

APS1070

Foundations of Data Analytics and
Machine Learning

Summer 2020

Wed July 15 / Week 10:

- *Linear regression*



Feedback (Thank you!)

- Lectures
 - Better linkage to projects
 - Questions at the end
 - Additional reading material
- Projects
 - Clearer instructions
- Q&A sessions
 - Better utilization

Final Quiz Scheduling

- If anyone has a timing conflict, please let me know this week (by email).
- Same format as for the midterm.

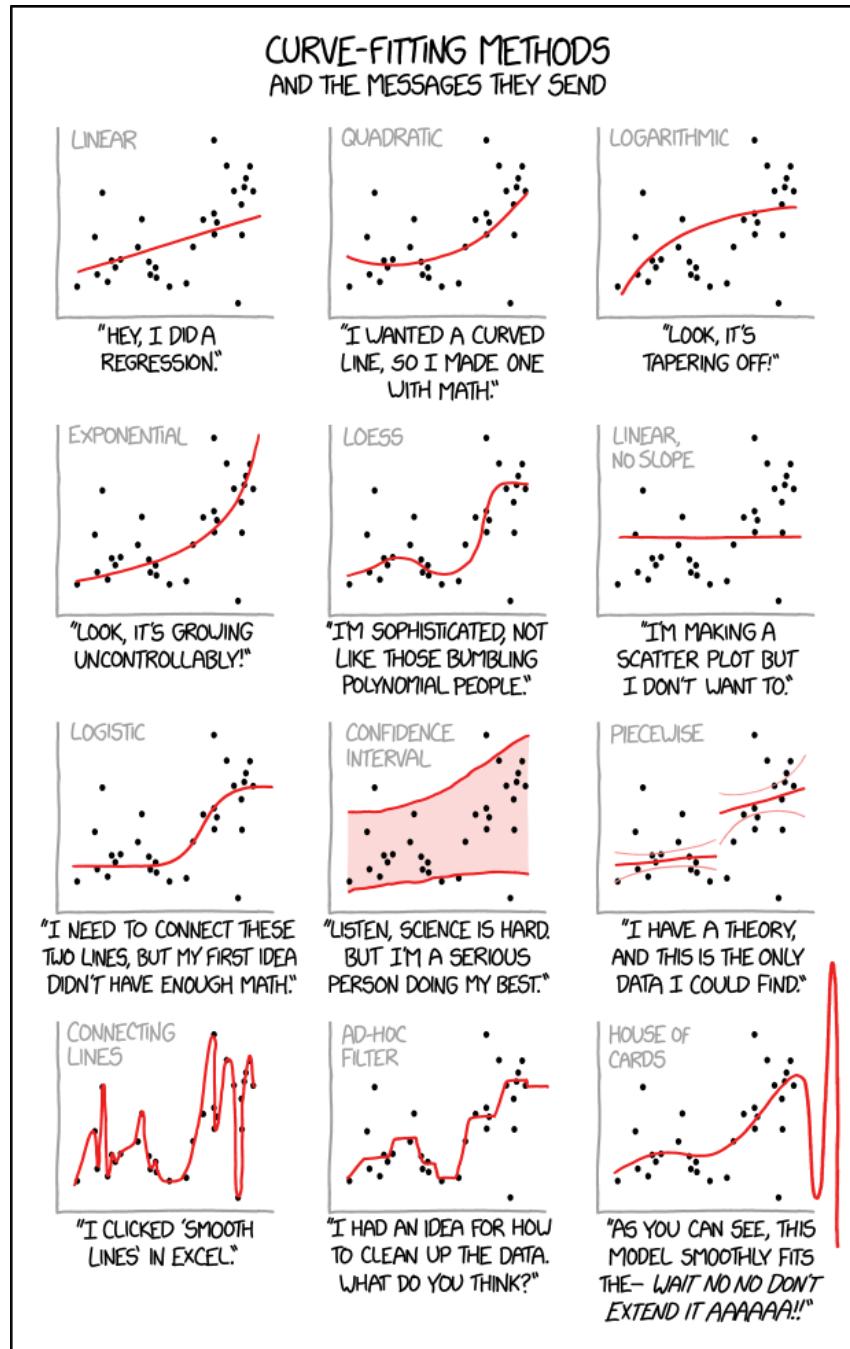
Tuesday, Aug 4	Wednesday, Aug 5
00:00	00:00
01:00	01:00
02:00	02:00
03:00	03:00
04:00	04:00
05:00	05:00
06:00	06:00
07:00	07:00
08:00	08:00
09:00	09:00
10:00	10:00
11:00	11:00
12:00	12:00
13:00 – Quiz Start!	13:00
14:00	14:00
15:00	15:00
16:00	16:00
17:00	17:00
18:00	18:00
19:00	19:00
20:00	20:00
21:00	21:00
22:00	22:00
23:00	23:00
23:59	23:59 – Quiz End!

Linear Regression

Note on slides

Certain slides presented herein are reproduced or modified from external sources -> special thanks to Roger Grosse and others this week!

Curve fitting to Model



Steps to make a prediction

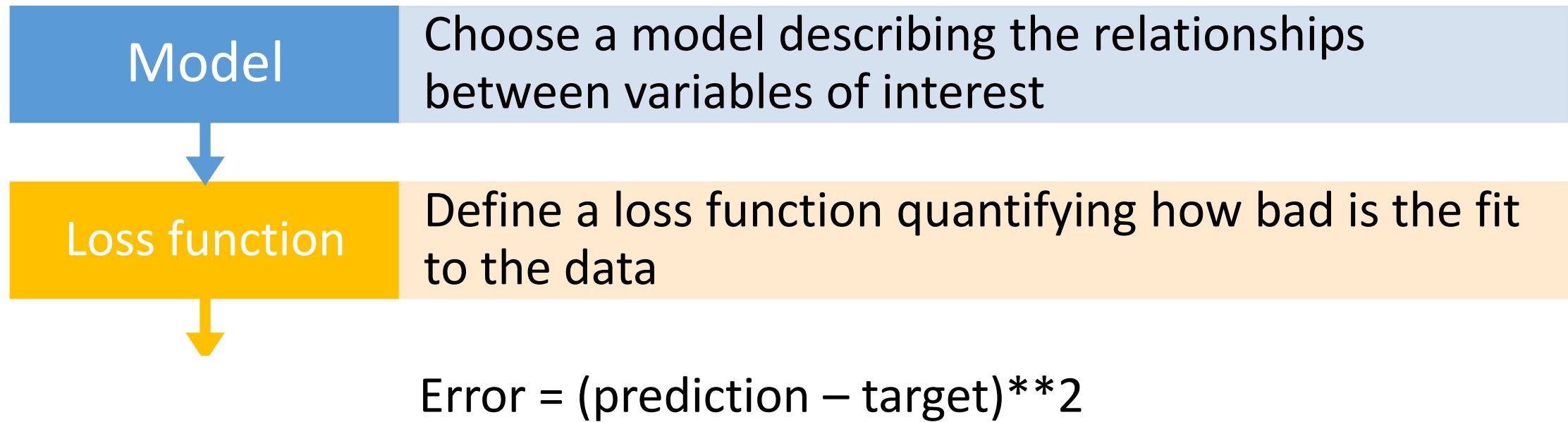
Model

Choose a model describing the relationships between variables of interest

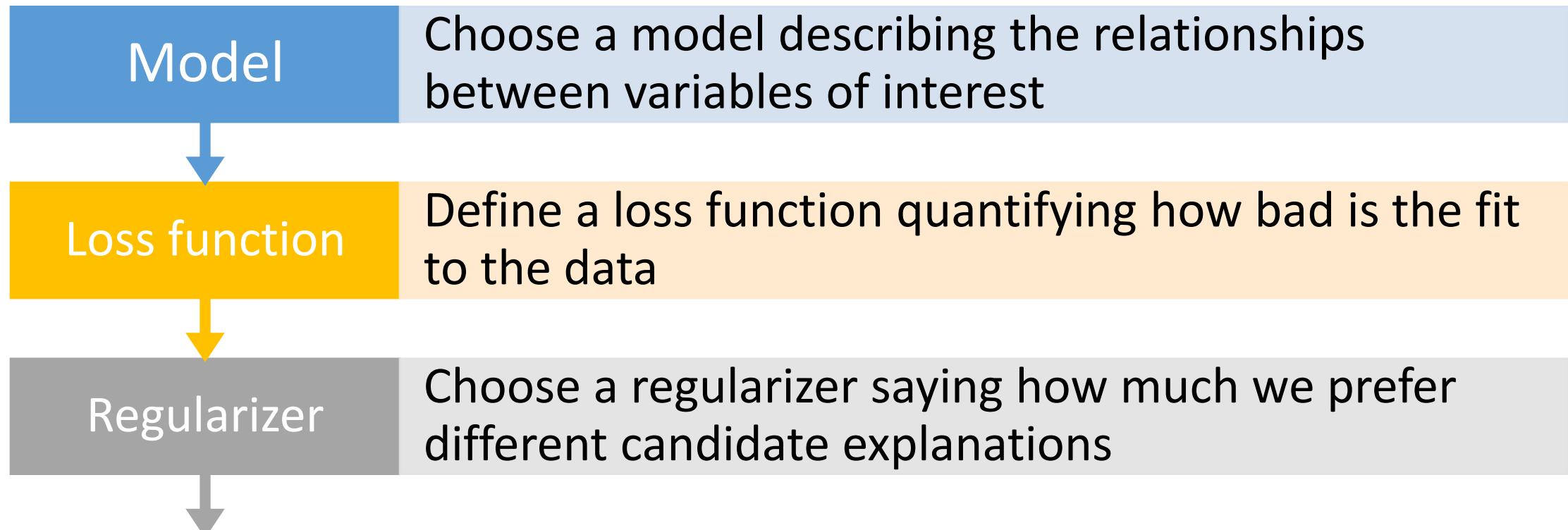


$$Y = ax + b$$

Steps to make a prediction

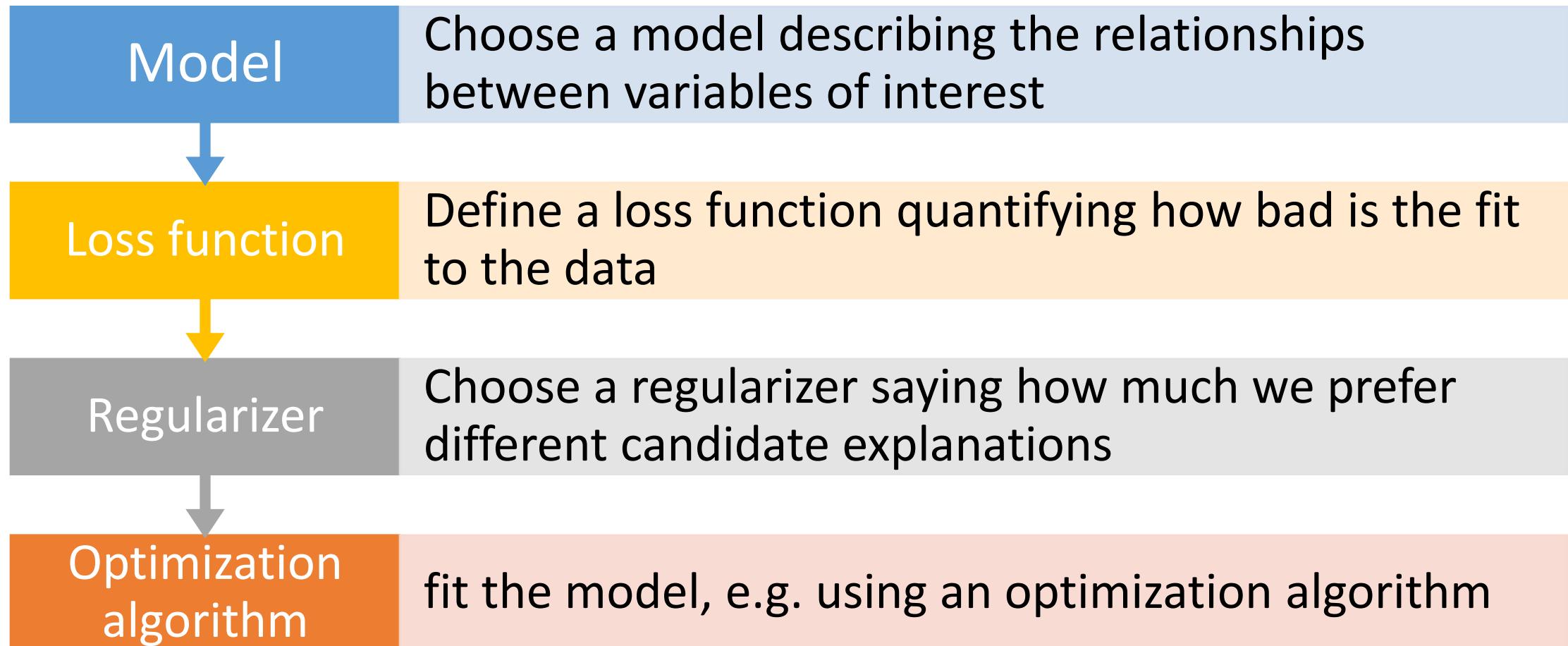


Steps to make a prediction

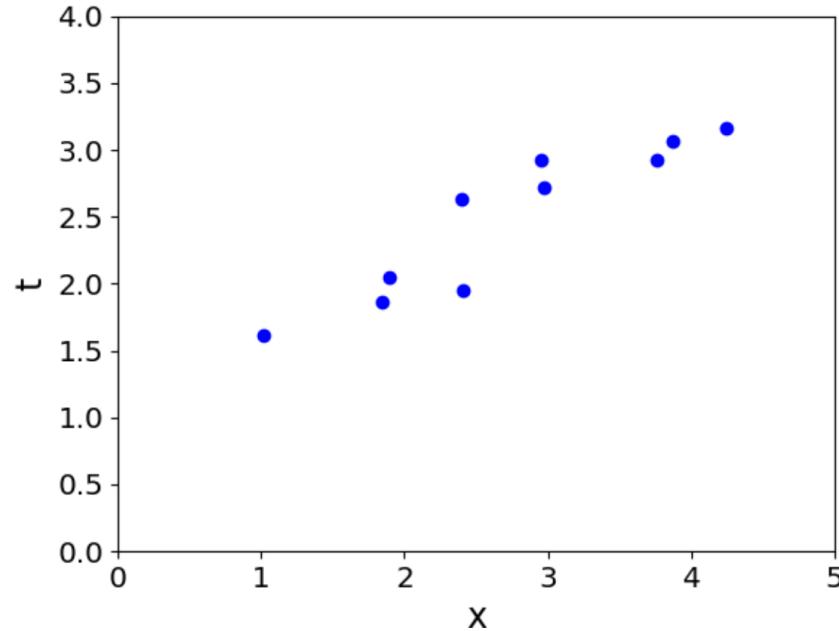


I like all the weights (coefficients) be smaller than 1

Steps to make a prediction

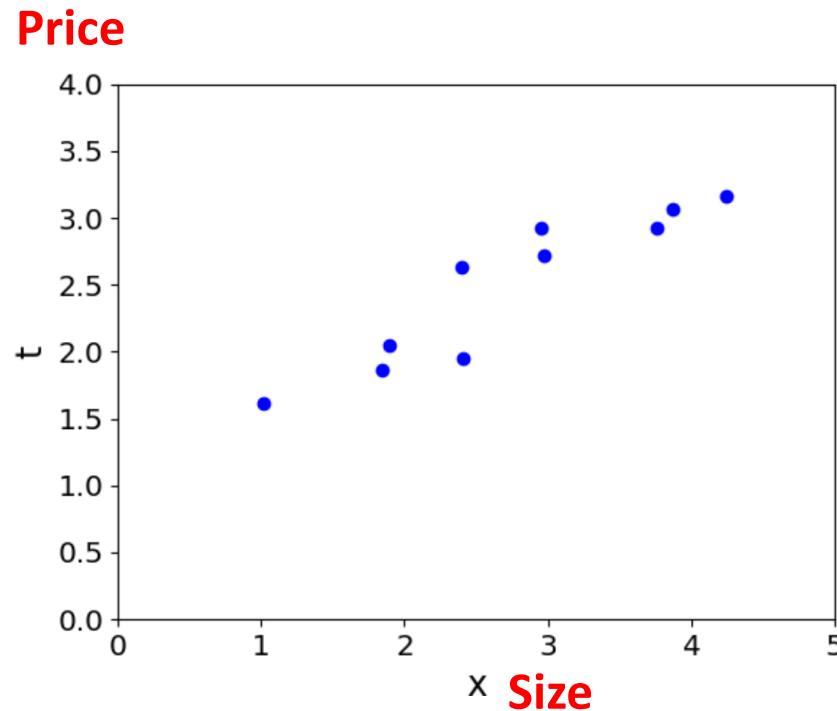


Problem setup



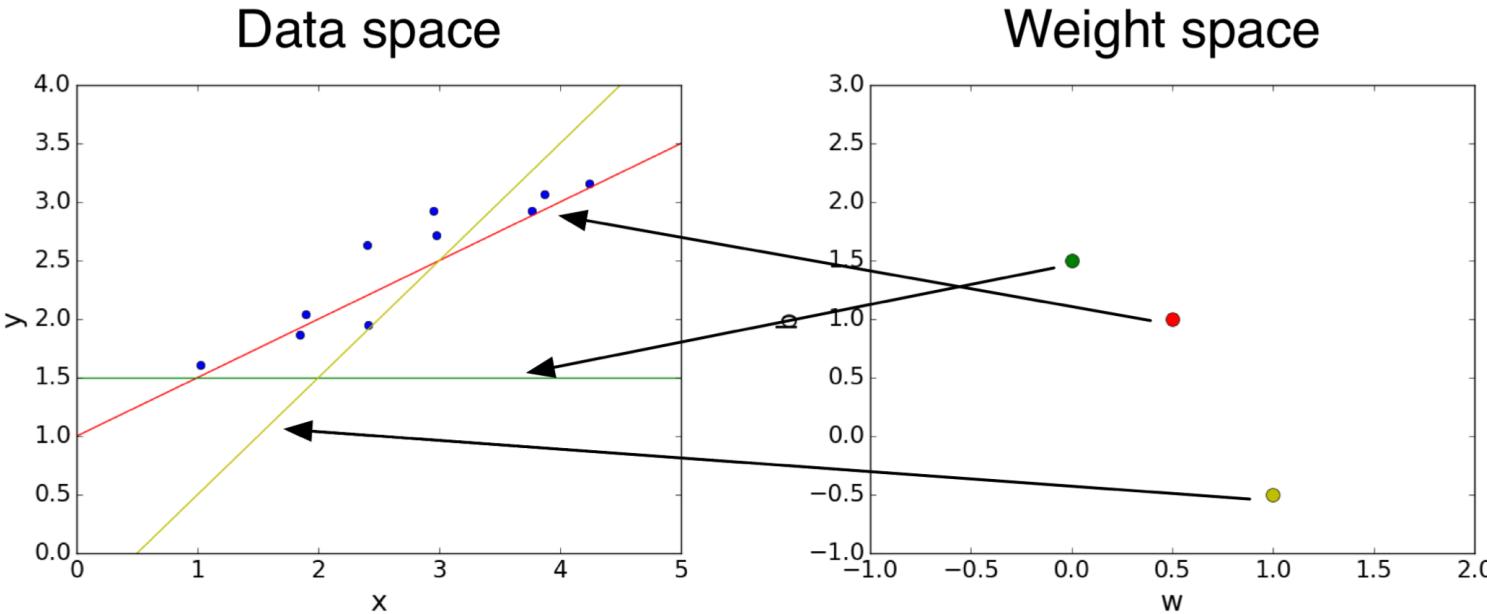
- Want to predict a scalar t as a function of a scalar x
- Given a dataset of pairs $\{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^N$
- The $\mathbf{x}^{(i)}$ are called **inputs**, and the $t^{(i)}$ are called **targets**.

Problem setup



- Want to predict a scalar t as a function of a scalar x
- Given a dataset of pairs $\{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^N$
- The $\mathbf{x}^{(i)}$ are called **inputs**, and the $t^{(i)}$ are called **targets**.

Linear Model



- **Model:** y is a linear function of x :

$$y = wx + b$$

- y is the **prediction**
- w is the **weight**
- b is the **bias**
- w and b together are the **parameters**
- Settings of the parameters are called **hypotheses**

Loss Function

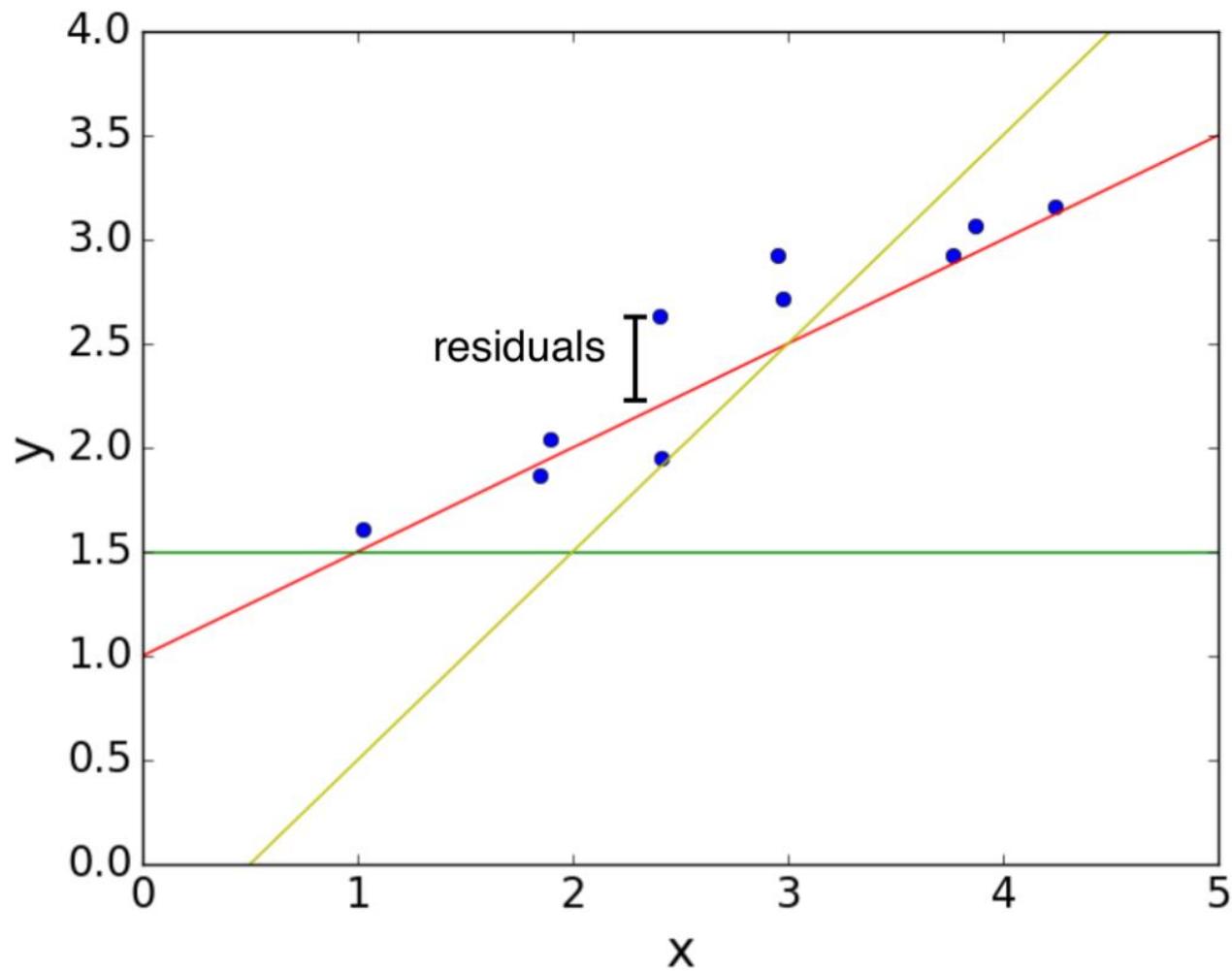
- Loss function: squared error (says how bad the fit is)

$$\mathcal{L}(y, t) = \frac{1}{2} (y - t)^2$$

predictions – targets (single example)

- $y - t$ is the residual, and we want to make this small in magnitude
- The $\frac{1}{2}$ factor is just to make the calculations convenient.
- Cost function: loss function averaged over all training examples

$$\begin{aligned}\mathcal{J}(w, b) &= \frac{1}{2N} \sum_{i=1}^N (y^{(i)} - t^{(i)})^2 \\ &= \frac{1}{2N} \sum_{i=1}^N (wx^{(i)} + b - t^{(i)})^2\end{aligned}$$



Multivariable regression

- Suppose we have multiple inputs x_1, \dots, x_D . This is referred to as **multivariable regression**.
- This is no different than the single input case, just harder to visualize.
- Linear model:

$$y = \sum_j w_j x_j + b$$

$$\begin{aligned} y &= w_1 x_1 + w_2 x_2 + w_3 x_3 + \dots + b \\ y &= (b)(1) + w_1 x_1 + w_2 x_2 + \dots \end{aligned}$$

Implementation

- Computing the prediction using a for loop:

```
y = b  
for j in range(M):  
    y += w[j] * x[j]
```

- For-loops in Python are slow, so we **vectorize** algorithms by expressing them in terms of vectors and matrices.

$$\mathbf{w} = (w_1, \dots, w_D)^\top \quad \mathbf{x} = (x_1, \dots, x_D)$$

$$y = \mathbf{w}^\top \mathbf{x} + b$$

- This is simpler and much faster:

```
y = np.dot(w, x) + b
```

Why vectorized?

Why vectorize?

- The equations, and the code, will be simpler and more readable. Gets rid of dummy variables/indices!
- Vectorized code is much faster
 - Cut down on Python interpreter overhead
 - Use highly optimized linear algebra libraries
 - Matrix multiplication is very fast on a Graphics Processing Unit (GPU)

Design Matrix

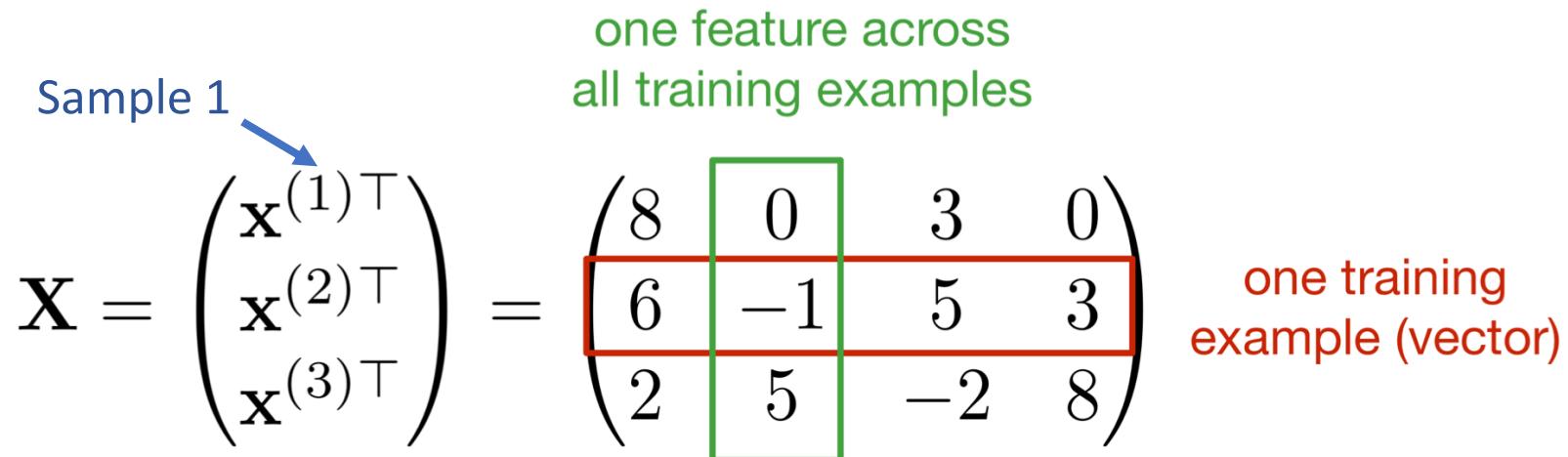
- We can take this a step further. Organize all the training examples into the **design matrix** \mathbf{X} with one row per training example, and all the targets into the **target vector** \mathbf{t} .

Sample 1

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{pmatrix} 8 & 0 & 3 & 0 \\ 6 & -1 & 5 & 3 \\ 2 & 5 & -2 & 8 \end{pmatrix}$$

one feature across
all training examples

one training
example (vector)



- Computing the predictions for the whole dataset:

$$\mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} \mathbf{w}^\top \mathbf{x}^{(1)} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{(N)} + b \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix} = \mathbf{y}$$

LOSS

- Computing the squared error cost across the whole dataset:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$$

$$\mathcal{J} = \frac{1}{2N} \|\mathbf{y} - \mathbf{t}\|^2$$

- In Python:

```
y = np.dot(X, w) + b
cost = np.sum((y - t) ** 2) / (2. * N)
```

Direct Solution

- We defined a cost function. This is what we'd like to minimize.
- Recall from calculus class: minimum of a smooth function (if it exists) occurs at a **critical point**, i.e. point where the derivative is zero.
- Multivariate generalization: set the partial derivatives to zero. We call this **direct solution**.

- **Partial derivatives:** derivatives of a multivariate function with respect to one of its arguments.

$$\frac{\partial}{\partial x_1} f(x_1, x_2) = \lim_{h \rightarrow 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h}$$

- To compute, take the single variable derivatives, pretending the other arguments are constant.
- Example: partial derivatives of the prediction y

$$y = \sum_j w_j x_j + b$$

$$\begin{aligned}\frac{\partial y}{\partial w_j} &= \frac{\partial}{\partial w_j} \left[\sum_{j'} w_{j'} x_{j'} + b \right] \\ &= x_j \\ \frac{\partial y}{\partial b} &= \frac{\partial}{\partial b} \left[\sum_{j'} w_{j'} x_{j'} + b \right] \\ &= 1\end{aligned}$$

- Chain rule for derivatives:

$$\begin{aligned}
 \frac{\partial \mathcal{L}}{\partial w_j} &= \frac{d\mathcal{L}}{dy} \frac{\partial y}{\partial w_j} \\
 &= \frac{d}{dy} \left[\frac{1}{2}(y - t)^2 \right] \cdot x_j \\
 &= (y - t)x_j \\
 \mathcal{L}(y, t) &= \frac{1}{2}(y - t)^2 \\
 y &= \sum_j w_j x_j + b \\
 \frac{\partial \mathcal{L}}{\partial b} &= y - t
 \end{aligned}$$

- Cost derivatives (average over data points):

$$\begin{aligned}
 \frac{\partial \mathcal{J}}{\partial w_j} &= \frac{1}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)} \\
 \frac{\partial \mathcal{J}}{\partial b} &= \frac{1}{N} \sum_{i=1}^N y^{(i)} - t^{(i)}
 \end{aligned}$$

Optimal weights: Direct Solution

- The minimum must occur at a point where the partial derivatives are zero.

$$\frac{\partial \mathcal{J}}{\partial w_j} = 0 \quad \frac{\partial \mathcal{J}}{\partial b} = 0.$$

- If $\partial \mathcal{J}/\partial w_j \neq 0$, you could reduce the cost by changing w_j .
- This turns out to give a system of linear equations, which we can solve efficiently.
- Optimal weights:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$$

- Linear regression is one of only a handful of models in this course that permit direct solution.

Optimal weights: Direct Solution

- The minimum must occur at a point where the partial derivatives are zero.

$$\frac{\partial \mathcal{J}}{\partial w_j} = 0 \quad \frac{\partial \mathcal{J}}{\partial b} = 0.$$

- If $\partial \mathcal{J} / \partial w_j \neq 0$, you could reduce the cost by changing w_j .
- This turns out to give a system of linear equations, which can be solved efficiently.
- Optimal weights:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$$

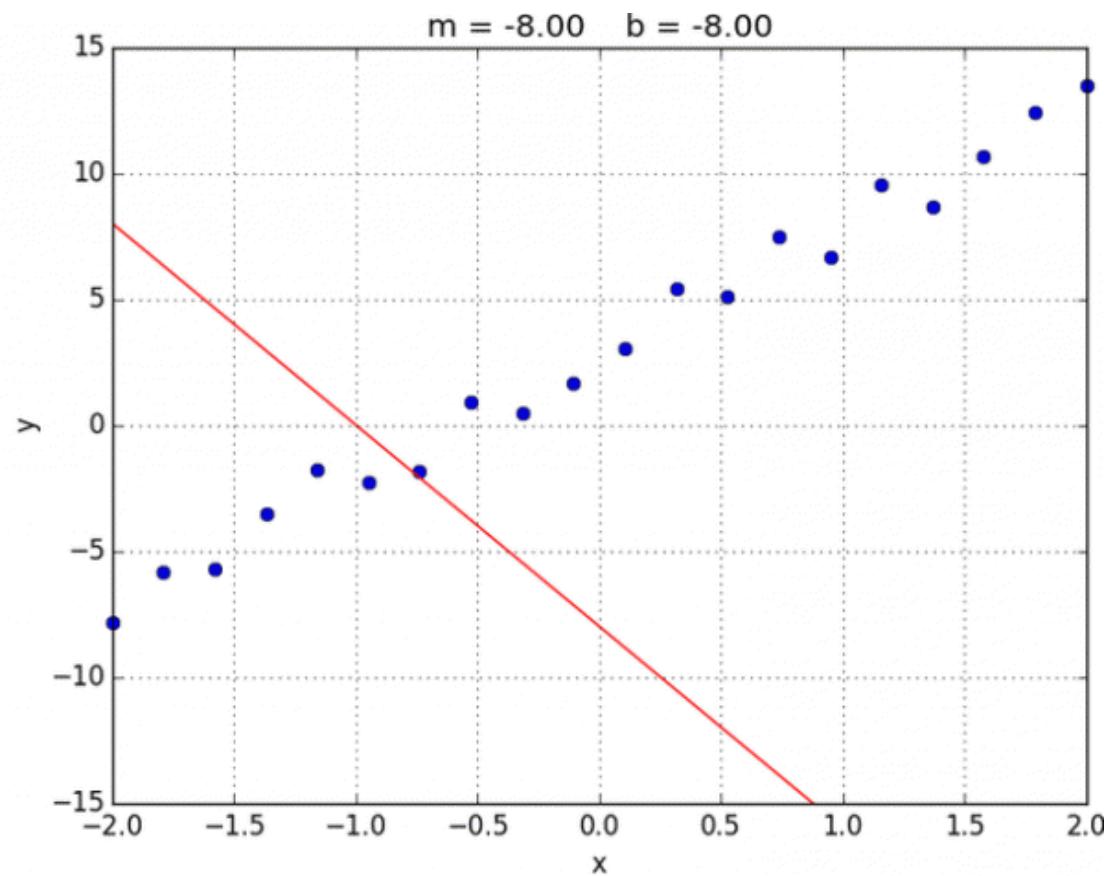
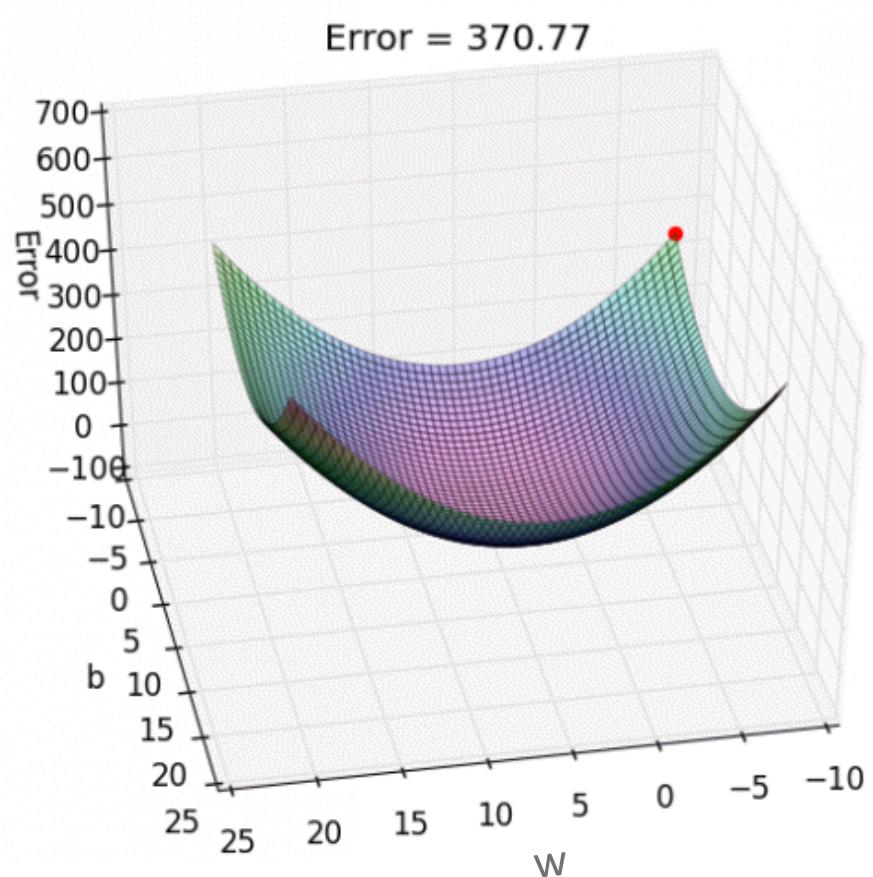


Show this!

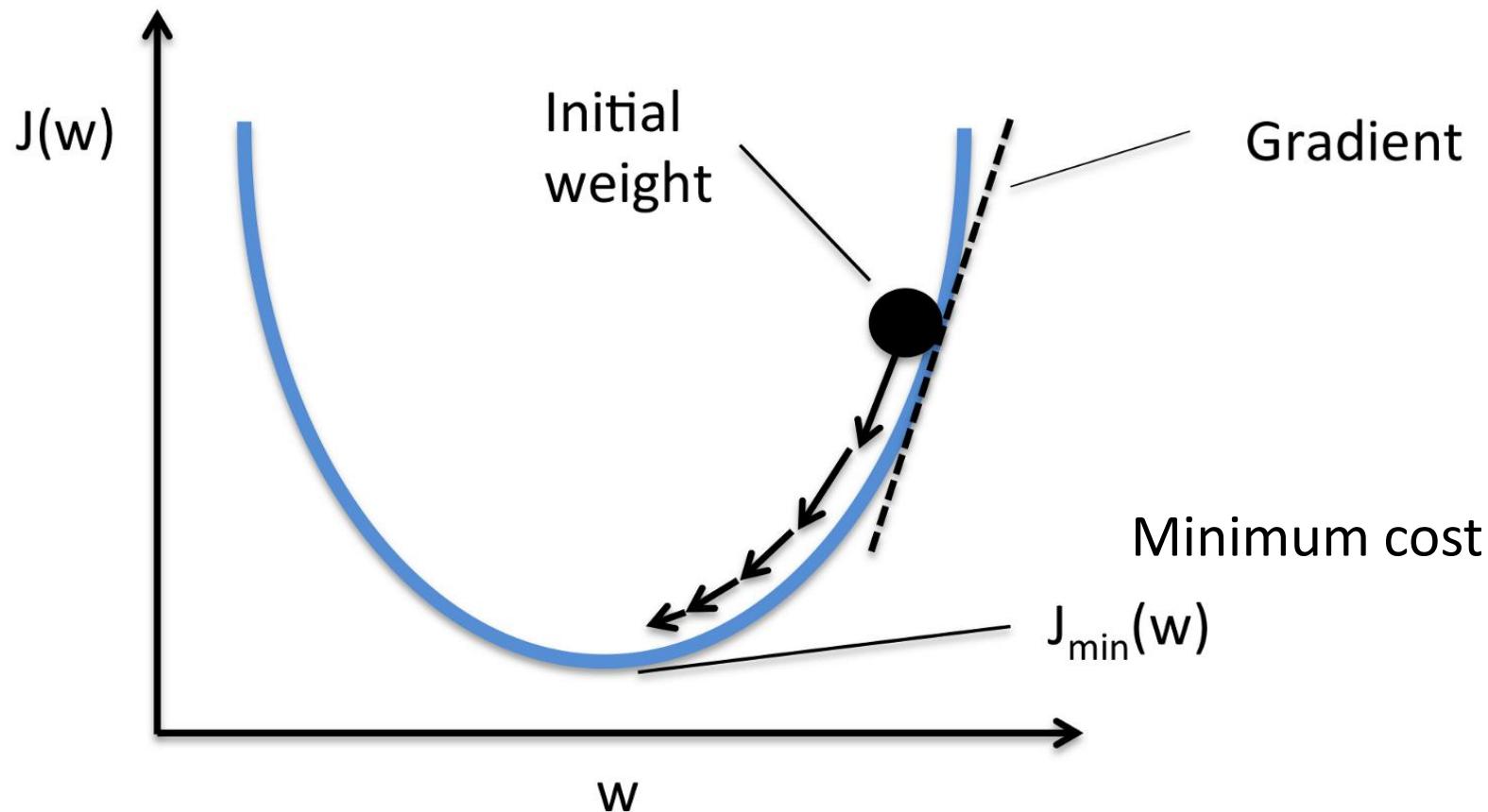
- Linear regression is one of only a handful of models in this course that permit direct solution.

Also maximum likelihood estimator
(least-square estimator)

Gradient Descent – convex!



Gradient Descent

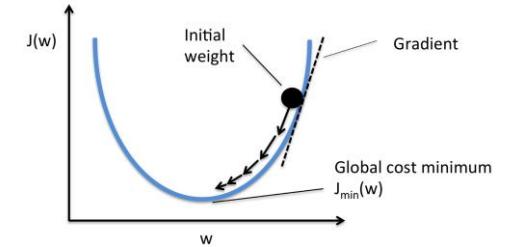


Gradient Descent

- Now let's see a second way to minimize the cost function which is more broadly applicable: **gradient descent**.
- Gradient descent is an **iterative algorithm**, which means we apply an update repeatedly until some criterion is met.
- We **initialize** the weights to something reasonable (e.g. all zeros) and repeatedly adjust them in the **direction of steepest descent**.

- Observe:
 - if $\partial \mathcal{J} / \partial w_j > 0$, then increasing w_j increases \mathcal{J} .
 - if $\partial \mathcal{J} / \partial w_j < 0$, then increasing w_j decreases \mathcal{J} .
- The following update decreases the cost function:

$$\begin{aligned}
 w_j &\leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} \\
 &= w_j - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)}
 \end{aligned}$$



- α is a **learning rate**. The larger it is, the faster \mathbf{w} changes.
 - We'll see later how to tune the learning rate, but values are typically small, e.g. 0.01 or 0.0001

- This gets its name from the **gradient**:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

- This is the direction of fastest increase in \mathcal{J} .
- Update rule in vector form:

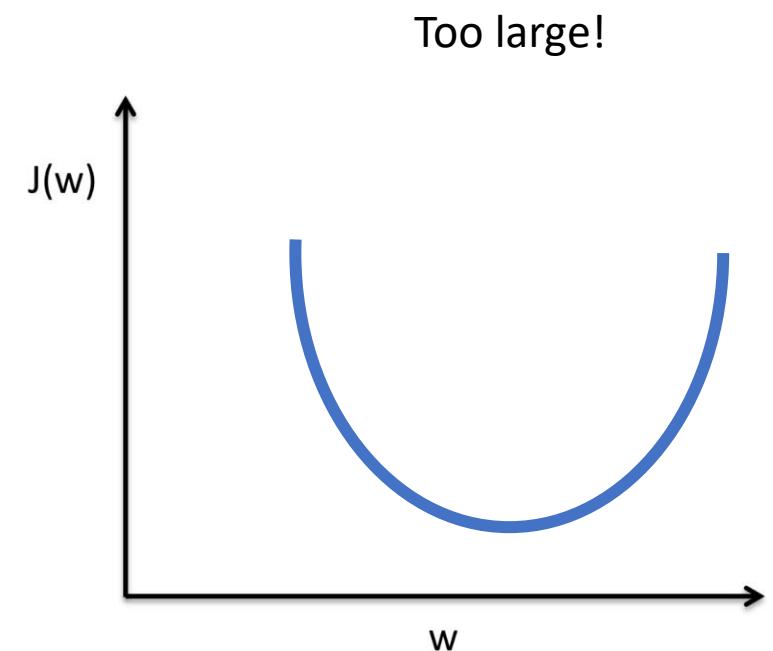
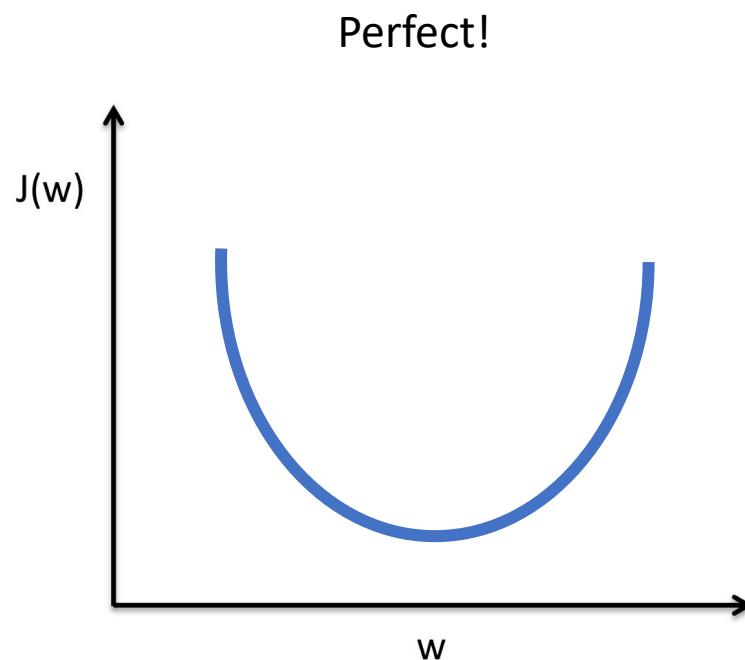
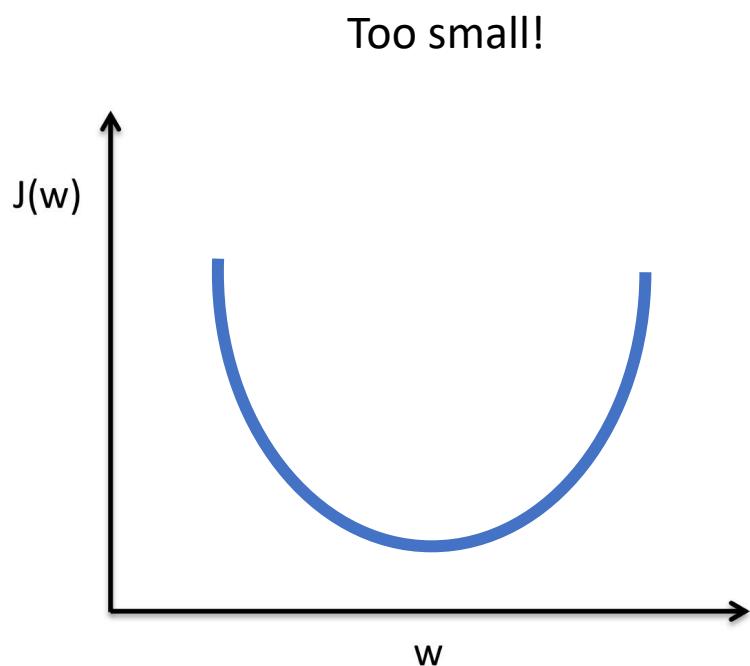
$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}} \\ &= \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) \mathbf{x}^{(i)} \end{aligned}$$

- Hence, gradient descent updates the weights in the direction of fastest *decrease*.

Do we need Gradient Descent (GD)?

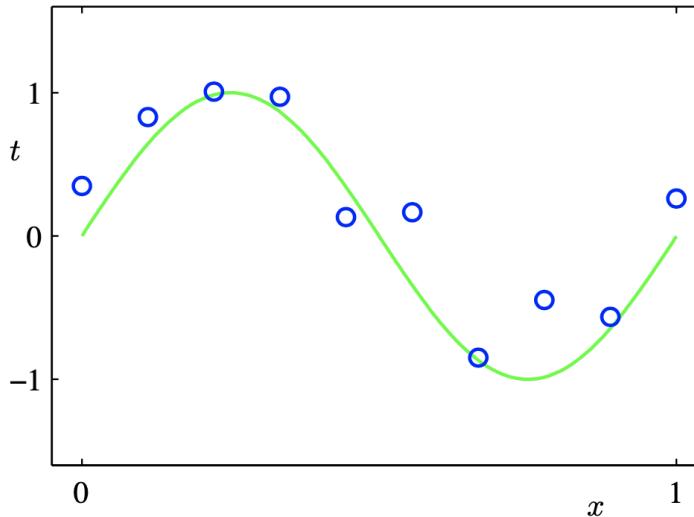
- Why gradient descent, if we can find the optimum directly?
 - GD can be applied to a much broader set of models
 - GD can be easier to implement than direct solutions, especially with automatic differentiation software
 - For regression in high-dimensional spaces, GD is more efficient than direct solution (matrix inversion is an $\mathcal{O}(D^3)$ algorithm).

Learning rate role?



Polynomial Regression

- Suppose we want to model the following data



-Pattern Recognition and Machine Learning, Christopher Bishop.

- One option: fit a low-degree polynomial; this is known as **polynomial regression**

$$y = w_3x^3 + w_2x^2 + w_1x + w_0$$

- Do we need to derive a whole new algorithm?

Feature mapping

- We get polynomial regression for free!
- Define the **feature map**

$$\psi(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}$$

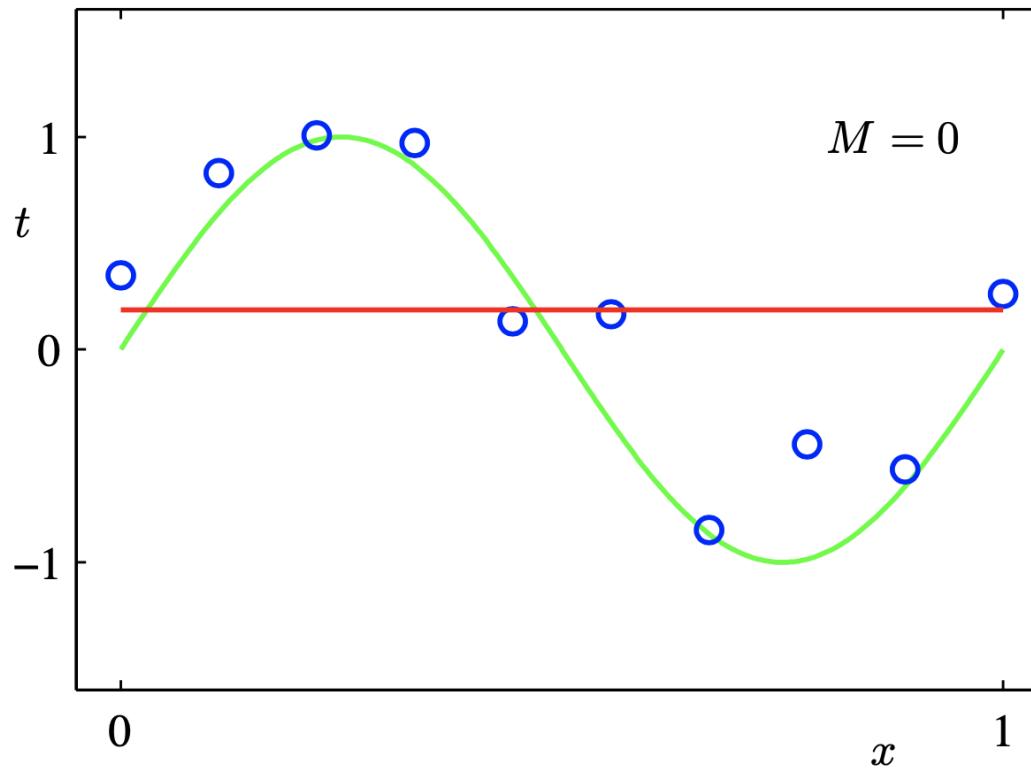
- Polynomial regression model:

$$y = \mathbf{w}^\top \psi(x)$$

- All of the derivations and algorithms so far in this lecture remain exactly the same!

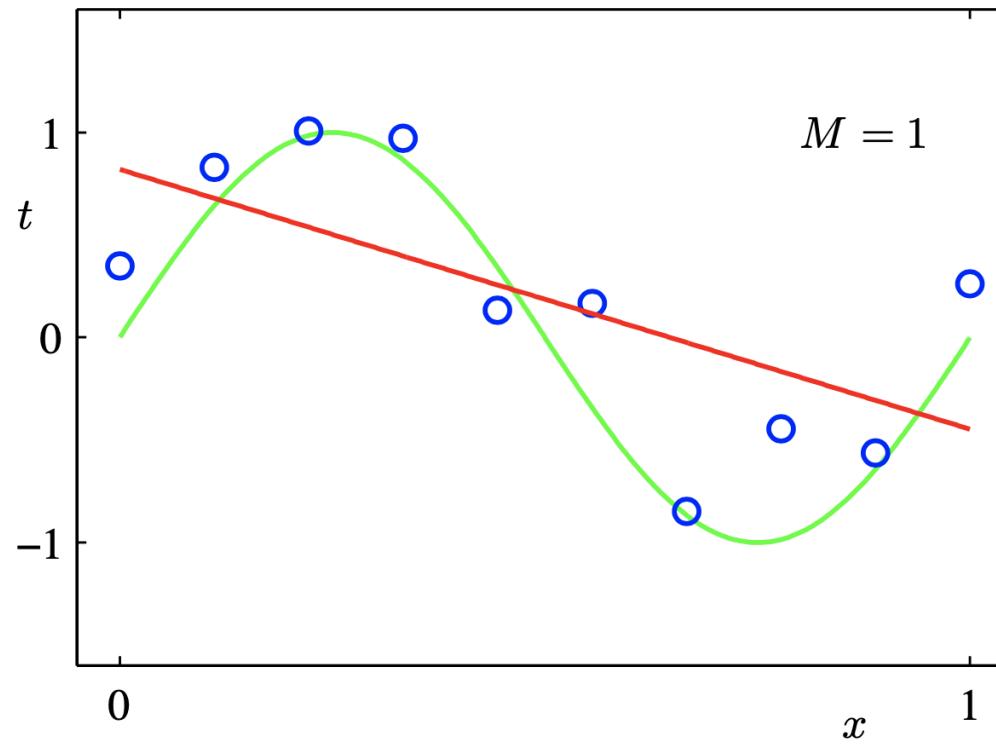
Fitting Polynomial

$$y = w_0$$



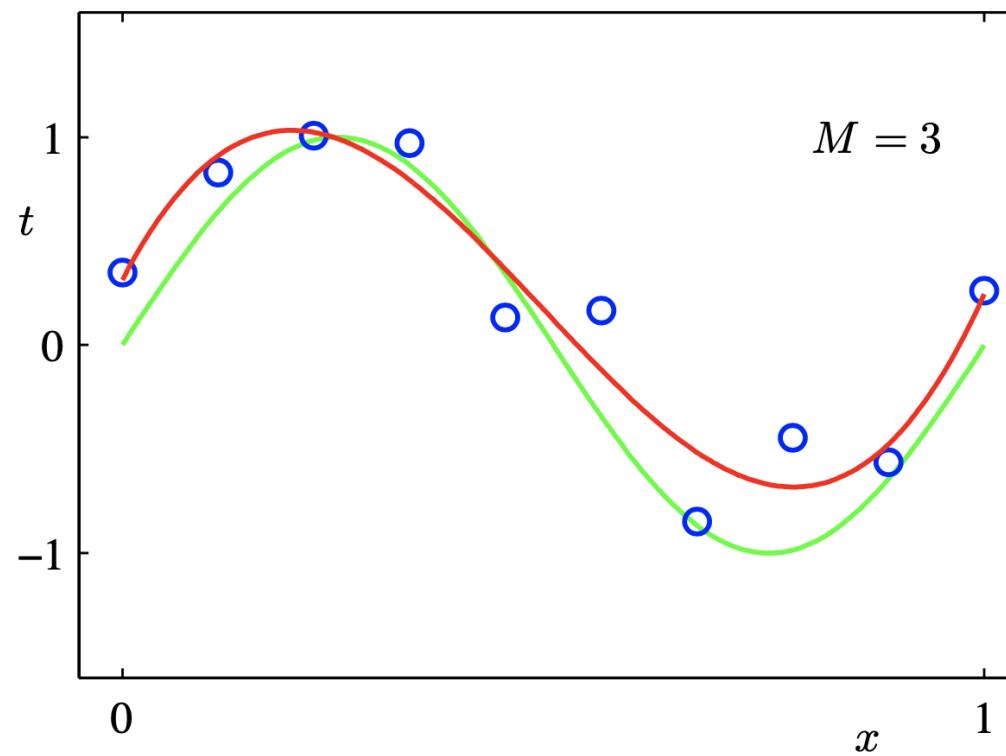
Fitting Polynomial

$$y = w_0 + w_1 x$$



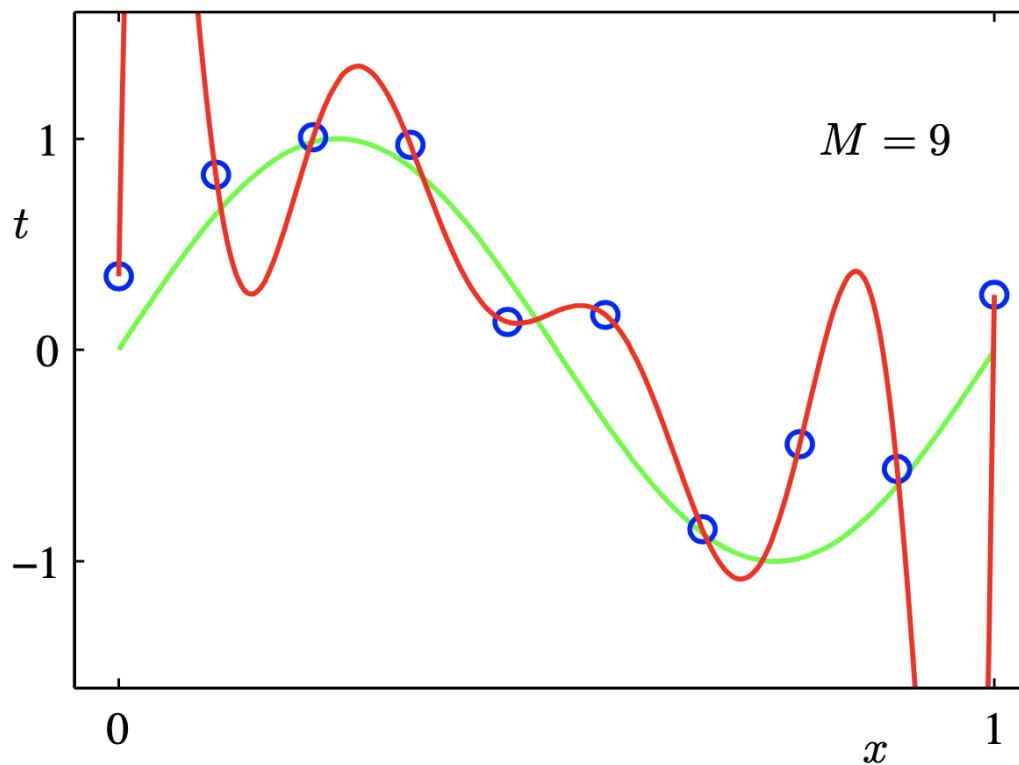
Fitting Polynomial

$$y = w_0 + w_1x + w_2x^2 + w_3x^3$$



Fitting Polynomial

$$y = w_0 + w_1x + w_2x^2 + w_3x^3 + \dots + w_9x^9$$



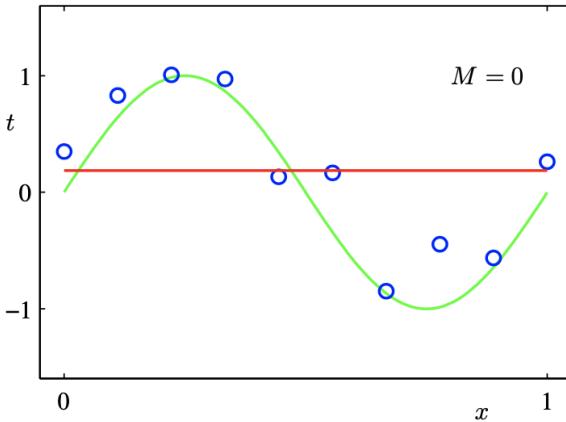
Fitting Polynomial

$$y = w_0 + w_1x + w_2x^2 + w_3x^3 + \dots + w_9x^9$$

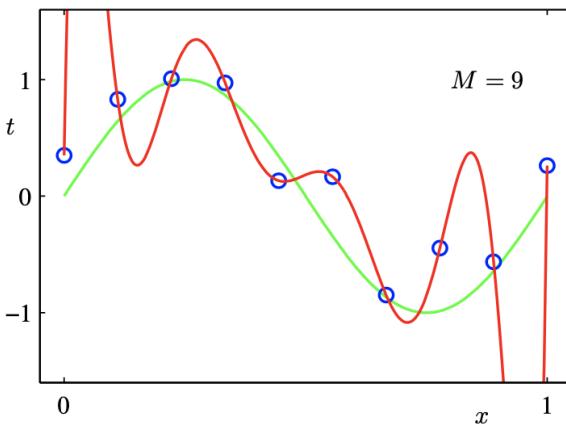


Generalization

Underfitting : model is too simple — does not fit the data.

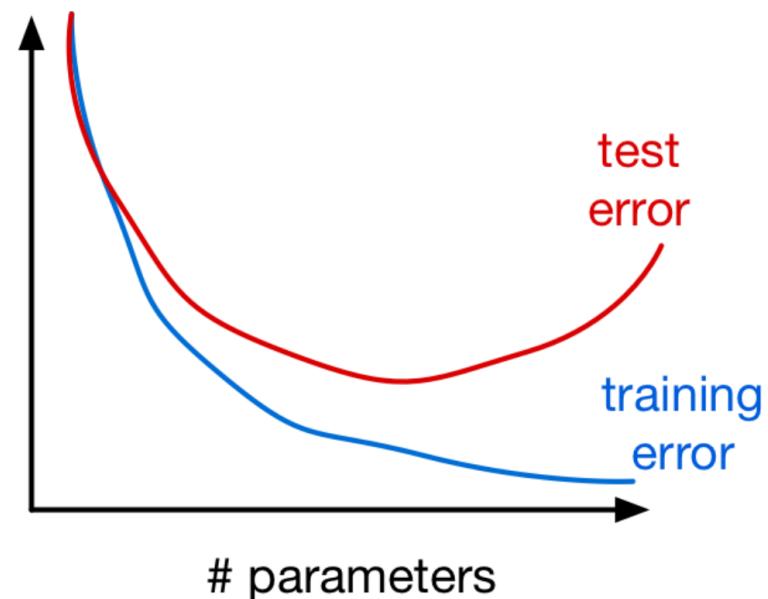
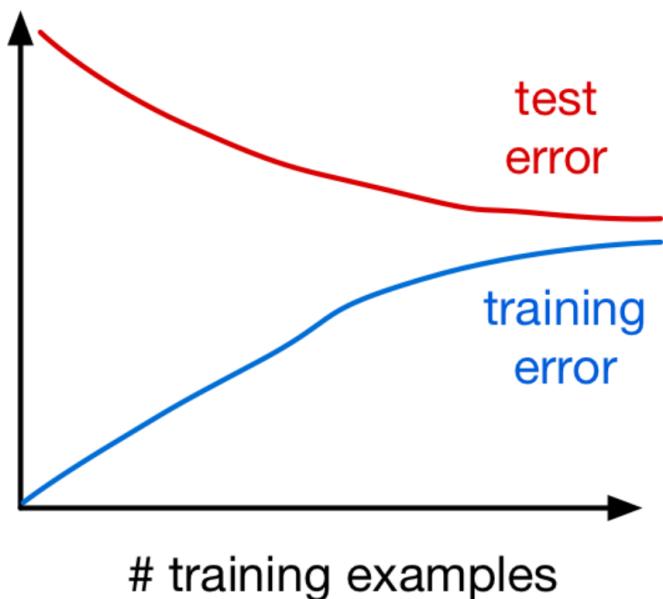


Overfitting : model is too complex — fits perfectly, does not generalize.



Generalization

- Training and test error as a function of # training examples and # parameters:

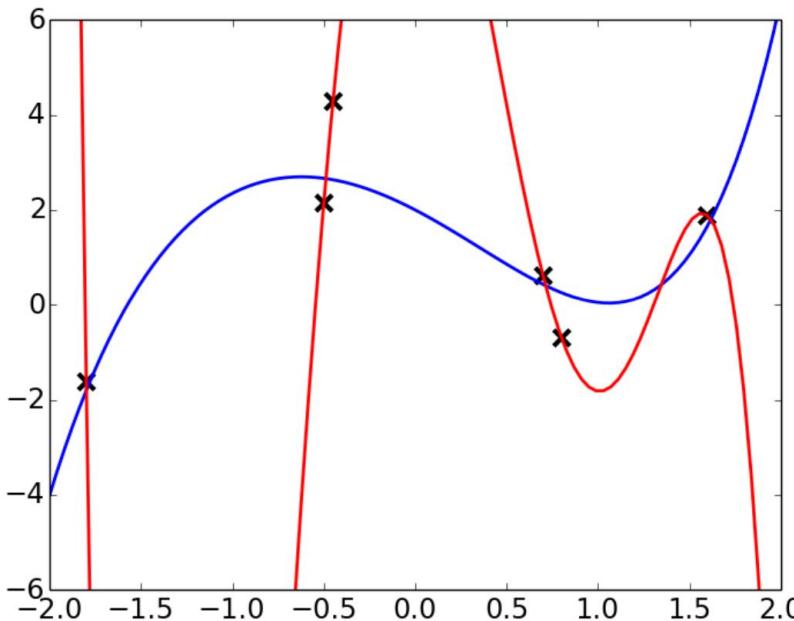


Regularization

- The degree of the polynomial is a hyperparameter, just like k in KNN. We can tune it using a validation set.
- But restricting the size of the model is a crude solution, since you'll never be able to learn a more complex model, even if the data support it.
- Another approach: keep the model large, but **regularize** it
 - **Regularizer:** a function that quantifies how much we prefer one hypothesis vs. another

L^2 Regularization

Observation: polynomials that overfit often have large coefficients.



$$y = 0.1x^5 + 0.2x^4 + 0.75x^3 - x^2 - 2x + 2$$

$$y = -7.2x^5 + 10.4x^4 + 24.5x^3 - 37.9x^2 - 3.6x + 12$$

So let's try to keep the coefficients small.

L^2 Regularization

Another reason we want weights to be small:

- Suppose inputs x_1 and x_2 are nearly identical for all training examples.
The following two hypotheses make nearly the same predictions:

$$\mathbf{w} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \mathbf{w} = \begin{pmatrix} -9 \\ 11 \end{pmatrix}$$

- But the second network might make weird predictions if the test distribution is slightly different (e.g. x_1 and x_2 match less closely).

L^2 Regularization

- We can encourage the weights to be small by choosing as our regularizer the L^2 penalty.

$$\mathcal{R}(\mathbf{w}) = \frac{1}{2} \|\mathbf{w}\|^2 = \frac{1}{2} \sum_j w_j^2.$$

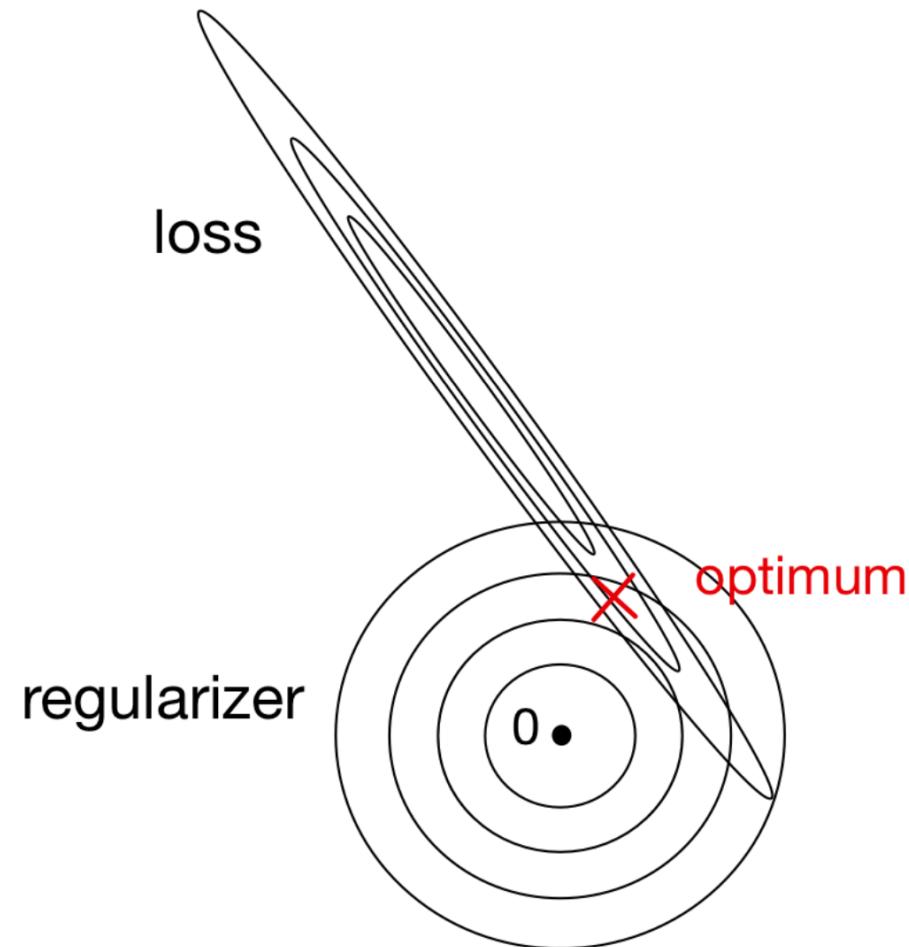
- Note: to be pedantic, the L^2 norm is Euclidean distance, so we're really regularizing the *squared* L^2 norm.
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights.

$$\mathcal{J}_{\text{reg}} = \mathcal{J} + \lambda \mathcal{R} = \mathcal{J} + \frac{\lambda}{2} \sum_j w_j^2$$

- Here, λ is a hyperparameter that we can tune using a validation set.

Visualization

- The geometric picture:



L² Regularization

- Recall the gradient descent update:

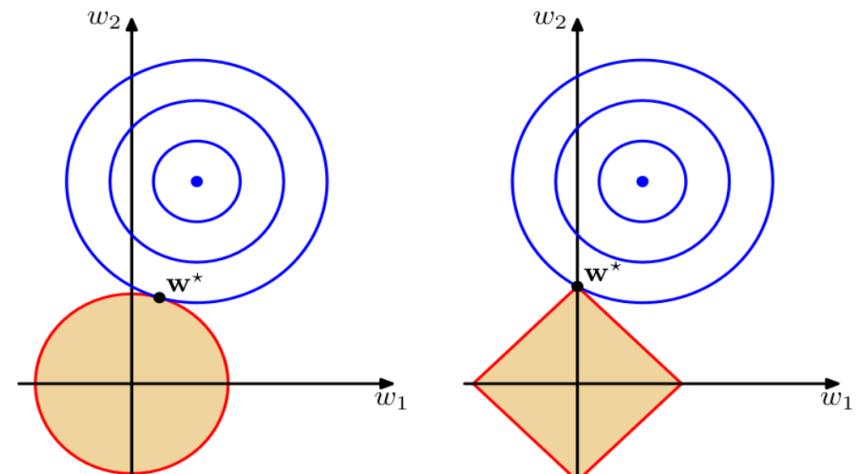
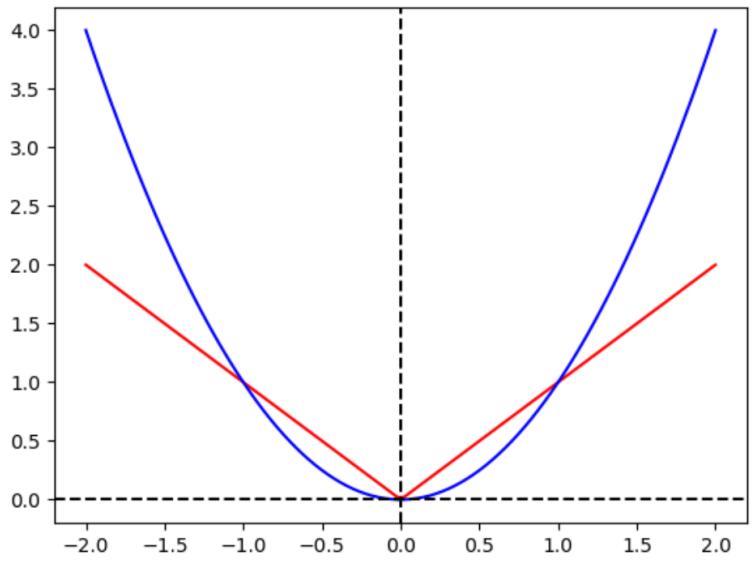
$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

- The gradient descent update of the regularized cost has an interesting interpretation as weight decay:

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\ &= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\ &= (1 - \alpha \lambda) \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}\end{aligned}$$

L^1 vs L^2 Regularization

- The L^1 norm, or sum of absolute values, is another regularizer that encourages weights to be exactly zero. (How can you tell?)
- We can design regularizers based on whatever property we'd like to encourage.



$$\mathcal{R} = \sum_i w_i^2$$

$$\mathcal{R} = \sum_i |w_i|$$

— Bishop, *Pattern Recognition and Machine Learning*

Batch size

- How much (training) data do you consider when performing a step?
 - Stochastic Gradient Descent – 1 data point
 - Mini-batch – part of your data
 - Full batch – all data
- Why does it matter?

