Linear Regression

PROF XIAOHUI XIE
SPRING 2019

CS 273P Machine Learning and Data Mining

Slides courtesy of Alex Ihler

Machine Learning

Linear Regression via Least Squares

Gradient Descent Algorithms

Direct Minimization of Squared Error

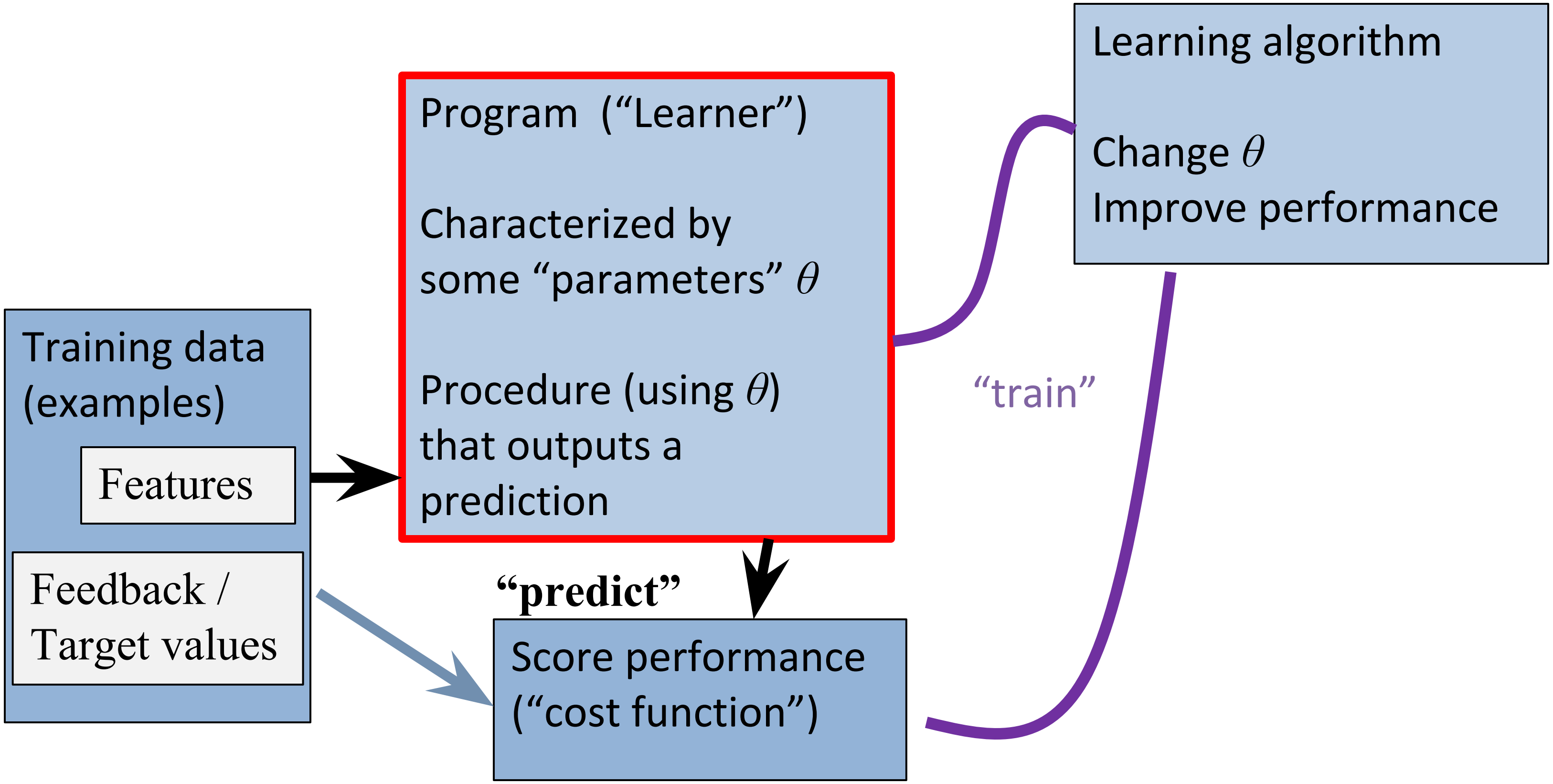
Regression with Non-linear Features

Bias, Variance, & Validation

Regularized Linear Regression

Supervised learning

- Notation
  - Features x
  - Targets y
  - Predictions $\hat{y} = f(x\ ;\ \theta)$
  - Parameters θ



Training data (examples)

Features

Feedback / Target values

Program ("Learner")

Characterized by some "parameters" θ

Procedure (using θ) that outputs a prediction

"predict"

Score performance ("cost function")

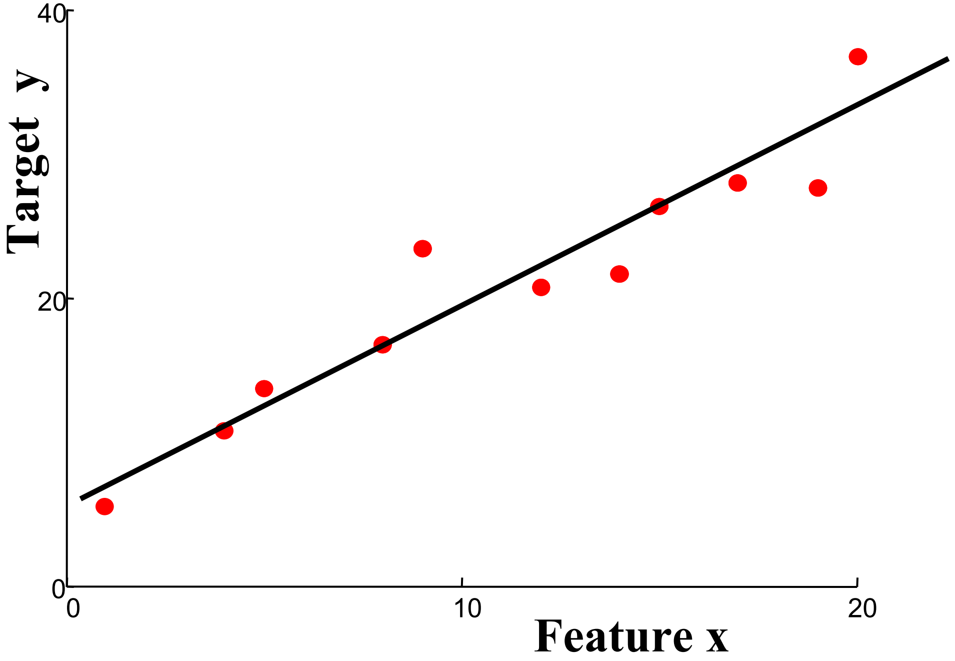
Learning algorithm

Change θ

Improve performance

"train"

Linear regression



"Predictor":

Evaluate line:

$$r = \theta_0 + \theta_1 x_1$$

return r

- Define form of function f(x) explicitly
- Find a good f(x) within that family

Notation

$$\hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots$$

Define "feature" $x_0 = 1$ (constant)

Then

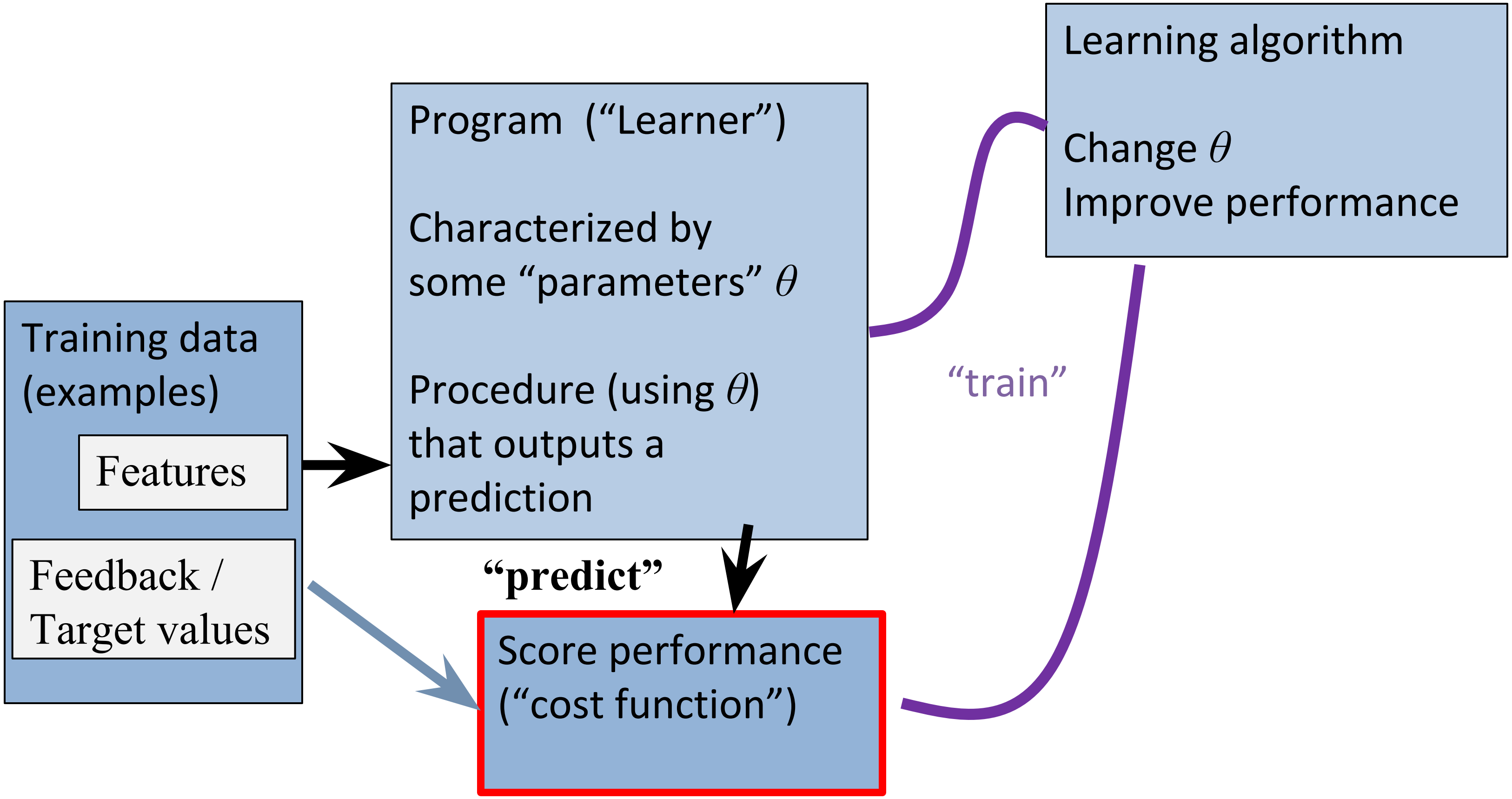
$$\hat{y}(x) = \theta\, x^T$$

$$\underline{\theta} = [\theta_0, \ldots, \theta_n]$$

$$\underline{x} = [1, x_1, \ldots, x_n]$$

Supervised learning

- Notation
  - Features x
  - Targets y
  - Predictions $\hat{y} = f(x ; \theta)$
  - Parameters θ



Learning algorithm

Change θ
Improve performance

Program ("Learner")

Characterized by some "parameters" θ

Procedure (using θ) that outputs a prediction

Training data (examples)

Features

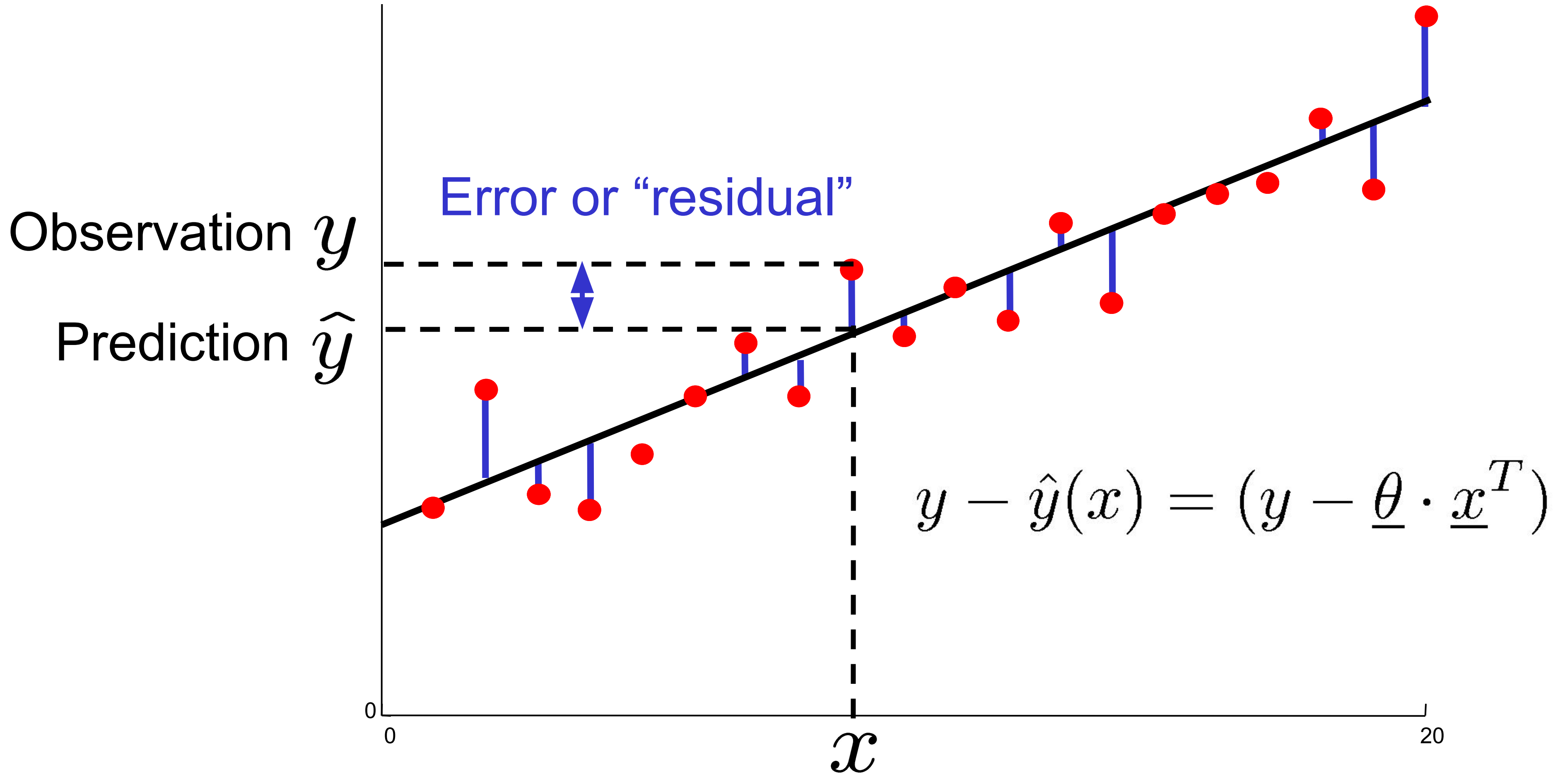
Feedback / Target values

"train"

"predict"

Score performance ("cost function")

Measuring error



Mean squared error

- How can we quantify the error?

$$\text{MSE},\ J(\underline{\theta}) = \frac{1}{m}\sum_j (y^{(j)} - \hat{y}(x^{(j)}))^2$$
$$= \frac{1}{m}\sum_j (y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)T})^2$$

- Could choose something else, of course…
  - Computationally convenient (more later)
  - Measures the variance of the residuals
  - Corresponds to likelihood under Gaussian model of "noise"

$$\mathcal{N}(y\ ;\ \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{ -\frac{1}{2\sigma^2}(y-\mu)^2 \right\}$$

MSE cost function

- Rewrite using matrix form

$$\text{MSE},\ J(\underline{\theta}) = \frac{1}{m}\sum_j \left(y^{(j)} - \hat{y}(x^{(j)})\right)^2$$

$$= \frac{1}{m}\sum_j \left(y^{(j)} - \underline{\theta}\cdot\underline{x}^{(j)T}\right)^2$$

$$\underline{\theta} = [\theta_0, \ldots, \theta_n]$$

$$\underline{y} = \left[y^{(1)} \ldots, y^{(m)}\right]^T$$

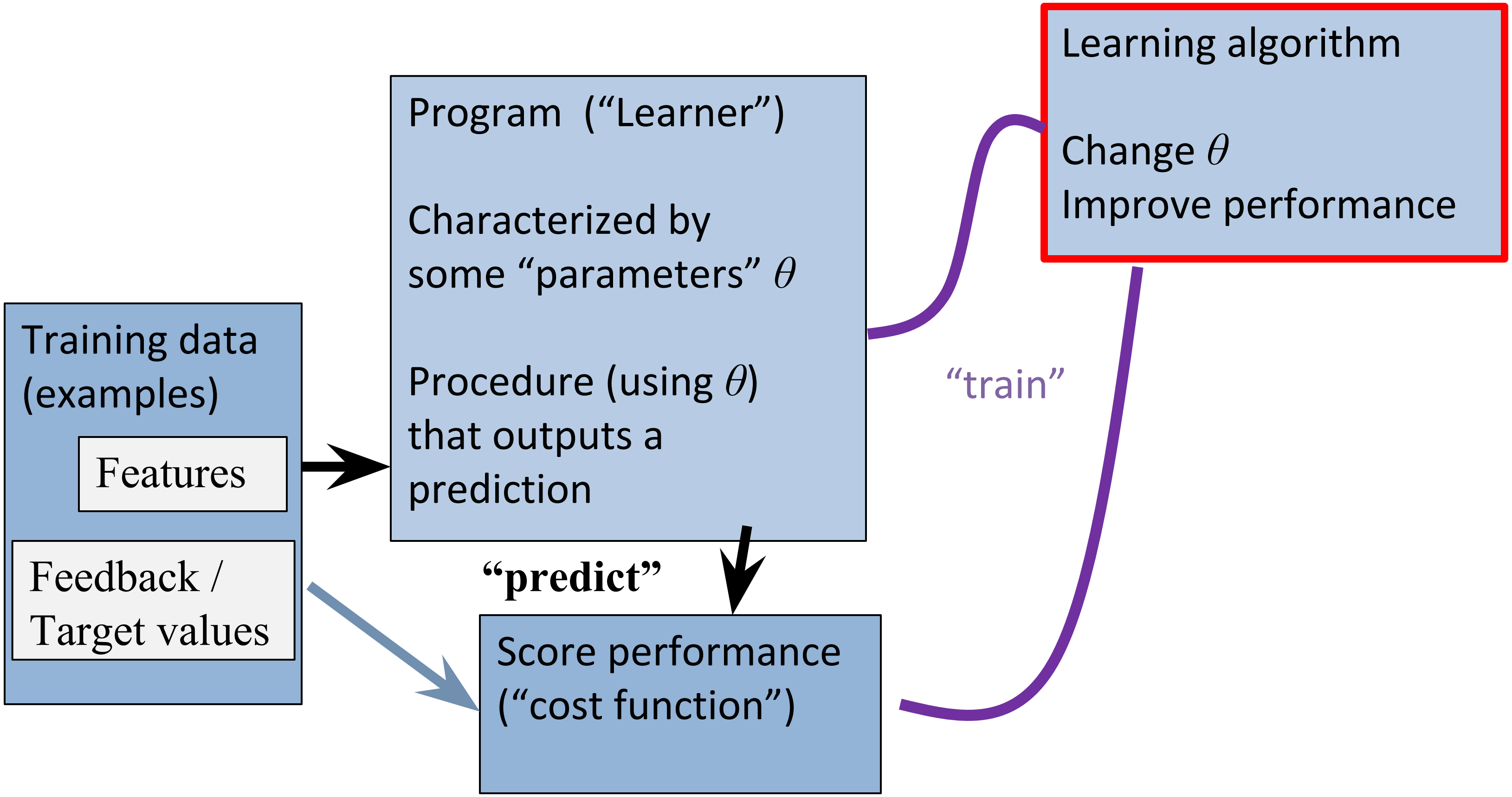
$$\underline{X} = \begin{bmatrix} x_0^{(1)} & \ldots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_0^{(m)} & \ldots & x_n^{(m)} \end{bmatrix}$$

$$J(\underline{\theta}) = \frac{1}{m}(\underline{y}^T - \underline{\theta}\,\underline{X}^T)\cdot(\underline{y}^T - \underline{\theta}\,\underline{X}^T)^T$$

```
# Python / NumPy:
e = Y – X.dot( theta.T );
J = e.T.dot( e ) / m  # = np.mean( e ** 2 )
```

Supervised learning

- Notation
  - Features x
  - Targets y
  - Predictions $\hat{y} = f(x\ ;\ \theta)$
  - Parameters θ



Learning algorithm

Change θ

Improve performance

Program (“Learner”)

Characterized by some “parameters” θ

Procedure (using θ) that outputs a prediction

Training data (examples)

Features

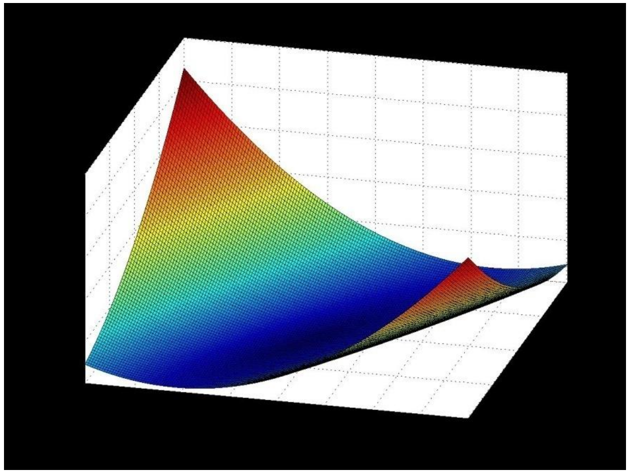
Feedback / Target values

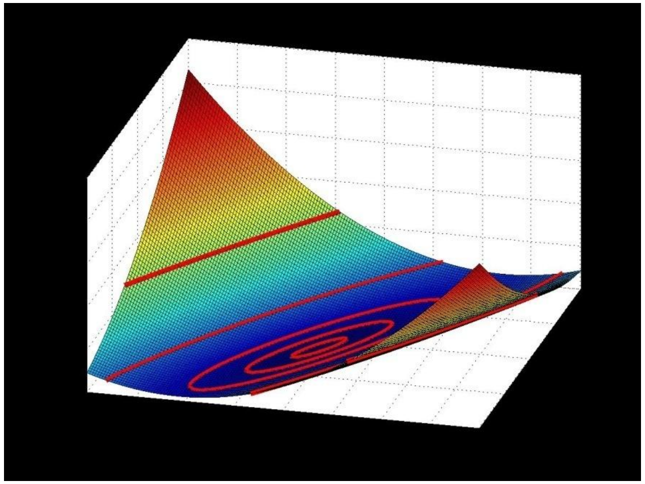
“train”

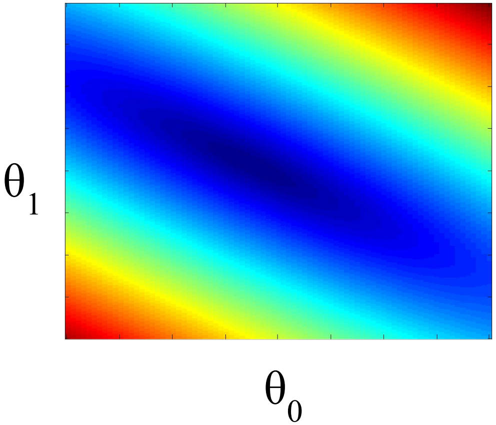
“predict”

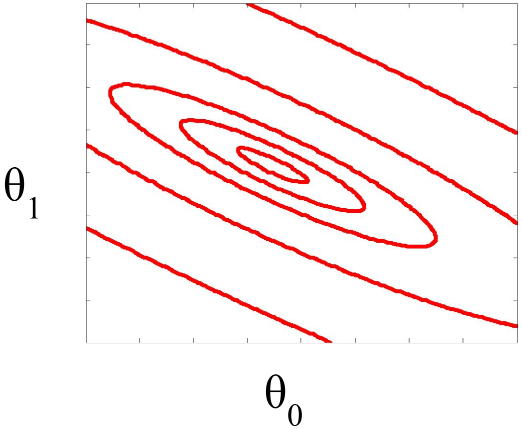
Score performance (“cost function”)

Visualizing the cost function





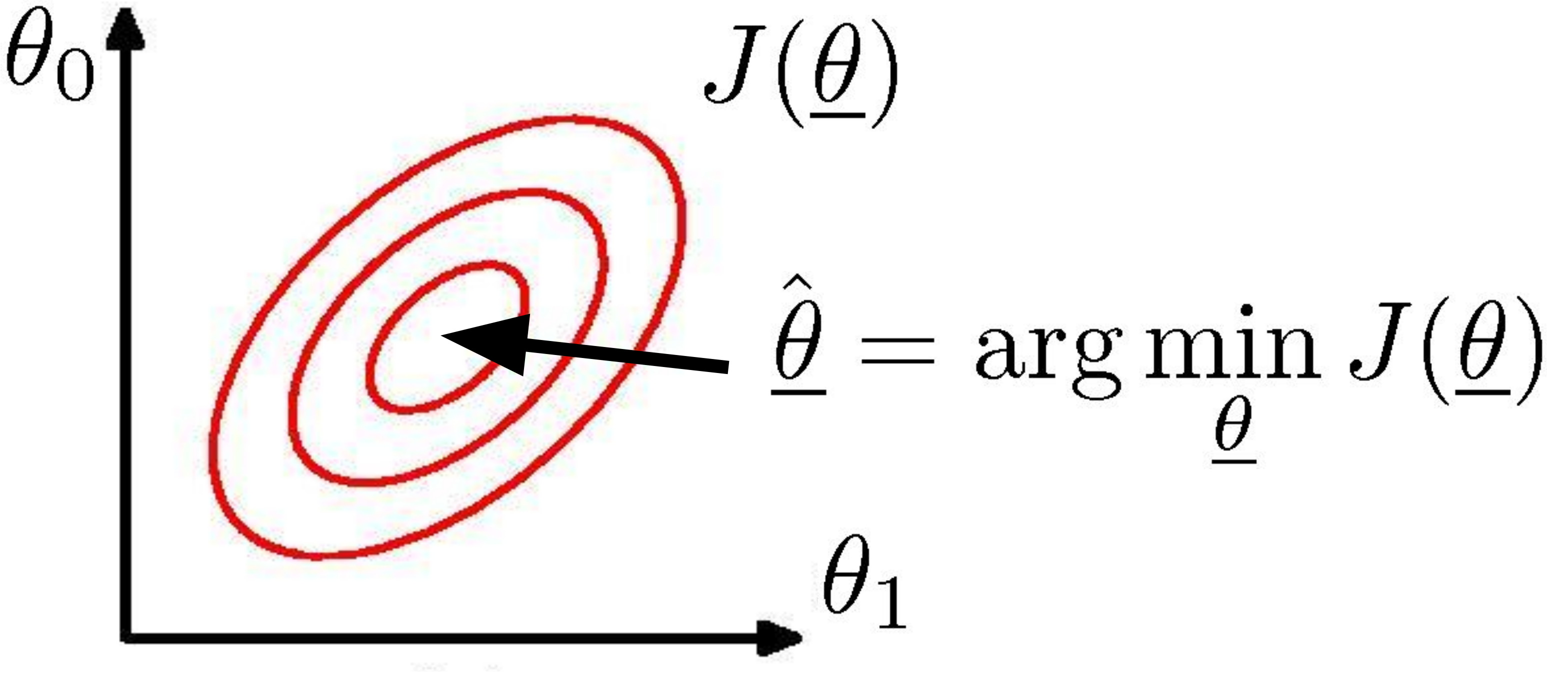




Finding good parameters

- Want to find parameters which minimize our error…

- Think of a cost “surface”: error residual for that θ …

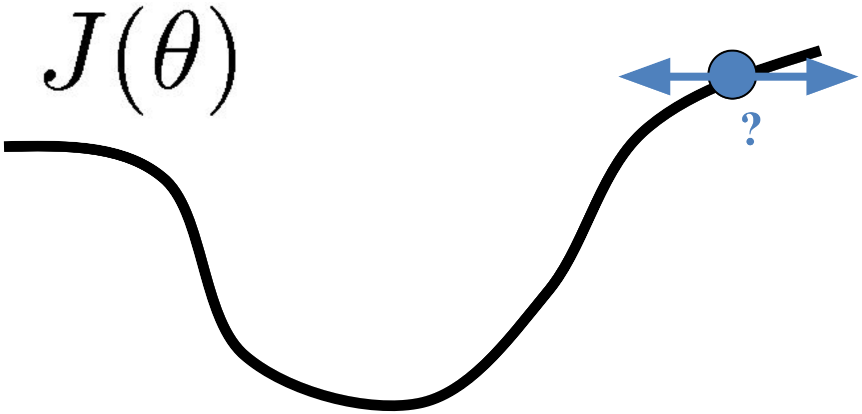


$$\underline{\hat{\theta}} = \arg\min_{\underline{\theta}} J(\underline{\theta})$$

Machine Learning

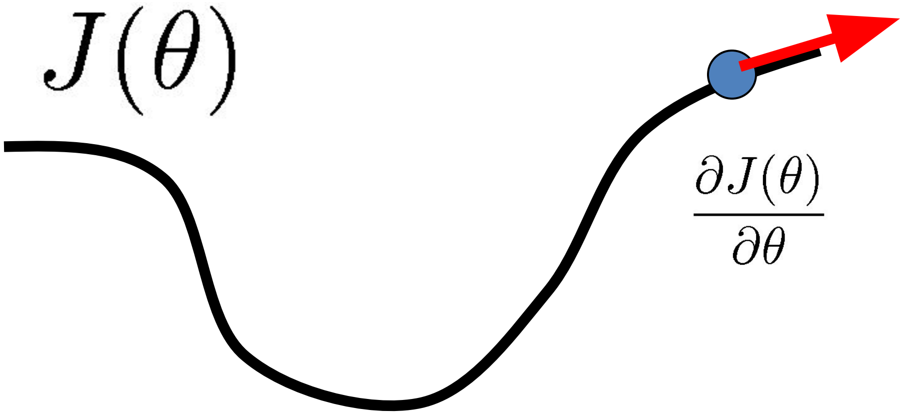
- Linear Regression via Least Squares
- **Gradient Descent Algorithms**
- Direct Minimization of Squared Error
- Regression with Non-linear Features
- Bias, Variance, & Validation
- Regularized Linear Regression

Gradient descent



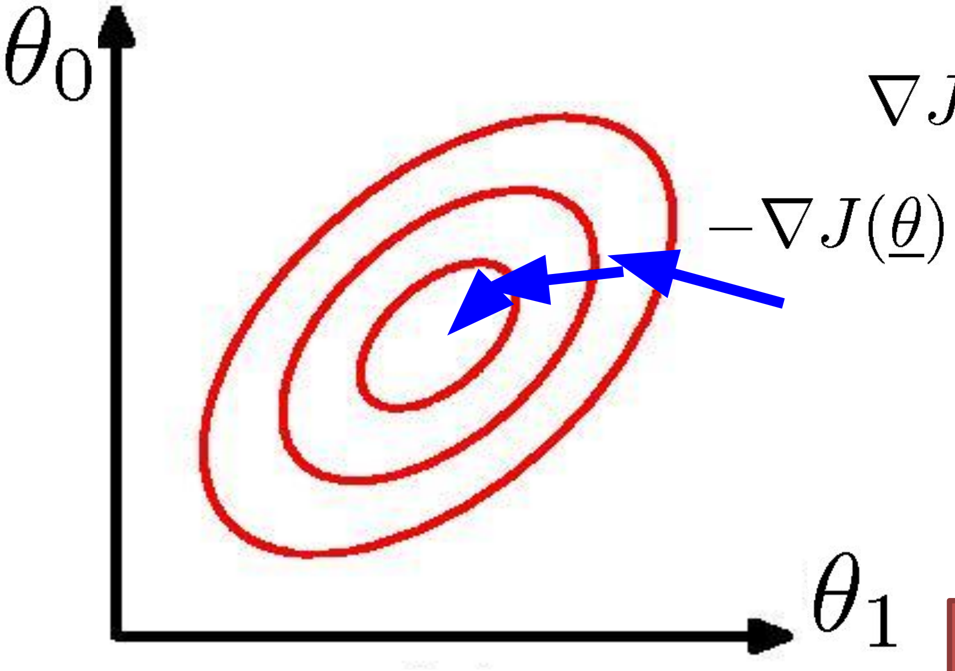
- How to change θ to improve J(θ)?
- Choose a direction in which J(θ) is decreasing

Gradient descent



- How to change θ to improve J(θ)?
- Choose a direction in which J(θ) is decreasing
- Derivative $\frac{\partial J(\theta)}{\partial \theta}$
- Positive => increasing
- Negative => decreasing

Gradient descent in more dimensions



- Gradient vector

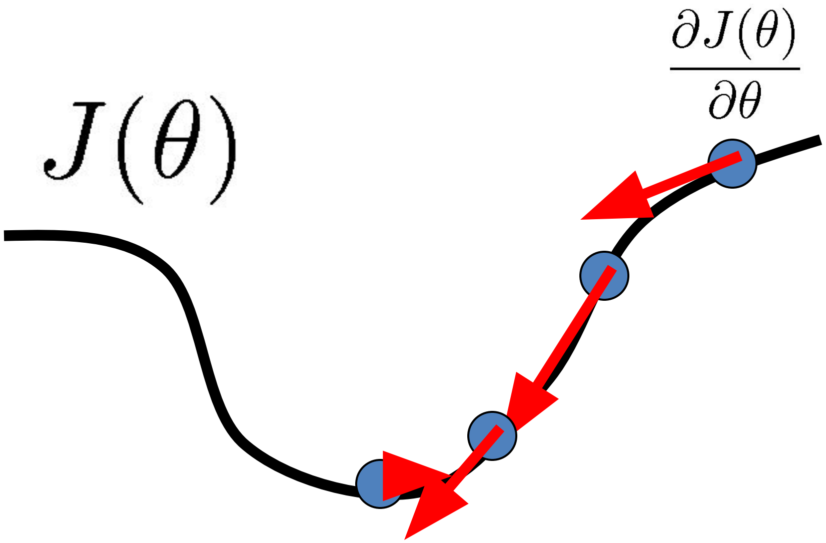
$$\nabla J(\underline{\theta}) = \left[\frac{\partial J(\underline{\theta})}{\partial \theta_0} \quad \frac{\partial J(\underline{\theta})}{\partial \theta_1} \quad \ldots \right]$$

Indicates direction of steepest ascent
(negative = steepest descent)

Gradient descent

- Initialization
- Step size α
  - Can change over iterations
- Gradient direction
- Stopping condition

```
Initialize $\theta$
Do{
 $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$
} while $(\alpha \| \nabla_\theta J \| > \varepsilon)$
```



Gradient for the MSE

- MSE

$$J(\underline{\theta}) = \frac{1}{m}\sum_j (y^{(j)} - \underline{\theta}\cdot\underline{x}^{(j)\,T})^2$$

- ∇ J = ?

$$J(\underline{\theta}) = \frac{1}{m}\sum_j (\overbrace{y^{(j)} - \theta_0\underline{x}_0^{(j)} - \theta_1\underline{x}_1^{(j)} - \ldots}^{e_j(\theta)})^2$$

$$\begin{aligned}\frac{\partial J}{\partial\theta_0} &= \frac{\partial}{\partial\theta_0}\frac{1}{m}\sum_j(\ e_j(\theta)\ )^2\\ &= \frac{1}{m}\sum_j\ \frac{\partial}{\partial\theta_0}(\ e_j(\theta)\ )^2\\ &= \frac{1}{m}\sum_j\ 2e_j(\theta)\ \frac{\partial}{\partial\theta_0}e_j(\theta)\end{aligned}$$

$$\begin{aligned}\frac{\partial}{\partial\theta_0}e_j(\theta) &= \underbrace{\frac{\partial}{\partial\theta_0}y^{(j)}}_{\mathbf{0}} - \frac{\partial}{\partial\theta_0}\theta_0 x_0^{(j)} - \underbrace{\frac{\partial}{\partial\theta_0}\theta_1 x_1^{(j)}}_{\mathbf{0}} - \ldots\\ &= -x_0^{(j)}\end{aligned}$$

Gradient for the MSE

- MSE

$$J(\underline{\theta}) = \frac{1}{m} \sum_j (y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)^T})^2$$

- ∇ J = ?

$$J(\underline{\theta}) = \frac{1}{m} \sum_j (\overbrace{y^{(j)} - \theta_0 \underline{x}_0^{(j)} - \theta_1 \underline{x}_1^{(j)} - \ldots}^{e_j(\theta)})^2$$

$$\nabla J(\underline{\theta}) = [\quad \frac{\partial J}{\partial \theta_0} \qquad \frac{\partial J}{\partial \theta_1} \qquad \ldots]$$

$$= [\frac{2}{m} \sum_j -e_j(\theta) x_0^{(j)} \qquad \frac{2}{m} \sum_j -e_j(\theta) x_1^{(j)} \qquad \ldots]$$

Gradient descent

- Initialization
- Step size α
  - Can change over iterations
- Gradient direction
- Stopping condition

```
Initialize $\theta$
Do{
 $\theta \leftarrow \theta - \alpha\nabla_\theta J(\theta)$
} while ($\alpha\|\nabla_\theta J\| > \varepsilon$)
```

$$J(\underline{\theta}) = \frac{1}{m}\sum_j \left(y^{(j)} - \underline{\theta}\cdot\underline{x}^{(j)^T}\right)^2$$

$$\nabla J(\underline{\theta}) = -\frac{2}{m}\sum_j \underbrace{\left(y^{(j)} - \underline{\theta}\cdot\underline{x}^{(j)^T}\right)}_{\text{Error magnitude \& direction for datum j}} \cdot \underbrace{[x_0^{(j)}\, x_1^{(j)}\, \ldots]}_{\text{Sensitivity to each param}}$$

Derivative of MSE

- Rewrite using matrix form

$$\nabla J(\underline{\theta}) = -\frac{2}{m}\sum_j (\underbrace{y^{(j)} - \underline{\theta}}_{\text{Error magnitude \& direction for datum j}} \cdot \underline{x}^{(j)}{}^T) \cdot [\underbrace{x_0^{(j)} x_1^{(j)}}_{\text{Sensitivity to each } \theta_i} \ldots]$$

$$\underline{\theta} = [\theta_0, \ldots, \theta_n]$$

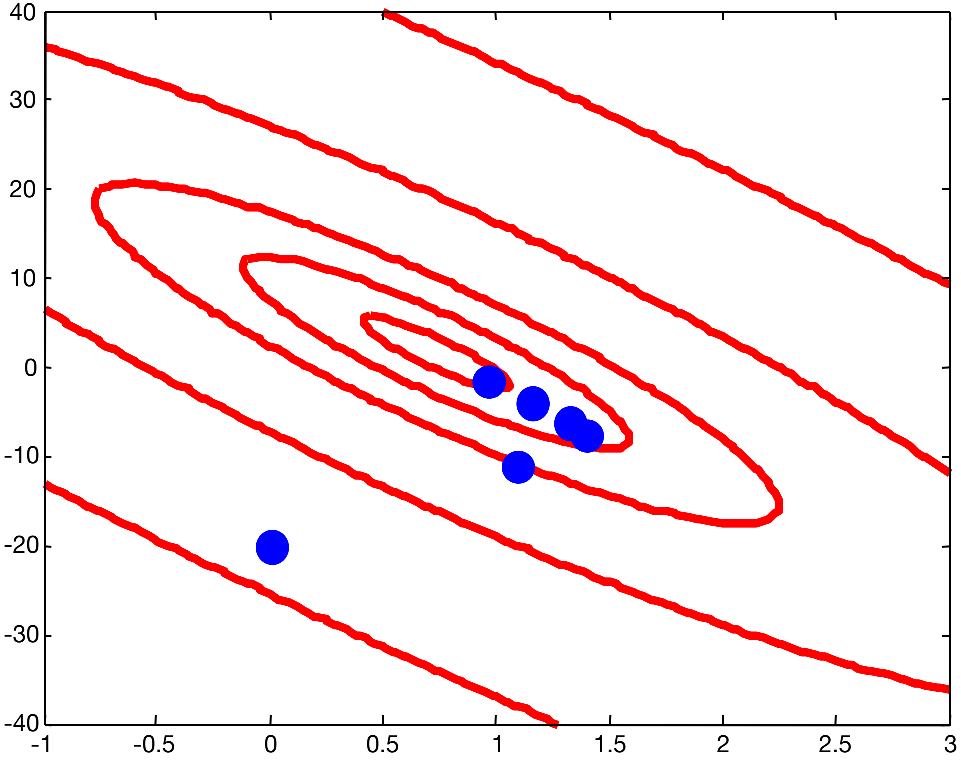
$$\underline{y} = \left[y^{(1)} \ldots, y^{(m)}\right]^T$$

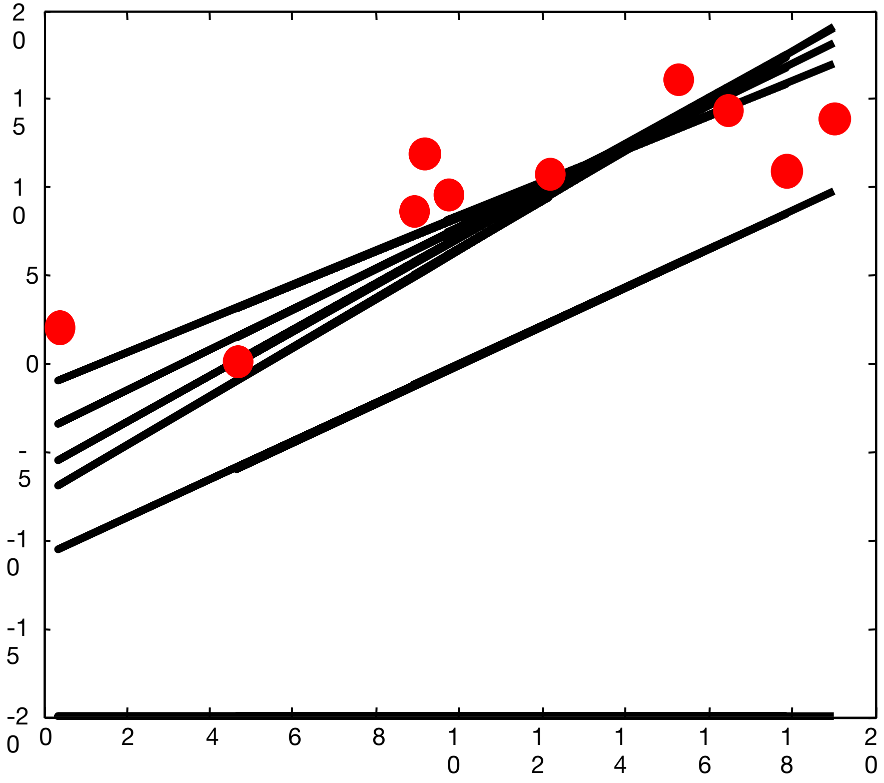
$$\underline{X} = \begin{bmatrix} x_0^{(1)} & \ldots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_0^{(m)} & \ldots & x_n^{(m)} \end{bmatrix}$$

$$\nabla J(\underline{\theta}) = -\frac{2}{m}(\underline{y}^T - \underline{\theta X}^T) \cdot \underline{X}$$

```
e = Y – X.dot( theta.T ) # error residual
DJ = - e.dot(X) * 2.0/m  # compute the gradient
theta -= alpha * DJ       # take a step
```

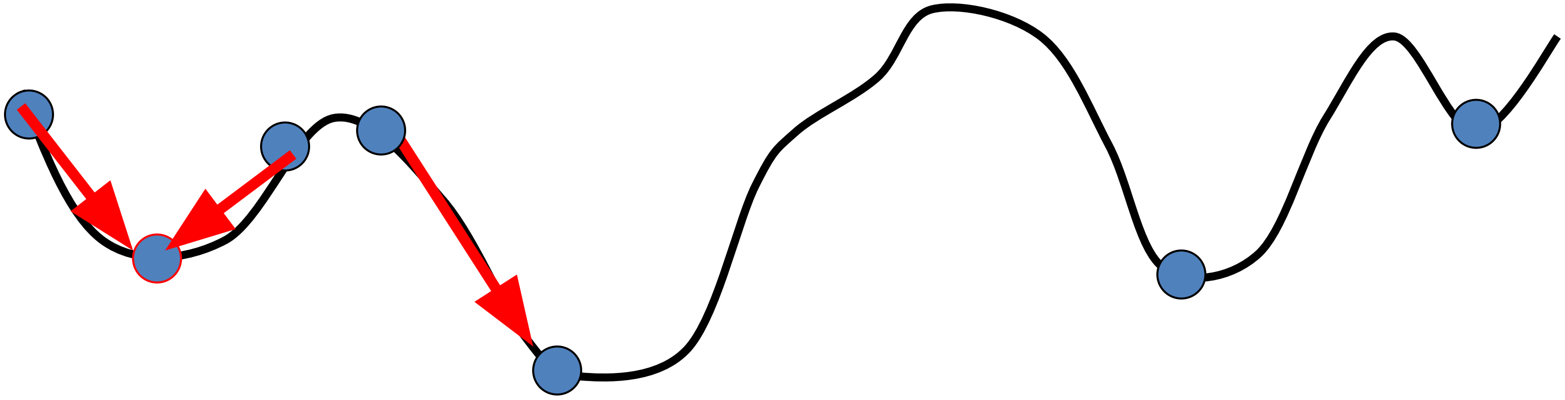
Gradient descent on cost function





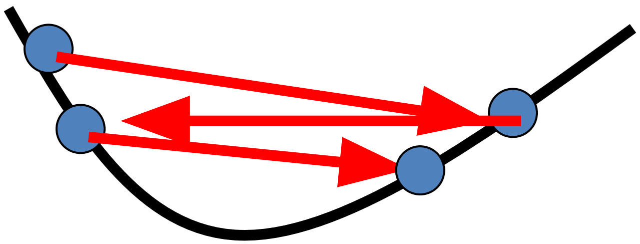
Comments on gradient descent

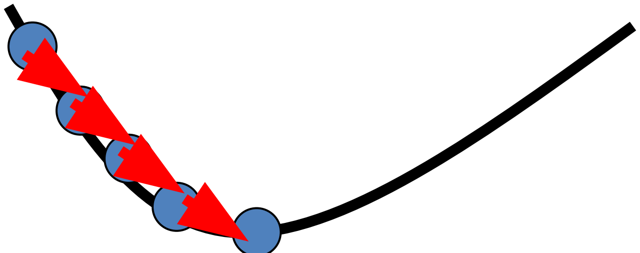
- Very general algorithm
  - We'll see it many times
- Local minima
  - Sensitive to starting point



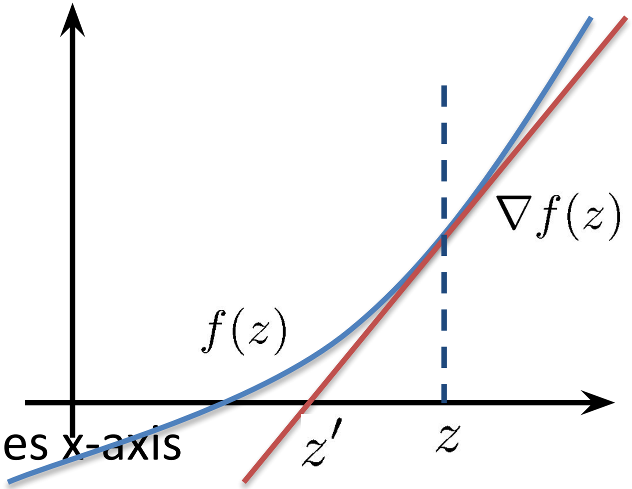
Comments on gradient descent

- Very general algorithm
  - We’ll see it many times
- Local minima
  - Sensitive to starting point
- Step size
  - Too large? Too small? Automatic ways to choose?
  - May want step size to decrease with iteration
  - Common choices:
    - Fixed
    - Linear: C/(iteration)
    - Line search / backoff (Armijo, etc.)
    - Newton’s method





Newton’s method



- Want to find the roots of f(x)
  - “Root”: value of x for which f(x)=0
- Initialize to *some* point x
- Compute the tangent at x & compute where it crosses x-axis

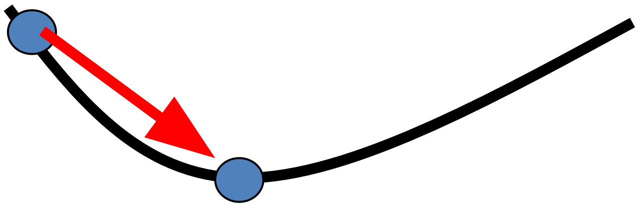
$$\nabla f(z) = \frac{0 - f(z)}{z' - z} \quad \Rightarrow \quad z' = z - \frac{f(z)}{\nabla f(z)}$$

- Optimization: find roots of rJ(µ)

$$\nabla\nabla J(\theta) = \frac{0 - \nabla J(\theta)}{\theta' - \theta} \quad \Rightarrow \quad \theta' = \theta - \frac{\nabla J(\theta)}{\nabla\nabla J(\theta)}$$

(“Step size” ¸ = 1/rrJ ; inverse curvature)

  - If converges, usually very fast
  - Works well for smooth, non-pathological functions, locally quadratic
  - For n large, may be computationally hard: O(n^2) storage, O(n^3) time



(Multivariate:
r J(µ) = gradient vector
r^2 J(µ) = matrix of 2nd derivatives
a/b = a b^{-1}, matrix inverse)

Stochastic / Online gradient descent

- MSE

$$J(\underline{\theta}) = \frac{1}{m}\sum_j J_j(\underline{\theta}), \qquad J_j(\underline{\theta}) = (y^{(j)} - \underline{\theta}\cdot \underline{x}^{(j)\,T})^2$$

- Gradient

$$\nabla J(\underline{\theta}) = \frac{1}{m}\sum_j \nabla J_j(\underline{\theta}) \qquad \nabla J_j(\underline{\theta}) = (y^{(j)} - \underline{\theta}\cdot \underline{x}^{(j)\,T})\cdot [x_0^{(j)}\, x_1^{(j)} \ldots]$$

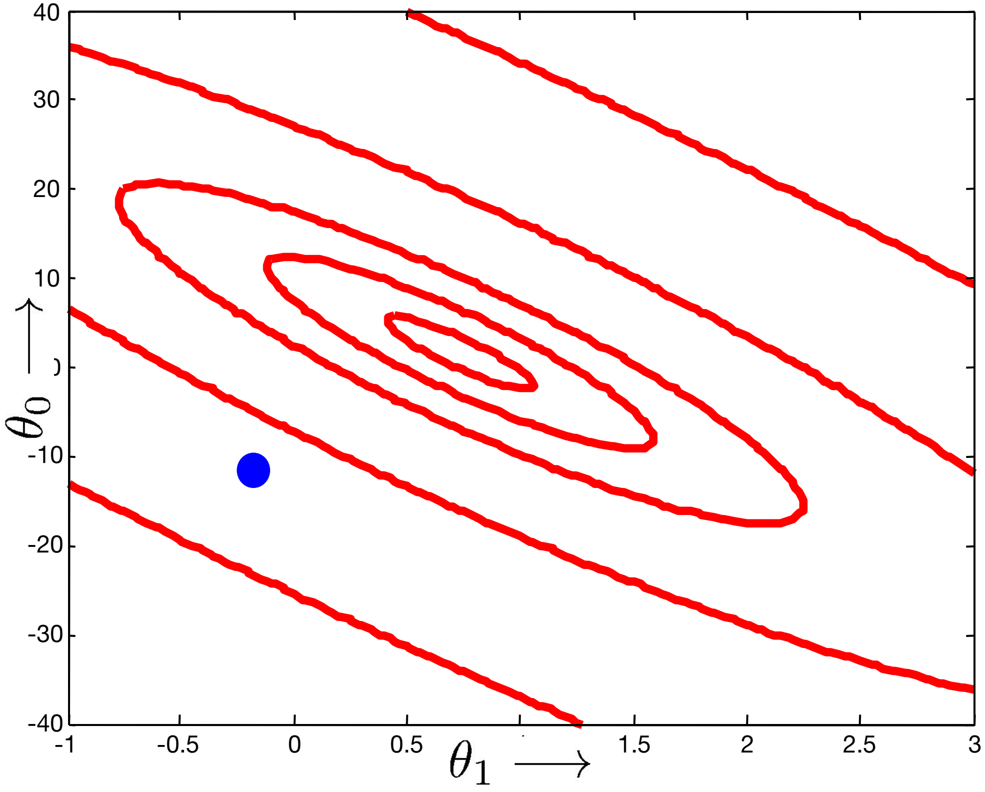
- Stochastic (or “online”) gradient descent:
  - Use updates based on individual datum j, chosen at random
  - At optima, $\mathbb{E}\big[\nabla J_j(\underline{\theta})\big] = \nabla J(\underline{\theta}) = 0$

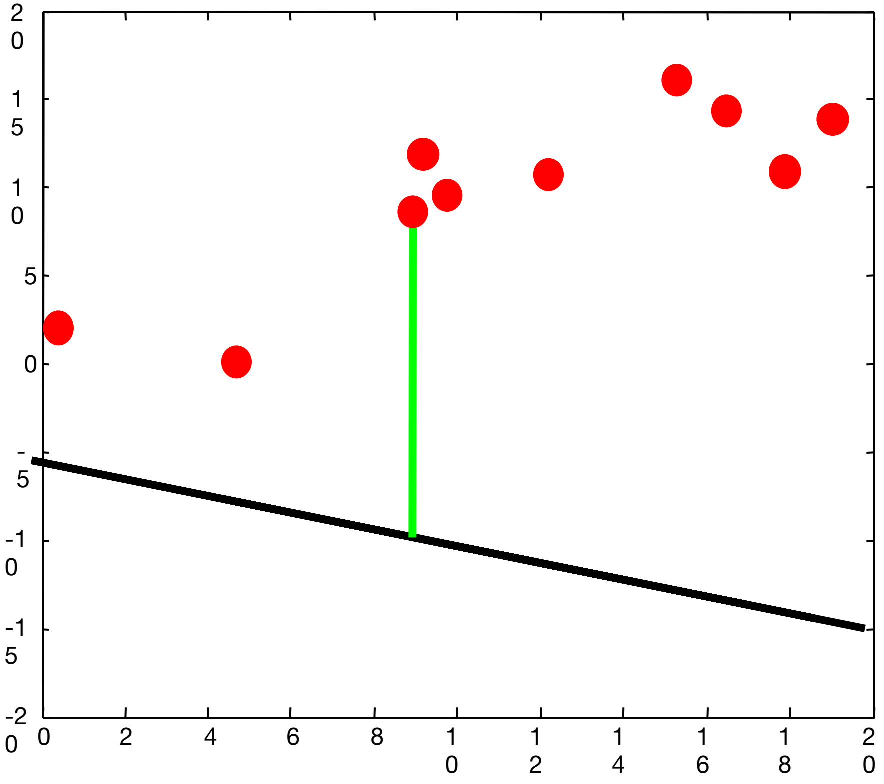
    (average over the data)

Online gradient descent

- Update based on one datum, and its residual, at a time

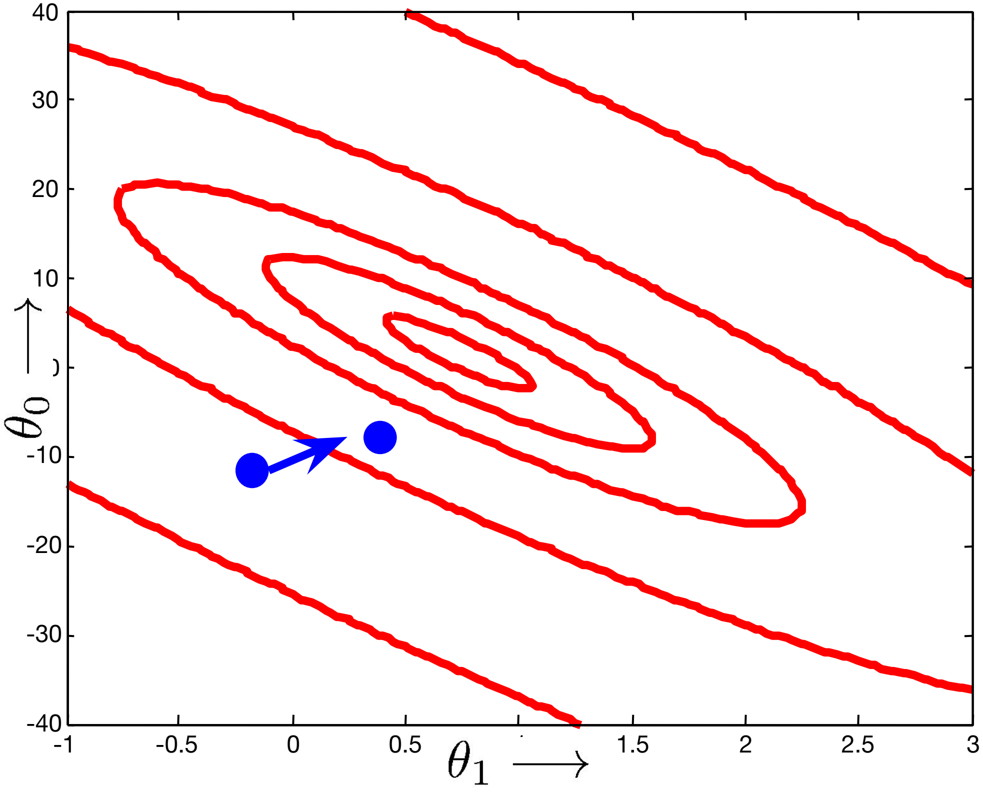
```
Initialize $\theta$
Do {
 for j=1:m
   $\theta \leftarrow \theta - \alpha \nabla_\theta J_j(\theta)$
} while (not done)
```

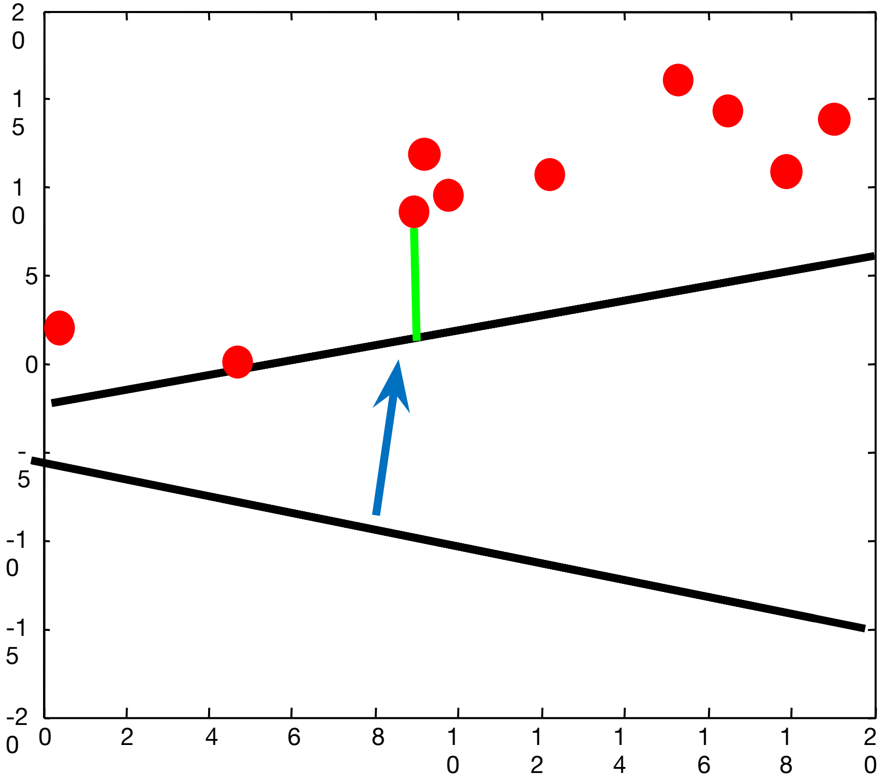




Online gradient descent

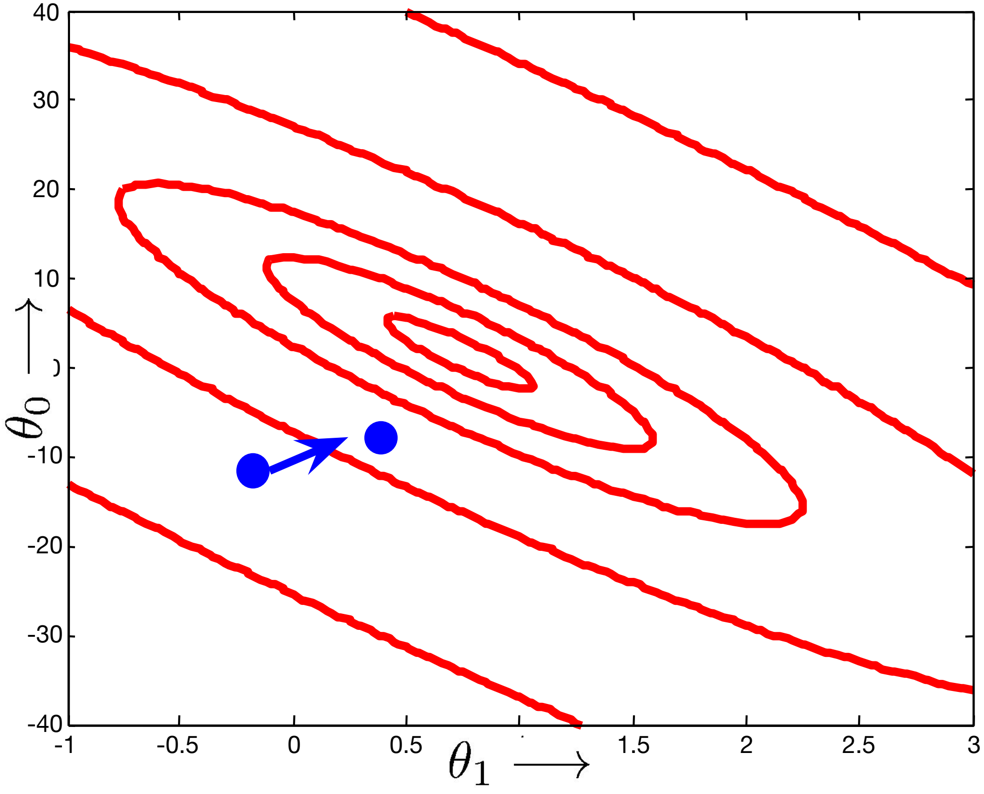
```
Initialize $\theta$
Do {
 for j=1:m
   $\theta \leftarrow \theta - \alpha\nabla_{\theta}J_{j}(\theta)$
} while (not done)
```

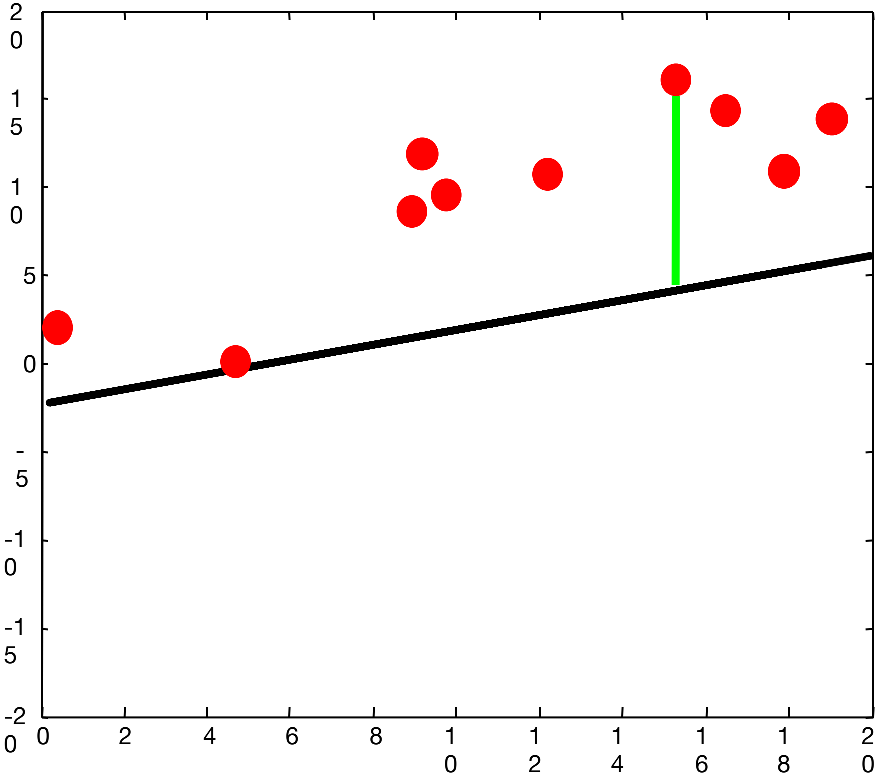




Online gradient descent

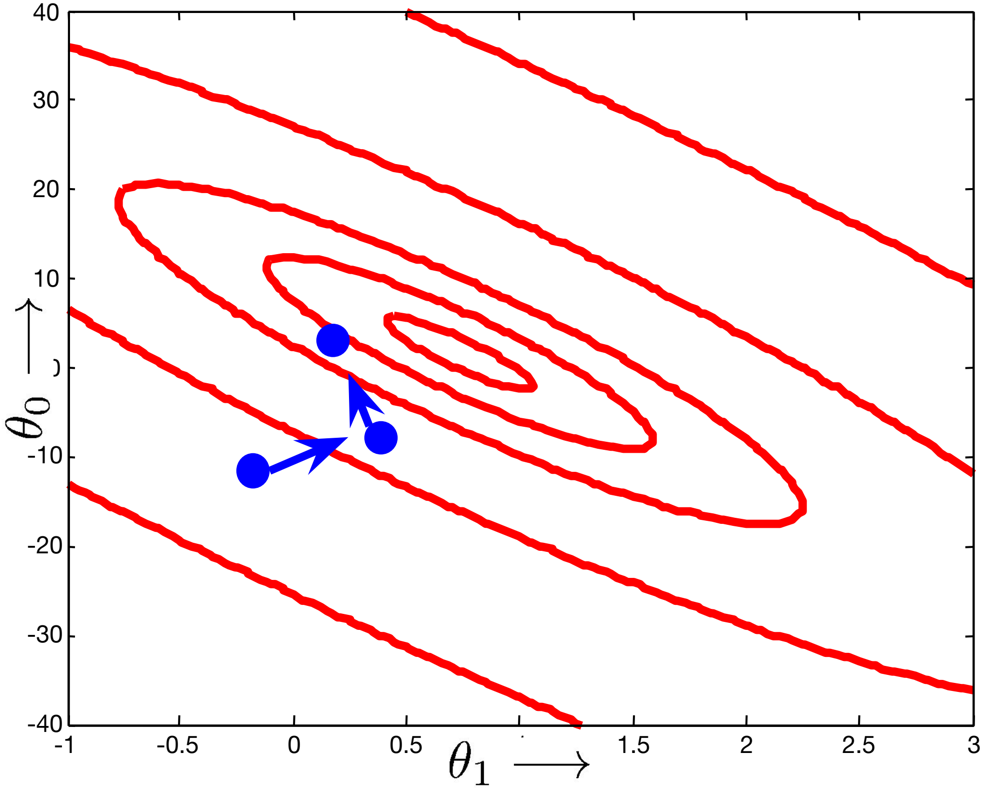
```
Initialize θ
Do {
 for j=1:m
   $\theta \leftarrow \theta - \alpha \nabla_\theta J_j(\theta)$
} while (not done)
```

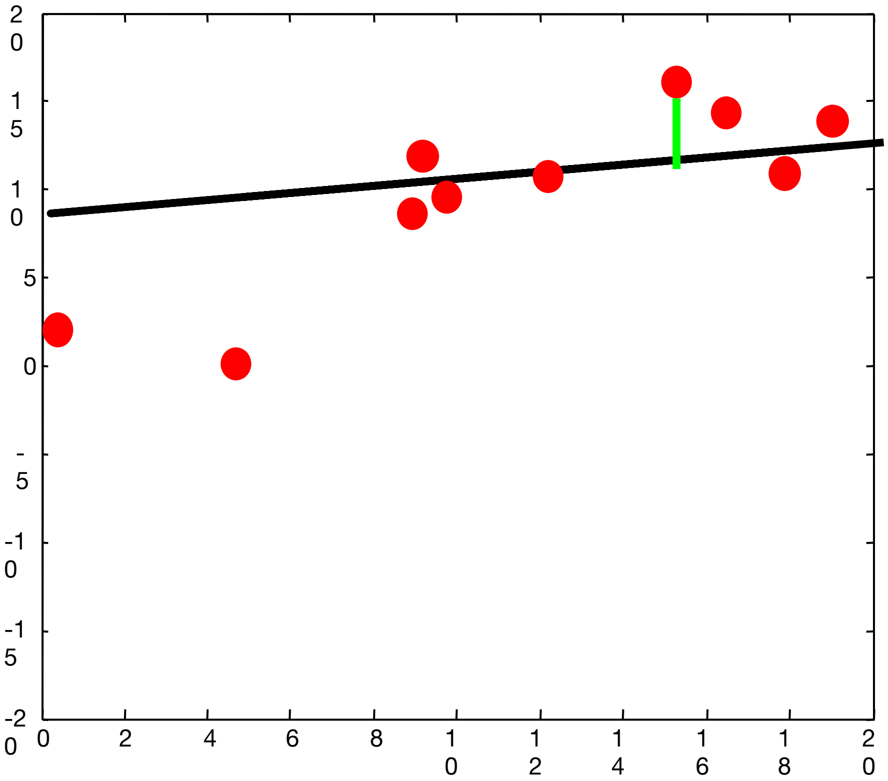




Online gradient descent

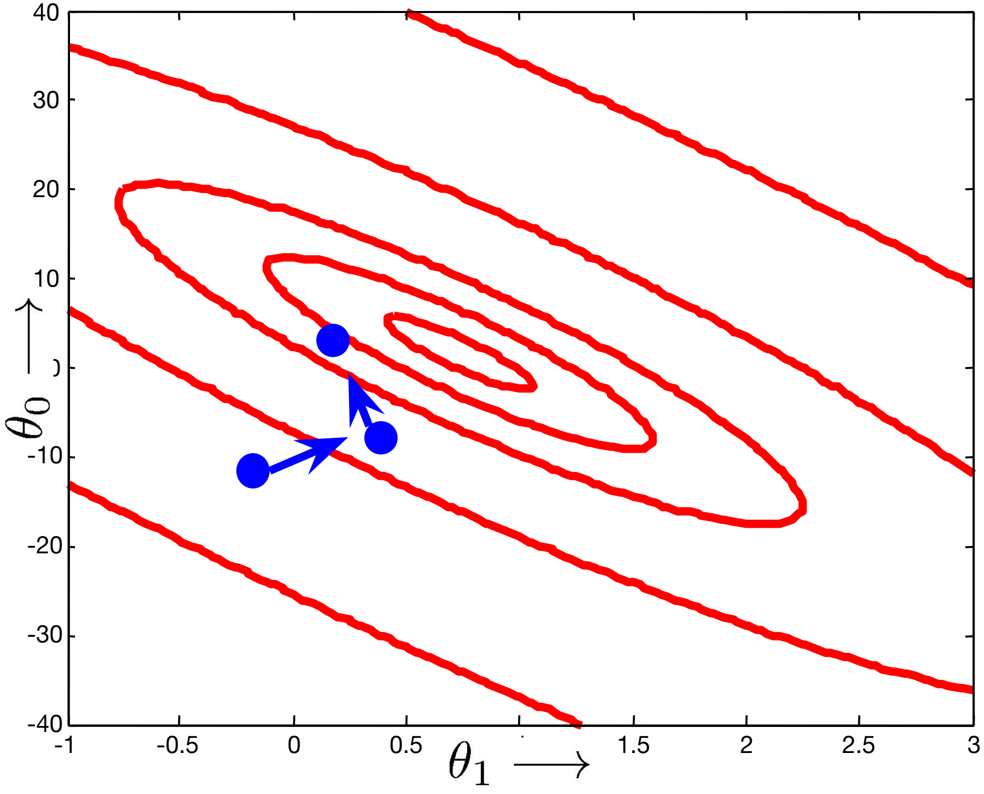
```
Initialize θ
Do {
 for j=1:m
   $\theta \leftarrow \theta - \alpha \nabla_\theta J_j(\theta)$
} while (not done)
```

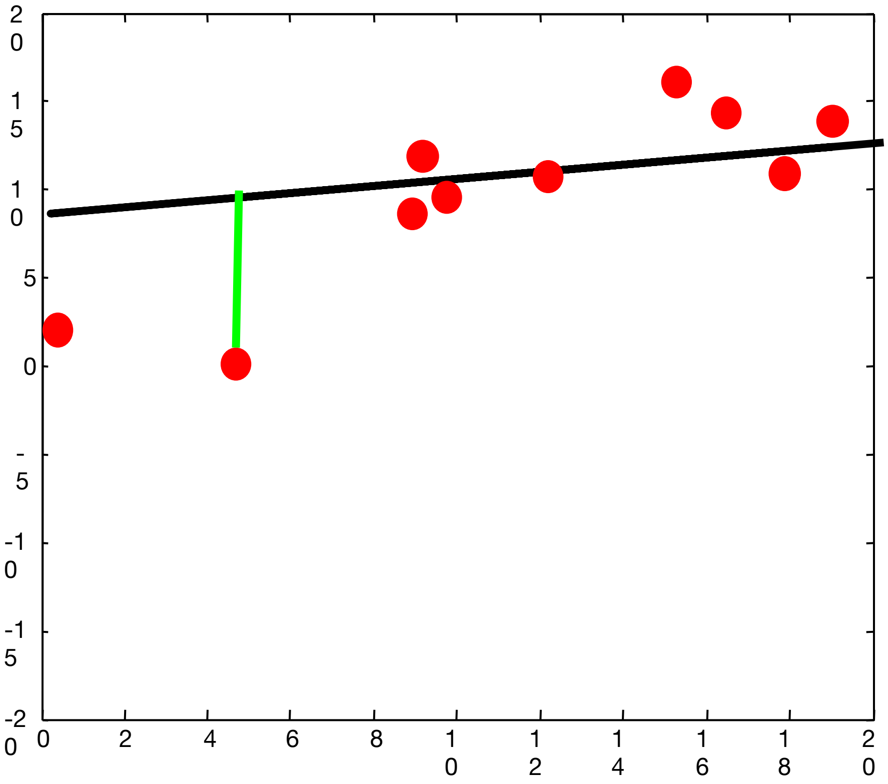




Online gradient descent

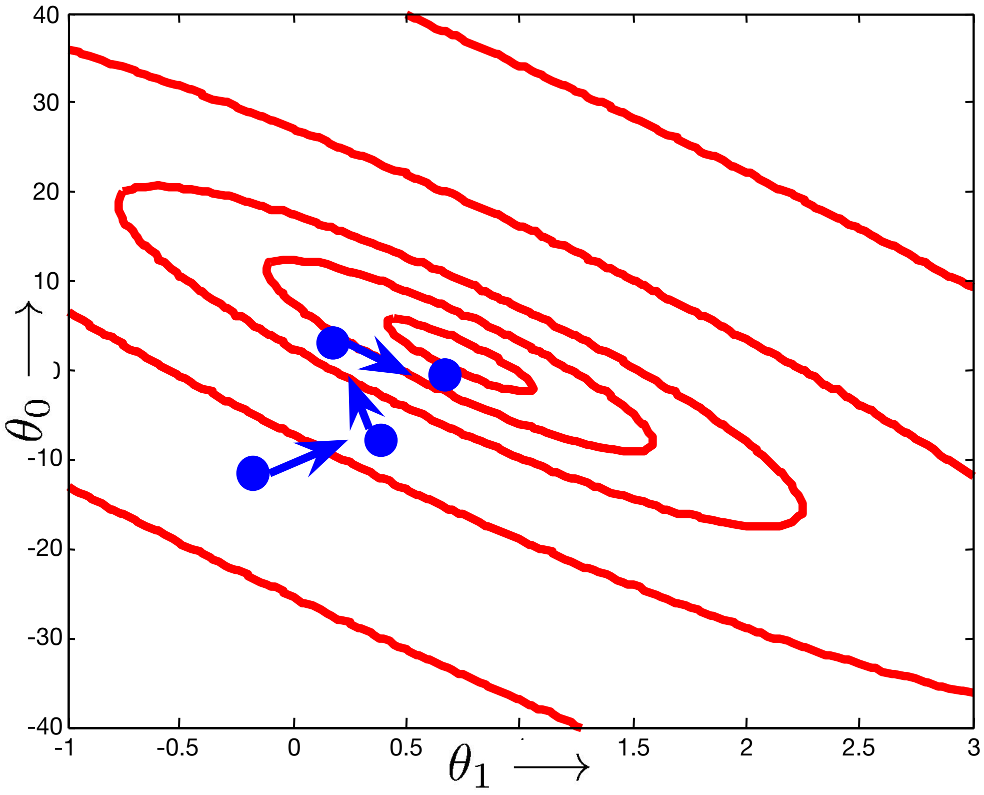
```
Initialize $\theta$
Do {
 for j=1:m
   $\theta \leftarrow \theta - \alpha \nabla_{\theta} J_{j}(\theta)$
} while (not done)
```

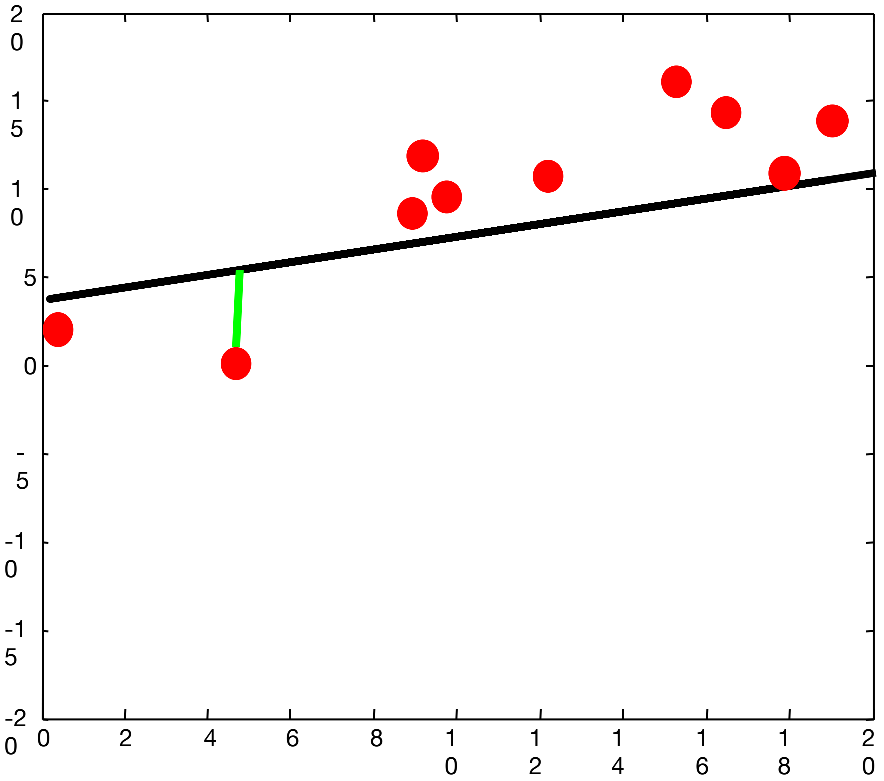




Online gradient descent

```
Initialize $\theta$
Do {
 for j=1:m
   $\theta \leftarrow \theta - \alpha \nabla_\theta J_j(\theta)$
} while (not done)
```





Online gradient descent

```
Initialize θ
Do {
 for j=1:m
   $\theta \leftarrow \theta - \alpha\nabla_\theta J_j(\theta)$
} while (not done)
```

- Benefits
  - Lots of data = many more updates per pass
  - Computationally faster
- Disadvantages
  - No longer strictly “descent”
  - Stopping conditions may be harder to evaluate

(Can use “running estimates” of J(.), etc.)

$$J_j(\underline{\theta}) = (y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)^T})^2$$

$$\nabla J_j(\underline{\theta}) = -2(y^{(j)} - \underline{\theta} \cdot \underline{x}^{(j)^T}) \cdot [x_0^{(j)} x_1^{(j)} \ldots]$$

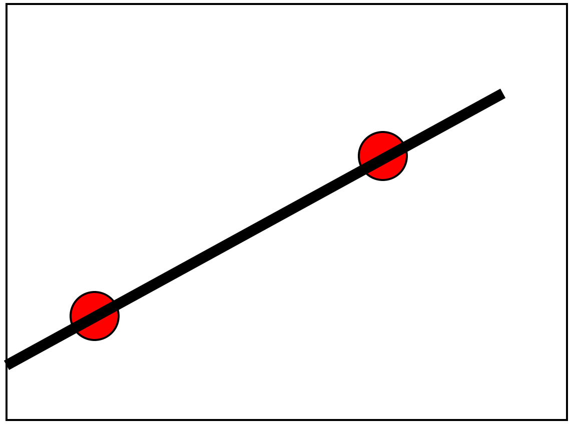
Machine Learning

- Linear Regression via Least Squares
- Gradient Descent Algorithms
- Direct Minimization of Squared Error
- Regression with Non-linear Features
- Bias, Variance, & Validation
- Regularized Linear Regression

MSE Minimum

- Consider a simple problem
  - One feature, two data points
  - Two unknowns: θ_0, θ_1
  - Two equations:

$$y^{(1)} = \theta_0 + \theta_1 x^{(1)}$$
$$y^{(2)} = \theta_0 + \theta_1 x^{(2)}$$



- Can solve this system directly:

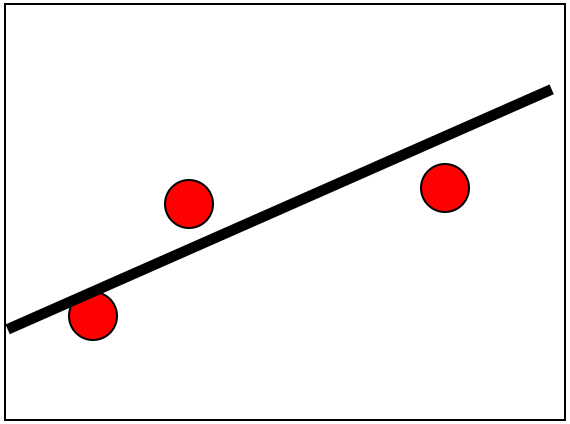
$$\underline{y}^T = \underline{\theta}\,\underline{X}^T \qquad \Rightarrow \qquad \underline{\hat{\theta}} = y^T (\underline{X}^T)^{-1}$$

- However, most of the time, m > n
  - There may be no linear function that hits all the data exactly
  - Instead, solve directly for minimum of MSE function

MSE Minimum

- Simplify with some algebra:

$$\nabla J(\underline{\theta}) = -\frac{2}{m}(\underline{y}^T - \underline{\theta X}^T) \cdot \underline{X} \quad = \quad \underline{0}$$



$$\underline{y}^T \underline{X} - \underline{\theta X}^T \cdot \underline{X} \quad = \quad \underline{0}$$

$$\underline{y}^T \underline{X} = \underline{\theta X}^T \cdot \underline{X}$$

$$\underline{\theta} \quad = \quad \underline{y}^T \underline{X}(\underline{X}^T \underline{X})^{-1}$$

- $X (X^T X)^{-1}$ is called the “pseudo-inverse”

- If X^T is square and full rank, this is the inverse
- If m > n: overdetermined; gives minimum MSE fit

Matlab MSE

- This is easy to solve in Matlab…

$$\underline{\theta} \quad = \quad \underline{y}^T \, \underline{X} (\underline{X}^T \, \underline{X})^{-1}$$

```
%  y = [y1 ; … ; ym]
%  X = [x1_0 … x1_m ; x2_0 … x2_m ; …]


% Solution 1: “manual”
   th = y’ * X * inv(X’ * X);


% Solution 2: “mrdivide”
   th = y’ / X’; % th*X’ = y  =>  th = y/X’
```

Python MSE

- This is easy to solve in Python / NumPy…

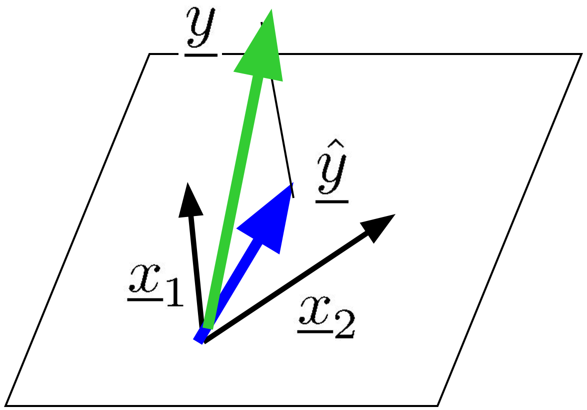
$$\underline{\theta} \quad = \quad \underline{y}^T \, \underline{X}(\underline{X}^T \, \underline{X})^{-1}$$

```
#  y = np.matrix( [[y1], … , [ym]] )
#  X = np.matrix( [[x1_0 … x1_n], [x2_0 … x2_n],
…] )

# Solution 1: “manual”
   th = y.T * X * np.linalg.inv(X.T * X)


# Solution 2: “least squares solve”
   th = np.linalg.lstsq(X, Y)
```

Normal equations

$$\nabla J(\underline{\theta}) = 0 \quad \Rightarrow \quad (\underline{y}^T - \underline{\theta X}^T) \cdot \underline{X} \quad = \quad \underline{0}$$

- Interpretation:
  - (y - θ X) = (y - yhat) is the vector of errors in each example
  - X are the features we have to work with for each example
  - Dot product = 0: orthogonal

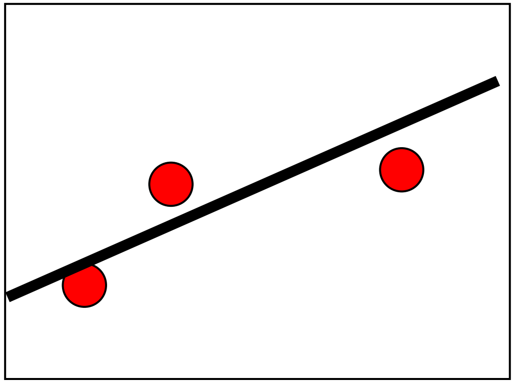


$$\underline{y}^T = [y^{(1)} \ldots y^{(m)}]$$
$$\underline{x}_i = [x_i^{(1)} \ldots x_i^{(m)}]$$

Normal equations

$$\nabla J(\underline{\theta}) = 0 \quad \Rightarrow \quad (\underline{y}^T - \underline{\theta X^T}) \cdot \underline{X} \quad = \quad \underline{0}$$

- Interpretation:
  - (y - θ X) = (y - yhat) is the vector of errors in each example
  - X are the features we have to work with for each example
  - Dot product = 0: orthogonal
- Example:



$$\underline{y} = [1 \quad 3 \quad 3]^T$$

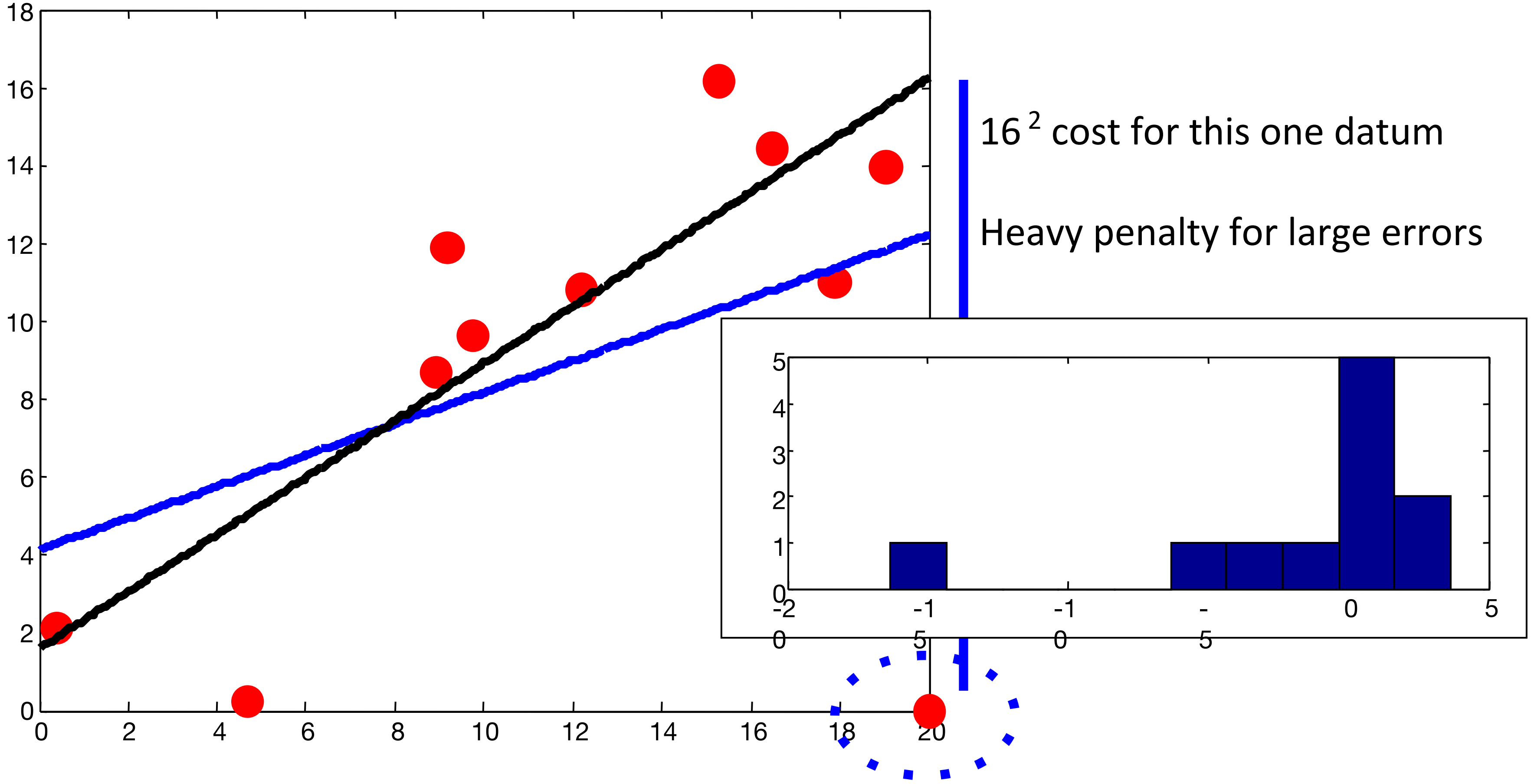
$$\underline{x}_0 = [1 \quad 1 \quad 1]^T \qquad \theta = [1.00 \quad 0.57]$$

$$\underline{x}_1 = [1 \quad 2 \quad 4]^T$$

$$\underline{e} = (y - \hat{y}) = [-0.57 \quad 0.85 \quad -0.28]^T$$

Effects of MSE choice

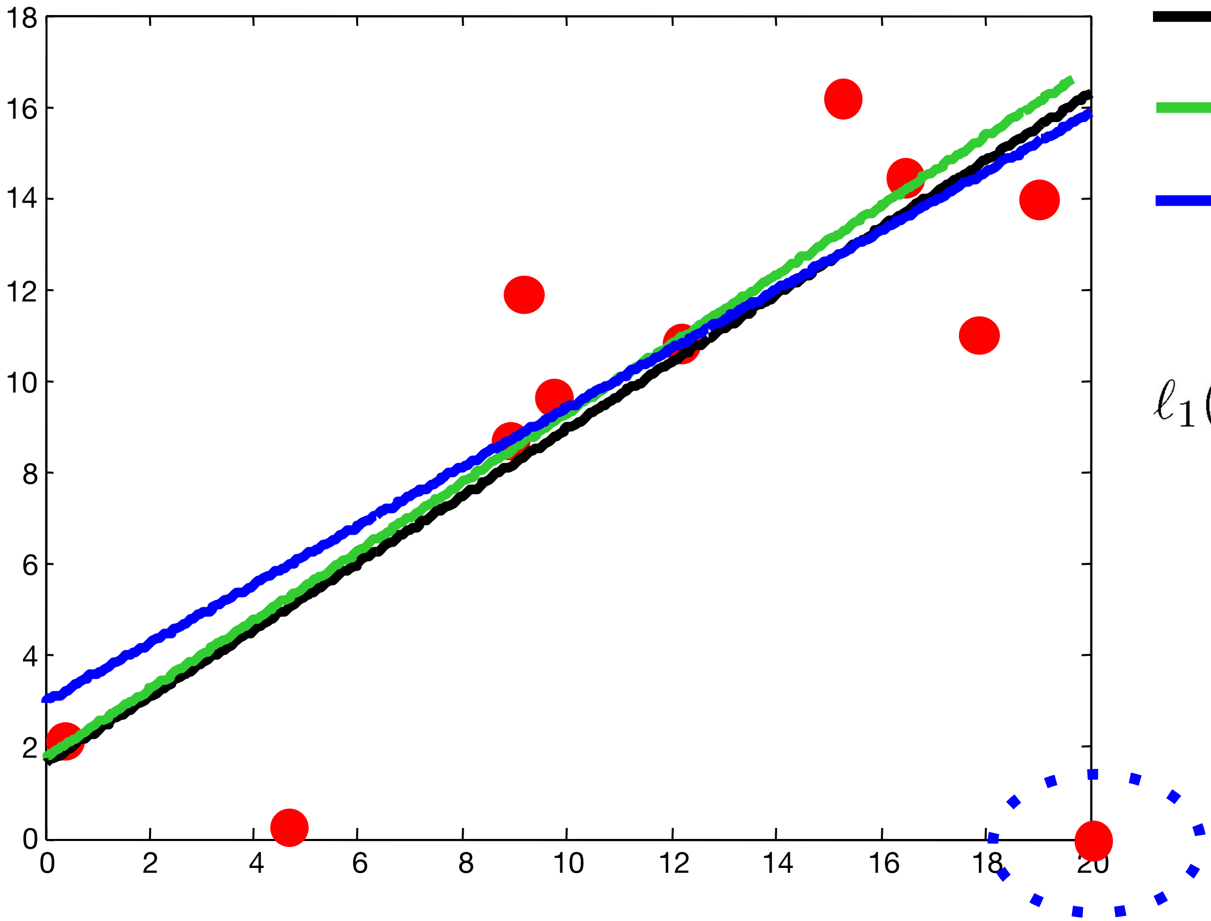
- Sensitivity to outliers



16^2 cost for this one datum

Heavy penalty for large errors

L1 error: Mean Absolute Error



$$\ell_1(\underline{\theta}) = \sum_j |y^{(j)} - \hat{y}(x^{(j)})|$$

$$= \sum_j |y - \underline{\theta} \cdot \underline{x}^T|$$

Cost functions for regression

$\ell_2 \;:\; (y - \hat{y})^2$ **(MSE)**

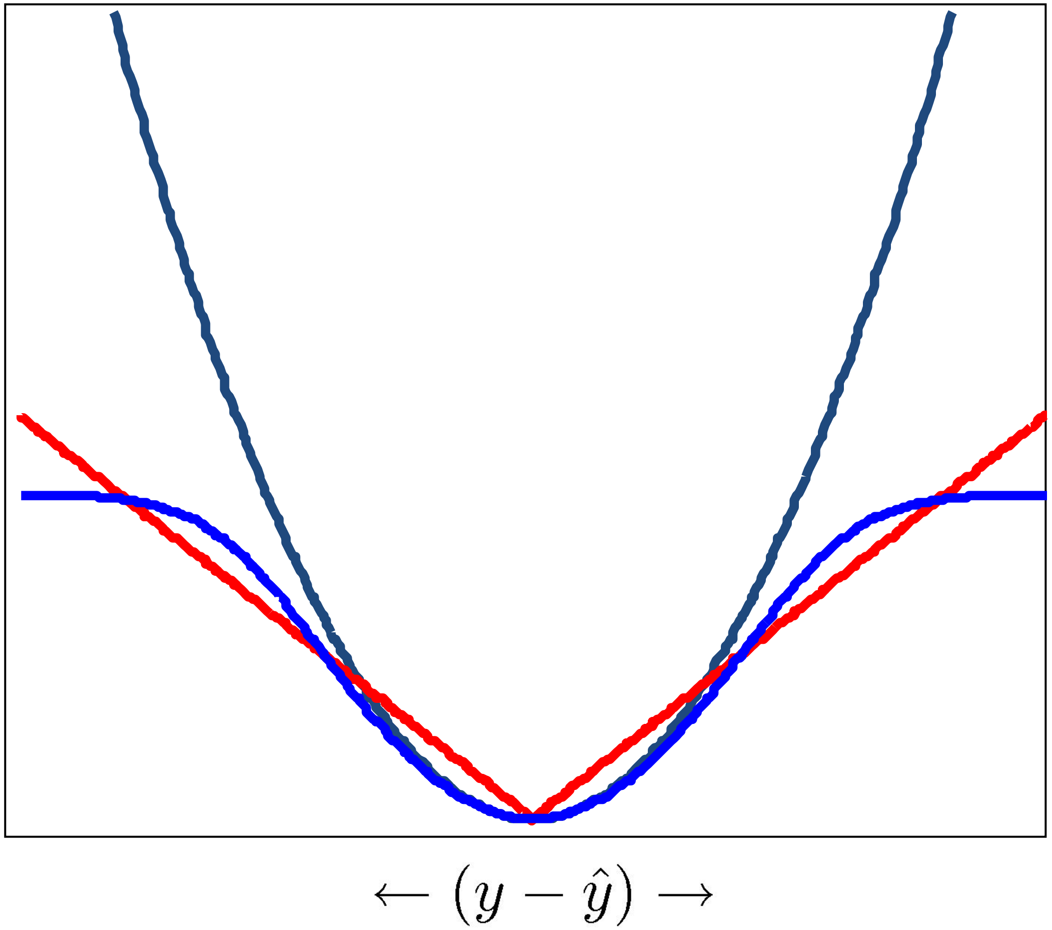
$\ell_1 \;:\; |y - \hat{y}|$ **(MAE)**

Something else entirely…

$$c - \log(\exp(-(y - \hat{y})^2) + c)$$

(???)

Arbitrary functions cannot be solved in closed form
- use gradient descent



Machine Learning

Linear Regression via Least Squares

Gradient Descent Algorithms

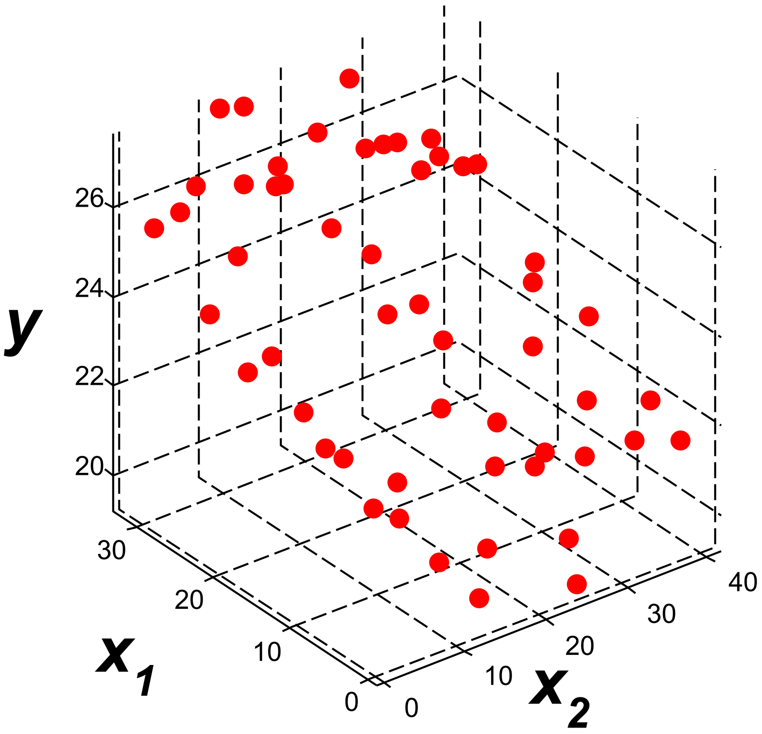
Direct Minimization of Squared Error

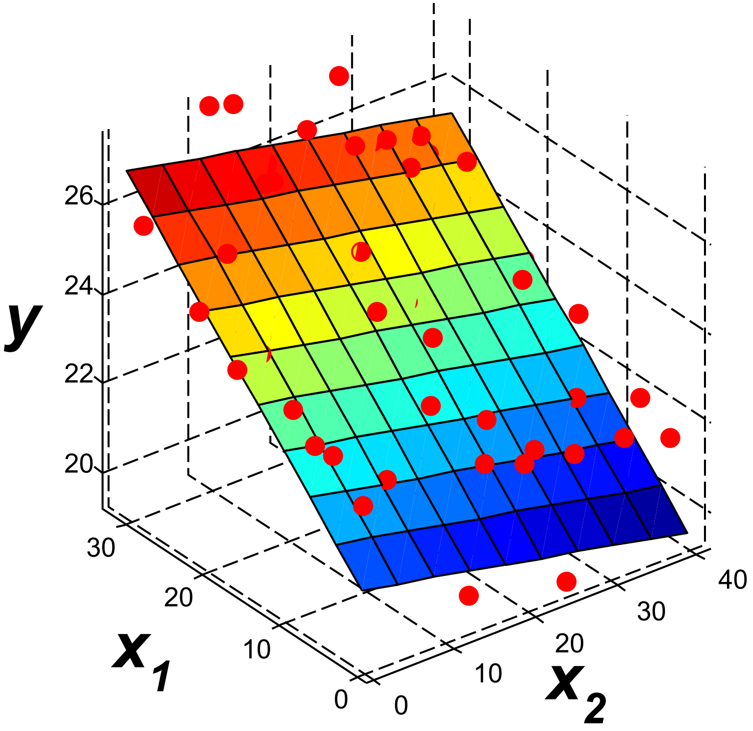
Regression with Non-linear Features

Bias, Variance, & Validation

Regularized Linear Regression

More dimensions?



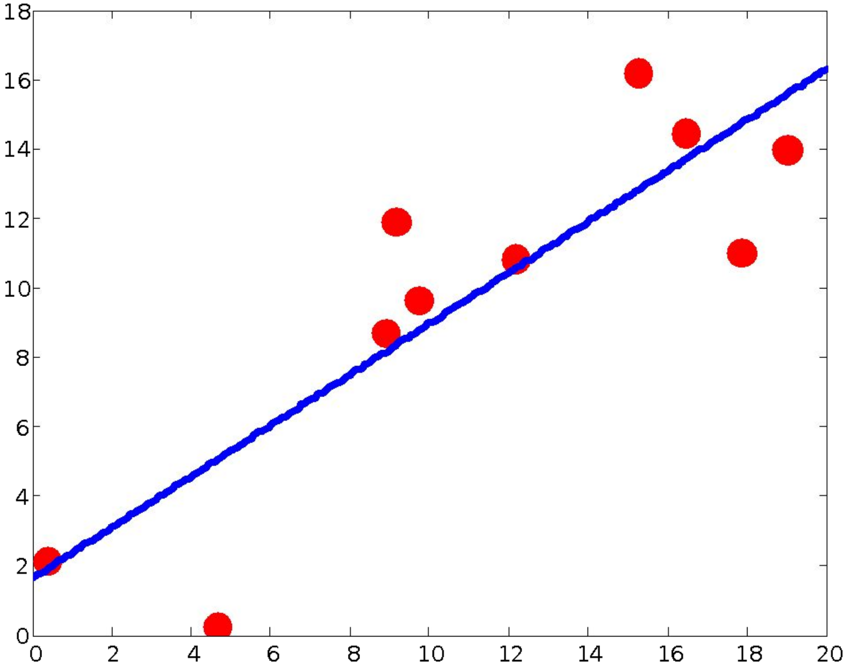


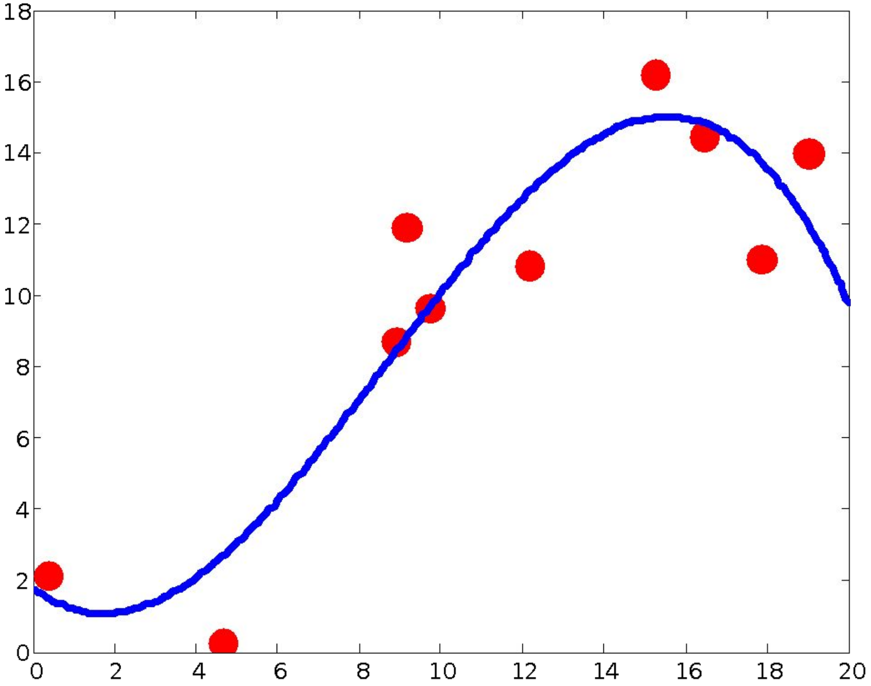
$$\hat{y}(x) = \underline{\theta} \cdot \underline{x}^T$$

$$\underline{\theta} = [\theta_0 \ \theta_1 \ \theta_2]$$
$$\underline{x} = [1 \ x_1 \ x_2]$$

Nonlinear functions

- What if our hypotheses are not lines?
  - Ex: higher-order polynomials





Nonlinear functions

- Single feature x, predict target y:

$$D = \{(x^{(j)}, y^{(j)})\}$$

$$\hat{y}(x) = \theta_0 + \theta_1\, x + \theta_2\, x^2 + \theta_3\, x^3$$

$$\Downarrow$$

Add features:

$$\Downarrow$$

$$D = \{([x^{(j)}, (x^{(j)})^2, (x^{(j)})^3], y^{(j)})\}$$

$$\hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$

Linear regression in new features

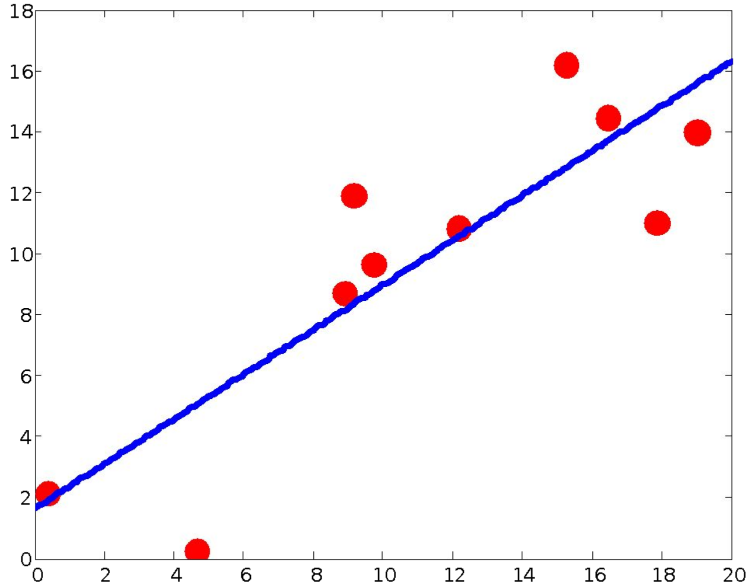
- Sometimes useful to think of “feature transform”

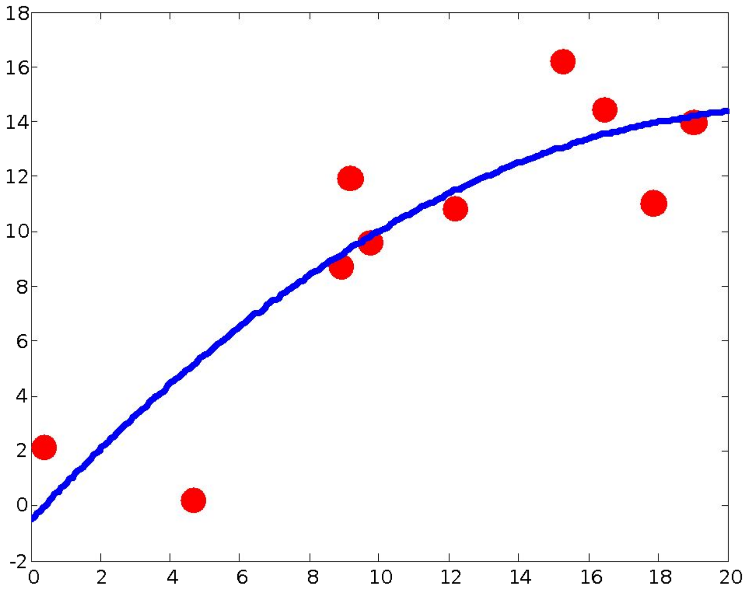
$$\Phi(x) = \left[1\,,\, x\,,\, x^2\,,\, x^3\,,\, \ldots\right]$$

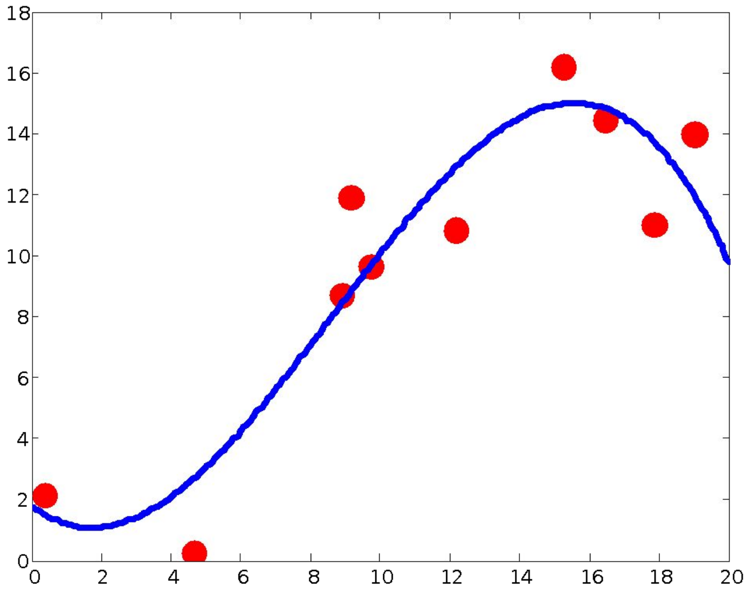
$$\hat{y}(x) = \underline{\theta} \cdot \Phi(x)$$

Higher-order polynomials

- Fit in the same way
- More “features”







Features

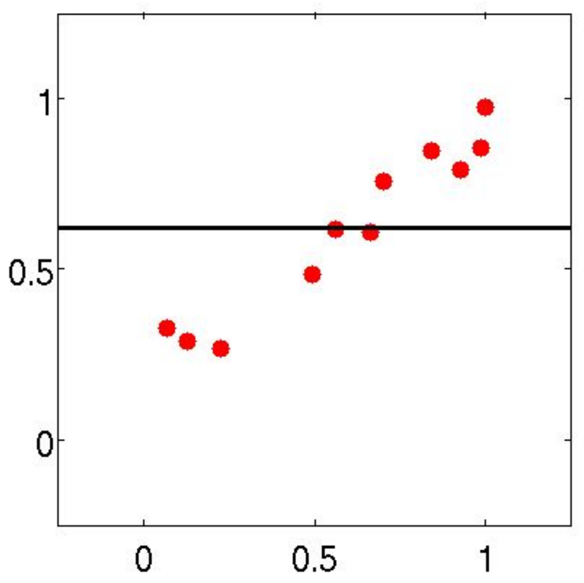
- In general, can use any features we think are useful

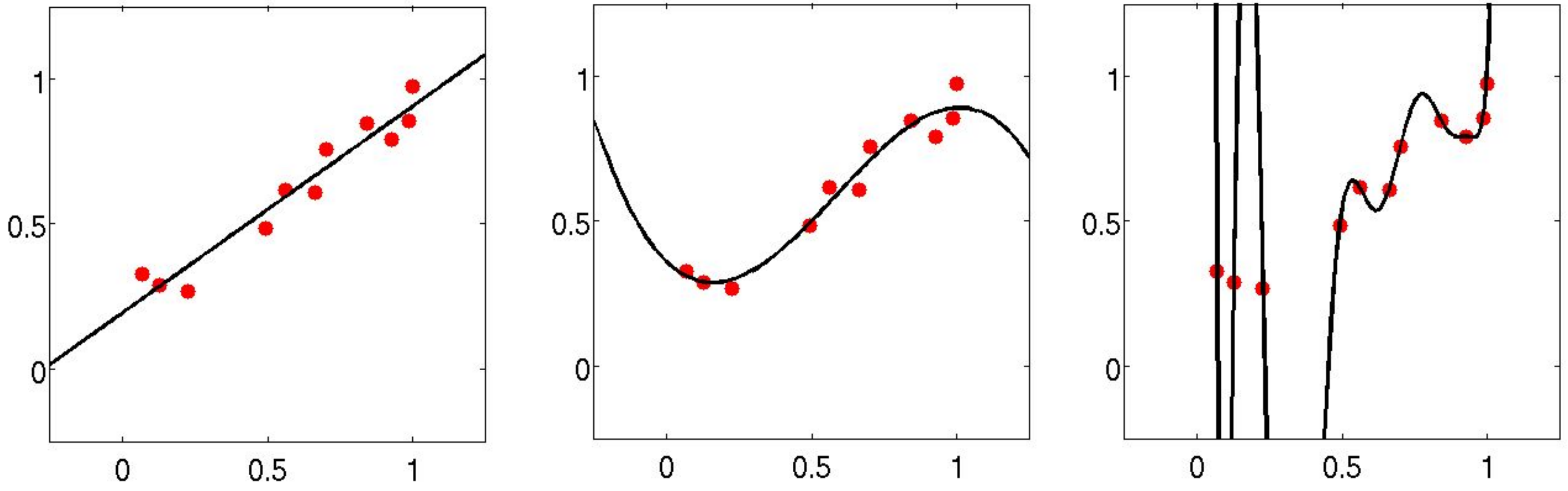
- Other information about the problem
  - Anything you can encode as fixed-length vectors of numbers
- Polynomial functions
  - Features [1, x, x^2, x^3, …]
- Other functions
  - 1/x, sqrt(x), x_1 * x_2, …

- "Linear regression" = linear in the parameters
  - Features we can make as complex as we want!

Higher-order polynomials

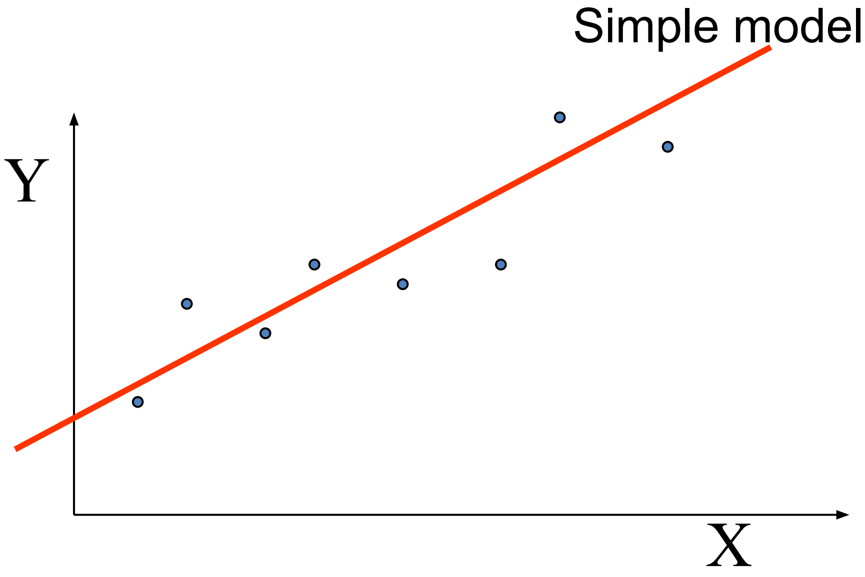
- Are more features better?
- “Nested” hypotheses
  - 2nd order more general than 1st,
  - 3rd order more general than 2nd, …
- Fits the observed data better

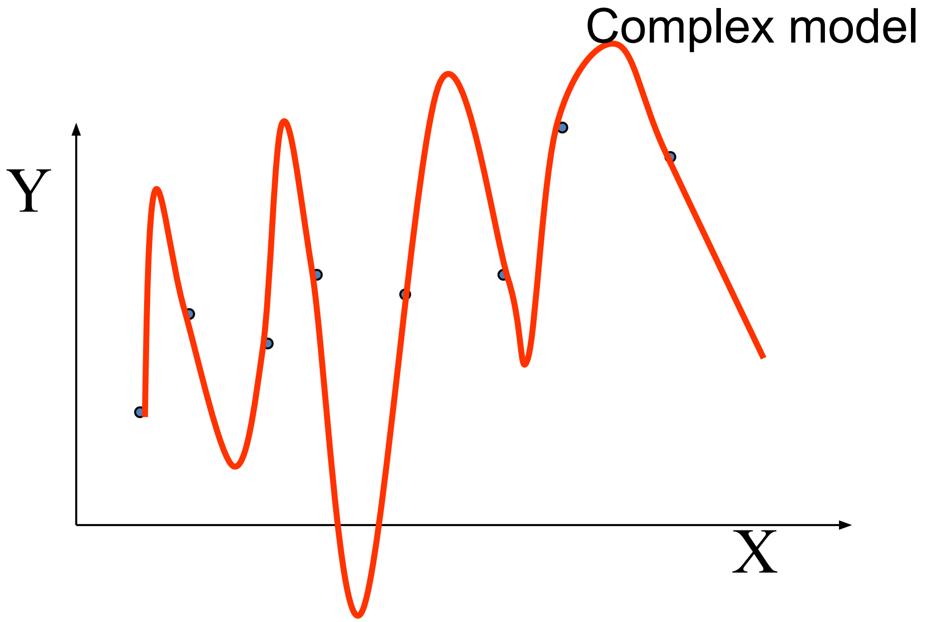




Overfitting and complexity

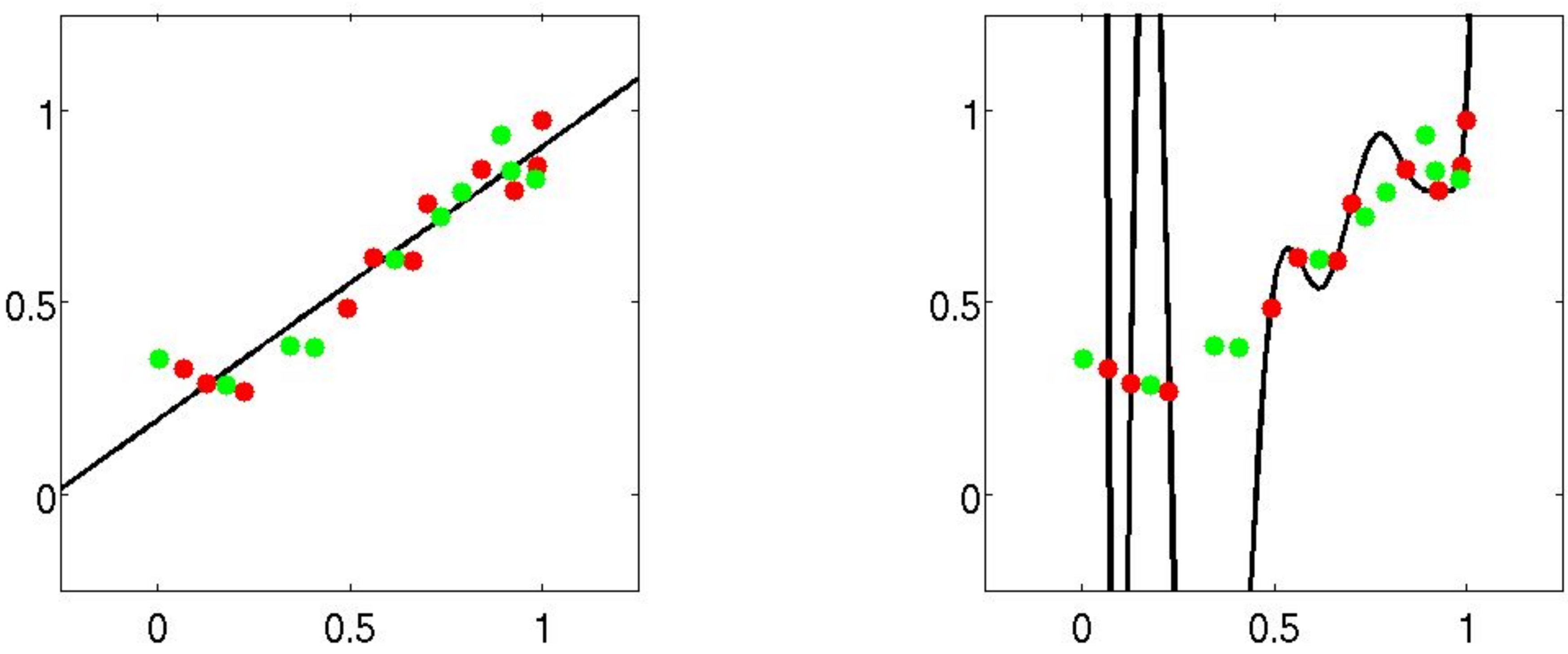
- More complex models will always fit the training data better
- But they may “overfit” the training data, learning complex relationships that are not really present





Test data

- After training the model
- Go out and get more data from the world
  - New observations (x,y)
- How well does our model perform?

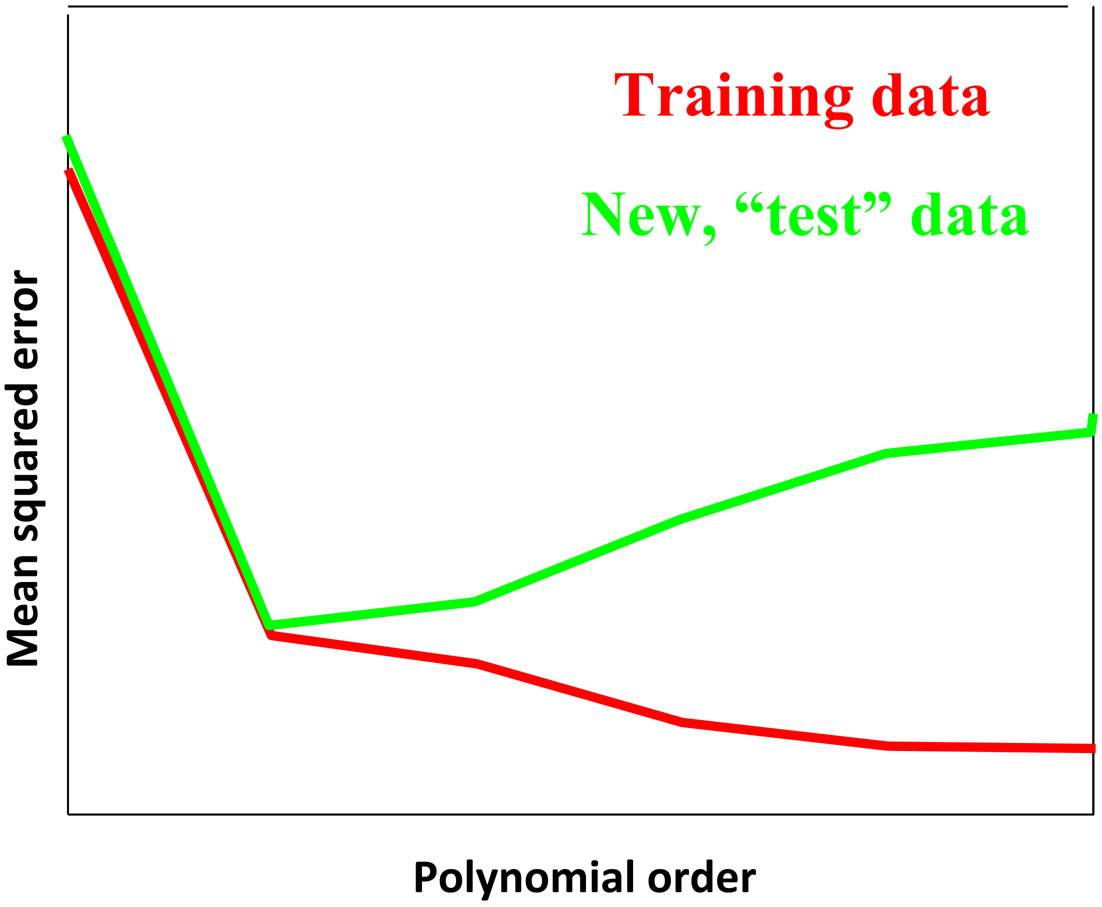


Training versus test error

- Plot MSE as a function of model complexity
  - Polynomial order

- Decreases
  - More complex function fits training data better

- What about new data?



- 0th to 1st order
  - Error decreases
  - Underfitting
- Higher order
  - Error increases
  - Overfitting

Machine Learning

Linear Regression via Least Squares

Gradient Descent Algorithms

Direct Minimization of Squared Error

Regression with Non-linear Features

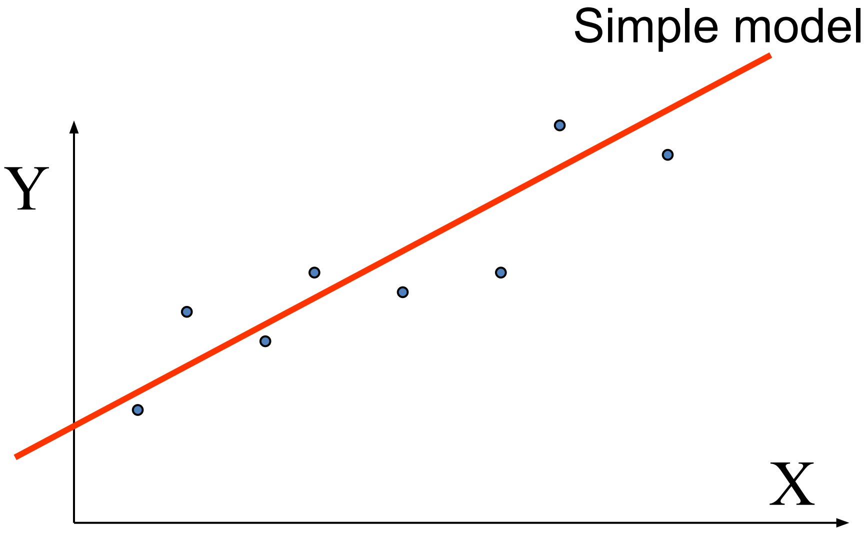
Bias, Variance, & Validation

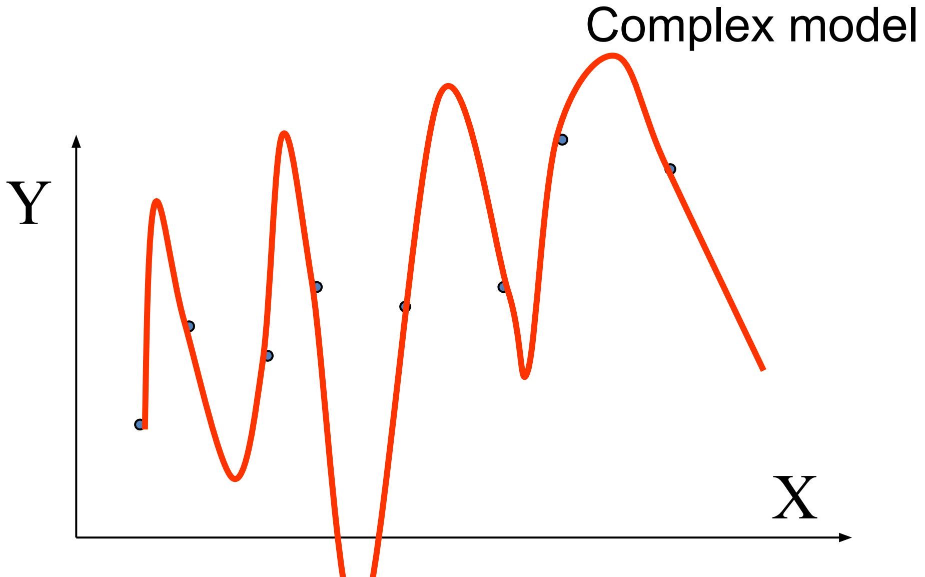
Regularized Linear Regression

Inductive bias

- The assumptions needed to predict examples we haven’t seen
- Makes us “prefer” one model over another
- Polynomial functions; smooth functions; etc

- Some bias is necessary for learning!

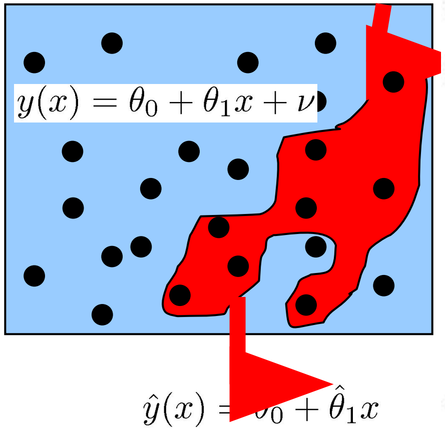




Bias & variance

“The world”

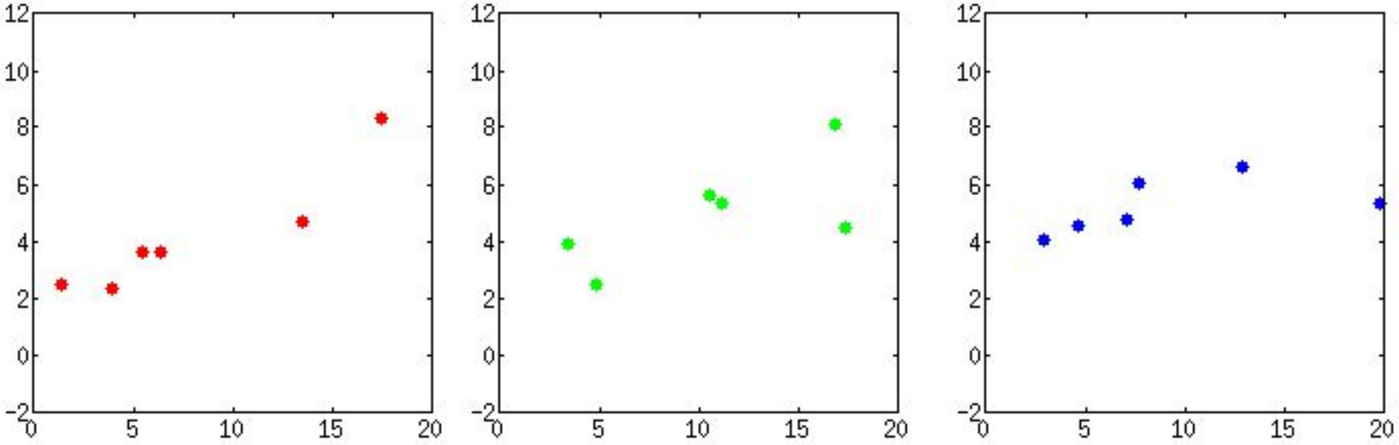
Data we observe



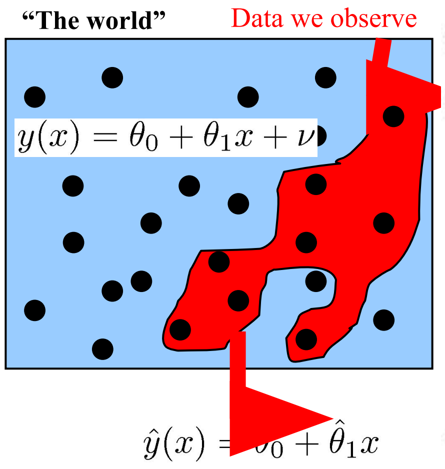
$y(x) = \theta_0 + \theta_1 x + \nu$

$\hat{y}(x) = \hat{\theta}_0 + \hat{\theta}_1 x$

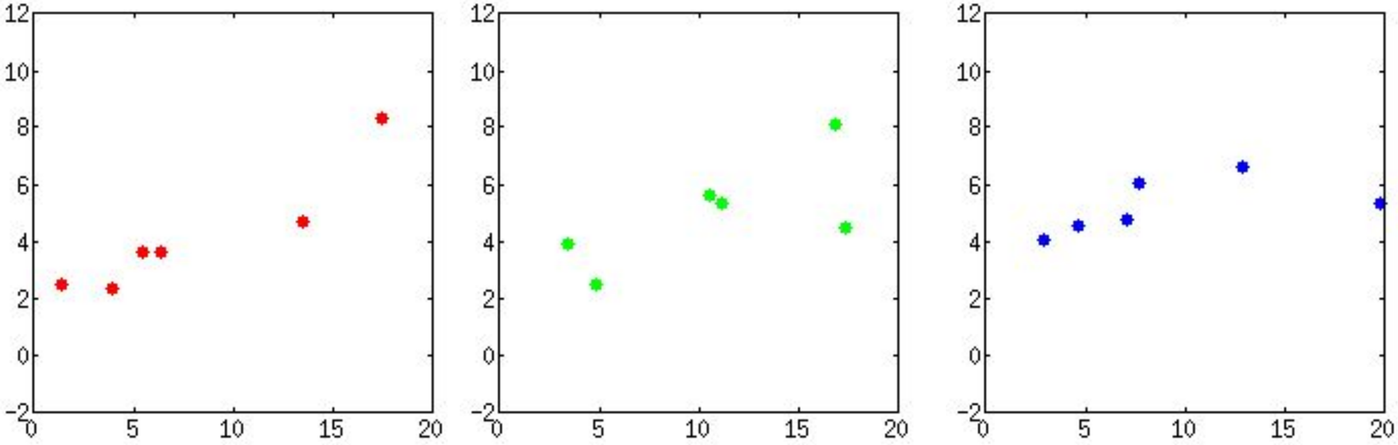
Three different possible data sets:



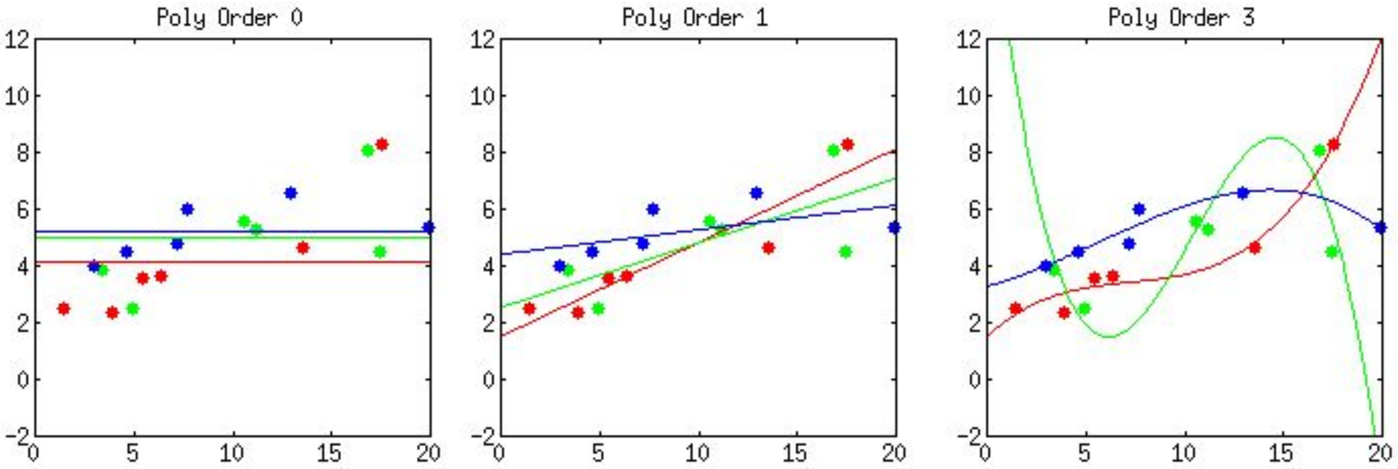
Bias & variance


"The world"
Data we observe
$y(x) = \theta_0 + \theta_1 x + \nu$
$\hat{y}(x) = \hat{\theta}_0 + \hat{\theta}_1 x$

Three different possible data sets:



Each would give different predictors for any polynomial degree:



Detecting overfitting

- Overfitting effect
  - Do better on training data than on future data
  - Need to choose the “right” complexity

- One solution: “Hold-out” data
- Separate our data into two sets
  - Training
  - Test
- Learn only on training data
- Use test data to estimate generalization quality
  - Model selection

- All good competitions use this formulation
  - Often multiple splits: one by judges, then another by you

Model selection

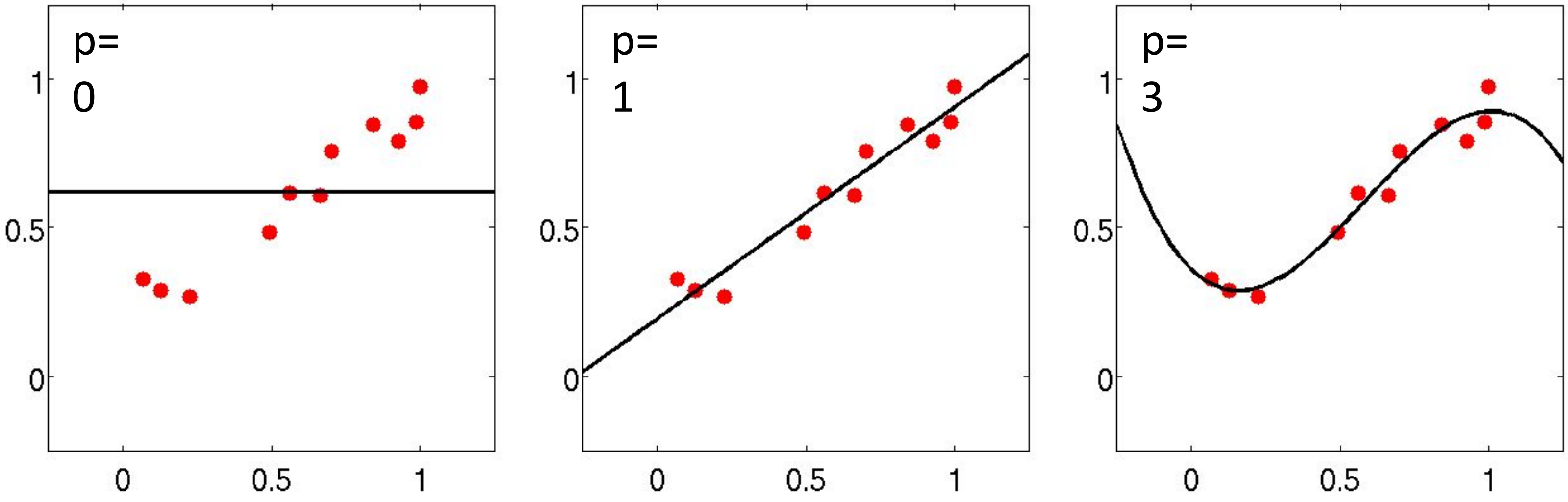
- Which of these models fits the data best?
  - p=0 (constant); p=1 (linear); p=3 (cubic); …
- Or, should we use KNN? Other methods?

- Model selection problem
  - Can’t use training data to decide (esp. if models are nested!)

- Want to estimate

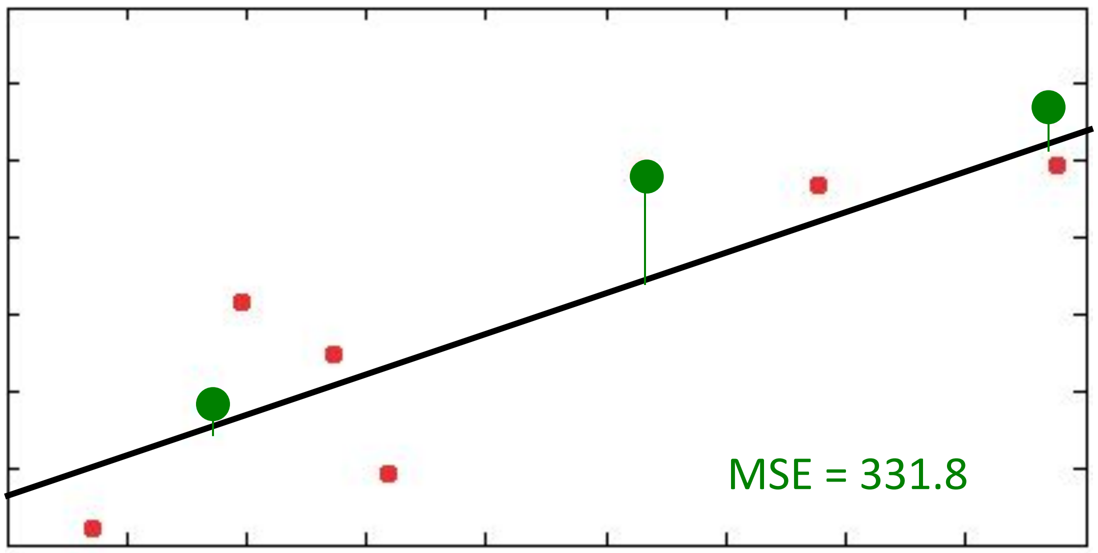
$$\mathbb{E}_{(x,y)}[J(y,\hat{y}(x\,;\,D))]$$

J = loss function (MSE)
D = training data set



Hold-out method

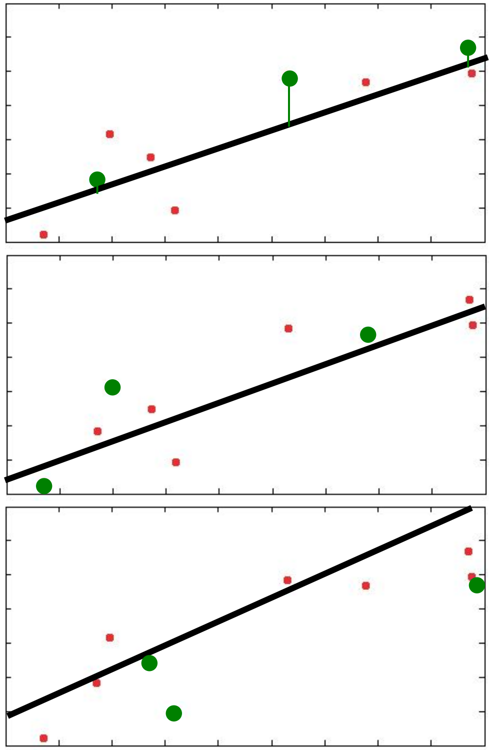
- Validation data
  - "Hold out" some data for evaluation (e.g., 70/30 split)
  - Train only on the remainder
- Some problems, if we have few data:
  - Few data in hold-out: noisy estimate of the error
  - More hold-out data leaves less for training!



	$x^{(i)}$	$y^{(i)}$
Training data	88	79
	32	-2
	27	30
	68	73
	7	-16
	20	43
Validation data	53	77
	17	16
	87	94

Cross-validation method

- K-fold cross-validation
  - Divide data into K disjoint sets
  - Hold out one set (= M / K data) for evaluation
  - Train on the others (= M*(K-1) / K data)



Split 1:
MSE = 331.8

Split 2:
MSE = 361.2

Split 3:
MSE = 669.8

3-Fold X-Val MSE
= 464.1

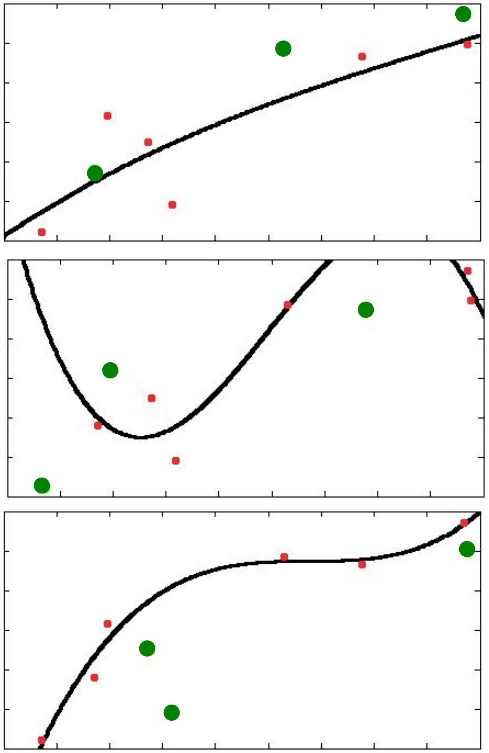
Training
data

Validation
data

$x^{(i)}$	$y^{(i)}$
88	79
32	-2
27	30
68	73
7	-16
20	43
53	77
17	16
87	94

Cross-validation method

- K-fold cross-validation
  - Divide data into K disjoint sets
  - Hold out one set (= M / K data) for evaluation
  - Train on the others (= M*(K-1) / K data)



Split 1:
MSE = 280.5

Split 2:
MSE = 3081.3

Split 3:
MSE = 1640.1

3-Fold X-Val MSE
= 1667.3

Training data

Validation data

$x^{(i)}$	$y^{(i)}$
88	79
32	-2
27	30
68	73
7	-16
20	43
53	77
17	16
87	94

Cross-validation

- Advantages:
  - Lets us use more (M) validation data

    (= less noisy estimate of test performance)

- Disadvantages:
  - More work
    - Trains K models instead of just one
  - Doesn’t evaluate any *particular* predictor
    - Evaluates K different models & averages
    - Scores *hyperparameters / procedure*, not an actual, specific predictor!

- Also: still estimating error for M’ < M data…

Learning curves

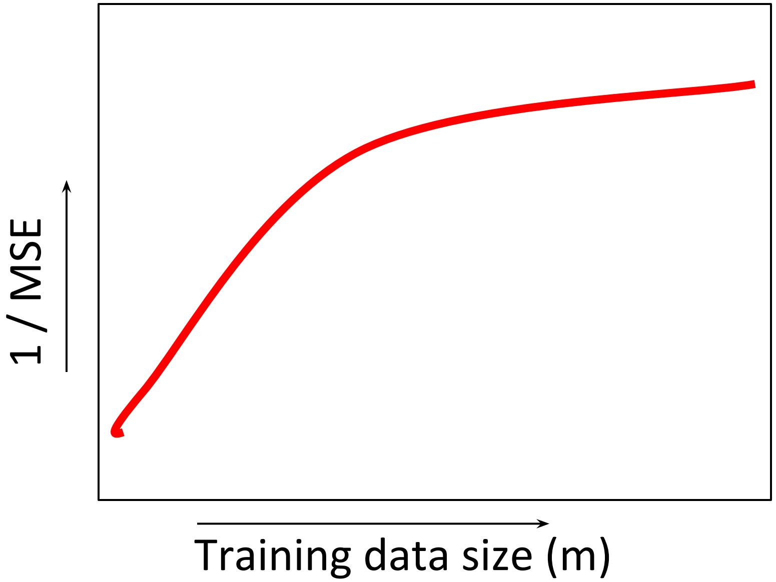
- Plot performance as a function of training size
  - Assess impact of fewer data on performance

    Ex: MSE0 - MSE (regression)

    or 1-Err (classification)

- Few data
  - More data significantly

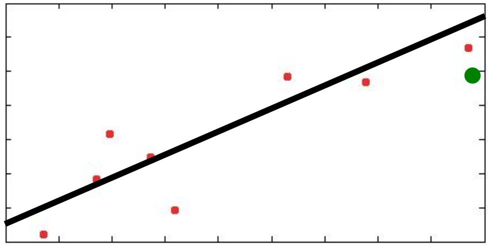
    improve performance
- “Enough” data
  - Performance saturates



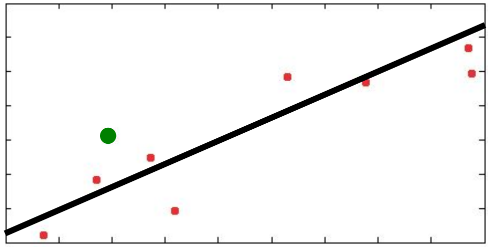
- If slope is high, decreasing *m* (for validation / cross-validation) might have a big impact…

Leave-one-out cross-validation

- When K=M (# of data), we get
  - Train on all data except one
  - Evaluate on the left-out data
  - Repeat M times (each data point held out once) and average



MSE = …



MSE = …

⋮

➡ LOO X-Val MSE = …

Training data

Validation data

$x^{(i)}$	$y^{(i)}$
88	79
32	-2
27	30
68	73
7	-16
20	43
53	77
17	16
87	94

Cross-validation Issues

- Need to balance:
  - Computational burden (multiple trainings)
  - Accuracy of estimated performance / error
- Single hold-out set:
  - Estimates performance with M’ < M data (important? learning curve?)
  - Need enough data to trust performance estimate
  - Estimates performance of a particular, trained learner
- K-fold cross-validation
  - K times as much work, computationally
  - Better estimates, still of performance with M’ < M data
- Leave-one-out cross-validation
  - M times as much work, computationally
  - M’ = M-1, but overall error estimate may have high variance

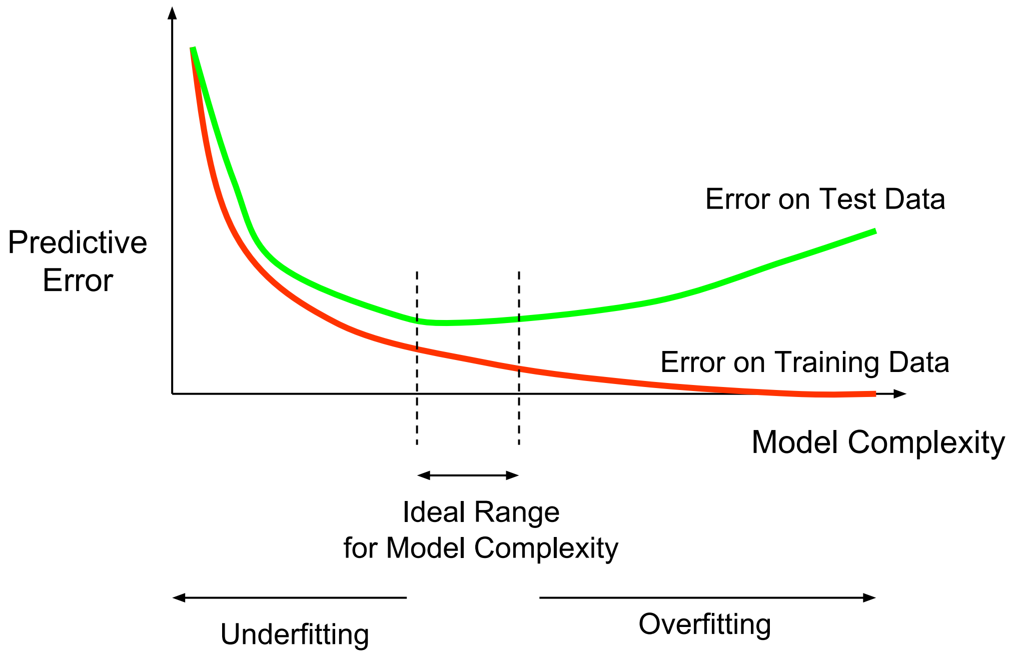
Machine Learning

- Linear Regression via Least Squares
- Gradient Descent Algorithms
- Direct Minimization of Squared Error
- Regression with Non-linear Features
- Bias, Variance, & Validation
- Regularized Linear Regression

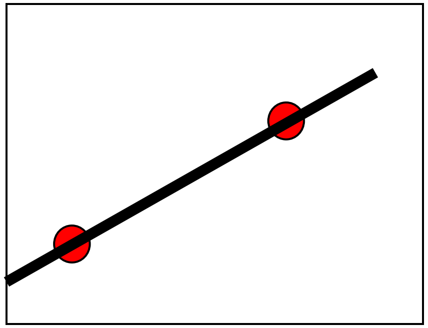
What to do about under/overfitting?

- Ways to increase complexity?
  - Add features, parameters
  - We'll see more…

- Ways to decrease complexity?
  - Remove features ("feature selection")
  - "Fail to fully memorize data"
    - Partial training
    - Regularization

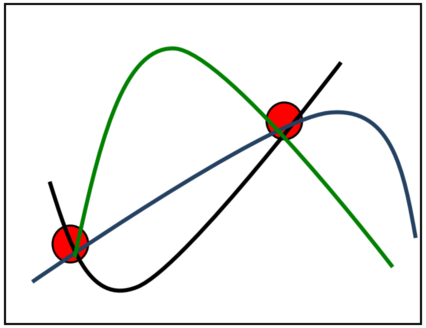


Linear regression



- Linear model, two data

- Quadratic model, two data?
  - Infinitely many settings with zero error
  - How to choose among them?



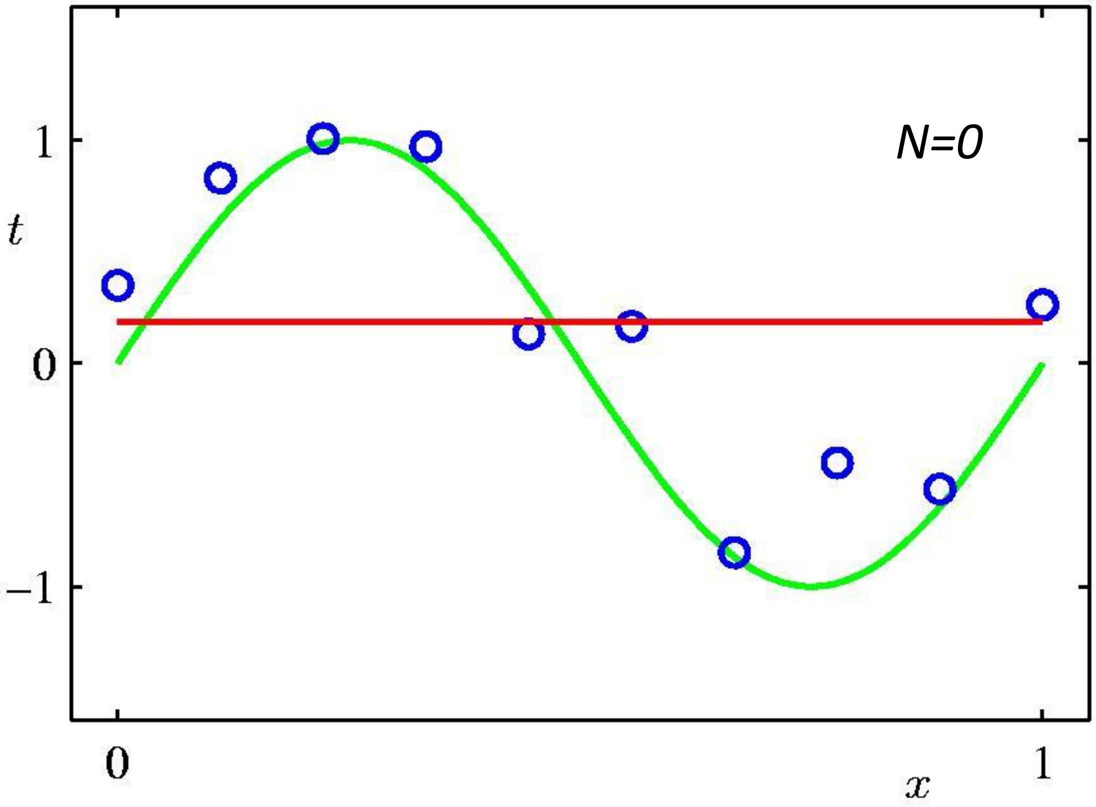
- Higher order coefficients = 0?
  - Uses knowledge of where features came from…

- Could choose e.g. minimum magnitude:

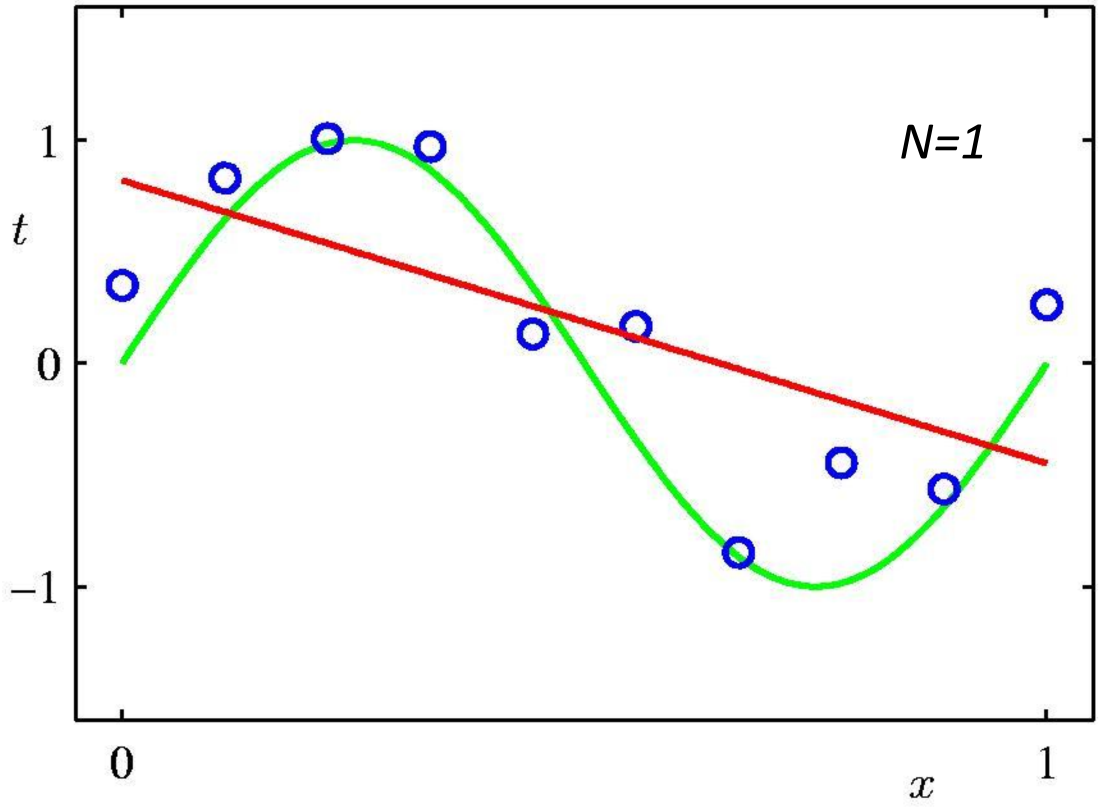
$$\min \underline{\theta}\underline{\theta}^T \quad s.t. \quad J(\underline{\theta}) = 0$$

- A type of *bias*: tells us which models to prefer

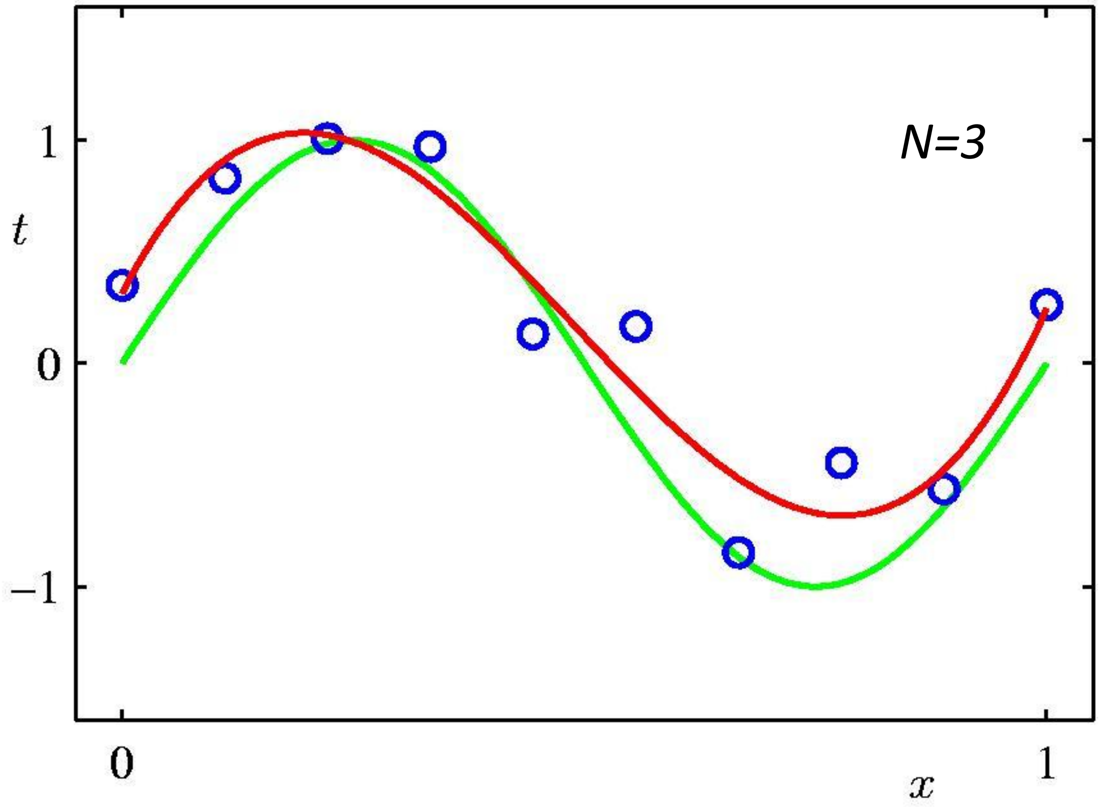
0^{th} Order Polynomial



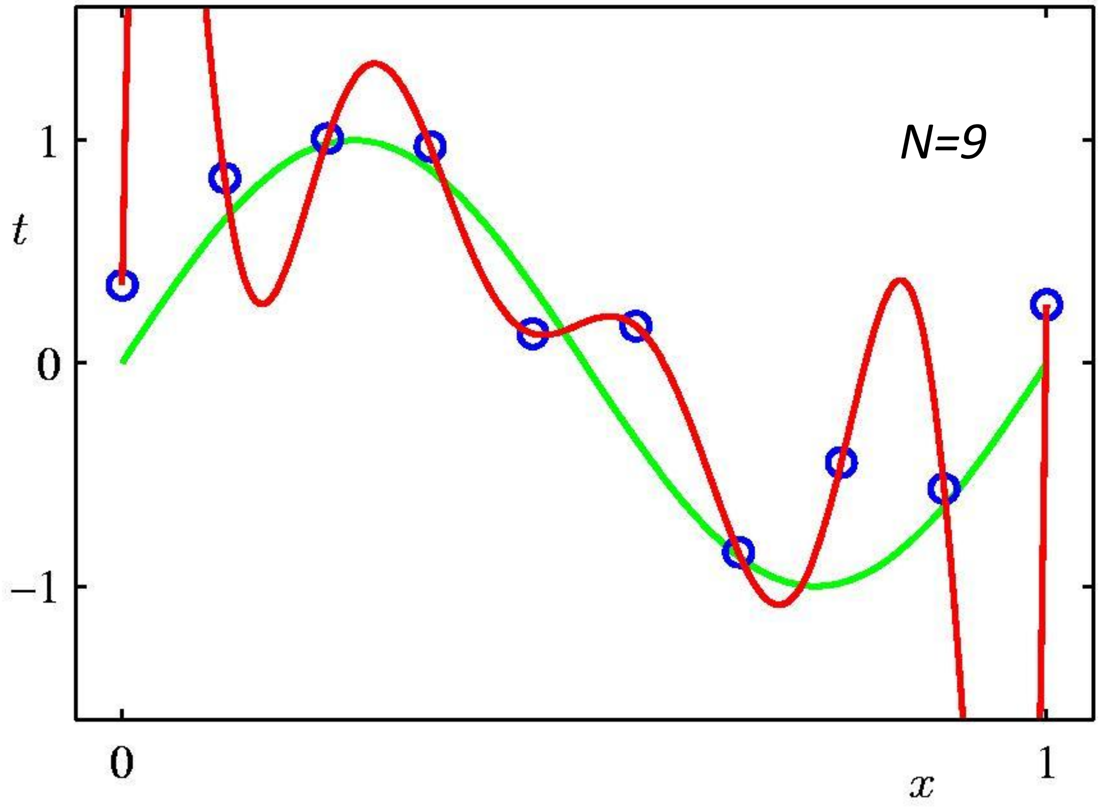
1^{st} Order Polynomial



3rd Order Polynomial



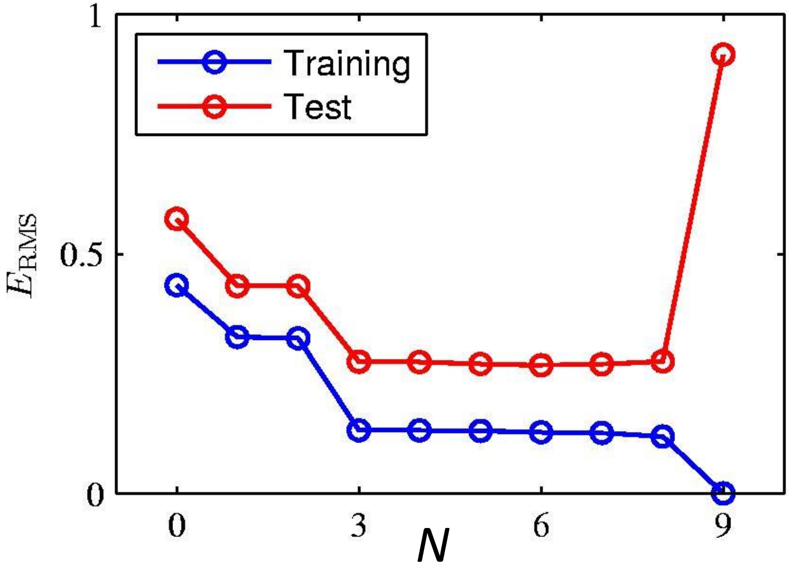
9th Order Polynomial



Estimated Polynomial Coefficients

Estimated Regression Coeffients θ

N=0	*N=1*	*N=3*	*N=9*
0.19	0.82	0.31	0.35
	-1.27	7.99	232.37
		-25.43	-5321.83
		17.37	48568.31
			-231639.30
			640042.26
			-1061800.52
			1042400.18
			-557682.99
			125201.43



Regularization

- Can modify our cost function J to add “preference” for certain parameter values

$$J(\underline{\theta}) = \frac{1}{2}(\underline{y} - \underline{\theta}\,\underline{X}^T) \cdot (\underline{y} - \underline{\theta}\,\underline{X}^T)^T + \alpha\,\theta\theta^T$$

- New solution (derive the same way)

$$\underline{\theta} \quad = \quad \underline{y}\,\underline{X}(\underline{X}^T\,\underline{X} + \alpha I)^{-1}$$

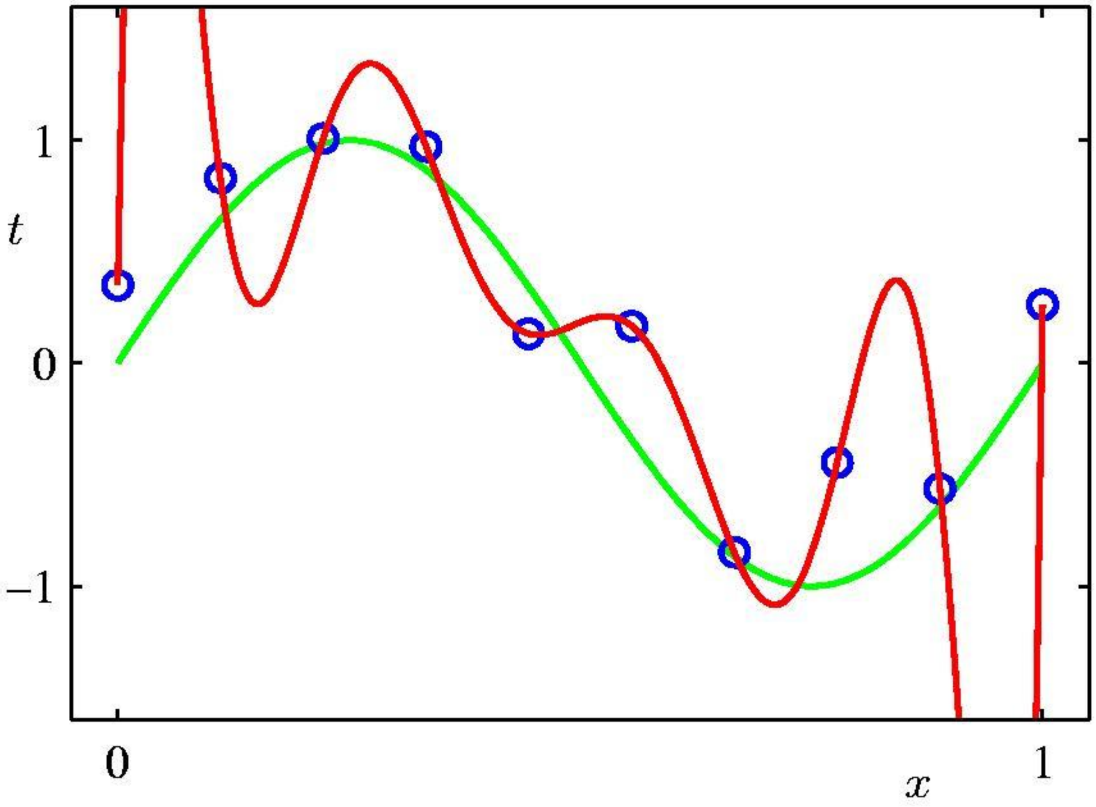
  - Problem is now well-posed for any degree

L_2 penalty:
“Ridge regression”

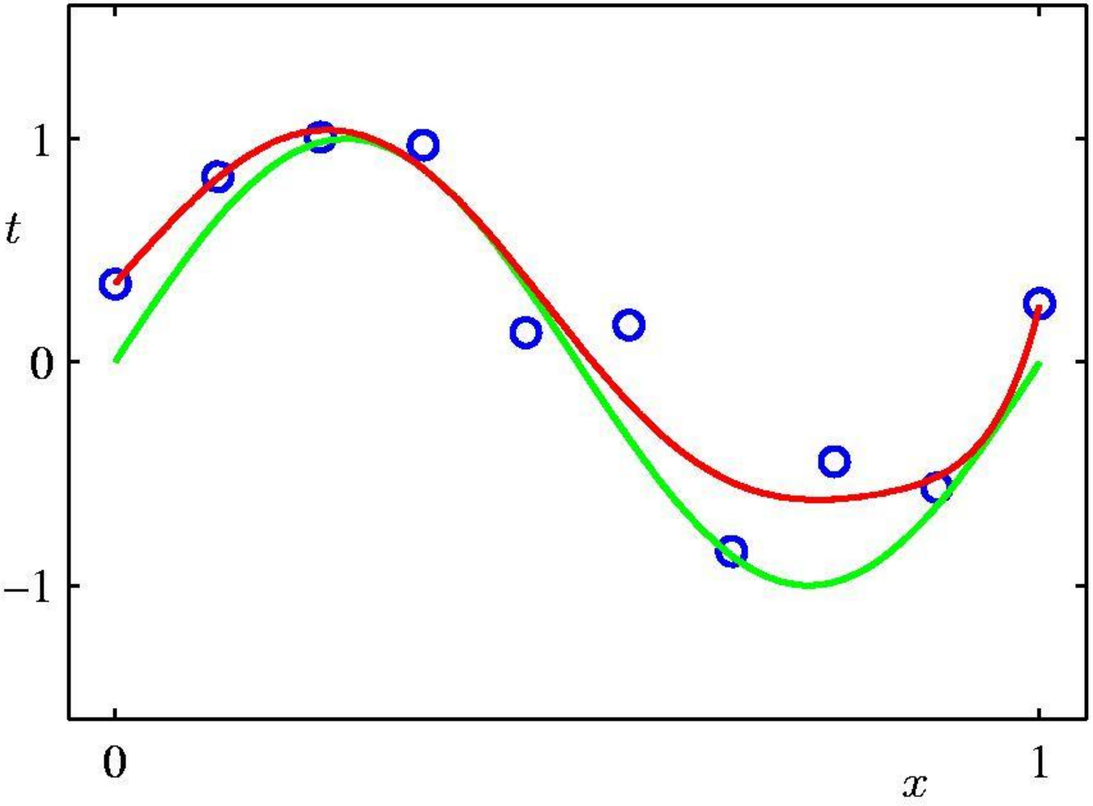
$$\theta\theta^T = \sum_i \theta_i^2$$

- Notes:
  - “Shrinks” the parameters toward zero
  - Alpha large: we prefer small theta to small MSE
  - Regularization term is independent of the data: paying more attention reduces our model variance

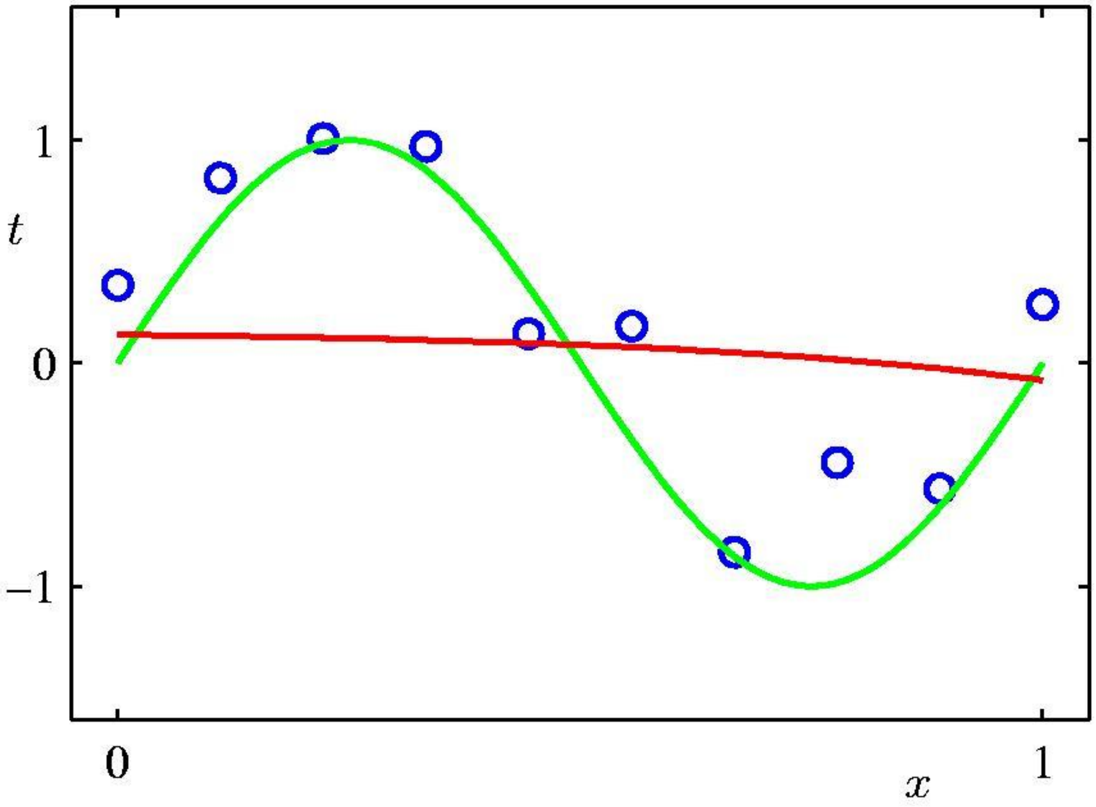
Regression: Zero Regularization



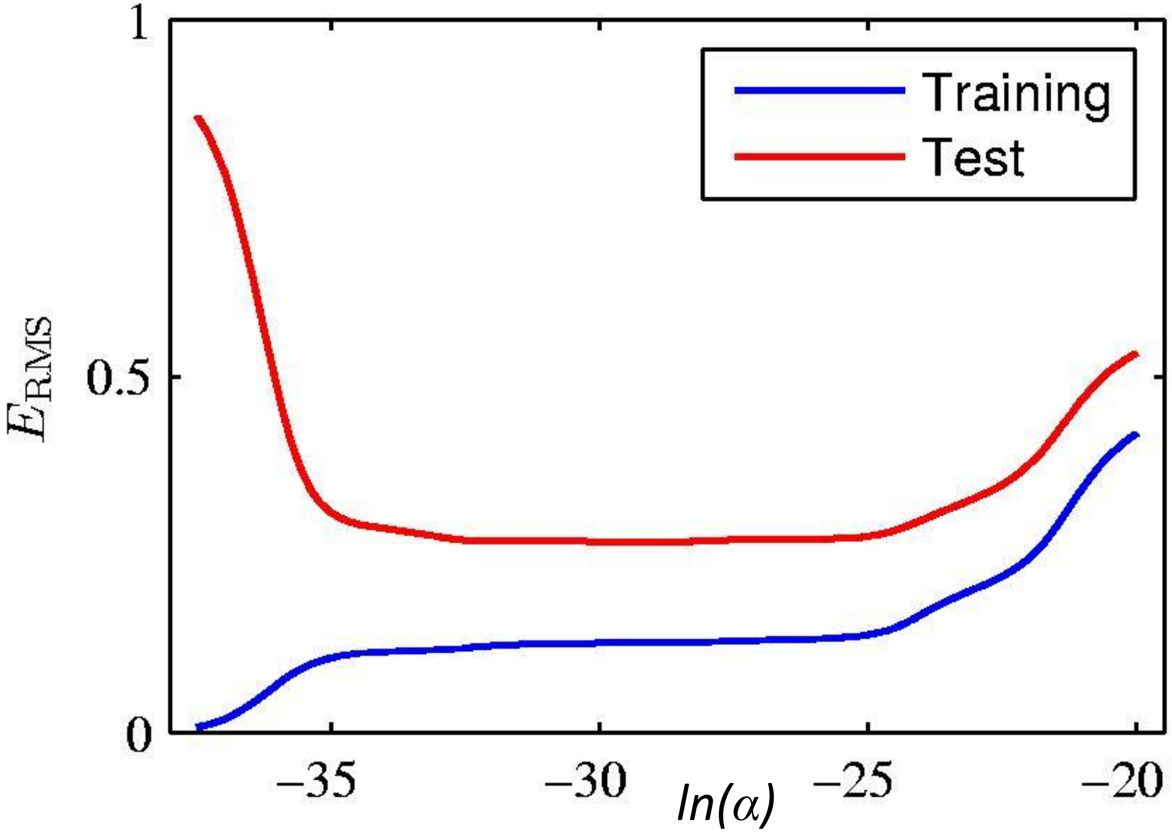
Regression: Moderate Regularization



Regression: Big Regularization



Impact of Regularization Parameter



Estimated Polynomial Coefficients

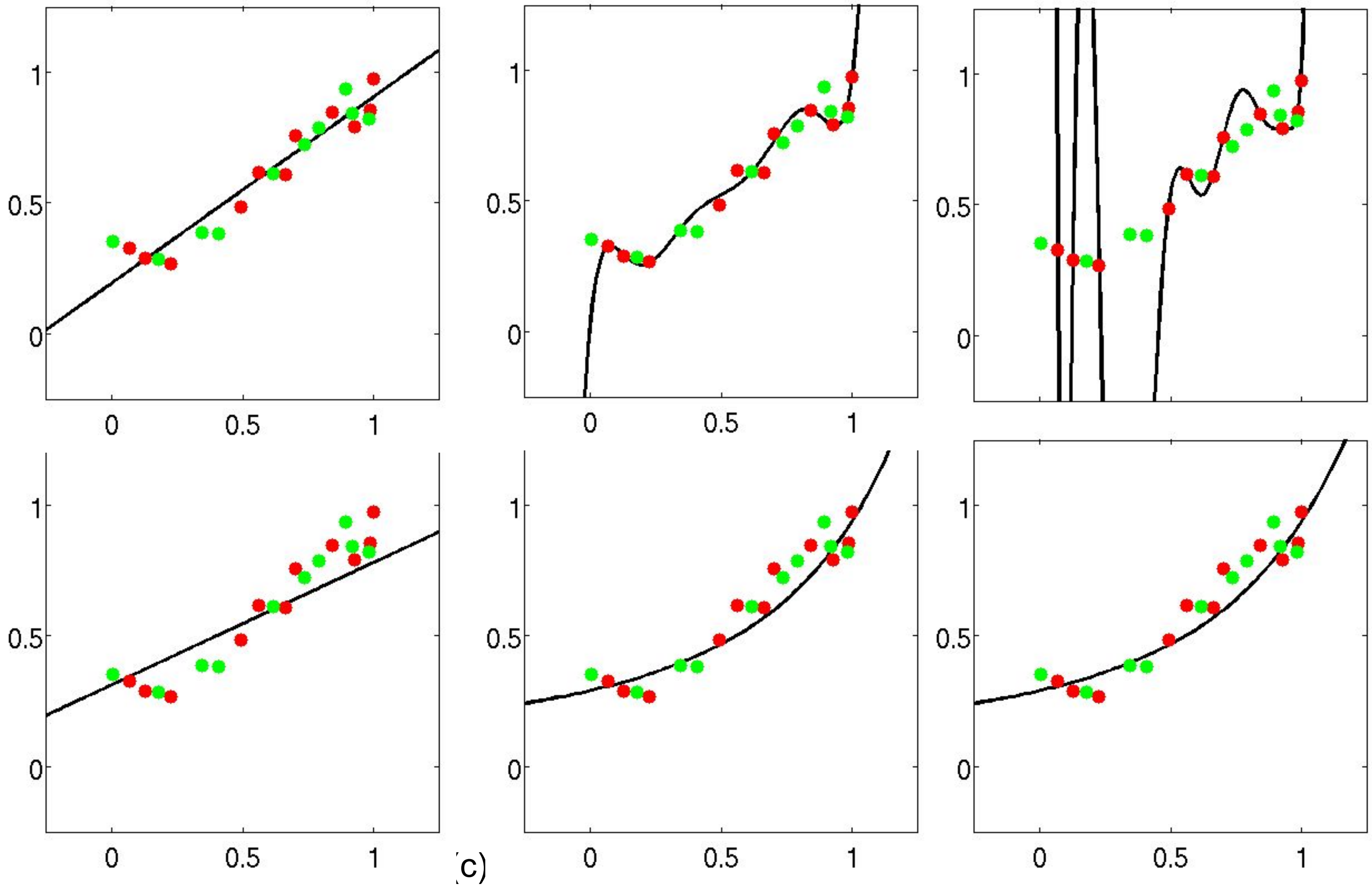
	α zero	α medium	α big
Estimated Regression Coeffients θ	0.35	0.35	0.13
	232.37	4.74	-0.05
	-5321.83	-0.77	-0.06
	48568.31	-31.97	-0.05
	-231639.30	-3.89	-0.03
	640042.26	55.28	-0.02
	-1061800.52	41.32	-0.01
	1042400.18	-45.95	-0.00
	-557682.99	-91.53	0.00
	125201.43	72.68	0.01

Regularization

- Compare between unreg. & reg. results

Alpha =0
(Unregularized)

Alpha =1

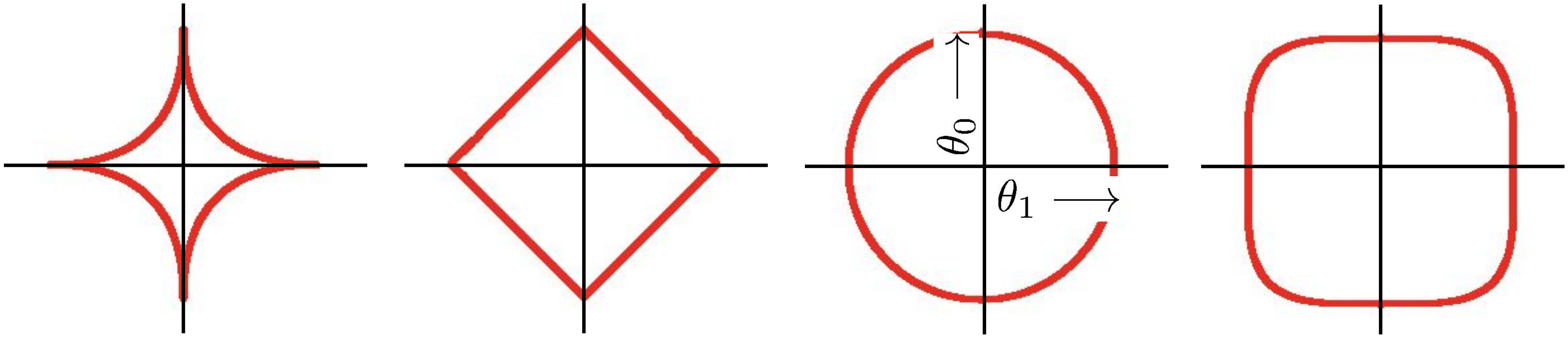


Different regularization functions

- More generally, for the L_p regularizer:

$$\Big(\sum_i |\theta_i|^p \Big)^{\frac{1}{p}}$$

Isosurfaces: $||\theta||_p$ = constant



p=0.5

p=1
Lasso

p=2
Quadratic

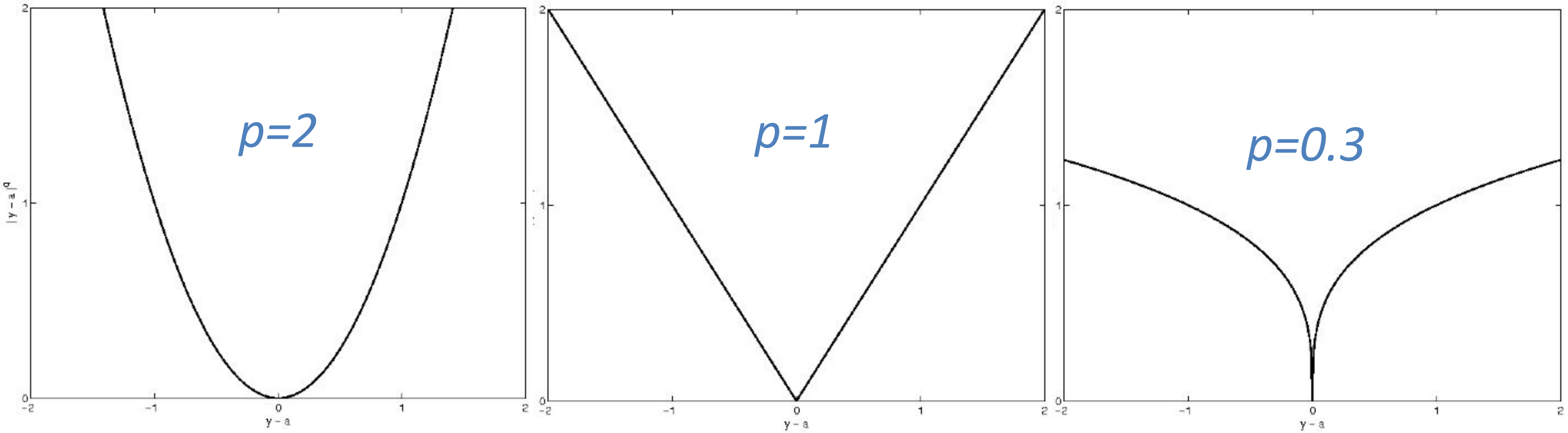
p=4

L_0 = limit as p goes to 0 : “number of nonzero weights”, a natural notion of complexity

Different regularization functions

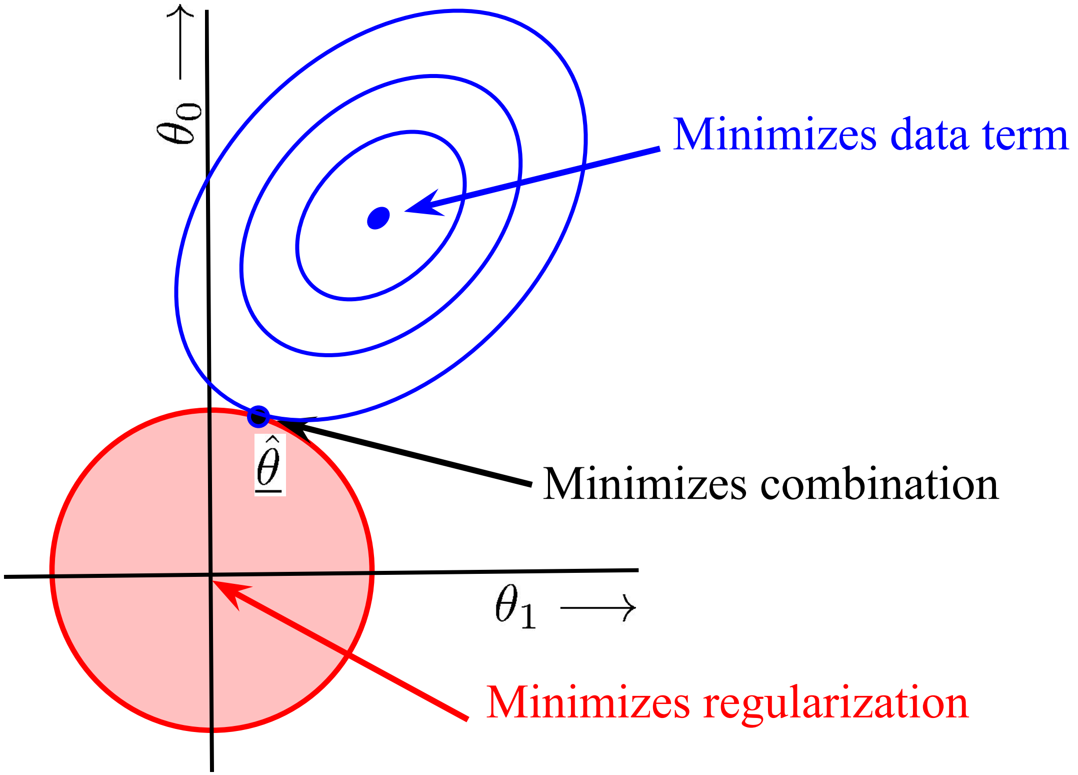
- More generally, for the L_p regularizer:

$$\Big(\sum_i |\theta_i|^p\Big)^{\frac{1}{p}}$$



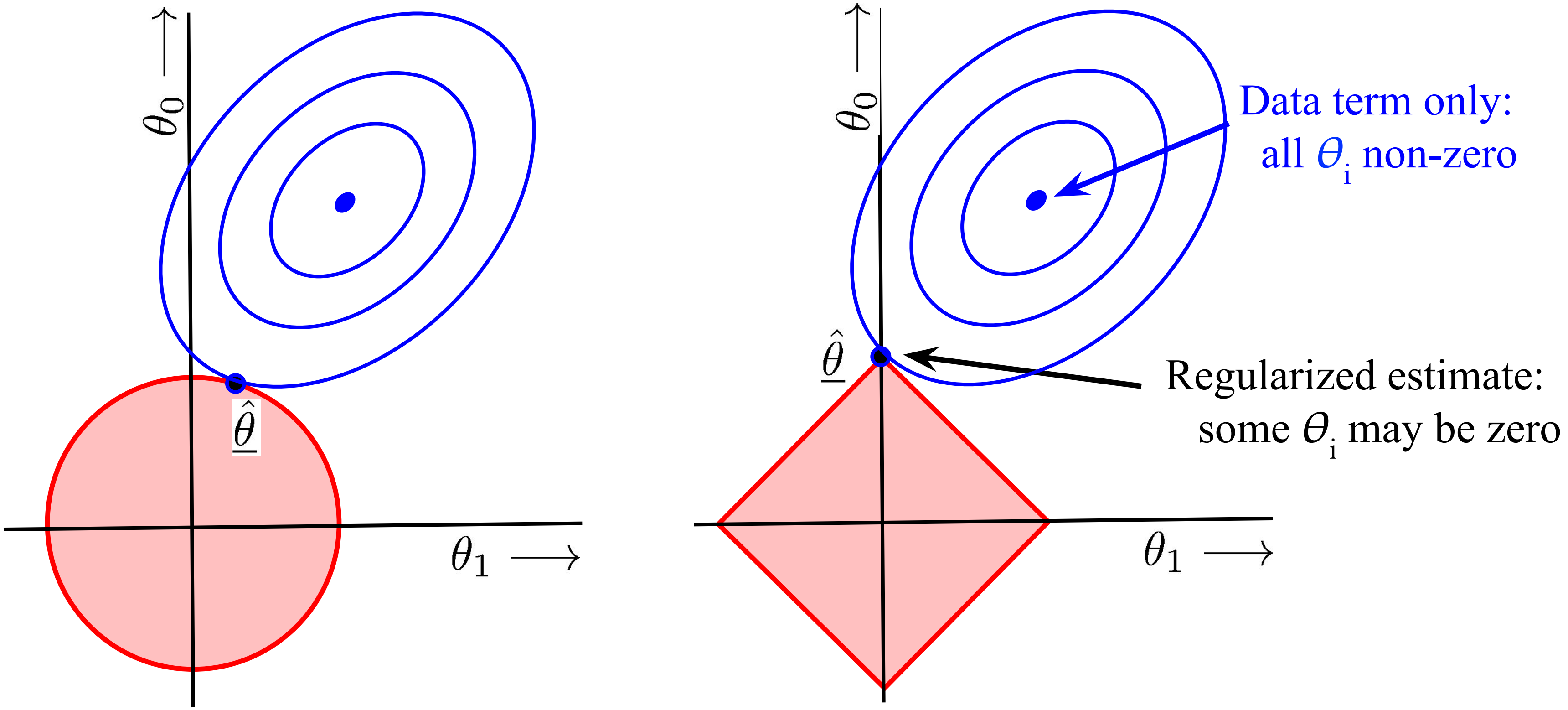
Regularization: L_2 vs L_1

- Estimate balances data term & regularization term



Regularization: L_2 vs L_1

- Estimate balances data term & regularization term
- Lasso tends to generate sparser solutions than a quadratic regularizer.

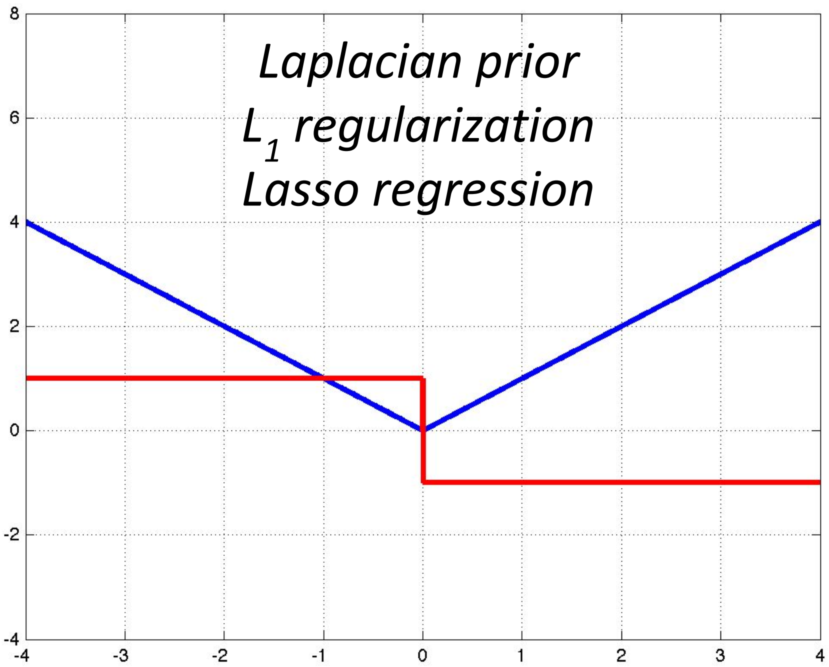

$\theta_0 \longrightarrow$
$\hat{\theta}$
$\theta_1 \longrightarrow$
$\theta_0 \longrightarrow$
$\hat{\theta}$
$\theta_1 \longrightarrow$

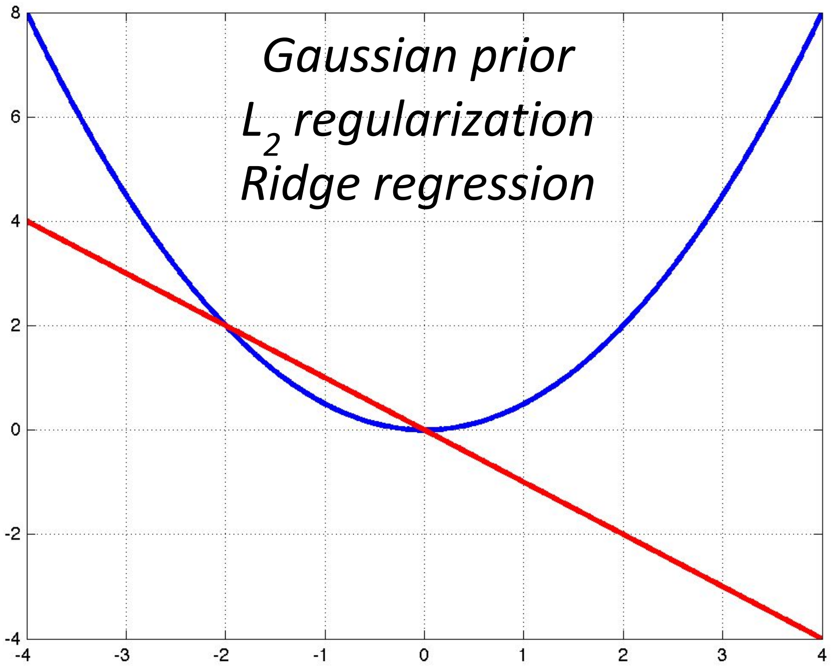
Data term only: all θ_i non-zero

Regularized estimate: some θ_i may be zero

Gradient-Based Optimization

- L_2 makes (all) coefficients smaller
- L_1 makes (some) coefficients exactly zero: *feature selection*





Objective Function: $f(\theta_i) = |\theta_i|^p$

Negative Gradient: $-f'(\theta_i)$

(Informal intuition: Gradient of L_1 objective not defined at zero)