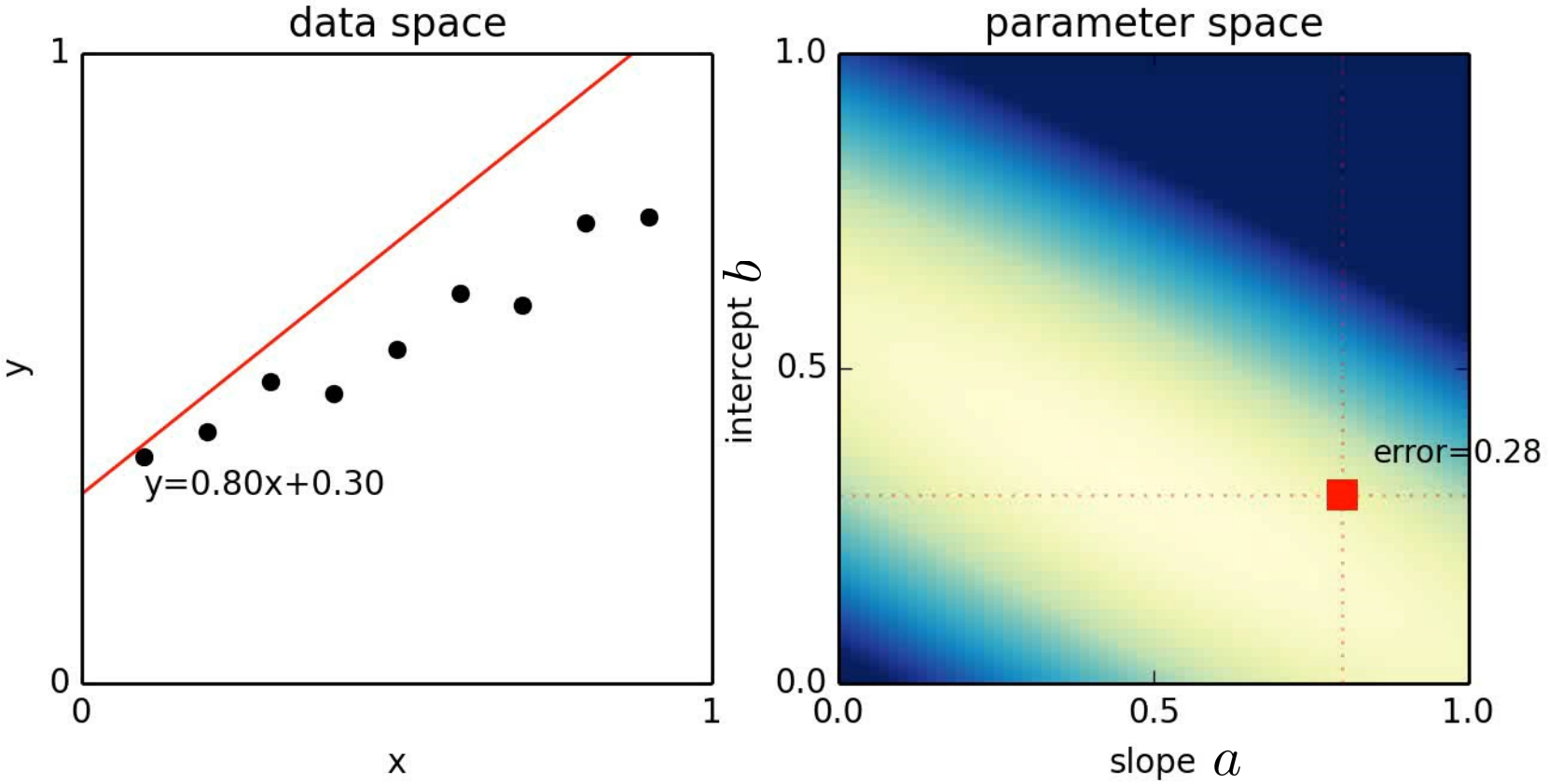


Error lowest when *a* is near 0.5

(this slide is a video)

Gradient Descent

learning rate = 0.05

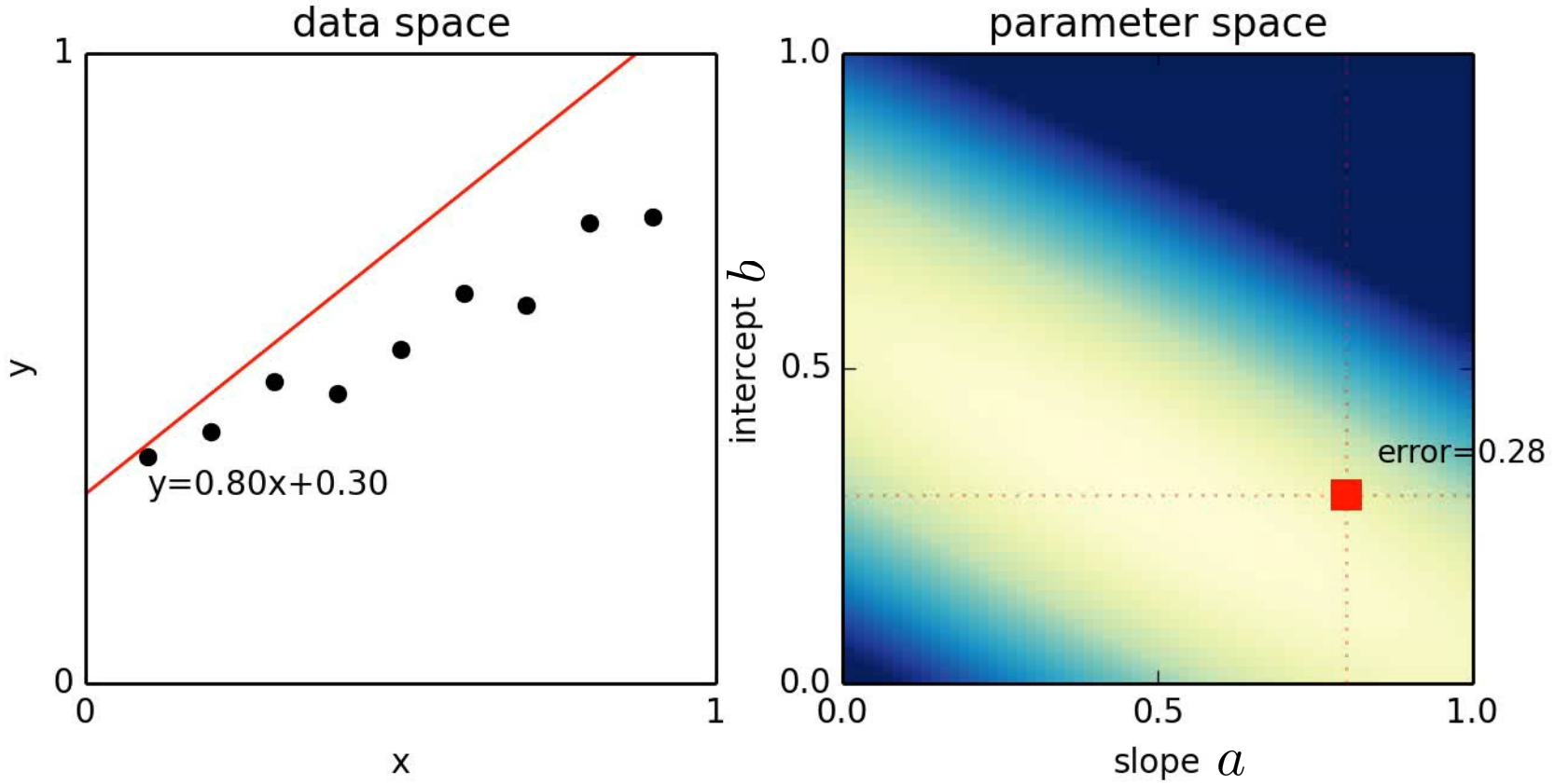


Basic gradient descent can 'learn' both *a* and *b*, but can require many steps.

(this slide is a video)

Gradient Descent

learning rate = **0.50**

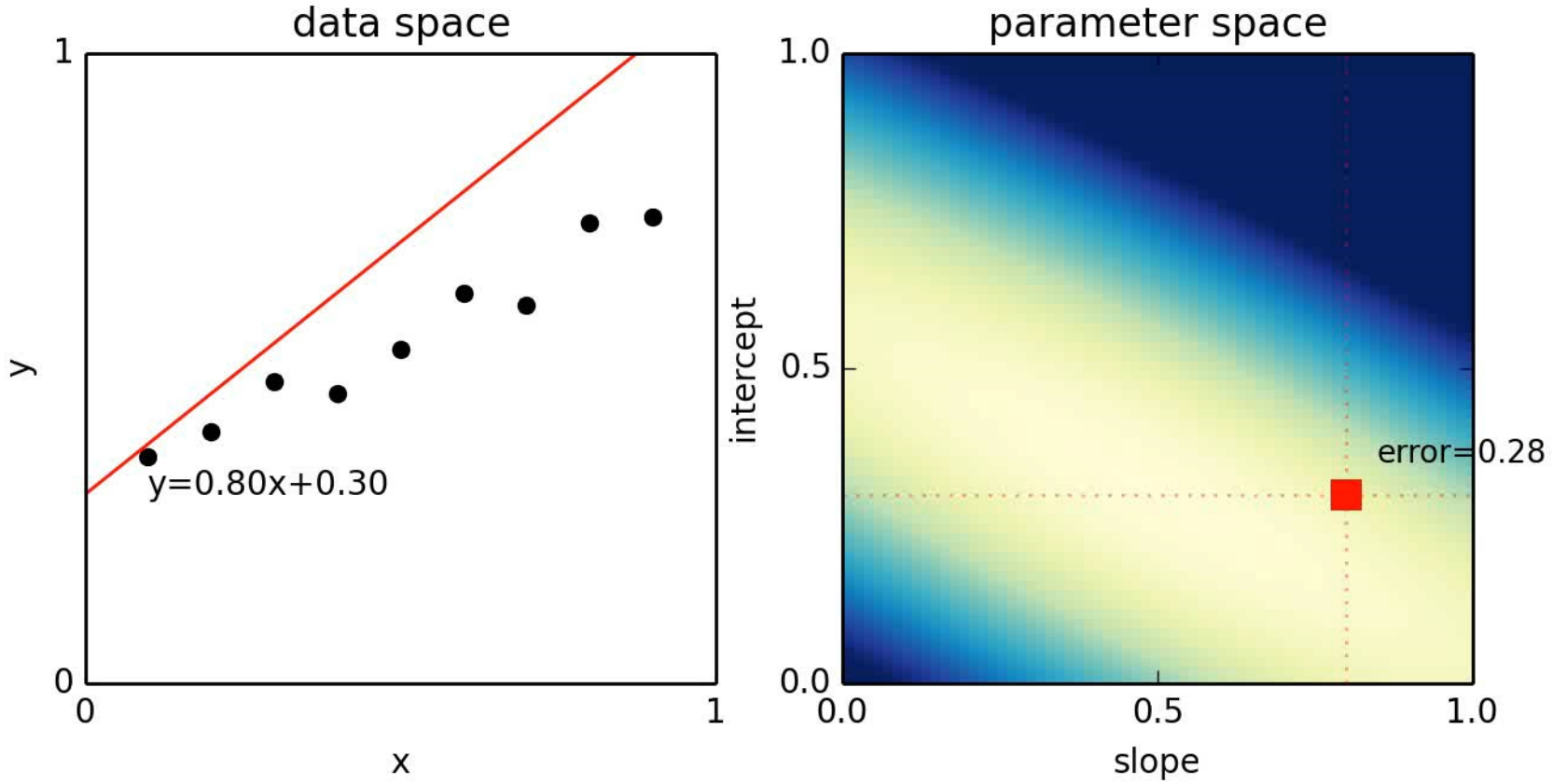


Bigger learning rate (step size) can help, but hard to guess best step size ahead of time

(this slide is a video)

Gradient Descent

learning rate = 0.05
momentum = **0.90**

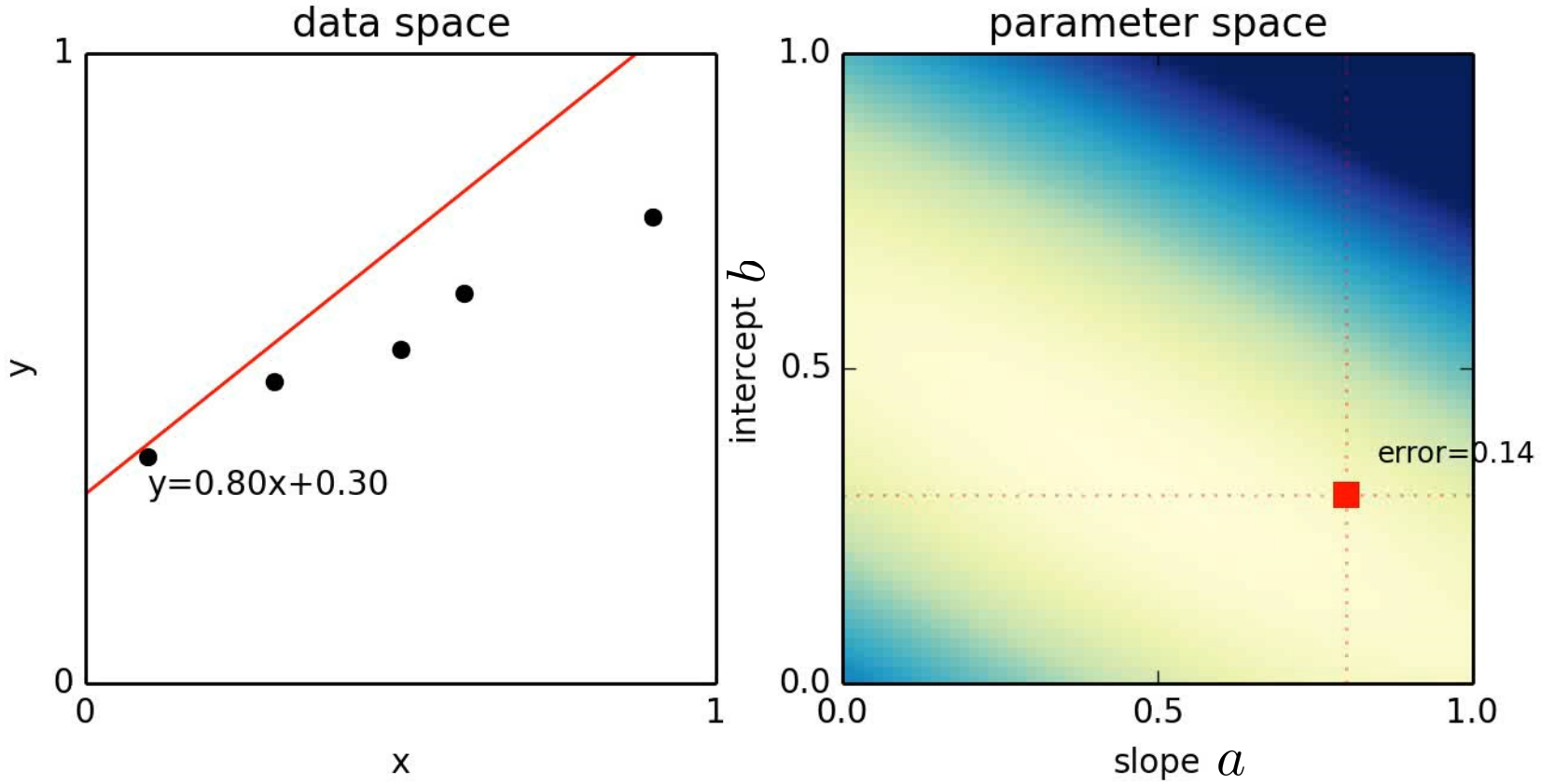


"Momentum" is a technique for adapting the learning rate in a smart way. (We'll introduce this with neural networks.)

(this slide is a video)

Stochastic Gradient

learning rate = 0.05
momentum = 0.90
mini-batch size = **5**



Sub-samples of data give “good enough” gradients faster

(this slide is a video)

```
a, b = .8, .3  # Initial parameters
da = db = 0

for i in range(200):
    # Sample 5 data points
    x, y = data.sample(5)

    # Calculate gradient (ga, gb)
    y_ = a * x + b
    ga = mean( (y_ - y) * x )
    gb = mean( (y_ - y)     )

    # Calculate step with momentum .9
    # and take step with rate .05
    da = .9*da - ga
    db = .9*db - gb
    a += .05*da
    b += .05*db
```

stochastic

gradient

descent

$$\frac{\partial \ell}{\partial a} = (\hat{y} - y)x$$

$$\frac{\partial \ell}{\partial b} = (\hat{y} - y)$$

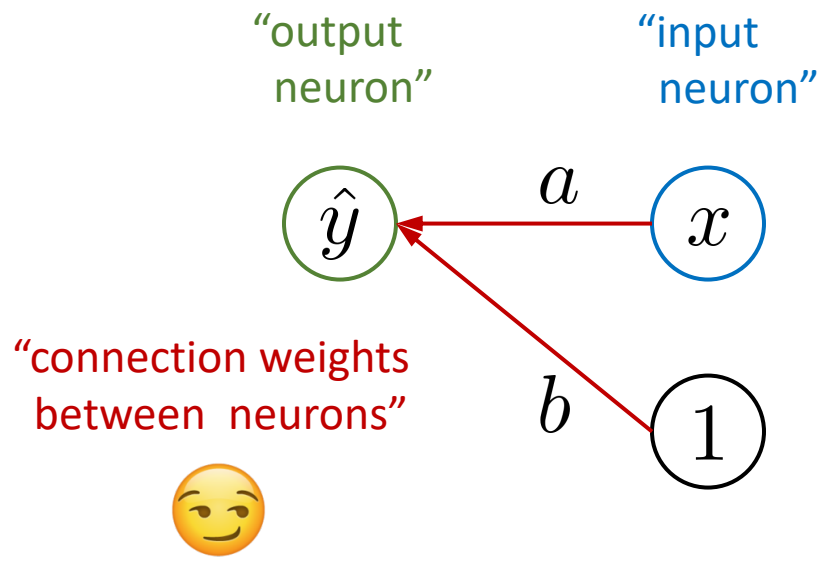
As special case of neural networks

"solving linear least squares by SGD"

$$\hat{y} = ax + b$$

… is equivalent to …

"training a 1-1 linear neural network using squared error loss"



😏

Linear least squares (LS) *setup*

- Model:

  Inputs ("features") $\mathbf{x} = \begin{bmatrix} 1 & x_1 & x_2 & \cdots & x_D \end{bmatrix}^T$

  Parameters ("weights") $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_D \end{bmatrix}^T$

  Output ("prediction") $\hat{y}(\mathbf{x}, \mathbf{w}) = \mathbf{x}^T \mathbf{w} = w_0 + \sum_{j=1}^{D} w_j x_j$

- Training:

  Training set: $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$

  Training loss: $\frac{1}{2} \sum_{i=1}^{N} \left(y_i - \hat{y}(\mathbf{x}_i, \mathbf{w}) \right)^2$

tune *this* very special kind of program to match *that* data

Linear least squares (LS) *learning*

- Loss is $\ell_{\mathrm{LS}}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\left(y_i - \mathbf{x}_i^T\mathbf{w}\right)^2 = \frac{1}{2}(\mathbf{y}-\mathbf{X}\mathbf{w})^T(\mathbf{y}-\mathbf{X}\mathbf{w})$

$$\text{where } \mathbf{X} = \begin{bmatrix}\mathbf{x}_1^T\\ \mathbf{x}_2^T\\ \vdots\\ \mathbf{x}_N^T\end{bmatrix} = \begin{bmatrix}1 & x_{11} & \cdots & x_{1D}\\ 1 & x_{21} & \cdots & x_{2D}\\ & & & \\ 1 & \vdots & \ddots & \vdots\\ 1 & x_{N1} & \cdots & x_{ND}\end{bmatrix}$$

- Gradient of loss is

$$\nabla\ell_{\mathrm{LS}}(\mathbf{w}) = \begin{bmatrix}\frac{\partial\ell}{\partial w_0}(\mathbf{w})\\ \frac{\partial\ell}{\partial w_1}(\mathbf{w})\\ \vdots\\ \frac{\partial\ell}{\partial w_D}(\mathbf{w})\end{bmatrix} = \begin{bmatrix}\sum_{i=1}^{N}\left(\mathbf{x}_i^T\mathbf{w}-y_i\right)\\ \sum_{i=1}^{N}\left(\mathbf{x}_i^T\mathbf{w}-y_i\right)x_{i1}\\ \vdots\\ \sum_{i=1}^{N}\left(\mathbf{x}_i^T\mathbf{w}-y_i\right)x_{iD}\end{bmatrix} = \sum_{i=1}^{N}\left(\mathbf{x}_i^T\mathbf{w}-y_i\right)\mathbf{x}_i$$

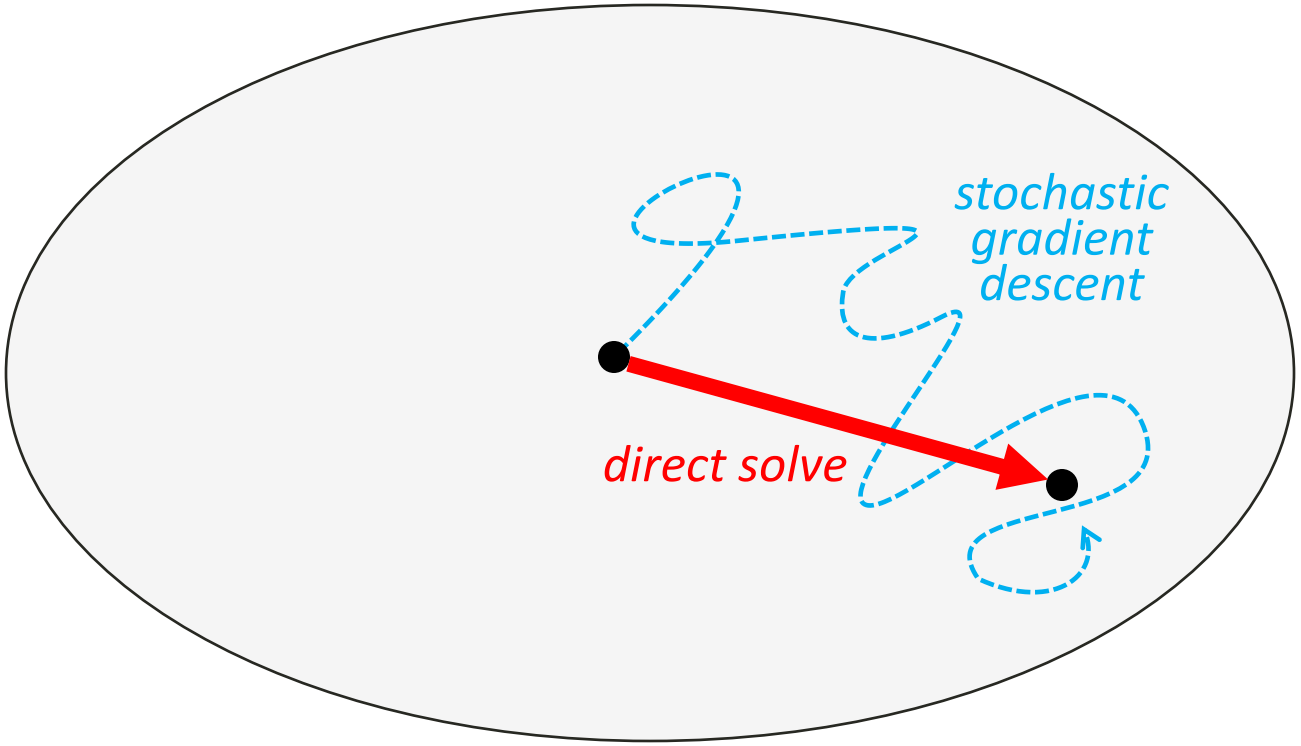
$$= (\mathbf{X}^T\mathbf{X})\mathbf{w} - \mathbf{X}^T\mathbf{y}$$

- Set gradient to zero, try to solve system of equations:

$$\mathbf{0} = \underbrace{(\mathbf{X}^T\mathbf{X})}_{\text{matrix}}\mathbf{w} - \underbrace{(\mathbf{X}^T\mathbf{y})}_{\text{vector}} \quad \Rightarrow \mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Solve linear system of equations!
No need to take gradient steps!

Linear least squares regression allows for *a direct* solution to the learning problem



(Nice! Not all models are amenable to direct solve!)

sklearn.linear_model.LinearRegression

class sklearn.linear_model.**LinearRegression**(*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=None*) [source]

Ordinary least squares Linear Regression.

LinearRegression fits a linear model with coefficients w = (w1, ..., wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

fit(*X*, *y*, *sample_weight=None*) [source]

Fit linear model.

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
Training data

y : *array-like of shape (n_samples,) or (n_samples, n_targets)*
Target values. Will be cast to X's dtype if necessary

Simple generalization of
linear least squares:

Linear *basis function* models

Linear basis function model *setup*

basically preprocessing your original features

- Model:

  Inputs ("original features") $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_D \end{bmatrix}^T$

  Basis function ("features") $\boldsymbol{\phi} = \begin{bmatrix} 1 & \phi_1(\mathbf{x}) & \phi_2(\mathbf{x}) & \cdots & \phi_M(\mathbf{x}) \end{bmatrix}^T$

  Parameters ("weights") $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_M \end{bmatrix}^T$

  Output ("prediction") $\hat{y}(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) = w_0 + \sum_{j=1}^{M} w_j \phi_j(\mathbf{x})$

- Training:

  Training set: $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$

  Training loss: $\frac{1}{2} \sum_{i=1}^{N} (y_i - \hat{y}(\mathbf{x}_i, \mathbf{w}))^2$

Still linear in $\mathbf{w}$ even if $\boldsymbol{\phi}$ not linear in $\mathbf{x}$!

e.g. $\phi(\mathbf{x}) = \begin{bmatrix} x_1 & x_2 & x_1 x_2 \end{bmatrix}^T$

Linear basis function model *learning*

- Loss is $\ell(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\left(y_i - \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}_i)\right)^2 = \frac{1}{2}(\mathbf{y}-\boldsymbol{\Phi}\mathbf{w})^T(\mathbf{y}-\boldsymbol{\Phi}\mathbf{w})$

$$\text{where } \boldsymbol{\Phi} = \begin{bmatrix}\boldsymbol{\phi}(\mathbf{x}_1)^T \\ \boldsymbol{\phi}(\mathbf{x}_2)^T \\ \vdots \\ \boldsymbol{\phi}(\mathbf{x}_N)^T\end{bmatrix} = \begin{bmatrix}1 & \phi_1(\mathbf{x}_1) & \cdots & \phi_M(\mathbf{x}_1) \\ 1 & \phi_1(\mathbf{x}_2) & \cdots & \phi_M(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(\mathbf{x}_N) & \cdots & \phi_M(\mathbf{x}_N)\end{bmatrix}$$

- Gradient of loss is

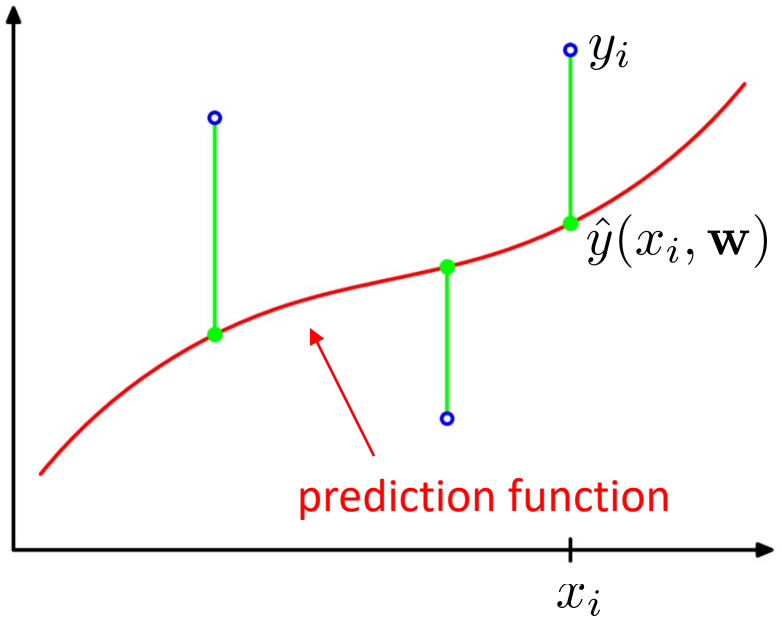
$$\nabla\ell(\mathbf{w}) = (\boldsymbol{\Phi}^T\boldsymbol{\Phi})\mathbf{w} - \boldsymbol{\Phi}^T\mathbf{y}$$

$$\Rightarrow \mathbf{w}^* = (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\mathbf{y}$$

Direct solution has same form because features $\boldsymbol{\Phi}$ are *fixed*, even if $\boldsymbol{\Phi}$ is a non-linear transformation of raw features $\mathbf{X}$

Example: Polynomial curve fitting

- Choose basis function $\phi(x) = \begin{bmatrix} 1 & x & x^2 & \cdots & x^M \end{bmatrix}^T$
- Prediction is $\hat{y}(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_m x^M = \mathbf{w}^T \boldsymbol{\phi}(x)$
- Loss is $\ell(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N} (y_i - \hat{y}(x_i, \mathbf{w}))^2$



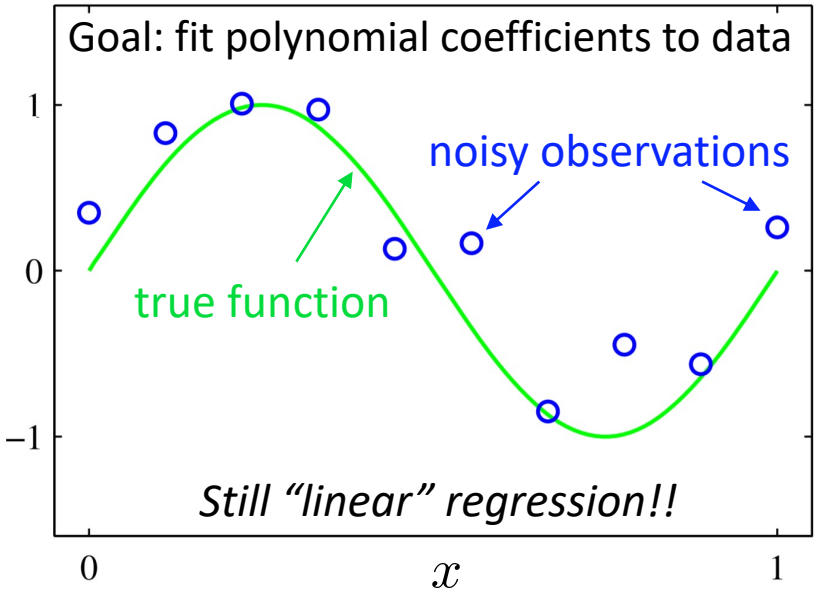


Image credit: Christopher M. Bishop

Example: Polynomial curve fitting

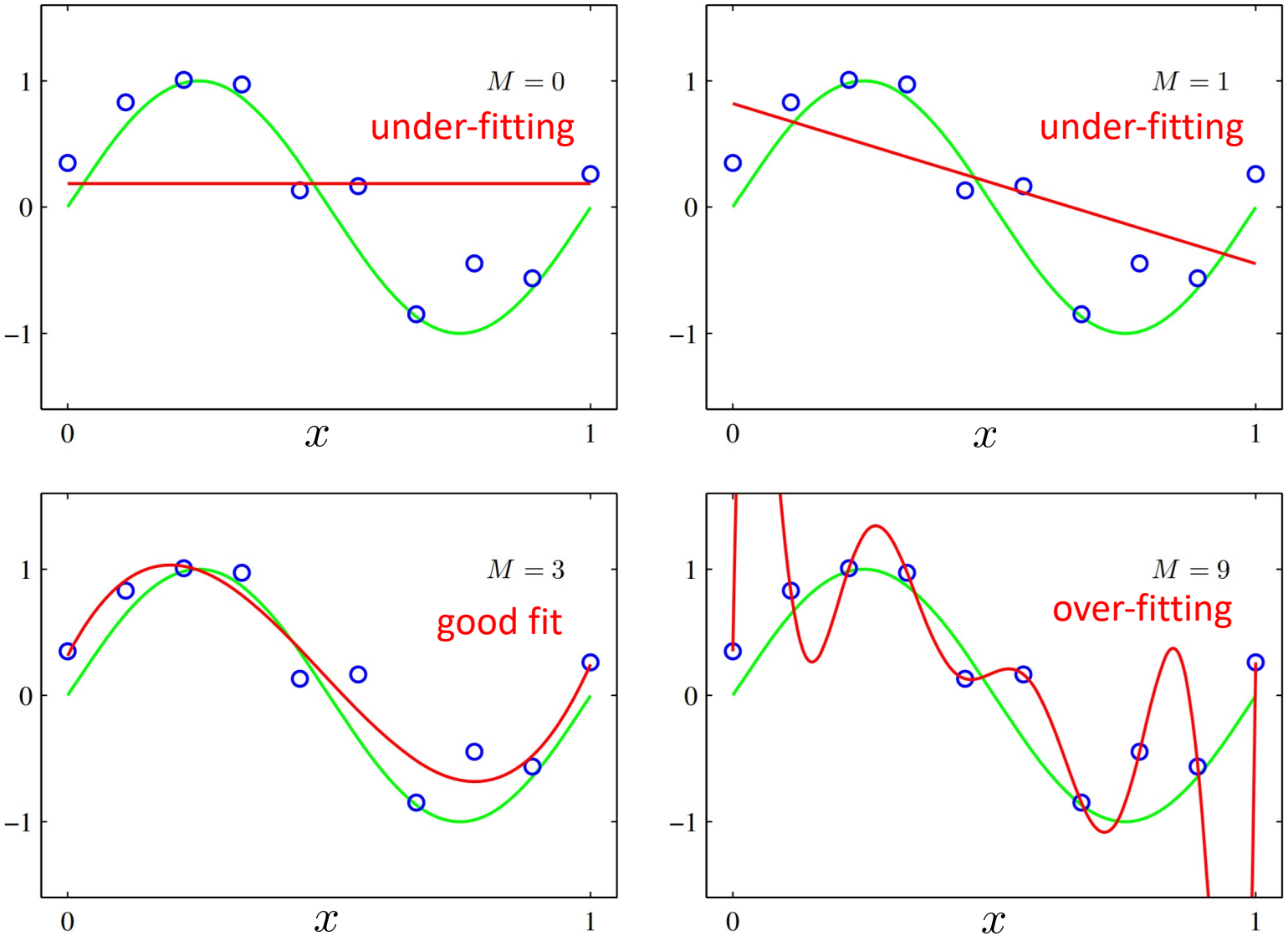
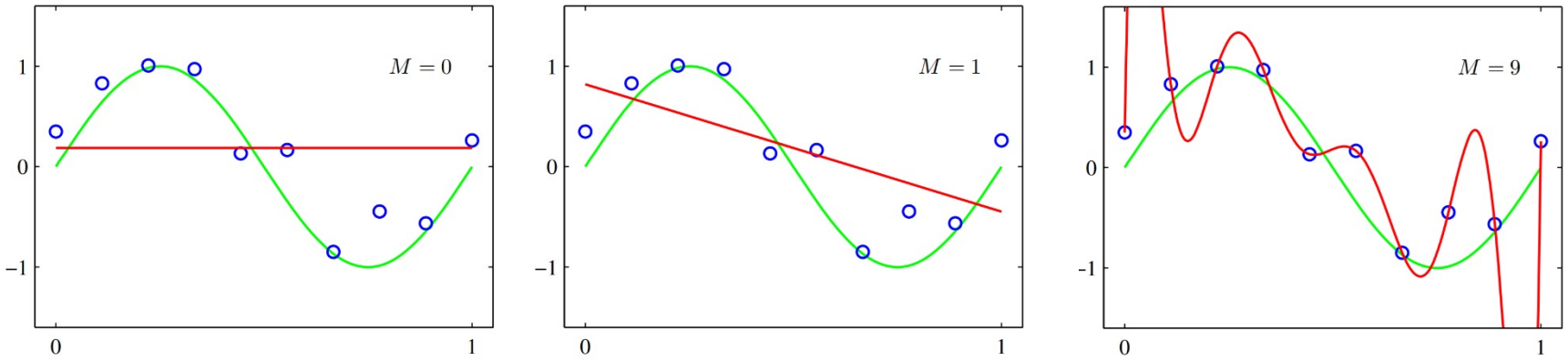


Image credit: Christopher M. Bishop

Example: Polynomial curve fitting

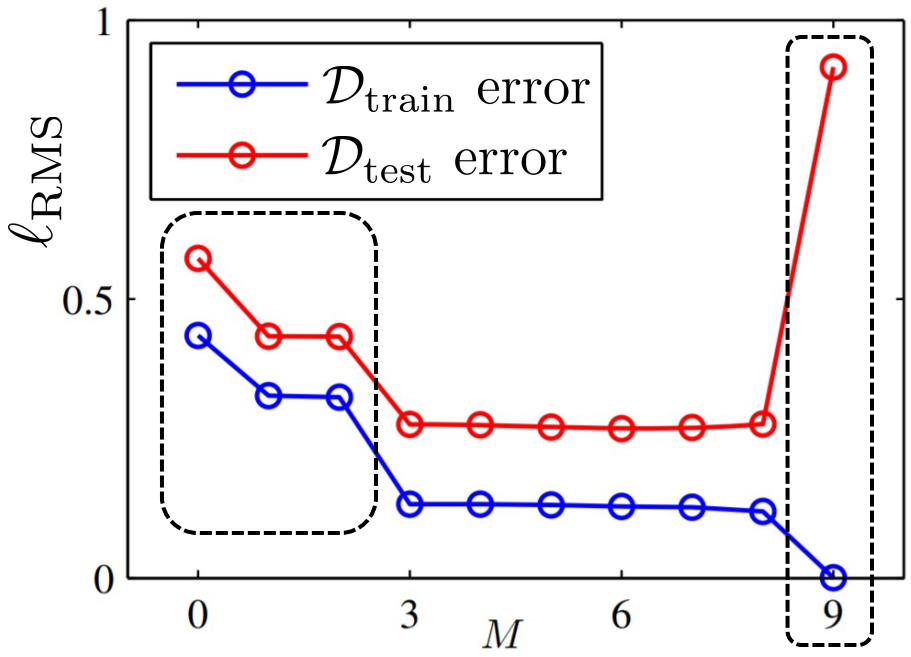


	$M=0$	$M=1$	$M=9$
$w_0^\star$	0.19	0.82	0.35
$w_1^\star$		-1.27	232.37
$w_2^\star$			-5321.83
$w_3^\star$			48568.31
$w_4^\star$			-231639.30
$w_5^\star$			640042.26
$w_6^\star$			-1061800.52
$w_7^\star$			1042400.18
$w_8^\star$			-557682.99
$w_9^\star$			125201.43

Image credit: Christopher M. Bishop

Symptoms of under- and over-fitting

- First, define *root mean squared* (RMS) error so we can compare error across multiple data sets $\mathcal{D}$:



$$\ell_{\mathrm{RMS}} = \sqrt{\frac{2}{N}\ell(\mathbf{w}^*)}$$

Defined w.r.t. a specific data set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ even if not shown.

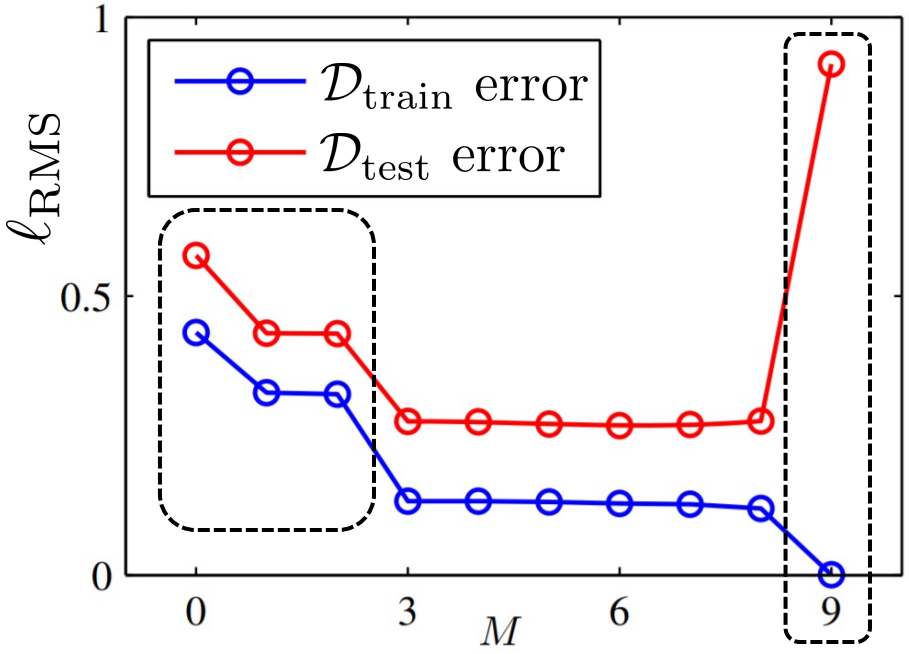
- Test data: new samples that were *not* used for learning

Image credit: Christopher M. Bishop

Symptoms of under- and over-fitting

high training error
and test error

low training error
but high test error



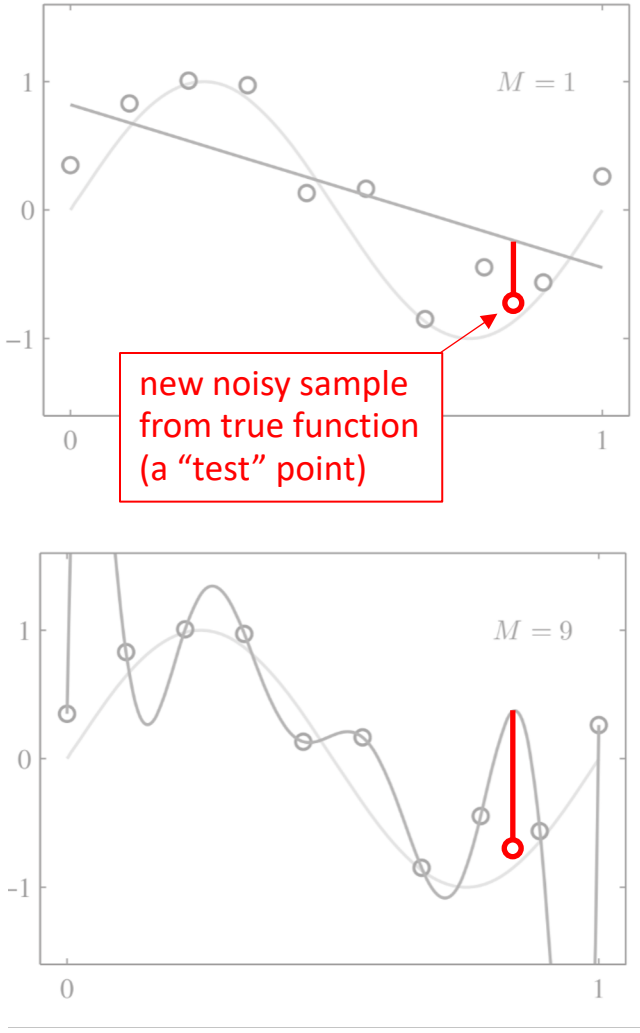


Image credit: Christopher M. Bishop

Limit over-fitting by *getting more data*

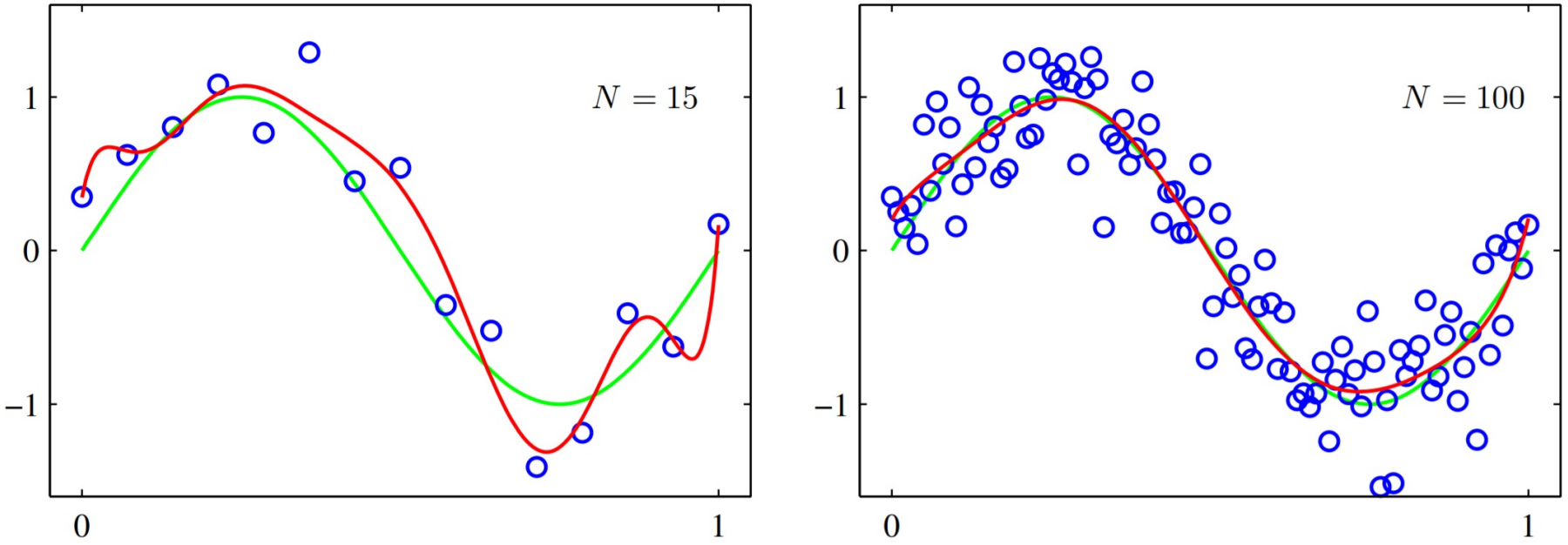


Figure 1.6 Plots of the solutions obtained by minimizing the sum-of-squares error function using the $M = 9$ polynomial for $N = 15$ data points (left plot) and $N = 100$ data points (right plot). We see that increasing the size of the data set reduces the over-fitting problem.

Image credit: Christopher M. Bishop

Limit over-fitting by "*regularization*"

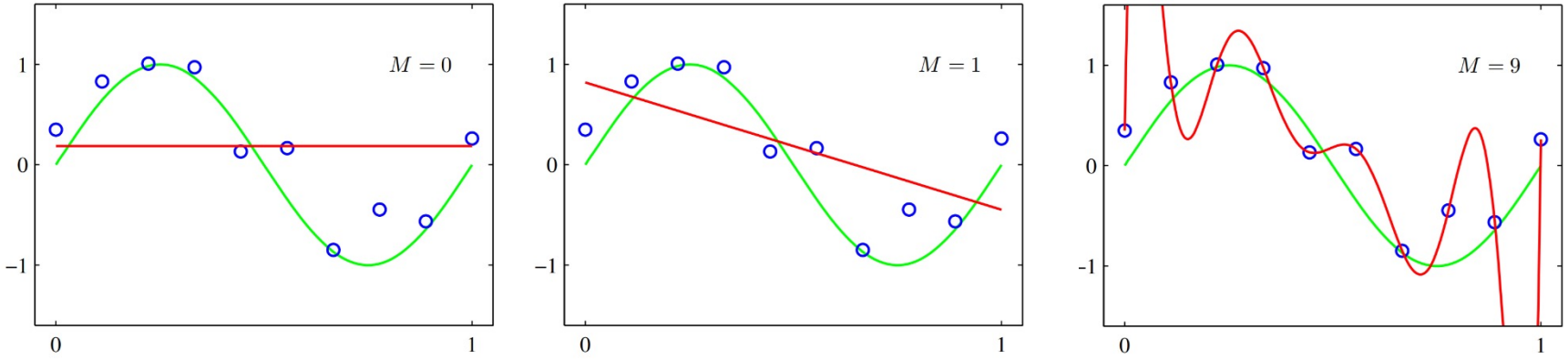


Table 1.1 Table of the coefficients $\mathbf{w}^\star$ for polynomials of various order. Observe how the typical magnitude of the coefficients increases dramatically as the order of the polynomial increases.

	$M=0$	$M=1$	$M=9$
$w_0^\star$	0.19	0.82	0.35
$w_1^\star$		-1.27	232.37
$w_2^\star$			-5321.83
$w_3^\star$			48568.31
$w_4^\star$			-231639.30
$w_5^\star$			640042.26
$w_6^\star$			-1061800.52
$w_7^\star$			1042400.18
$w_8^\star$			-557682.99
$w_9^\star$			125201.43

Image credit: Christopher M. Bishop

Regularized least squares (RLS)

- "L2" regularized loss is $\tilde{\ell}(\mathbf{w}) = \ell(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$

$$\text{where 2-norm is } \|\mathbf{w}\|_2 = \sqrt{\mathbf{w}^T\mathbf{w}} \text{ and } \lambda \geq 0$$

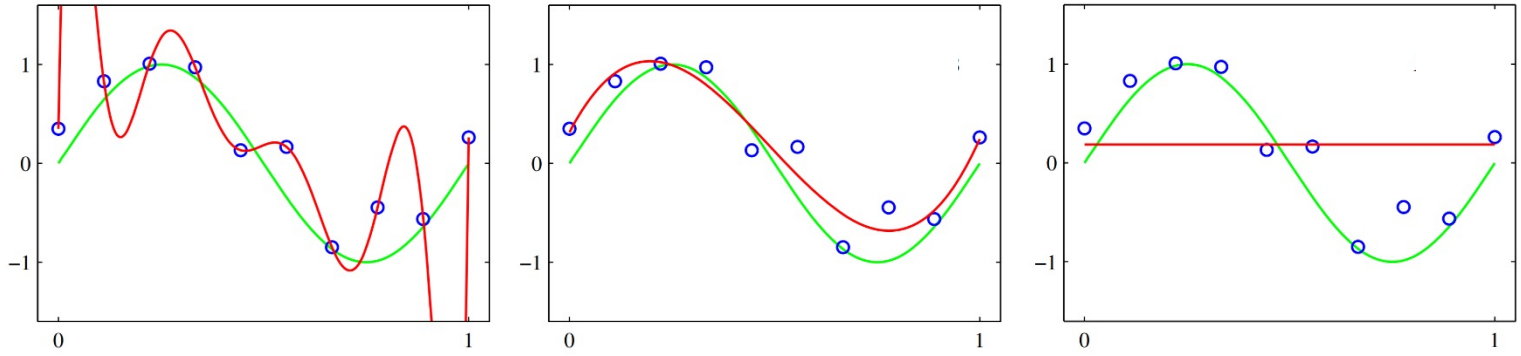
- Gradient: $\nabla\tilde{\ell}(\mathbf{w}) = \nabla\ell(\mathbf{w}) + \lambda\mathbf{w}$

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

- Direct solution: $\mathbf{w}^* = (\mathbf{\Phi}^T\mathbf{\Phi} + \lambda\mathbf{I})^{-1}\mathbf{\Phi}^T\mathbf{y}$

Bishop equation (1.4) p.10, equation (3.24) p.144

Regularized least squares (RLS)



	$\lambda = 0$	$\lambda = 1.5 \times 10^{-8}$	$\lambda = 1$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01

Image credit: Christopher M. Bishop

Regularized least squares (RLS)

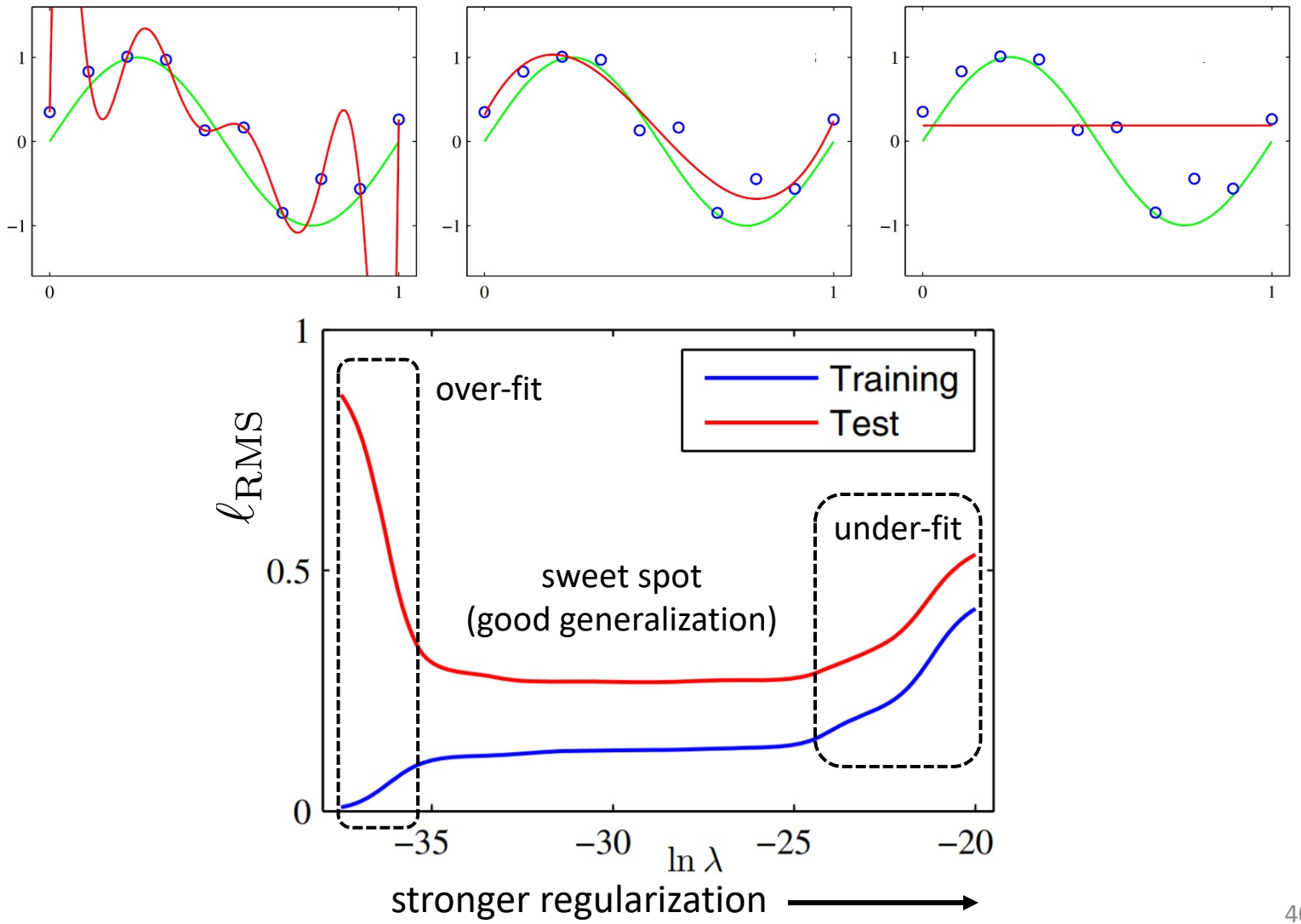


Image credit: Christopher M. Bishop

sklearn.linear_model.Ridge

class `sklearn.linear_model.` **Ridge**(*alpha=1.0*, ***, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *random_state=None*) [source]

Linear least squares with l2 regularization.

L2 penalty on weights is also called "**ridge regression**"

Minimizes the objective function:

```
||y - Xw||^2_2 + alpha * ||w||^2_2
```

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape (n_samples, n_targets)).

Read more in the User Guide.

sklearn calls scaling factor "alpha." Bishop uses "lambda" symbol. Don't get confused!

Parameters:

alpha : *{float, ndarray of shape (n_targets,)}, default=1.0*

Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `1 / (2C)` in other linear models such as **`LogisticRegression`** or **`sklearn.svm.LinearSVC`**. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

Regularized least squares (RLS)

- "L1" regularized loss is $\tilde{\ell}(\mathbf{w}) = \ell(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|_1$

$$\text{where 1-norm is } \|\mathbf{w}\|_1 = \sum_{j=1}^{M} |w_j| \text{ and } \lambda \geq 0$$

sklearn.linear_model.Lasso

class `sklearn.linear_model.` **Lasso**(*alpha=1.0, *, fit_intercept=True, normalize=False, precompute=False, copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic'*) [source]

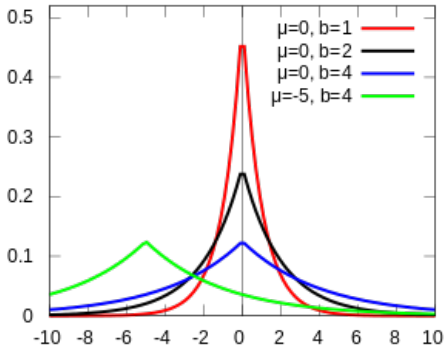
Linear Model trained with L1 prior as regularizer (aka the Lasso)

The optimization objective for Lasso is:

L1 penalty on weights is also called "**lasso regularization**"

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Probabilistic view:
MAP estimation using
Laplace prior on weights
(rather than Gaussian)



sklearn.linear_model.ElasticNet

class `sklearn.linear_model.` **`ElasticNet`**(*alpha=1.0, *, l1_ratio=0.5, fit_intercept=True, normalize=False, precompute=False, max_iter=1000, copy_X=True, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic'*) [source]

Linear regression with combined L1 and L2 priors as regularizer.

Minimizes the objective function:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

If you are interested in controlling the L1 and L2 penalty separately, keep in mind that this is equivalent to:

```
a * L1 + b * L2
```

where:

Combined L1 and L2 penalties
Is called “Elastic Net”

```
alpha = a + b and l1_ratio = a / (a + b)
```

Probabilistic perspective on linear regression

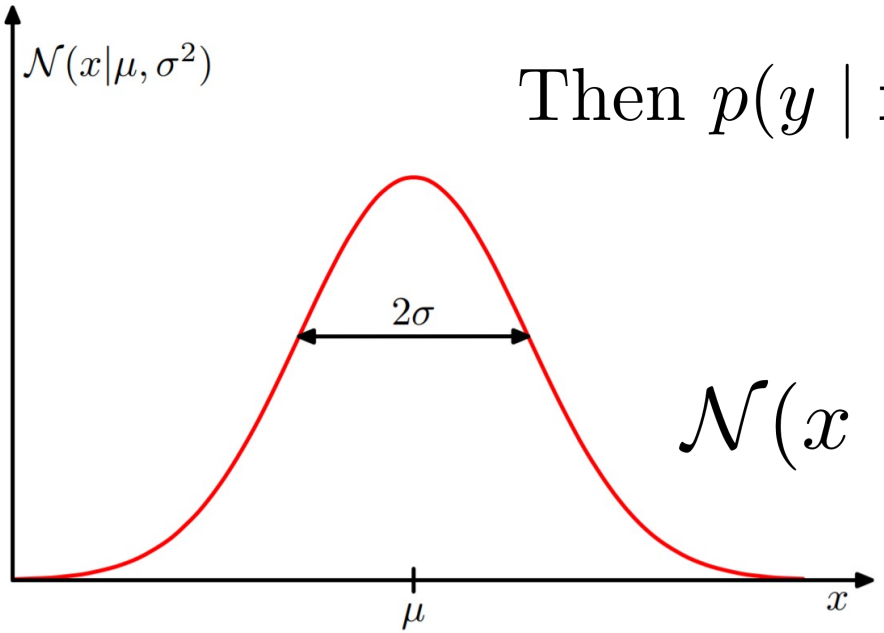
Probabilistic perspective

- Probabilistic motivation for *least squares:*

Assume observation $y = \hat{y}(\mathbf{x}, \mathbf{w}) + \epsilon$

where stochastic error $\epsilon \sim \mathcal{N}(0, \sigma^2)$

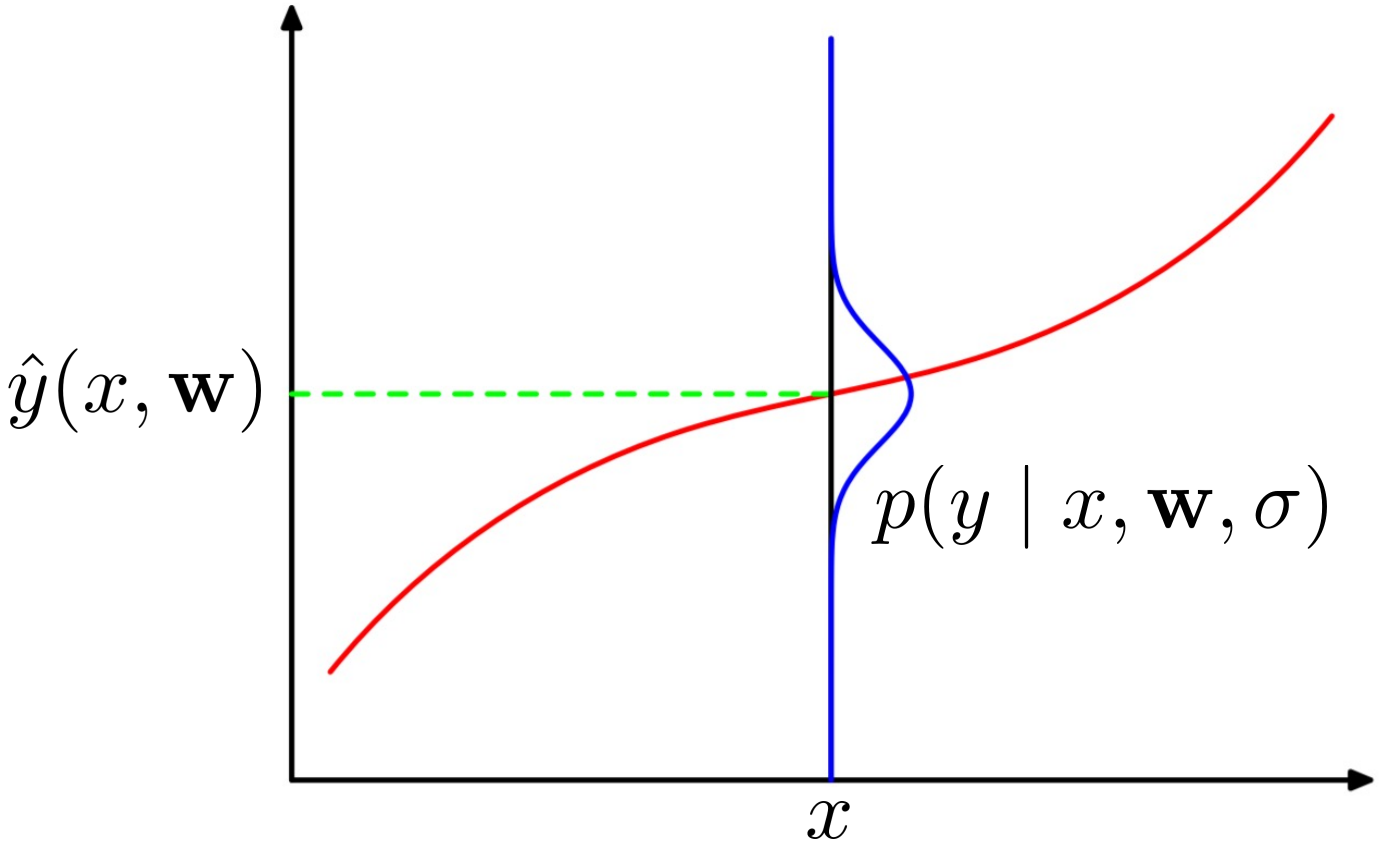
Then $p(y \mid \mathbf{x}, \mathbf{w}, \sigma) = \mathcal{N}(y \mid \hat{y}(\mathbf{x}, \mathbf{w}), \sigma^2)$.



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

Probabilistic perspective

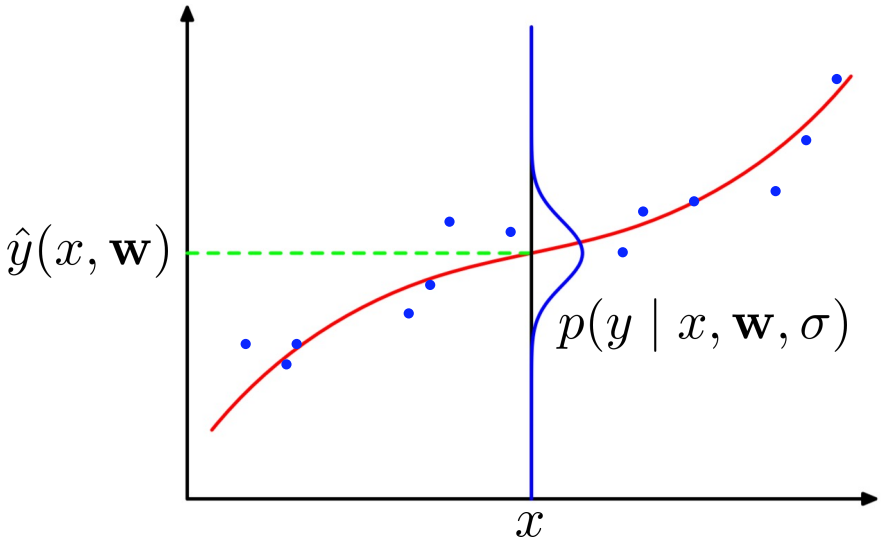
- Probabilistic motivation for *least squares:*



Probabilistic perspective

- Given data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ the probability of observing all y_i is

$$p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \sigma) = \prod_{i=1}^{N} \mathcal{N}(y_i \mid \hat{y}(\mathbf{x}_i, \mathbf{w}), \sigma^2)$$



Probabilistic perspective

- Given data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ the probability of observing all y_i is

$$p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \sigma) = \prod_{i=1}^{N} \mathcal{N}(y_i \mid \hat{y}(\mathbf{x}_i, \mathbf{w}), \sigma^2)$$

- We want to find $\mathbf{w}$ that maximizes the probability of the observed outputs

Probabilistic perspective

- When $p(\mathbf{y} \mid \mathbf{w})$ varies as a function of $\mathbf{w}$ for fixed $\mathbf{y}$, it is called a *likelihood function*
  - As function of $\mathbf{w}$, it is *not* a probability distribution (*e.g.* does not sum to 1)

- Maximizing $p(\mathbf{y} \mid \mathbf{w})$ w.r.t. $\mathbf{w}$ is called *maximum likelihood* and the resulting parameters $\mathbf{w}_{\mathrm{ML}}$ is called a *maximum likelihood estimate* of model parameters.

Maximum likelihood estimation (MLE)

Rather than maximize likelihood...

$$\mathbf{w}_{\mathrm{ML}} = \arg\max_{\mathbf{w}} \; p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \sigma)$$

...mathematically convenient to minimize negative of log of likelihood...

$$\mathbf{w}_{\mathrm{ML}} = \arg\min_{\mathbf{w}} \{ \; -\ln p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \sigma) \; \}$$

still “maximum likelihood”!

Negative log-likelihood (NLL)

$$-\ln p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \sigma) = -\sum_{i=1}^{N} \ln \mathcal{N}(y_i \mid \hat{y}(\mathbf{x}_i, \mathbf{w}), \sigma^2)$$

$$= \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \hat{y}(\mathbf{x}_i, \mathbf{w}))^2 + N \ln \sigma + \frac{N}{2} \ln 2\pi$$

scale factor doesn't affect argmin of ***w***

constant w.r.t. ***w***

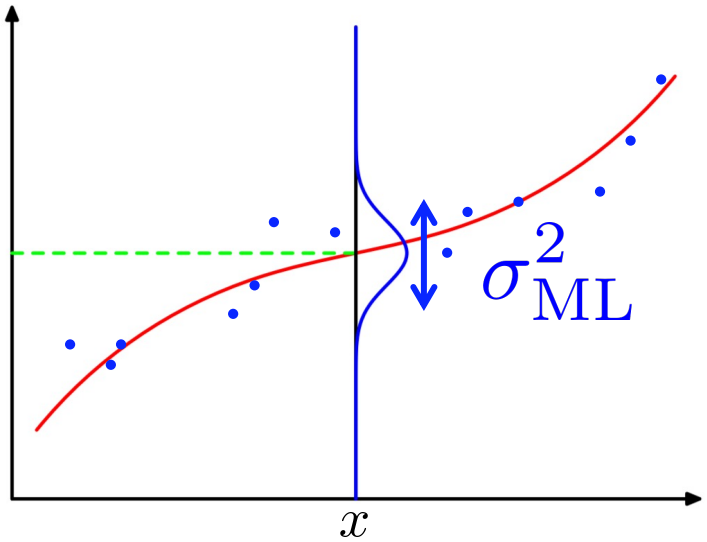
$$\mathbf{w}_{\mathrm{ML}} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} (y_i - \hat{y}(\mathbf{x}_i, \mathbf{w}))^2$$

We recovered least squares!

Maximum likelihood variance

- Can then compute maximum likelihood estimate for σ_{ML} too

$$\sigma^2_{\mathrm{ML}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}(\mathbf{x}_i, \mathbf{w}_{\mathrm{ML}}))^2$$



Maximum *a posteriori* (MAP)

- Regularization has probabilistic interpretation!
- Assume zero-mean Gaussian prior on $\mathbf{w} \in \mathbb{R}^M$

$$p(\mathbf{w} \mid \alpha) = \mathcal{N}(\mathbf{w} \mid \mathbf{0}, \tfrac{1}{\alpha}\mathbf{I}) = \tfrac{1}{Z} e^{-\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}}$$

$\frac{\alpha}{2}\|\mathbf{w}\|_2^2$

$$\text{where } \alpha > 0, Z = \left(2\pi \tfrac{1}{\alpha}\right)^{M/2}$$

- What $\mathbf{w}$ has maximum *posterior probability*?

$$p(\mathbf{w} \mid \mathbf{X}, \mathbf{y}, \sigma, \alpha) \propto p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \sigma) p(\mathbf{w} \mid \alpha)$$

posterior likelihood prior

Maximum *a posteriori* (MAP)

- Taking NLL: $-\ln\{\, p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \sigma) p(\mathbf{w} \mid \alpha) \,\}$

$$= -\ln p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, \sigma) - \ln p(\mathbf{w} \mid \alpha)$$

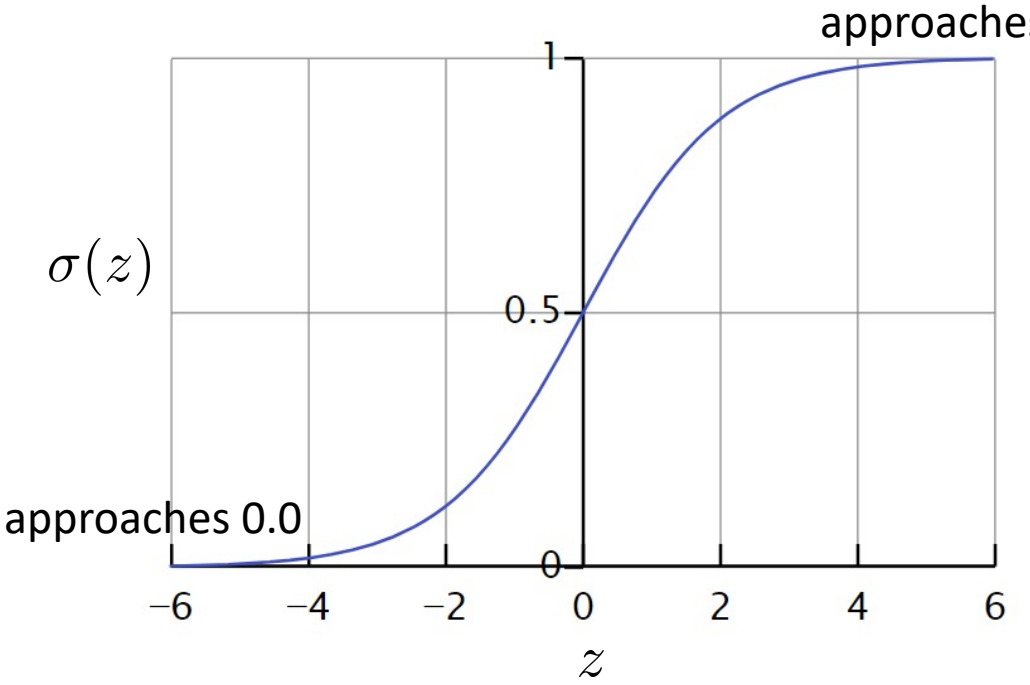
$$= \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \hat{y}(\mathbf{x}_i, \mathbf{w}))^2 + \cancel{N \ln \sigma} + \cancel{\frac{N}{2} \ln 2\pi} + \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} + \cancel{\ln Z}$$

- Gives regularized least squares for $\lambda = \alpha\sigma^2$
- MAP estimators $\mathbf{w}_{\mathrm{MAP}}, \sigma_{\mathrm{MAP}}$

Logistic regression

Logistic regression (LR)

- Often used for *binary* (2-class) *classification*, but technically still *regression*
- Based on *logistic sigmoid* function:



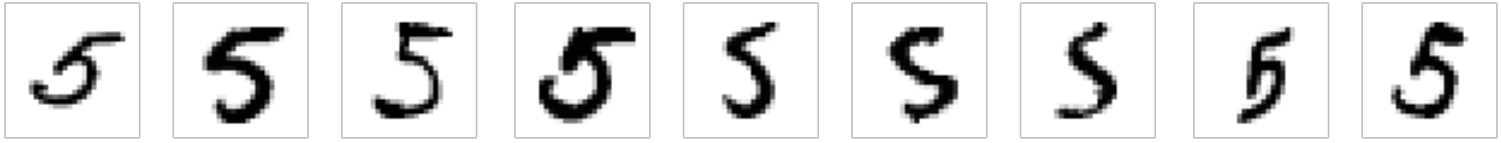
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Image credit: Qef (talk), Public Domain

Binary classification

- Given data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ where the $y_i \in \{0, 1\}$ are *class labels*
- Example: hand-written digit classification:

$\mathbf{x}_i$

$y_i = 1$ 

$y_i = 0$ 

sklearn.linear_model.LogisticRegression ¶

class `sklearn.linear_model.` **`LogisticRegression`**(*penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None*) [source]

This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **Note that regularization is applied by default**. It can handle

Parameters:

penalty : *{'l1', 'l2', 'elasticnet', 'none'}, default='l2'*

Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver. If 'none' (not supported by the liblinear solver), no regularization is applied.

"No regularization" corresponds to finding a *maximum likelihood* estimate of parameters

tol : *float, default=1e-4*

Tolerance for stopping criteria.

Need this because no direct solution (no closed form) is possible

C : *float, default=1.0*

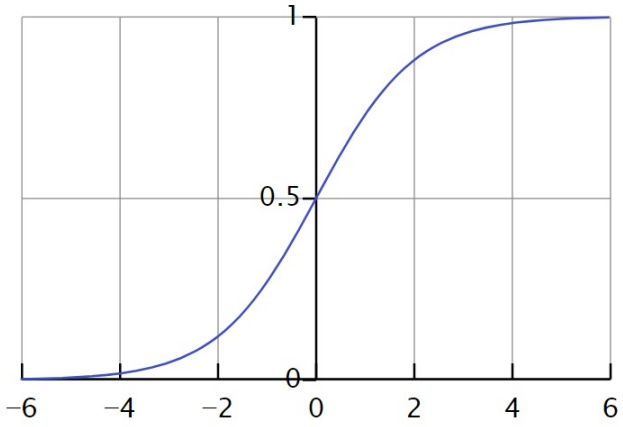
(or set this to a huge value like 10^8 to use very weak regularization)

Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

Logistic regression (LR)

- Apply sigmoid to output of linear regression machine:

$$\hat{y}(\mathbf{x}, \mathbf{w}) = \sigma(\underbrace{\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})})$$



if large positive value, predict close to 1.0
if large negative value, predict close to 0.0
if value nearly zero, predict close to 0.5

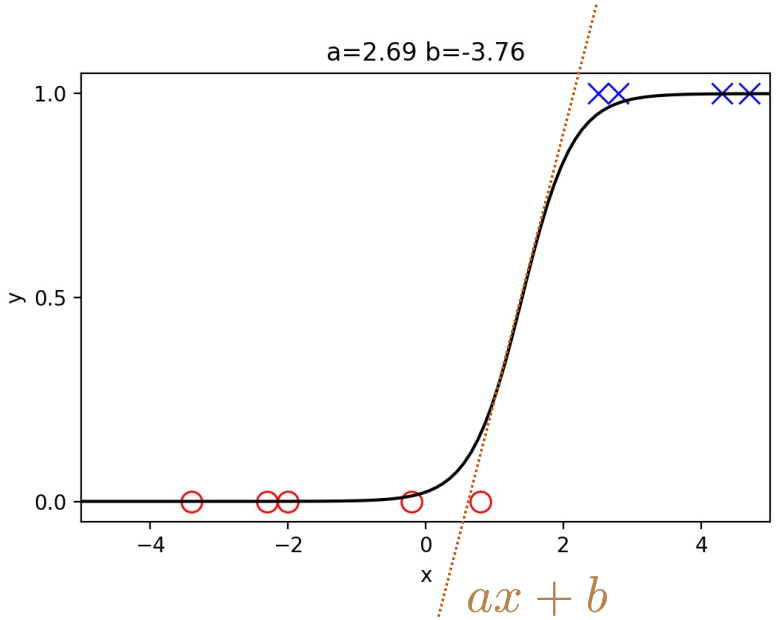
If we simply let $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$ then reduces to $\hat{y}(\mathbf{x}, \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x})$, a basic logistic model with no feature transformation $\boldsymbol{\phi}(\cdot)$

Image credit: Qef (talk), Public Domain

Logistic regression (LR)

- Probabilistic interpretation: LR models the posterior probability of class membership

$$p(y = 1 \mid \mathbf{x}, \mathbf{w}) = \sigma(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}))$$
$$p(y = 0 \mid \mathbf{x}, \mathbf{w}) = 1 - \sigma(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}))$$



$$\hat{y}(x, a, b) = \sigma(ax + b)$$

Likelihood function for LR

- Probability of observed class labels can be written as

$$p(y_i \mid \mathbf{x}_i, \mathbf{w}) = \begin{cases} \hat{y}_i & \text{if } y_i = 1 \\ 1 - \hat{y}_i & \text{if } y_i = 0 \end{cases}$$

$$\begin{aligned} \text{where } \hat{y}_i &= \hat{y}(\mathbf{x}_i, \mathbf{w}) \\ &= \sigma(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i)) \\ &= p(y = 1 \mid \mathbf{x}_i, \mathbf{w}) \end{aligned}$$

- Equivalently: $p(y_i \mid \mathbf{x}_i, \mathbf{w}) = \hat{y}_i^{y_i}(1 - \hat{y}_i)^{1-y_i}$

Likelihood function for LR

- Joint probability of *all* observed class labels can be written as

$$p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) = \prod_{i=1}^{N} \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1-y_i}$$

$$\begin{aligned} \text{where } \hat{y}_i &= \hat{y}(\mathbf{x}_i, \mathbf{w}) \\ &= \sigma(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i)) \\ &= p(y = 1 \mid \mathbf{x}_i, \mathbf{w}) \end{aligned}$$

Use true class label $y_i \in \{0, 1\}$ to contribute one of p(y=1) *or* p(y=0) term to the product. (Other term 1.0)

Maximum likelihood estimate for LR

- Sigmoid has simple derivative

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\mathrm{d}\sigma}{\mathrm{d}z}(z) = \sigma(z)(1 - \sigma(z))$$

- Many terms cancel out in NLL gradient

Maximum likelihood estimate for LR

- Negative log likelihood (NLL):

$$\ell_{\mathrm{LR}}(\mathbf{w}) = -\ln p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) = -\sum_{i=1}^{N} y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i)$$

- Gradient of NLL

$$\nabla \ell_{\mathrm{LR}}(\mathbf{w}) = \nabla_{\mathbf{w}} \{ -\ln p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) \} = \sum_{i=1}^{N} (\hat{y}_i - y_i) \boldsymbol{\phi}(\mathbf{x}_i)$$

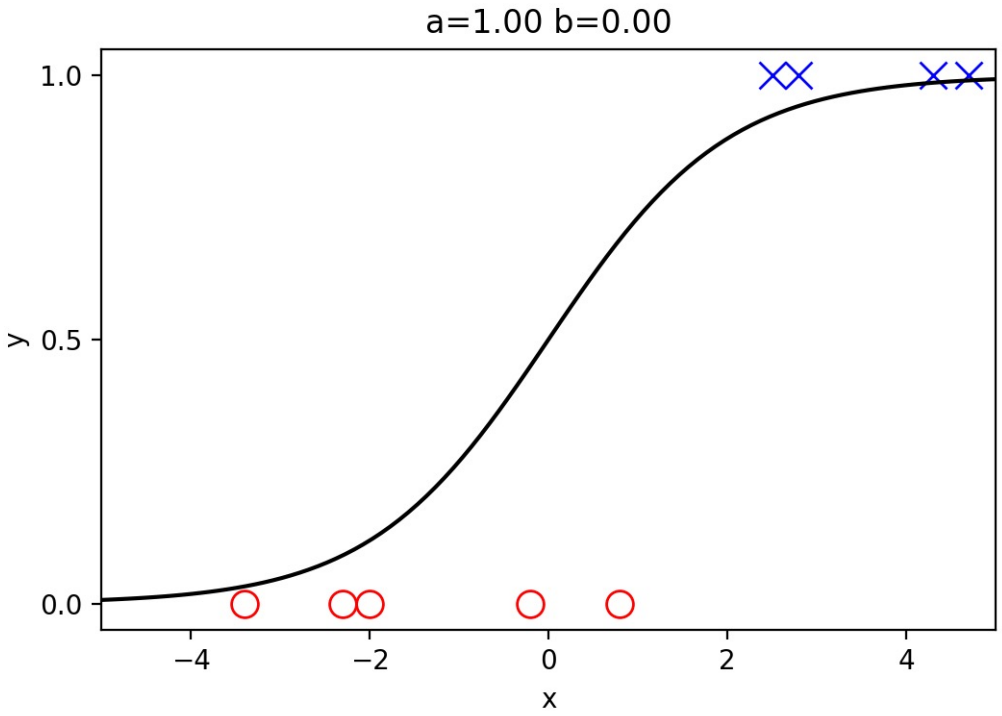
gradient takes exactly same form as for sum of squares!

$$= \sum_{i=1}^{N} (\sigma(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_i)) - y_i) \boldsymbol{\phi}(\mathbf{x}_i)$$

Basic $\nabla \ell_{\mathrm{LR}}(\mathbf{w}) = \sum_{i=1}^{N} \left(\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i\right) \mathbf{x}_i$ is just like basic $\nabla \ell_{\mathrm{LS}}(\mathbf{w}) = \sum_{i=1}^{N} \left(\mathbf{w}^T \mathbf{x}_i - y_i\right) \mathbf{x}_i$

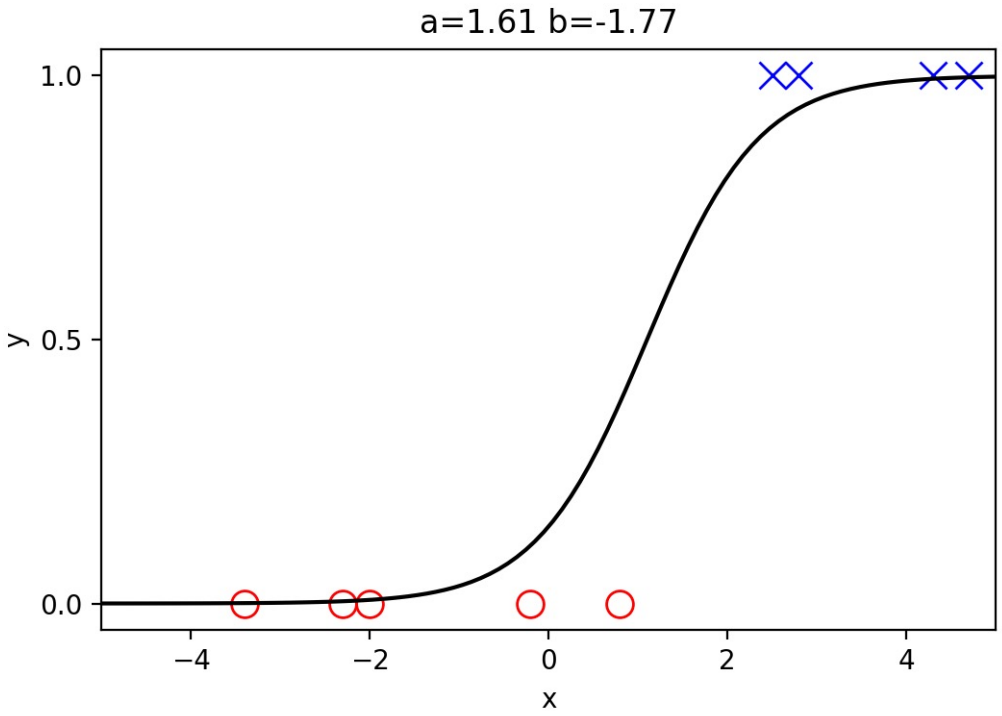
Maximum likelihood estimate for LR

- No “direct” learning algorithm for LR
- Learning algorithms are gradient-based



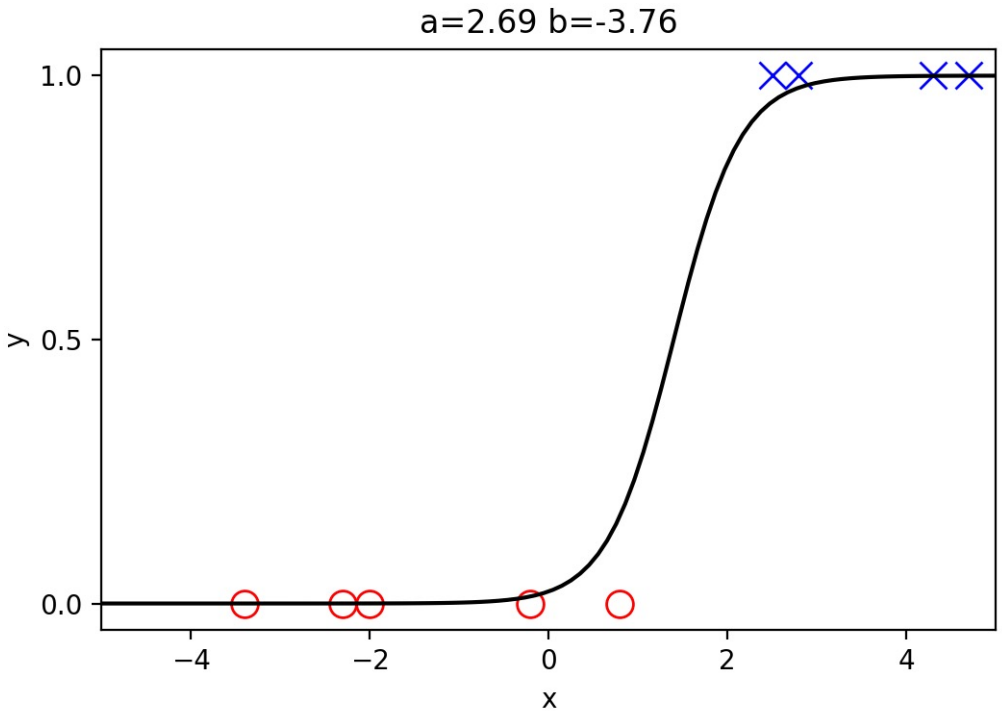
Maximum likelihood estimate for LR

- No “direct” learning algorithm for LR
- Learning algorithms are gradient-based



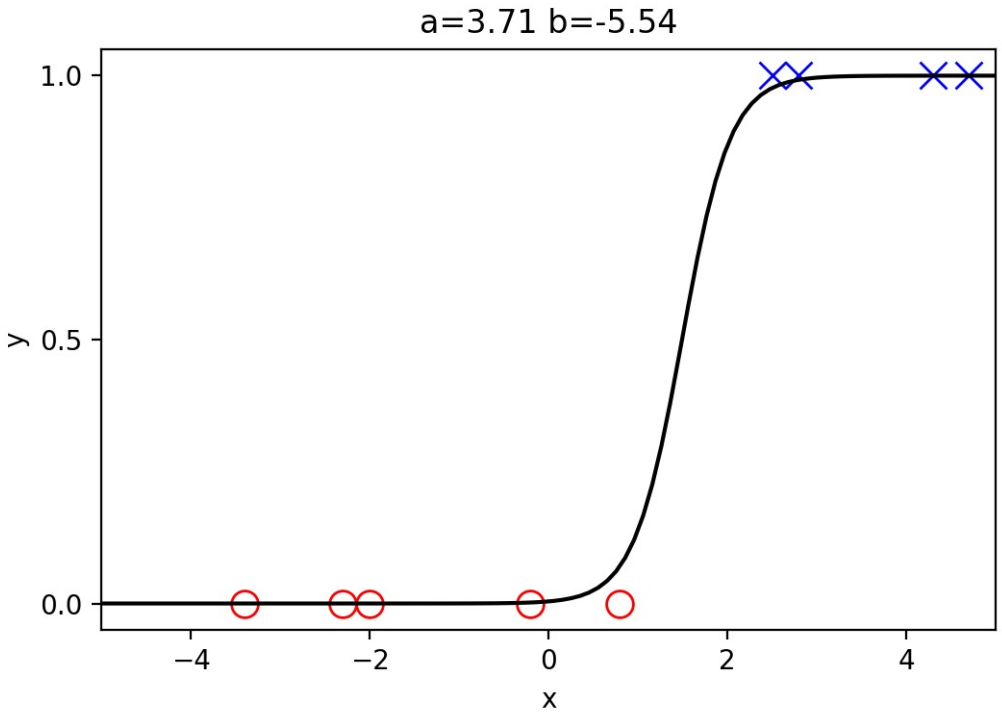
Maximum likelihood estimate for LR

- No “direct” learning algorithm for LR
- Learning algorithms are gradient-based

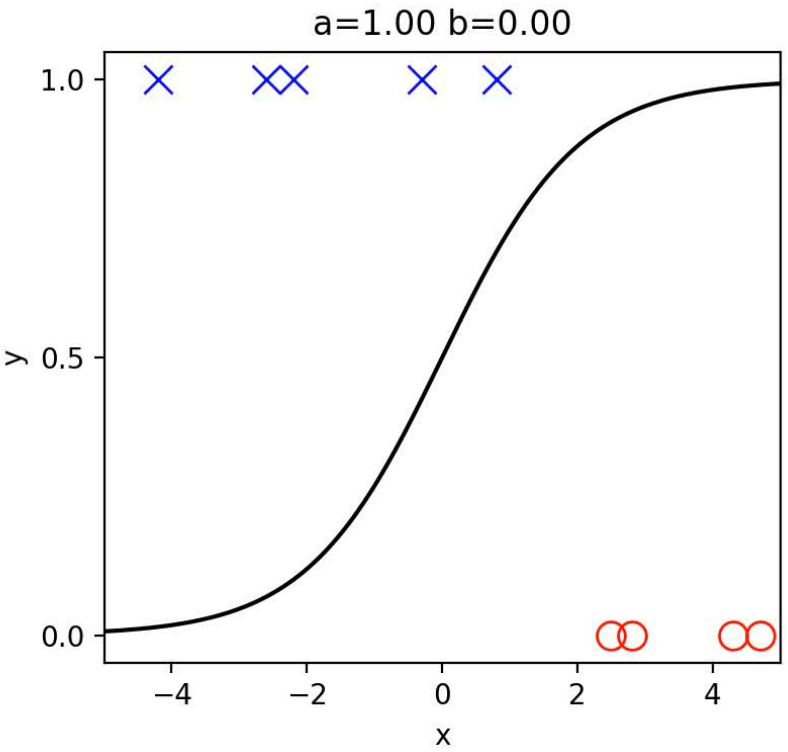


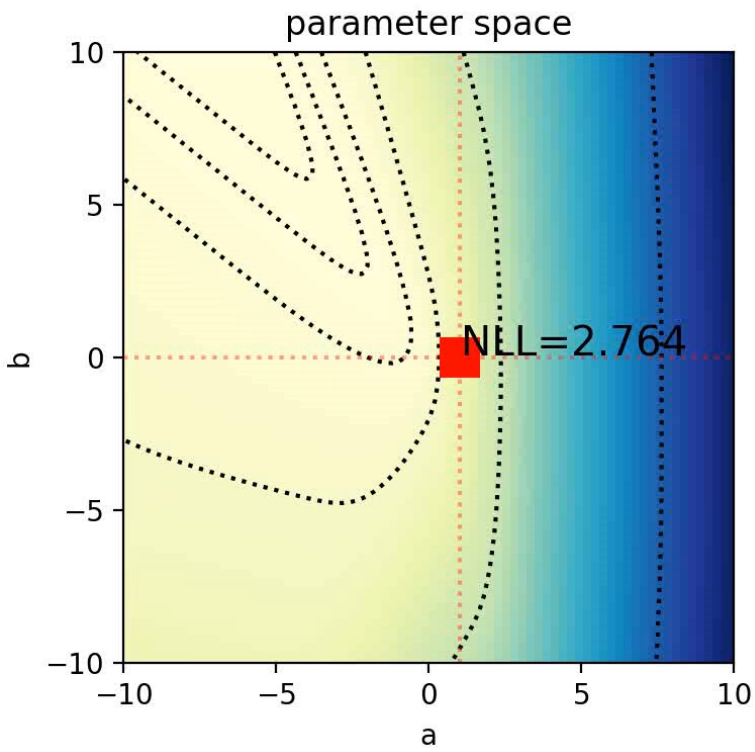
Maximum likelihood estimate for LR

- No “direct” learning algorithm for LR
- Learning algorithms take many gradient steps



Maximum likelihood estimate for LR

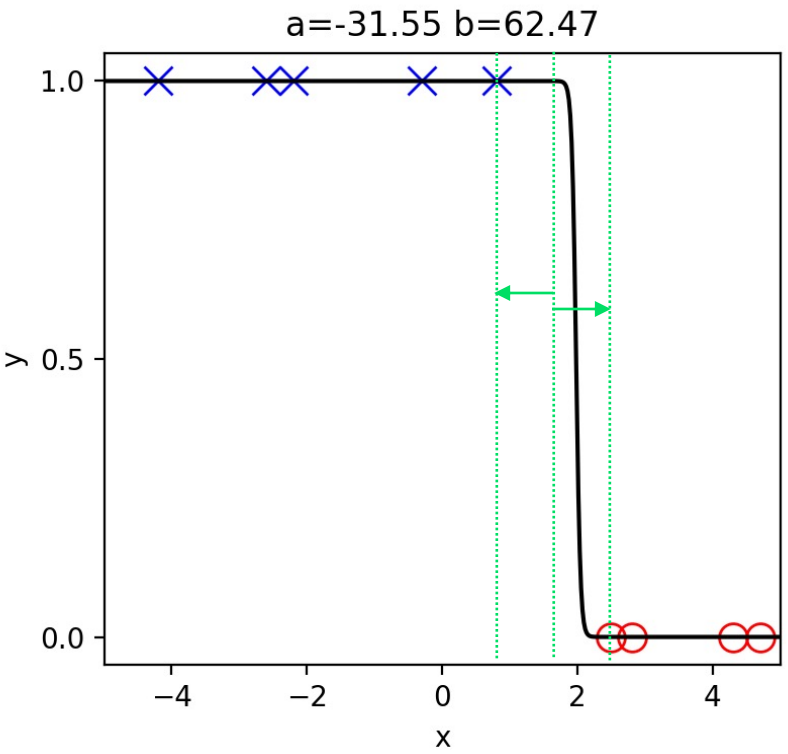


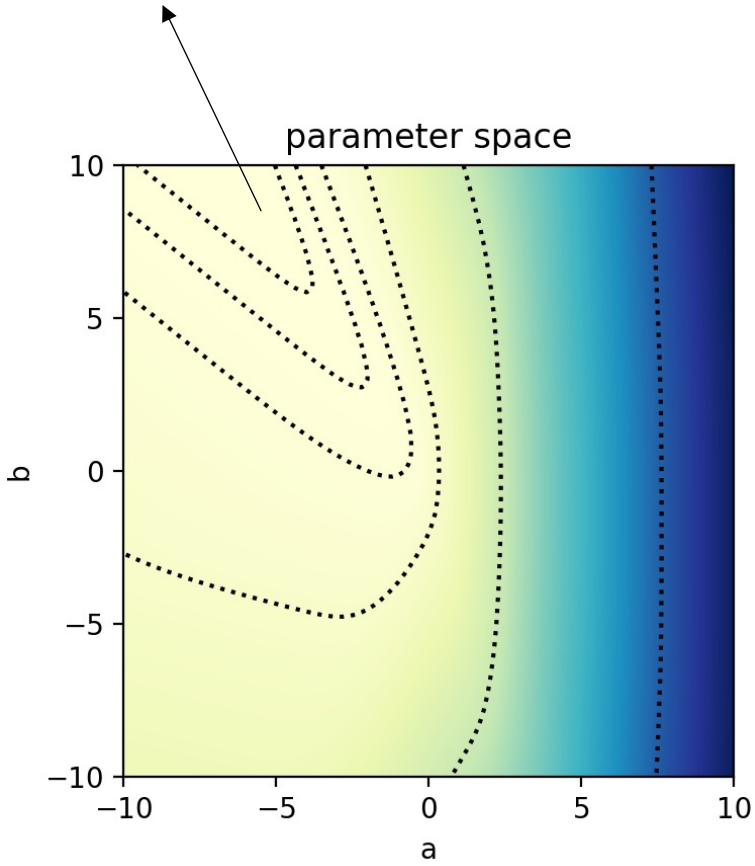


(this slide is a video)

Maximum likelihood estimate for LR

(after 1,000,000 gradient steps)

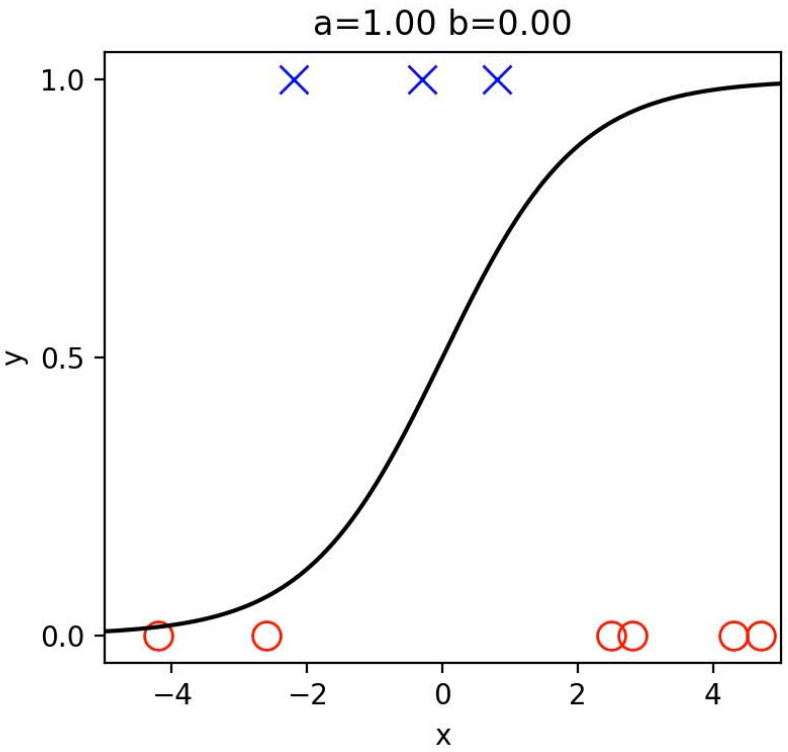


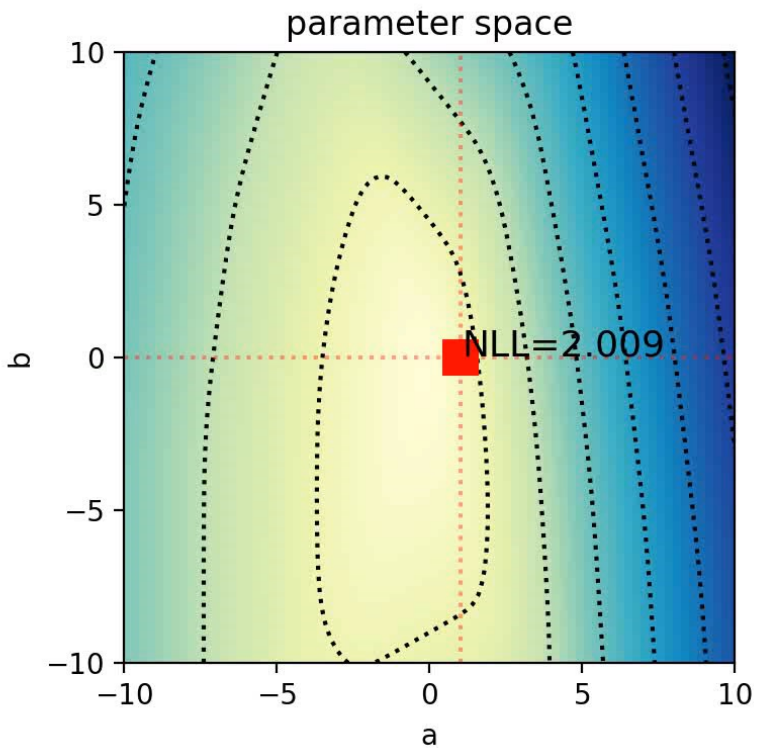


transition gets *sharper*

transition biased (happens closer to O class than to X class)

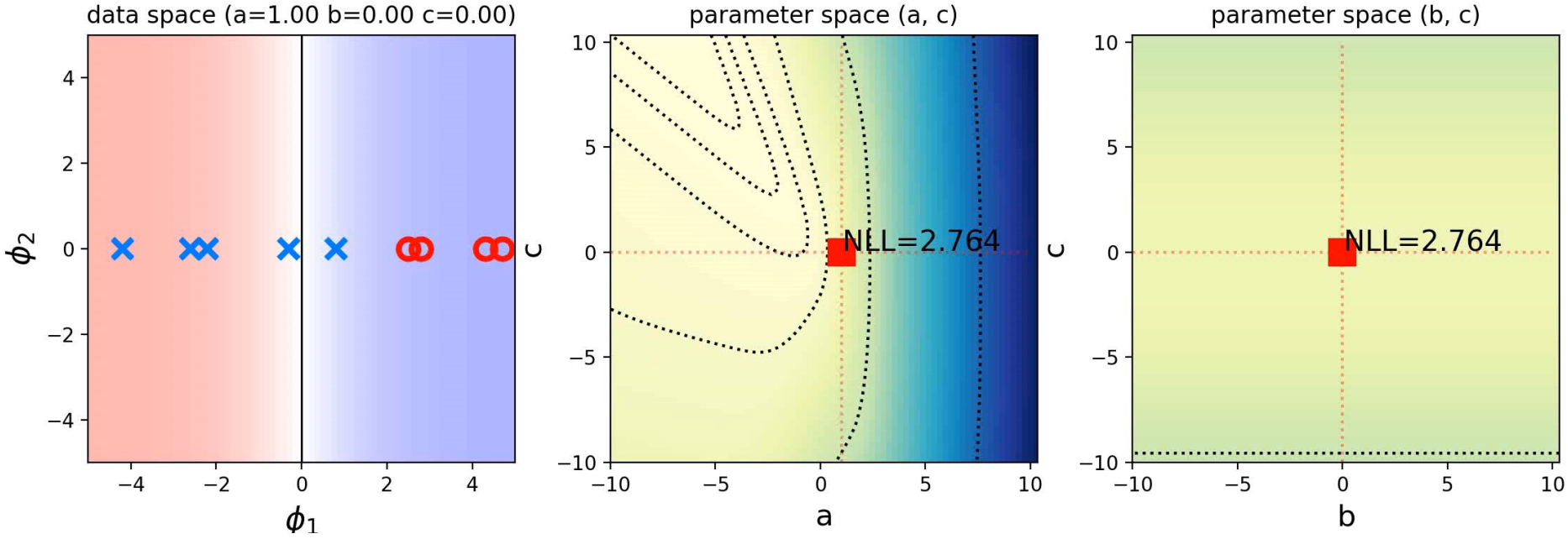
Non-separable data





(this slide is a video)

Moving to higher dimensions

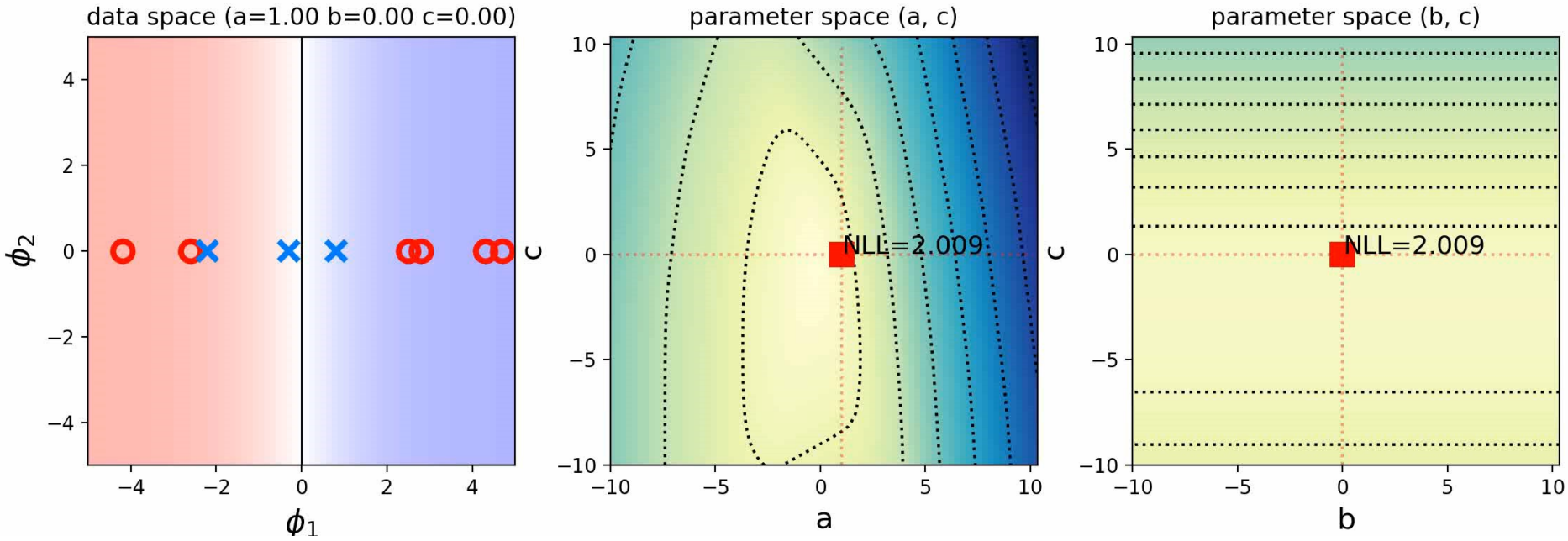


$$\boldsymbol{\phi}(x) = \begin{bmatrix} x & 0 & 1 \end{bmatrix}^T \qquad \mathbf{w} = \begin{bmatrix} a & b & c \end{bmatrix}^T$$

(separable version of the earlier data)

(this slide is a video)

Moving to higher dimensions

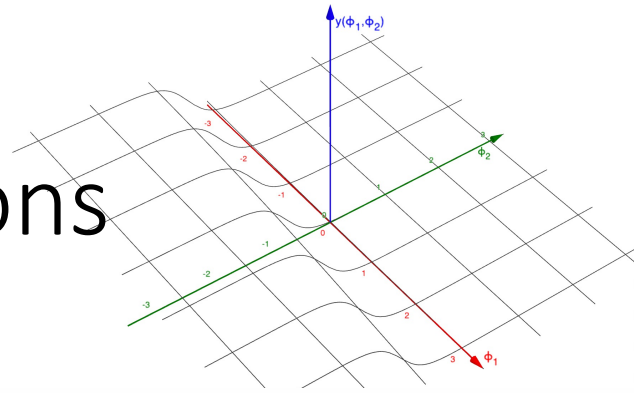


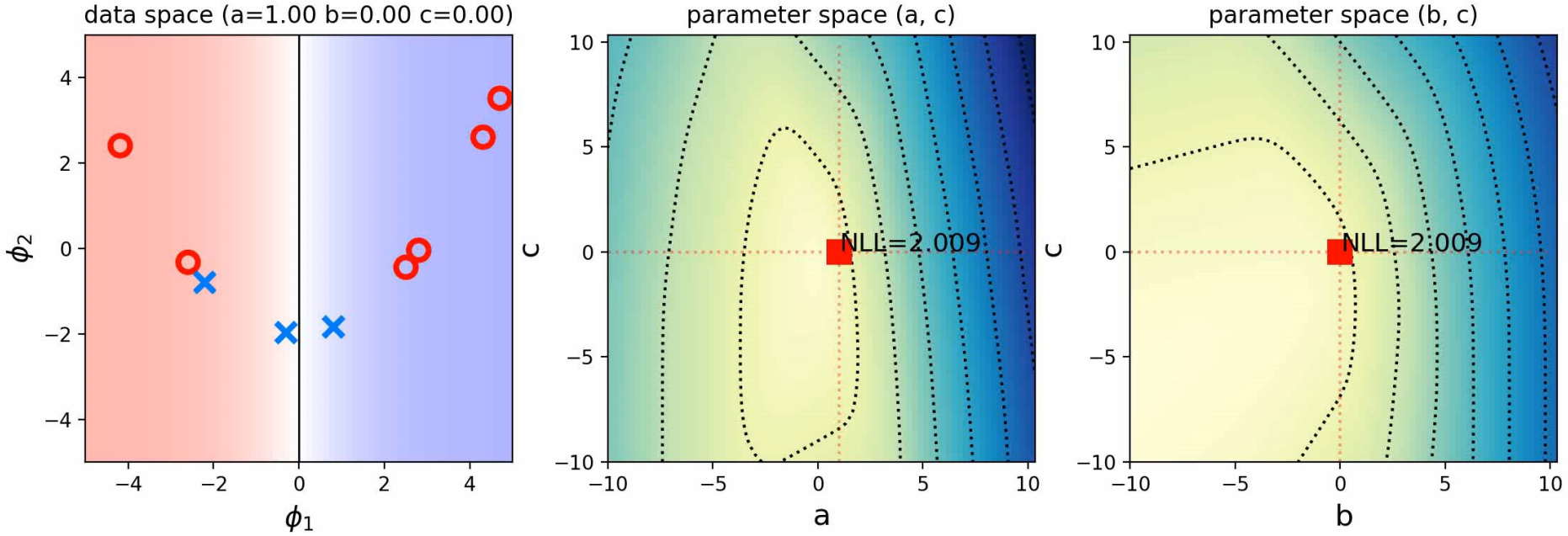
$$\boldsymbol{\phi}(x) = \begin{bmatrix} x & 0 & 1 \end{bmatrix}^T \qquad \mathbf{w} = \begin{bmatrix} a & b & c \end{bmatrix}^T$$

(non-separable version of the data)

(this slide is a video)

Moving to higher dimensions



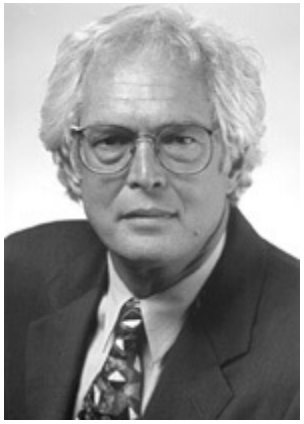


$$\boldsymbol{\phi}(x) = \begin{bmatrix} x & \frac{1}{4}x^2 - 2 & 1 \end{bmatrix}^T \quad \mathbf{w} = \begin{bmatrix} a & b & c \end{bmatrix}^T$$

(becomes separable in higher dimension)

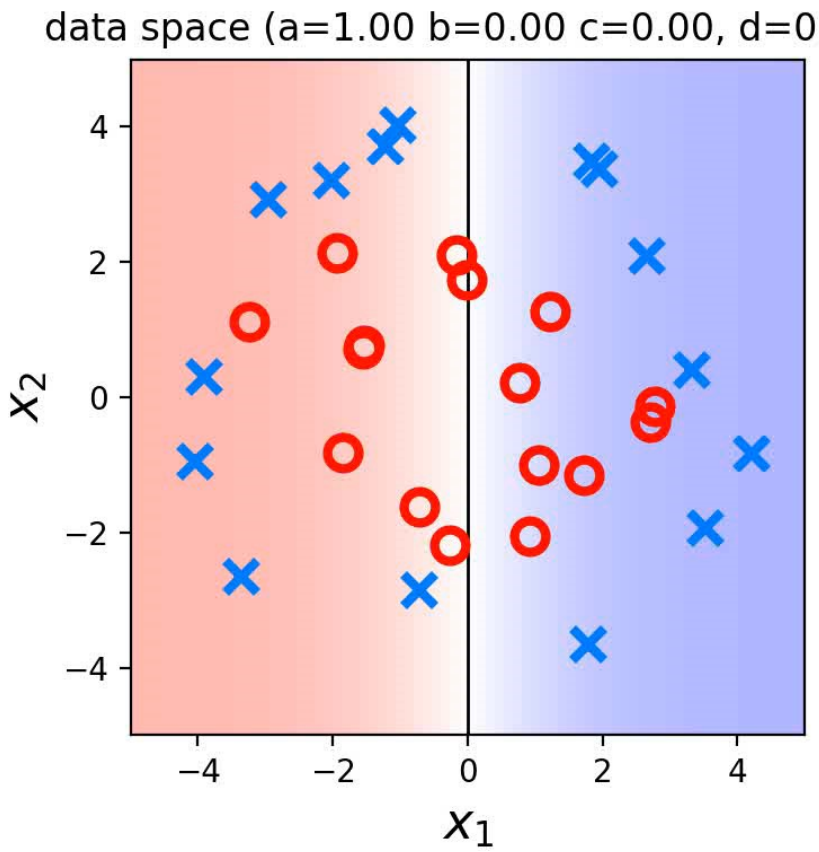
(this slide is a video)

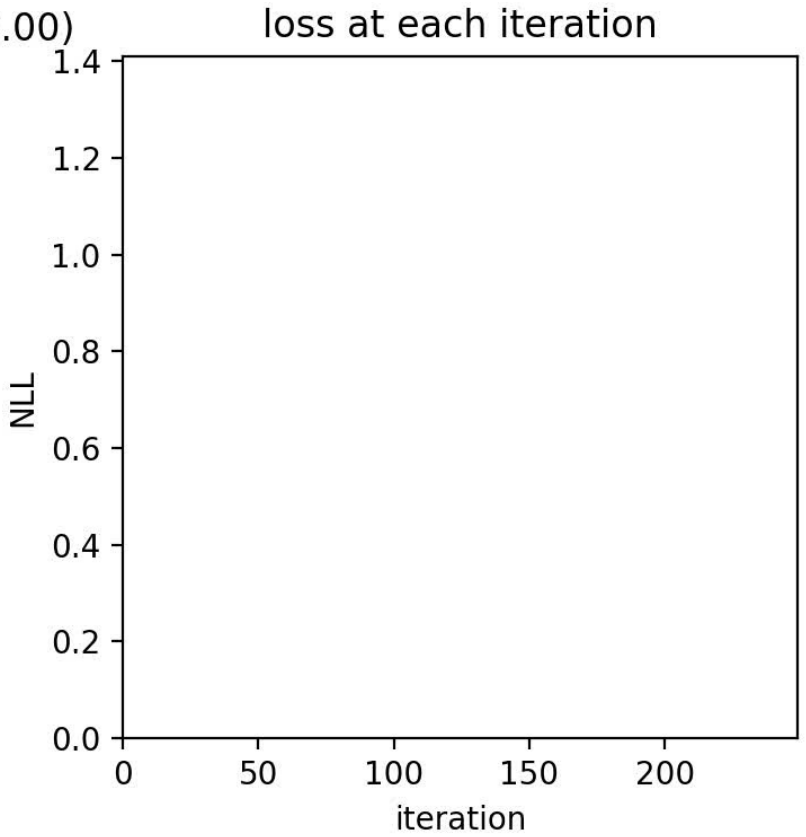
Cover's theorem (1965)



"A complex pattern-classification problem, cast in a high-dimensional space nonlinearly, is more likely to be linearly separable than in a low-dimensional space"

Moving to higher dimensions





$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} x_1 & x_2 & x_1^2 + x_2^2 & 1 \end{bmatrix}^T \qquad \mathbf{w} = \begin{bmatrix} a & b & c & d \end{bmatrix}^T$$

(this slide is a video)

Logistic regression separates classes better than basic linear regression

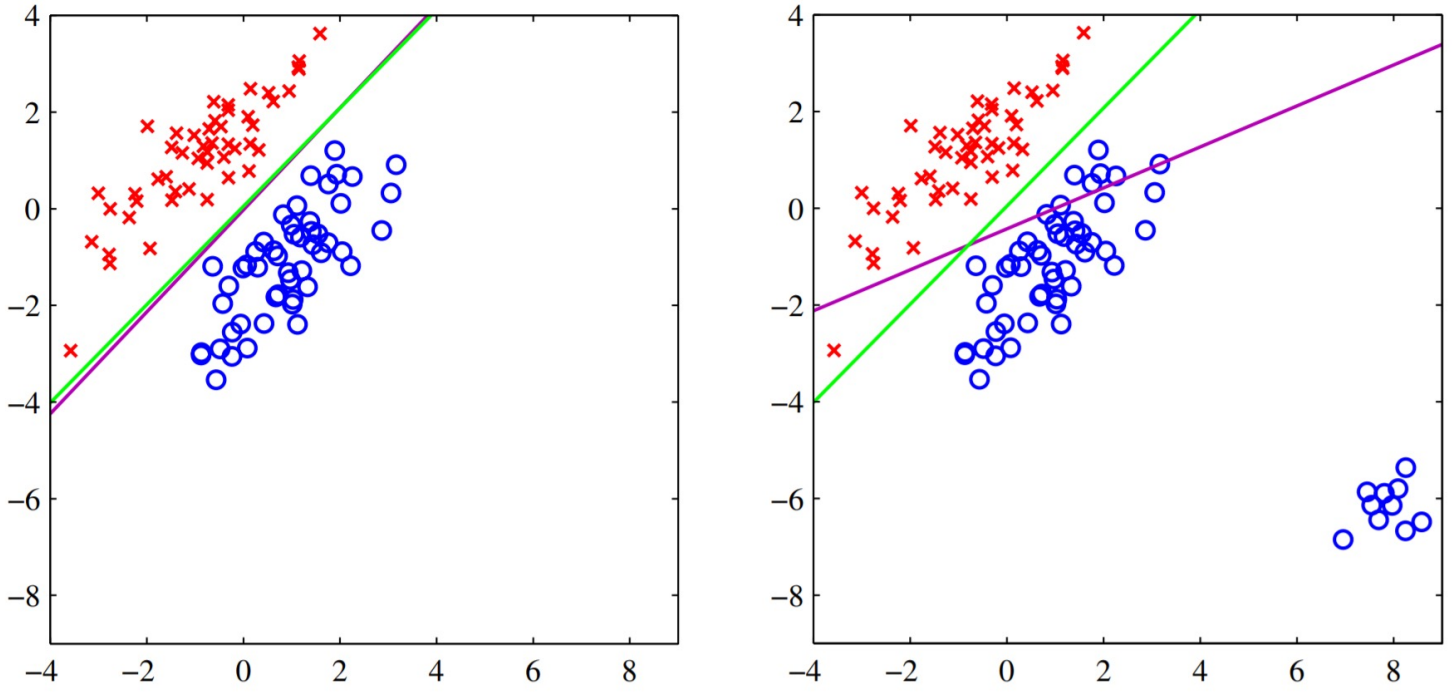
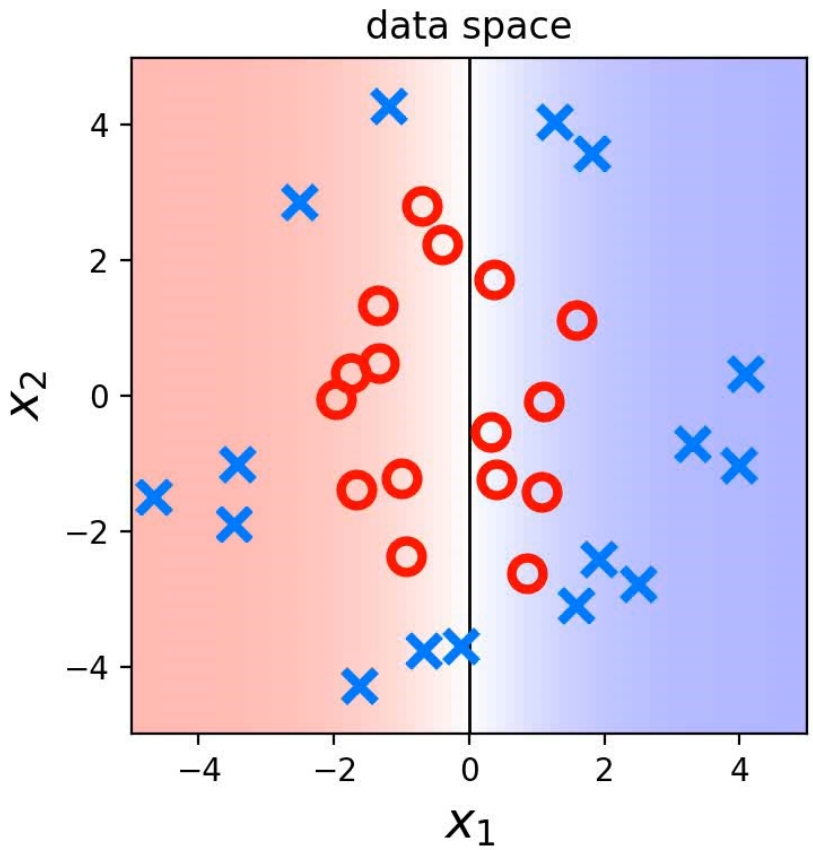
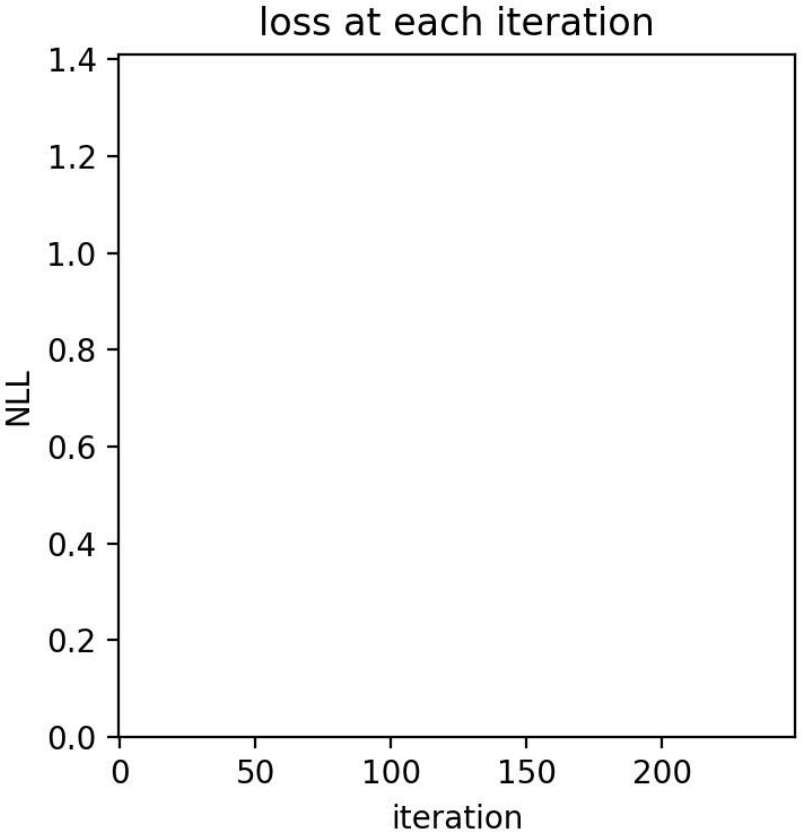


Figure 4.4 The left plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundary found by least squares (magenta curve) and also by the logistic regression model (green curve), which is discussed later in Section 4.3.2. The right-hand plot shows the corresponding results obtained when extra data points are added at the bottom left of the diagram, showing that least squares is highly sensitive to outliers, unlike logistic regression.

Polynomial basis (2nd degree)

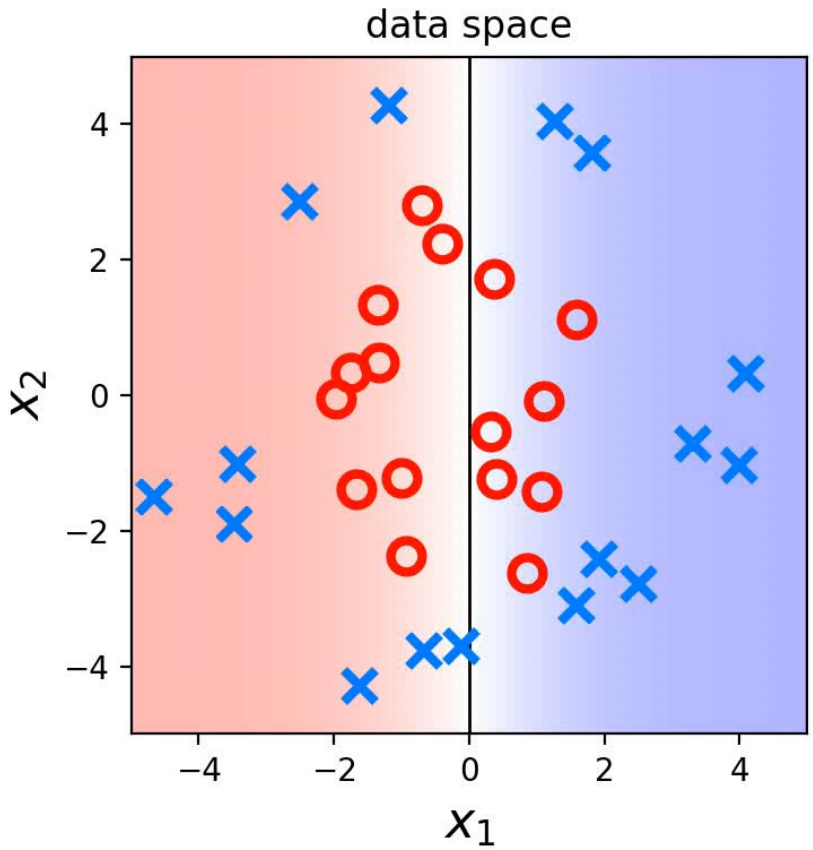


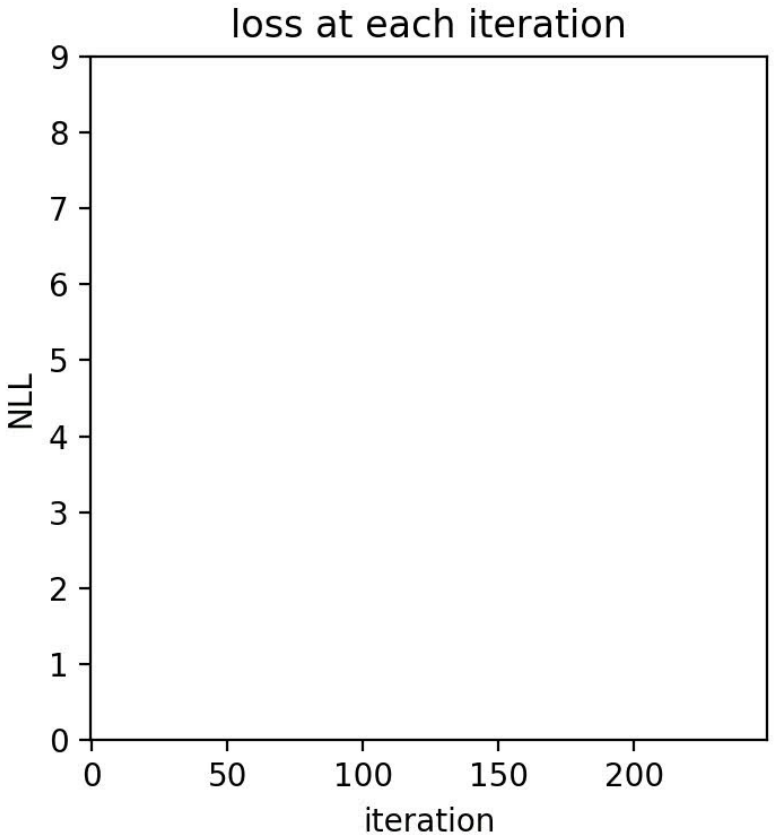


$$\phi(\mathbf{x}) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1 x_2 & x_2^2 \end{bmatrix}^T$$

(this slide is a video)

Polynomial basis (4th degree)



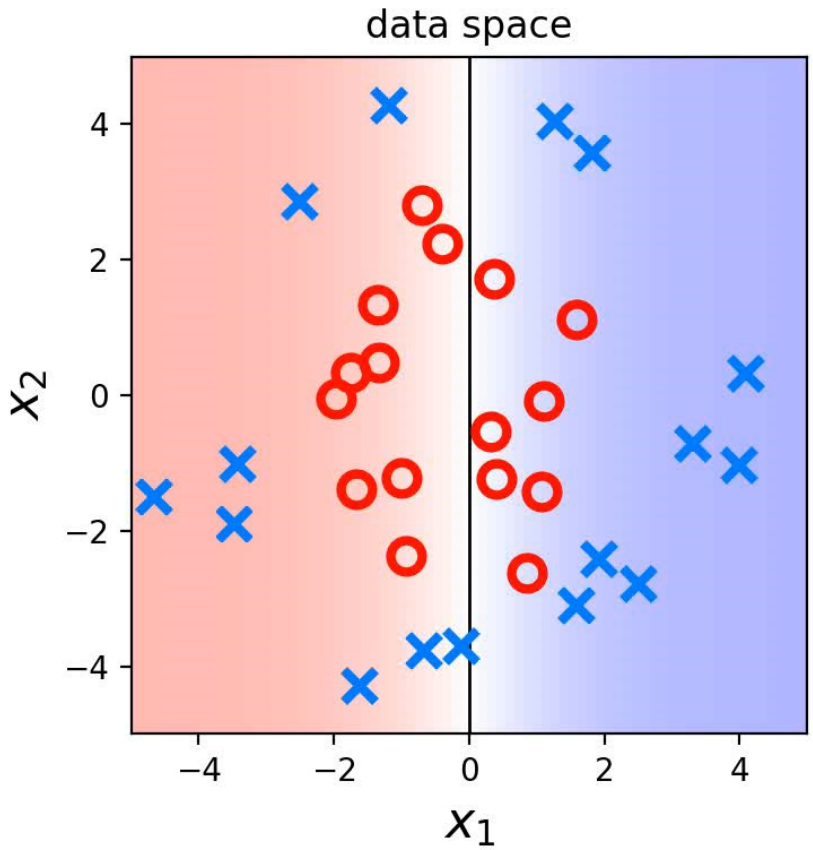


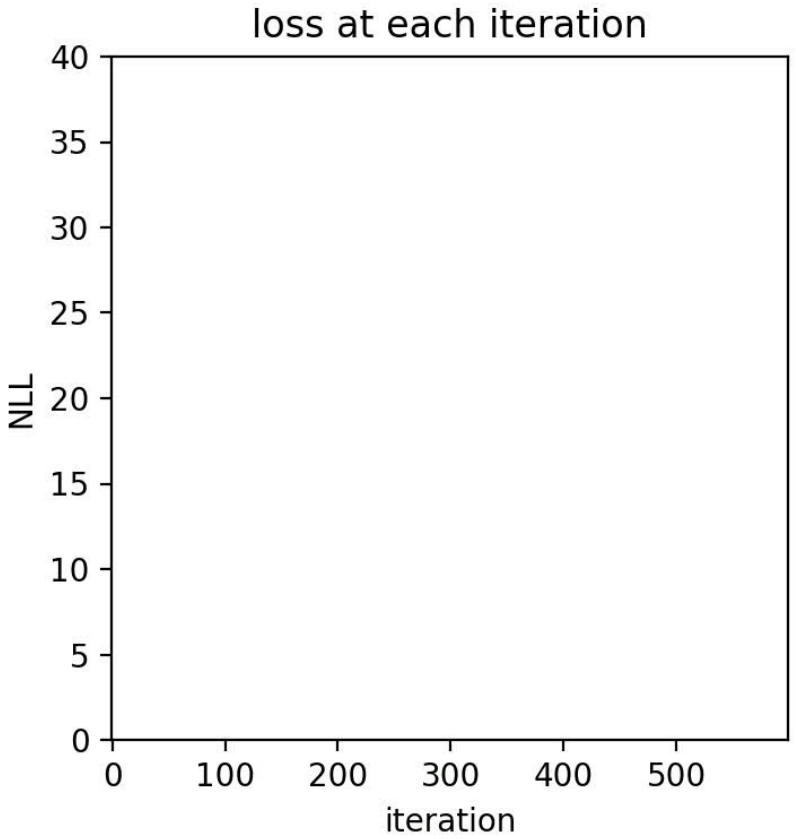
$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 & \cdots & x_1^2x_2^2 & x_1x_2^3 & x_2^4 \end{bmatrix}^T$$

(this slide is a video)

Polynomial basis (5th degree)

OVERFITTING!!





$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 & \cdots & x_1^2x_2^3 & x_1x_2^4 & x_2^5 \end{bmatrix}^T$$

(this slide is a video)

Why did gradient descent fail so badly on previous slide?

Range of $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$ values ranged from -5 to +5

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1 x_2 & x_2^2 & \cdots & x_1^2 x_2^3 & x_1 x_2^4 & x_2^5 \end{bmatrix}^T$$

That means these features can have huge values!

$$\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + \cdots + w_{18} x_1^2 x_2^3 + w_{19} x_1 x_2^4 + w_{20} x_2^5$$

Which means LR sensitive to *these* weights

$$\nabla \ell_{\mathrm{LR}}(\mathbf{w}) = \sum_{i=1}^{N} (\hat{y}_i - y_i) \boldsymbol{\phi}(\mathbf{x}_i)$$

Gradient much stronger for high-degree terms, so gradient descent focuses on tuning *them*!

Feature normalization

- **Problem:** Many learning algorithms fail when are features $\phi_i(\mathbf{x})$ and $\phi_j(\mathbf{x})$ that take values on very different scales or with very different biases
  - scales: [-0.001,0.001] vs [-1000,1000]
  - biases: [-1, 1] vs [999, 1001]
- **Idea:** Preprocess the features so that they have comparable scale and mean value across training set
- **Normalization:** Shift and scale each feature so that its mean is zero and its variance is one.

Feature normalization

- For basic model apply to $\mathbf{X}_{\text{train}}$
- For basis function model apply to $\mathbf{\Phi}_{\text{train}}$

```
X = np.array([[1, 2, 3],
              [0, 0, 2],
              [0, 0, 1]])
sklearn.preprocessing.scale(X)
```

```
array([[ 1.4142,  1.4142,  1.2247],
       [-0.7071, -0.7071,  0.    ],
       [-0.7071, -0.7071, -1.2247]])
```

Important: need to remember what these values were, so that test data can be shifted and scaled the same way that the training samples! These are effectively two extra parameters in your **model**!

```
X_normalized = (X - X.mean(axis=0)) / X.std(axis=0)
X_normalized
```

```
array([[ 1.4142,  1.4142,  1.2247],
       [-0.7071, -0.7071,  0.    ],
       [-0.7071, -0.7071, -1.2247]])
```

Feature normalization

The `preprocessing` module further provides a utility class `StandardScaler` that implements the `Transformer` API to compute the mean and standard deviation on a training set so as to be able to later reapply the same transformation on the testing set. This class is hence suitable for use in the early steps of a `sklearn.pipeline.Pipeline`:

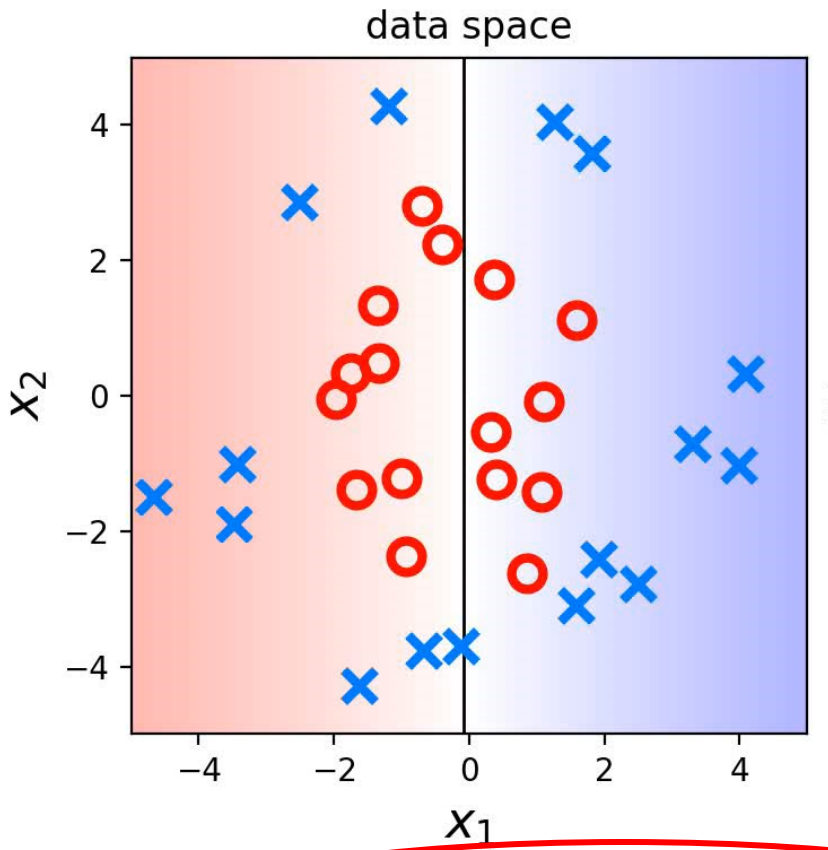
```
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> scaler
StandardScaler(copy=True, with_mean=True, with_std=True)

>>> scaler.mean_
array([1. ..., 0. ..., 0.33...])

>>> scaler.scale_
array([0.81..., 0.81..., 1.24...])

>>> scaler.transform(X_train)
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

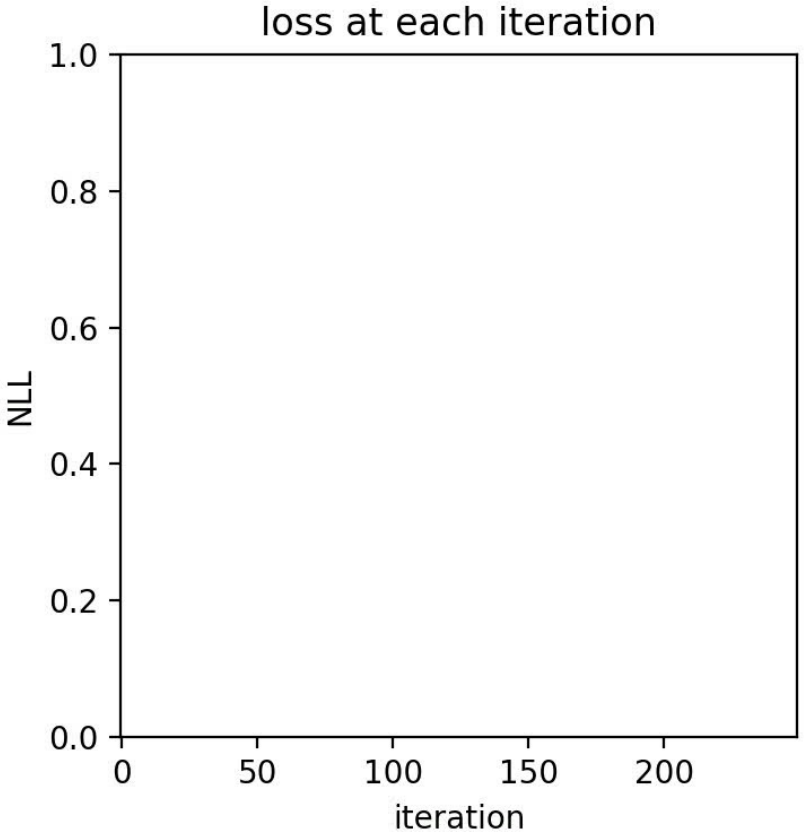
This object remembers the values so they can be applied to test data later on

The scaler instance can then be used on new data to transform it the same way it did on the training set:

```
>>> X_test = [[-1., 1., 0.]]
>>> scaler.transform(X_test)
array([[-2.44...,  1.22..., -0.26...]])
```

Feature normalization at work





$$\phi(\mathbf{x}) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 & \cdots & x_1^2x_2^3 & x_1x_2^4 & x_2^5 \end{bmatrix}^T$$

Gradient descent now pays attention to these features!

(this slide is a video)

Beyond linear models

- Almost all the ideas in this slide deck are directly applicable to non-linear models, too.
- This includes ideas of regression, classification, logistic function, basis functions & dimensionality, probabilistic justifications, feature processing, under- and over-fitting, gradient descent.
  - Do not come away thinking that these ideas are solely about “linear models”!
- The ideas were presented in the context of linear models only to make understanding them *easier*.

PRML Readings (“Bishop book”)

§1.0.0 *Introduction*

§1.1.0 *Example: Polynomial Curve Fitting*

§1.2.0 *Probability Theory*

§1.2.1 *Probability densities*

§1.2.2 *Expectations and covariances*

§1.2.3 *Bayesian probabilities*

§1.2.4 *The Gaussian distribution*

§1.2.5 *Curve fitting re-visited*

PRML Readings (“Bishop book”)

§2.1.0 *Binary Variables*

§3.1.0 *Linear Basis Function Models*

§3.1.1 *Maximum likelihood and least squares*

§3.1.2 *Geometry of least squares*

§3.1.4 *Regularized least squares*

§3.1.5 *Multiple outputs*

§3.1.2 *Geometry of least squares*

§4.3.2 *Logistic regression*