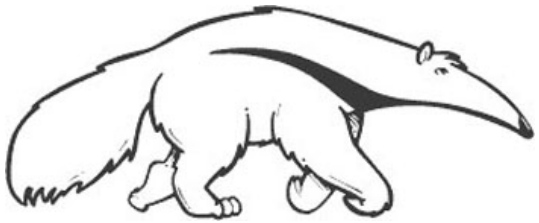
+

CS178: Machine Learning and Data Mining

Midterm Review

Prof. Alexander Ihler



BREN:ICS
INFORMATION AND COMPUTER SCIENCES
UNIVERSITY of CALIFORNIA IRVINE

Midterm Format

In Class Exam on Monday, November 5th:

- 50 minutes long
- Will start at 11:00am sharp (be here early)

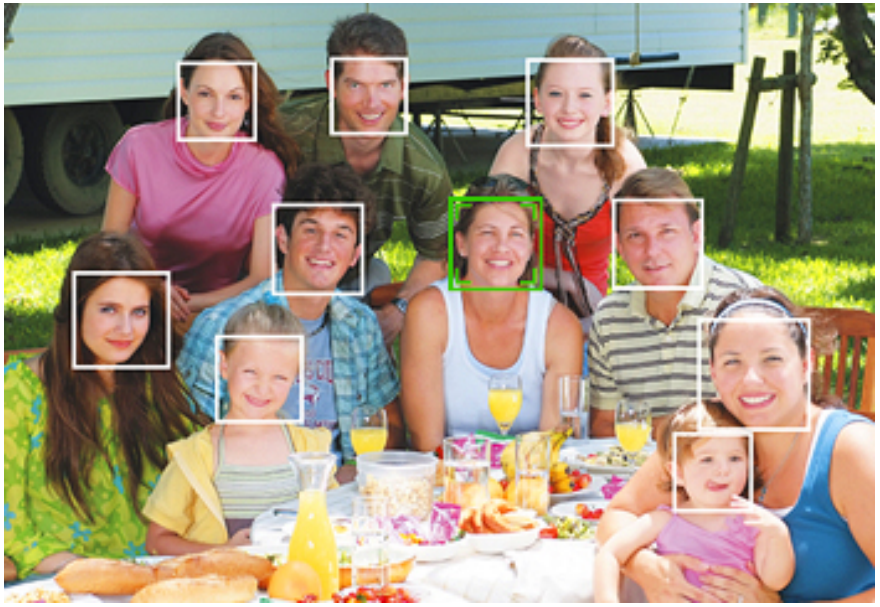
What you may bring:

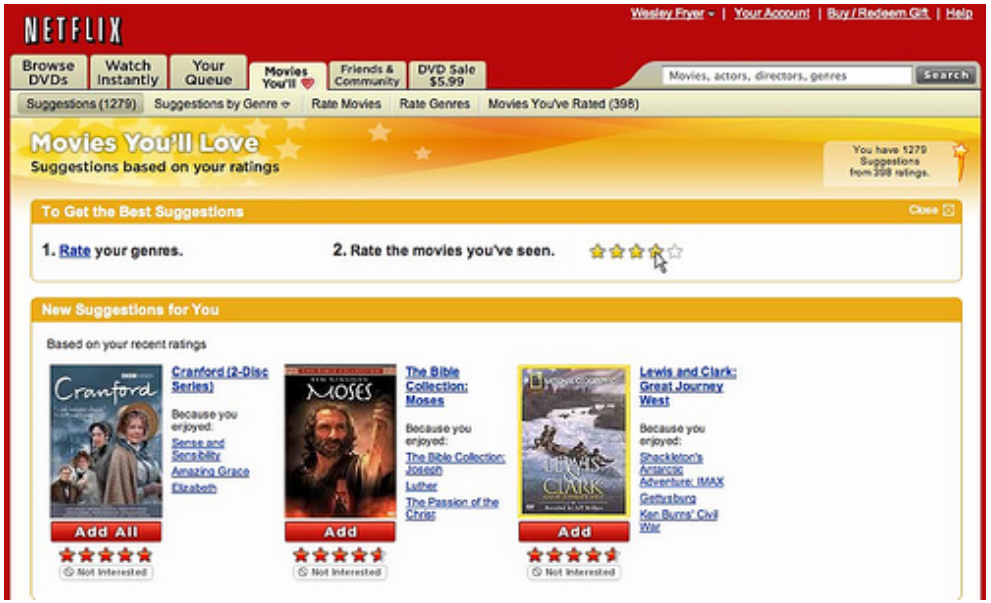
- One (double-sided) 8.5x11-inch sheet of handwritten notes
- Pencil and/or pen
- Scratch sheets will be part of the exam booklet
- *Electronic devices are not allowed*

Note: seating assignment by exam!

Types of prediction problems

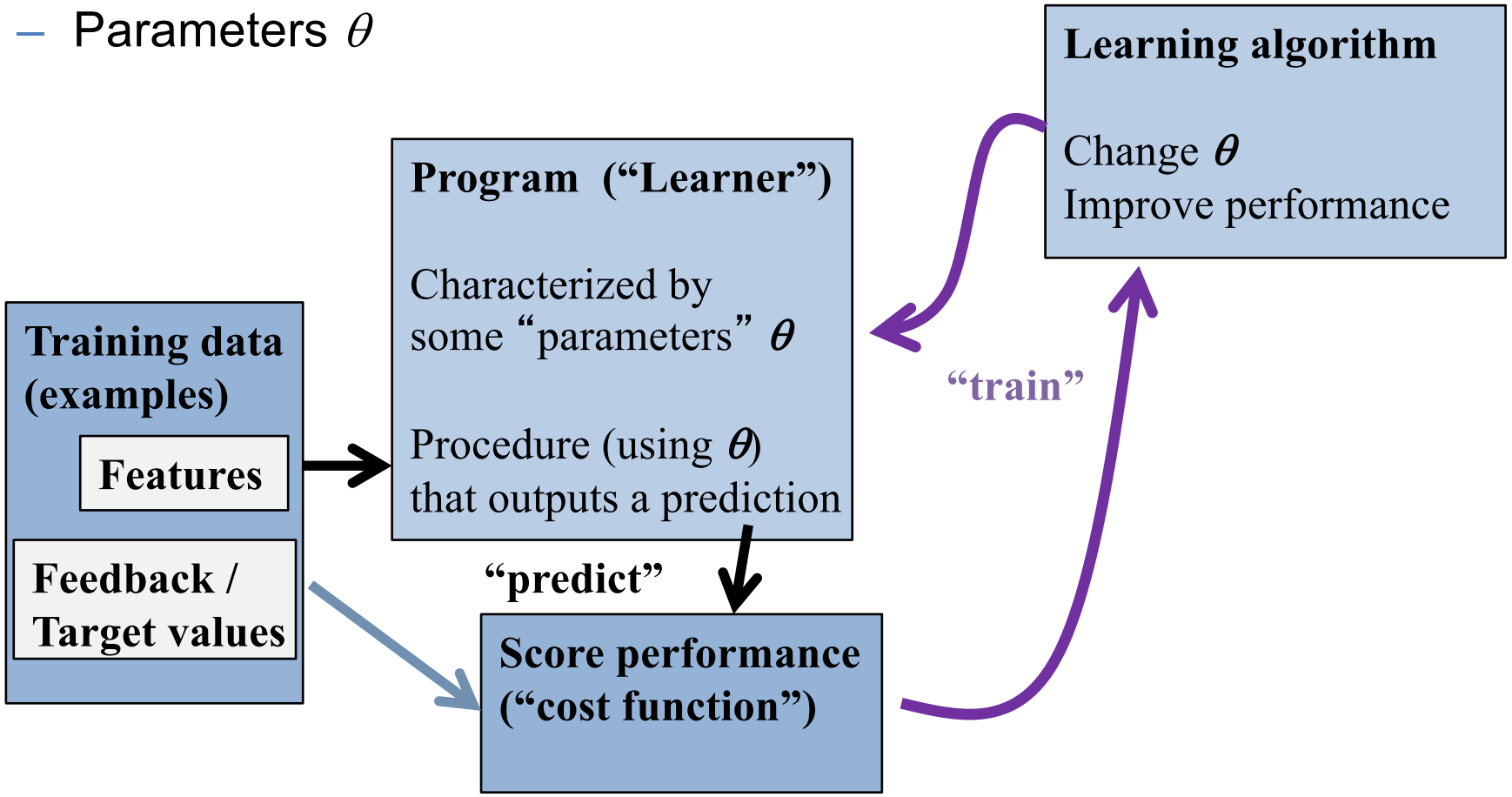
- Supervised learning
  - "Labeled" training data
  - Every example has a desired target value (a "best answer")
  - Reward prediction being close to target
  - **Classification:** a discrete-valued prediction (often: action / decision)
  - **Regression:** a continuous-valued prediction





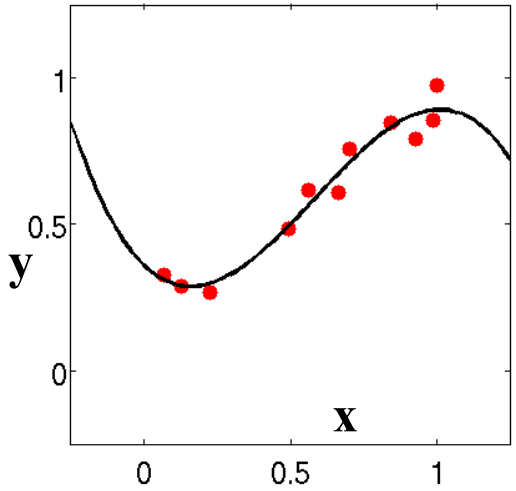
Supervised learning

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



Regression vs. Classification

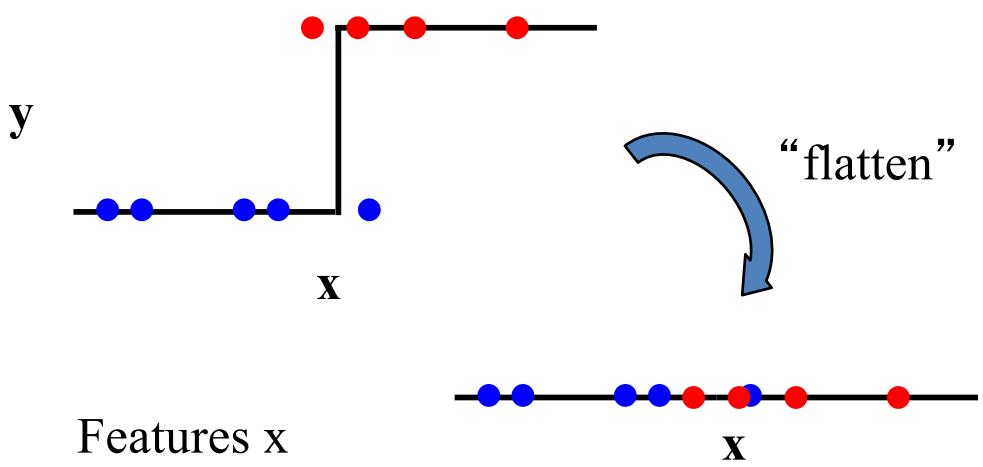
Regression



Features x
Real-valued target y

Predict continuous function $\hat{y}(x)$

Classification



Features x
Discrete class c
(usually 0/1 or +1/-1)
Predict discrete function $\hat{y}(x)$

Machine Learning

Overfitting & Cross-Validation

Nearest Neighbors

Linear Classification

Naïve Bayes Classifiers

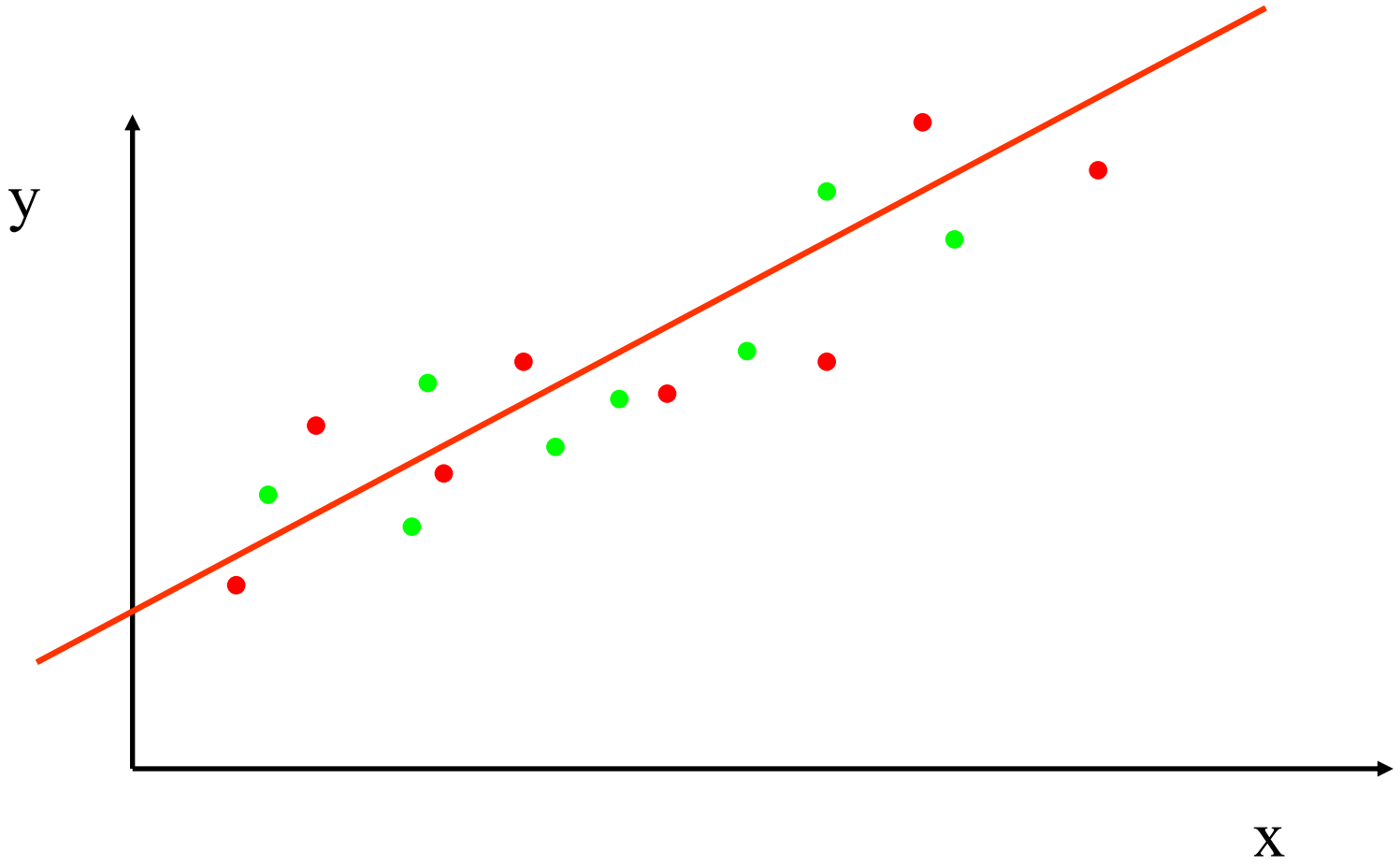
Support Vector Machines

Linear Regression

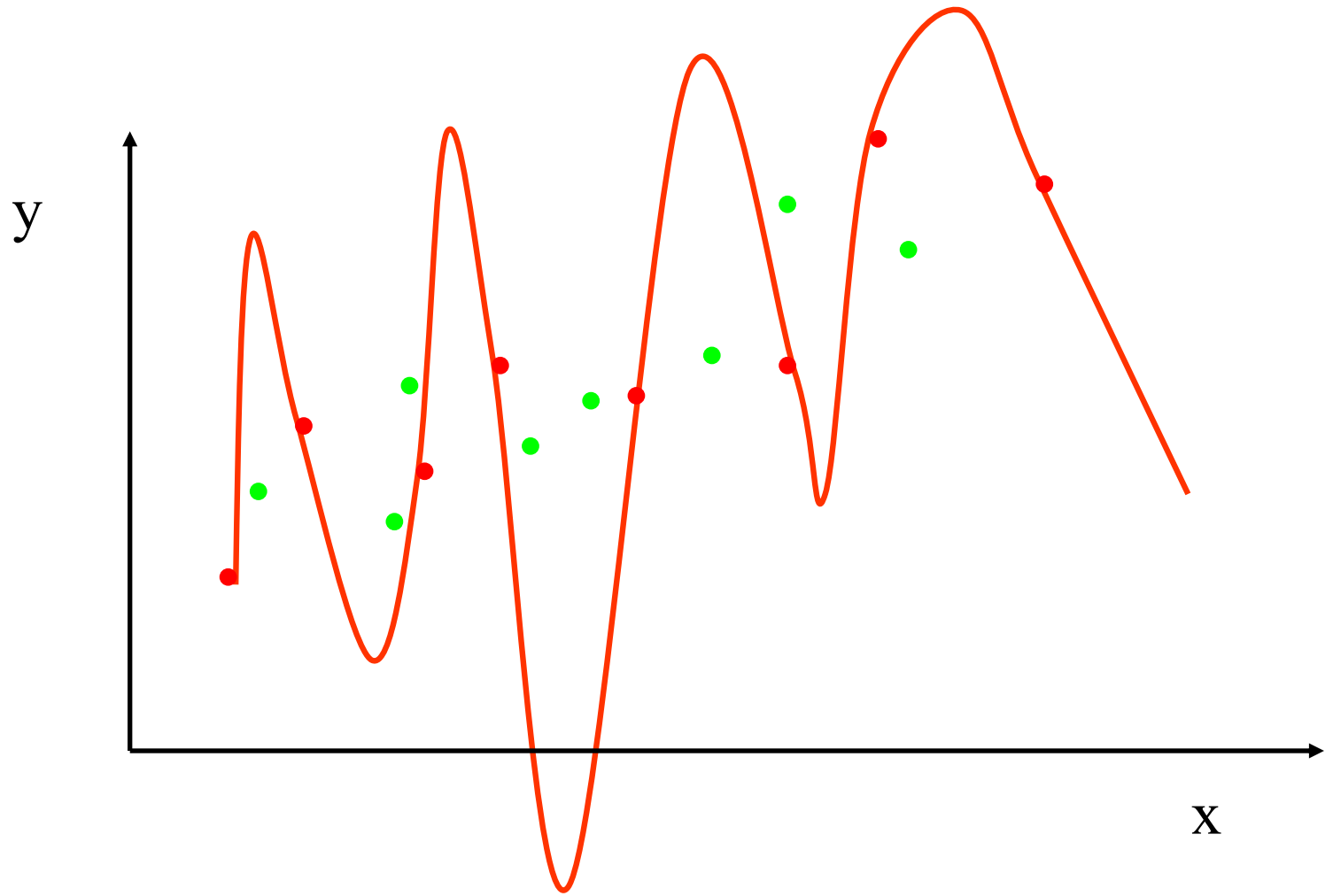
Neural Networks

Overfitting and complexity

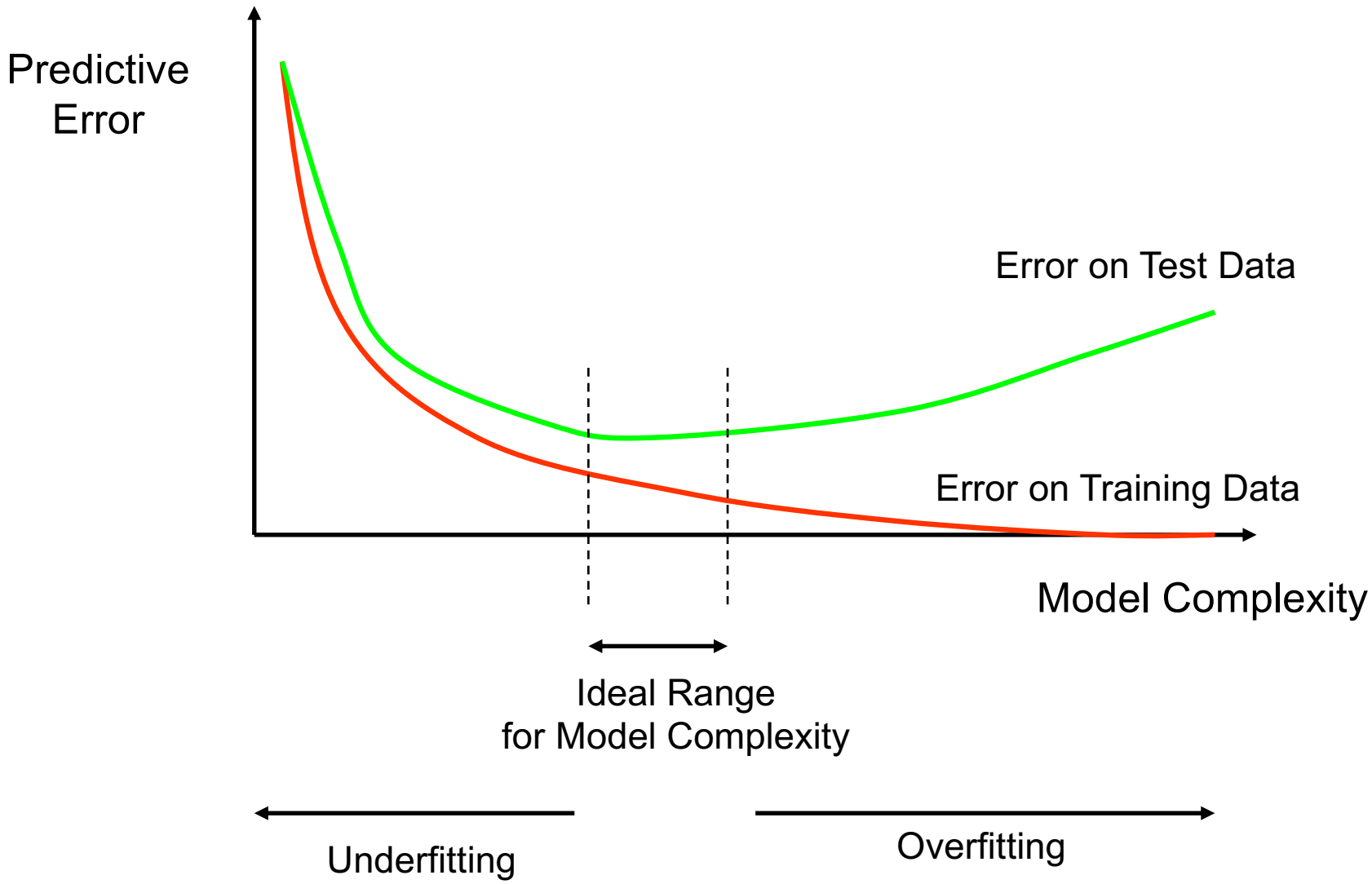
Simple model: Y= aX + b + e



Overfitting and complexity

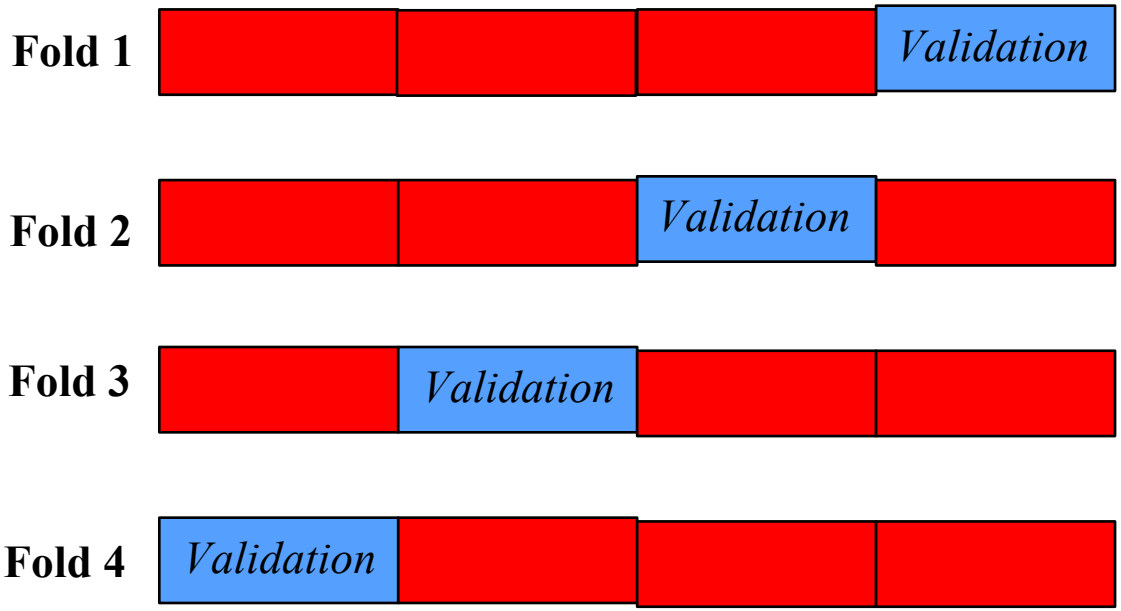


How Overfitting affects Prediction



Cross-Validation

- Divide training data into K equal-sized *folds*
- Train on K-1 folds, evaluate on remainder
- Pick model with best average performance across K trials



How many folds?

- *Bias:* Too few, and effective training dataset much smaller
- *Variance:* Too many, and test performance estimates noisy
- *Cost:* Must run training algorithm once per fold (parallelizable)
- *Practical rule of thumb:* 5-fold or 10-fold cross-validation
- *Theoretically troubled:* Leave-one-out cross-validation, K=N

Machine Learning

Overfitting & Cross-Validation

Nearest Neighbors

Linear Classification

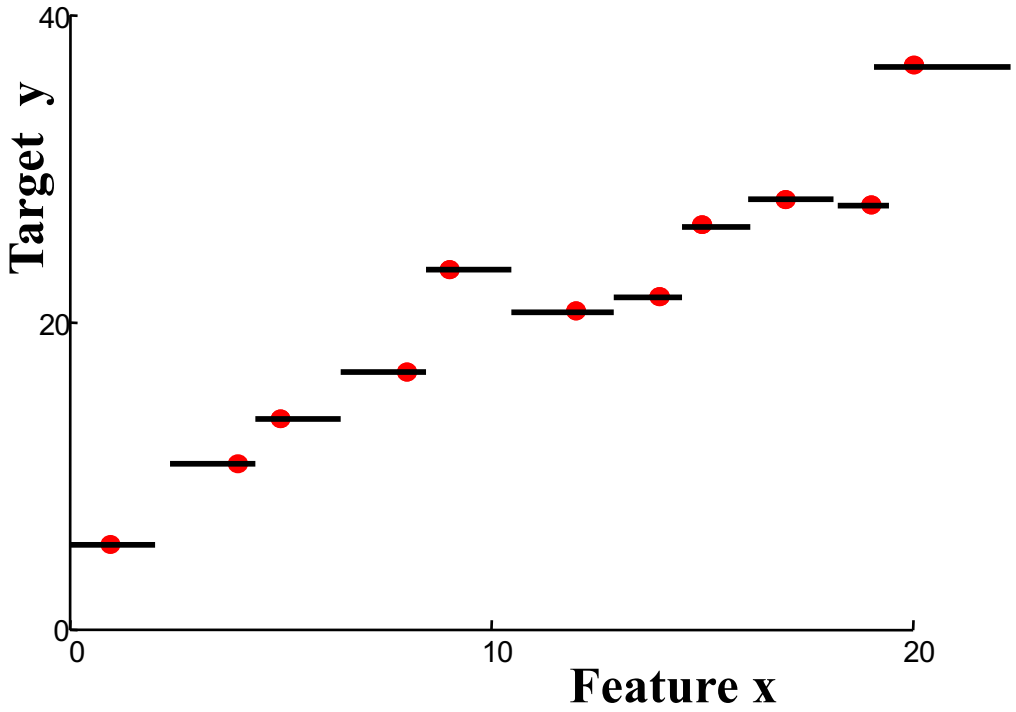
Naïve Bayes Classifiers

Support Vector Machines

Linear Regression

Neural Networks

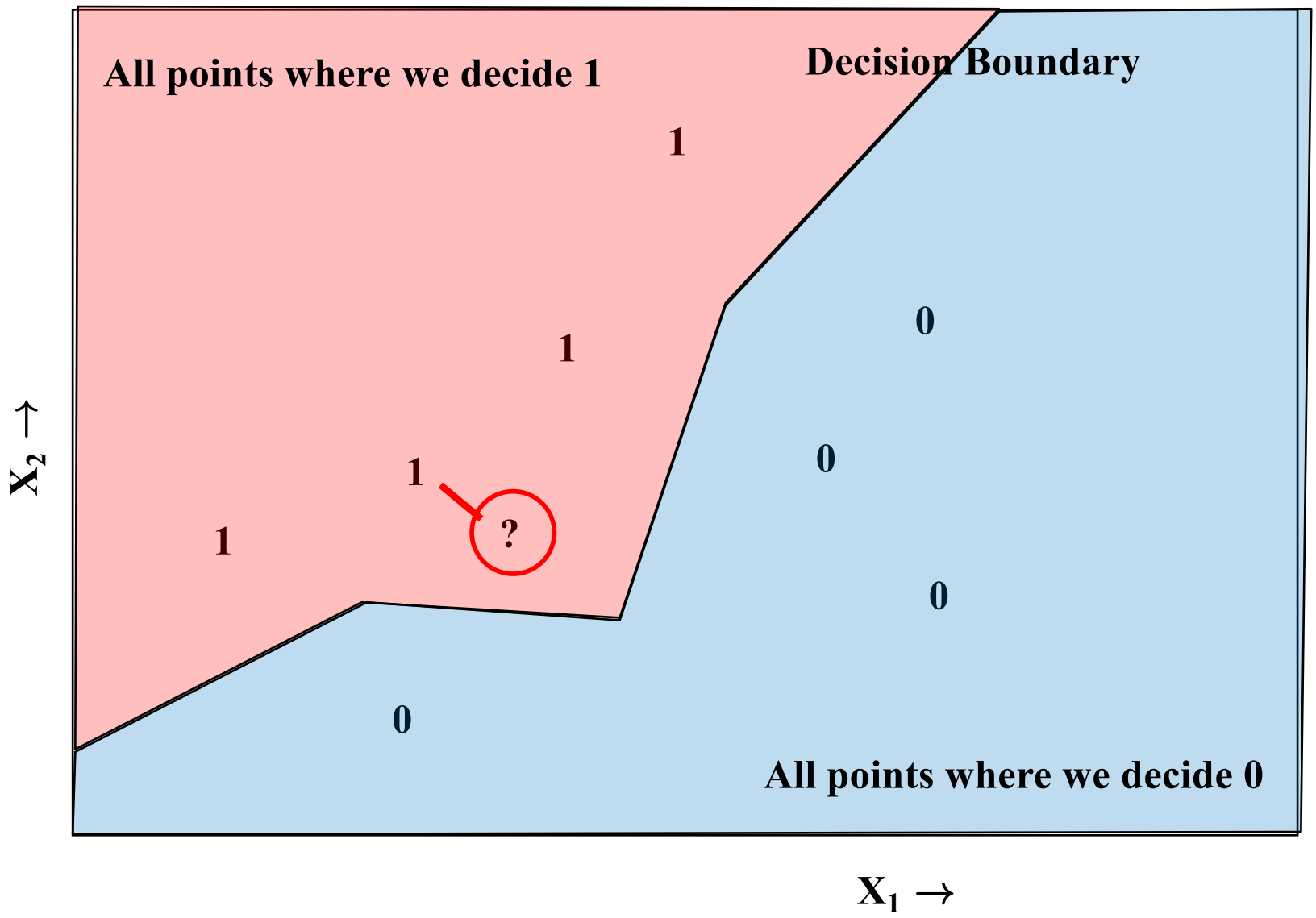
Nearest neighbor regression



"Predictor":
Given new features:
Find nearest example
Return its value

- Find training datum $x^{(i)}$ closest to $x^{(new)}$; predict $y^{(i)}$
- Defines an (implict) function f(x)
- "Form" is piecewise constant

Nearest neighbor classifier



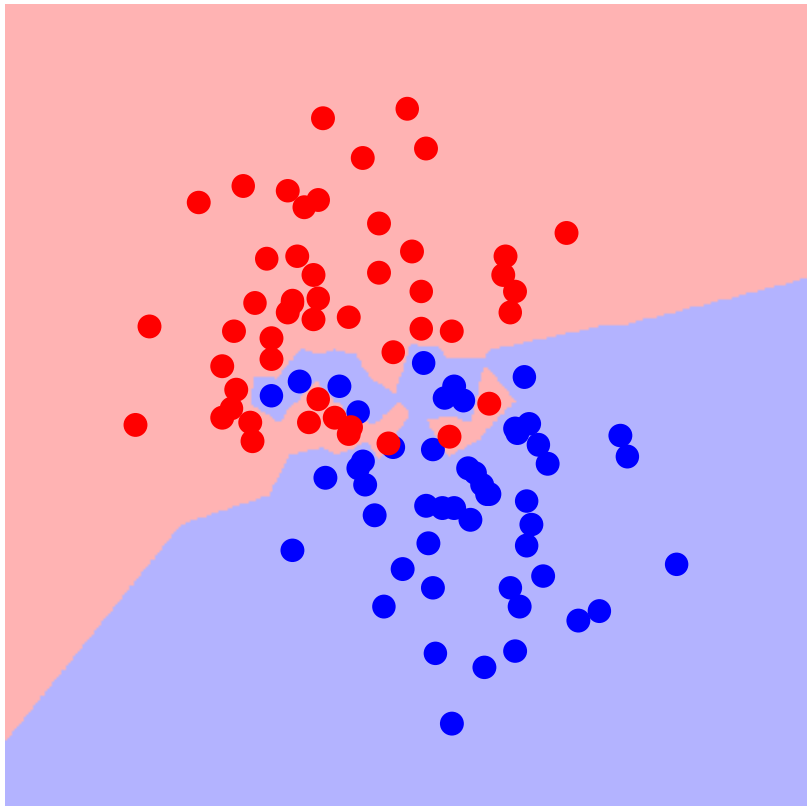
K-Nearest Neighbor (kNN) Classifier

- Find the k-nearest neighbors to $\underline{x}$ in the data
  - i.e., rank the feature vectors according to Euclidean distance
  - select the k vectors which are have smallest distance to $\underline{x}$
- Regression
  - Predict by averaging the y-values of the k closest training examples
- Classification
  - ranking yields k feature vectors and a set of k class labels
  - Predict the class label which is most common in this set ("vote")
  - Note: for two-class problems, if k is odd (k=1, 3, 5, …) there will never be any "ties"; otherwise, just use (any) tie-breaking rule
- "Training" is trivial: just use training data as a lookup table, and search to classify a new datum
- "Testing" can be computationally expensive: must find nearest neighbors within a potentially large training set

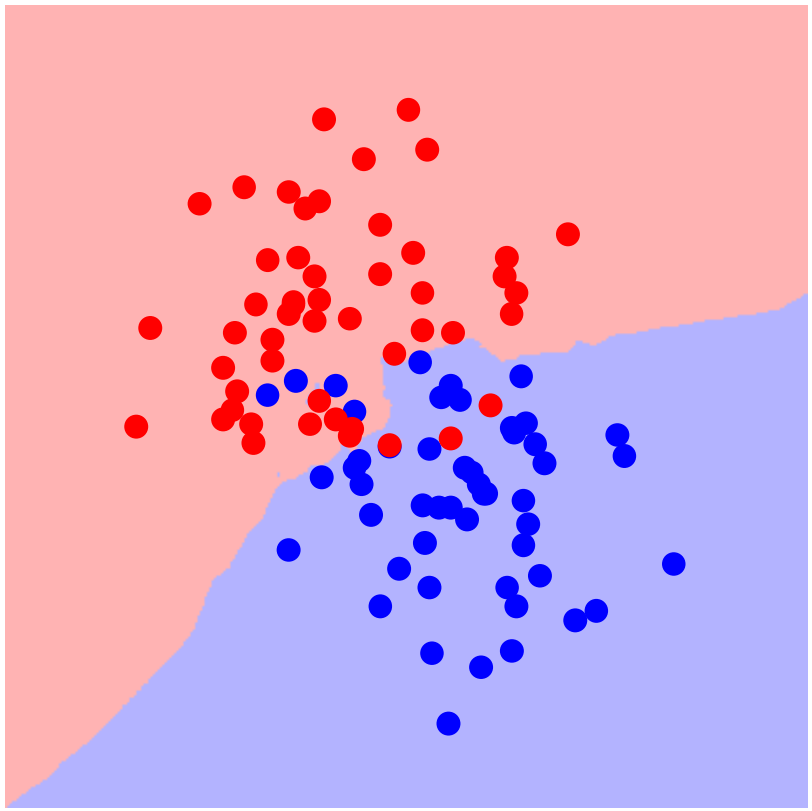
kNN Decision Boundary

- Piecewise linear decision boundary
- Increasing k “simplifies” decision boundary
  - Majority voting means less emphasis on individual points

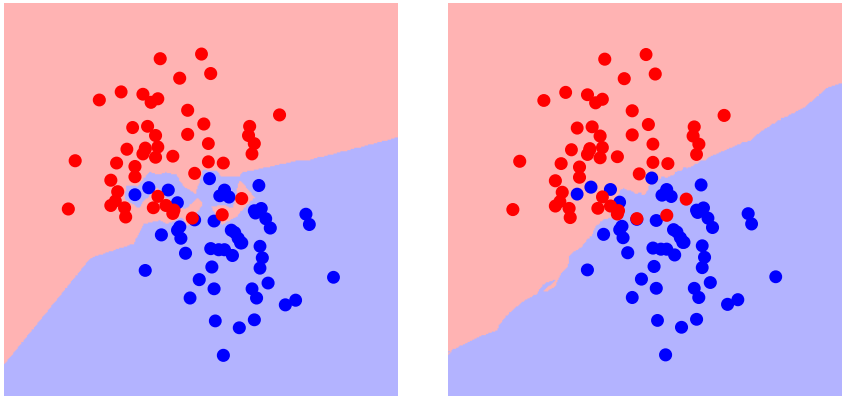
K = 1

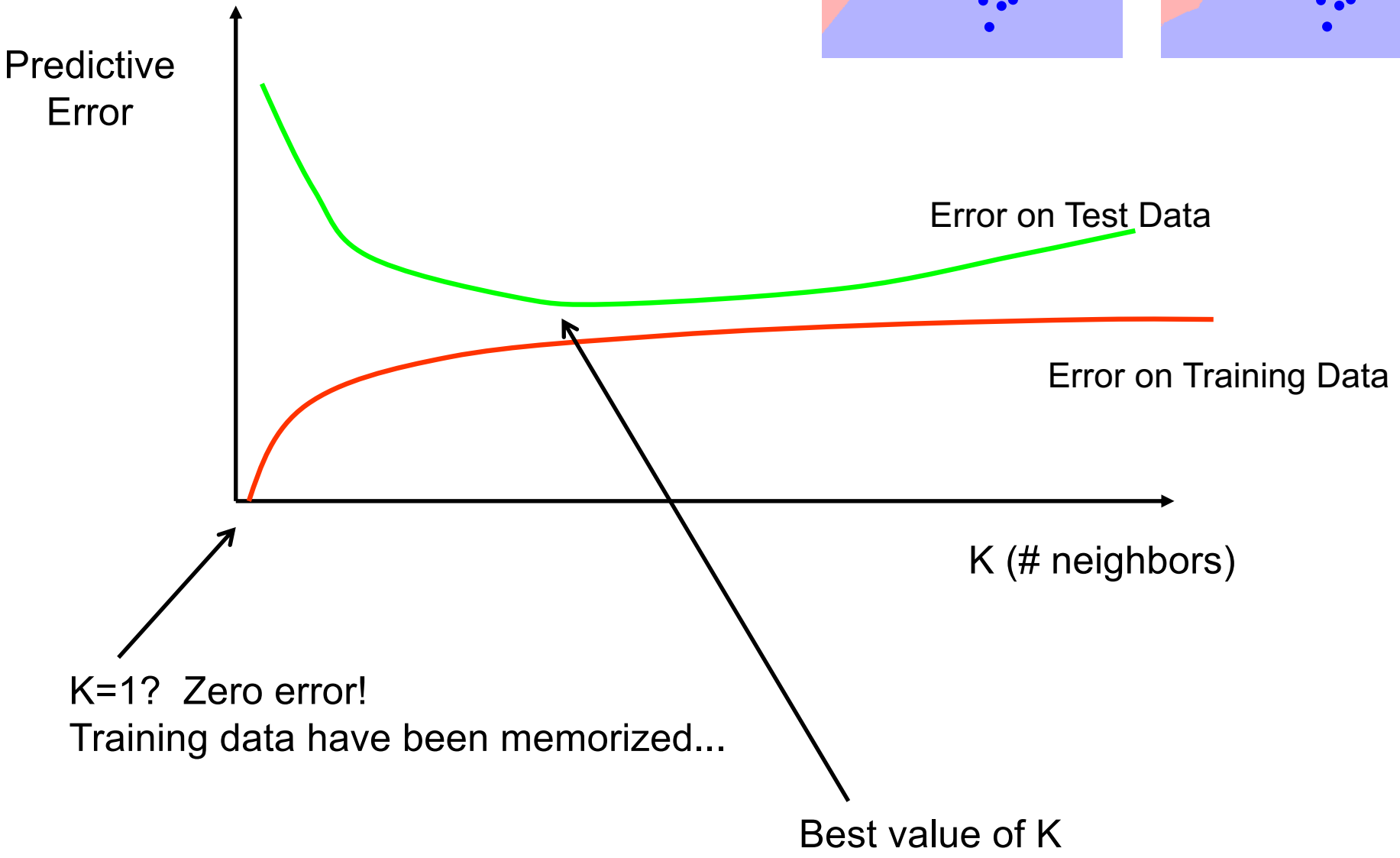


K = 7



Error rates and K





Machine Learning

Overfitting & Cross-Validation

Nearest Neighbors

Linear Classification

Naïve Bayes Classifiers

Support Vector Machines

Linear Regression

Neural Networks

Bayes classifiers

- Estimate p(y) = [p(y=0) , p(y=1) …]
- Estimate p(x | y=c) for each class c
- Calculate p(y=c | x) using Bayes rule
- Choose the most likely class c

- For a discrete x, can represent as a contingency table…
  - What about if we have more discrete features?

Features	# bad	# good
X=0	42	15
X=1	338	287
X=2	3	5

p(y)	383/690	307/690

⇒

p(x | y=0)	p(x | y=1)
42 / 383	15 / 307
338 / 383	287 / 307
3 / 383	5 / 307

⇒

p(y=0|x)	p(y=1|x)
.7368	.2632
.5408	.4592
.3750	.6250

Joint Bayes and Overfitting

- Estimate probabilities from the data
  - E.g., how many times (what fraction) did each outcome occur?

A	B	C	p(A,B,C | y=1)
0	0	0	4/10
0	0	1	1/10
0	1	0	0/10
0	1	1	0/10
1	0	0	1/10
1	0	1	2/10
1	1	0	1/10
1	1	1	1/10

- *m* data << 2^n parameters?
- What about the zeros?
  - We learn that certain combinations are impossible?
  - What if we see these later in test data?
- Overfitting!

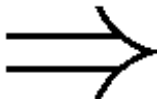
Independence and Naïve Bayes

- Another option: reduce the model complexity by assuming the features are independent of one another
- Independence:
- $p(a,b) = p(a)\, p(b)$
- $p(x_1, x_2, \ldots x_N \mid y=1) = p(x_1 \mid y=1)\, p(x_2 \mid y=1) \ldots p(x_N \mid y=1)$
- Only need to estimate each individually

A	p(A|y=1)
0	.4
1	.6

B	p(B |y=1)
0	.7
1	.3

C	p(C |y=1)
0	.1
1	.9



A	B	C	p(A,B,C | y=1)
0	0	0	.4 * .7 * .1
0	0	1	.4 * .7 * .9
0	1	0	.4 * .3 * .1
0	1	1	…
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Machine Learning

Overfitting & Cross-Validation

Nearest Neighbors

Linear Classification

Naïve Bayes Classifiers

Support Vector Machines

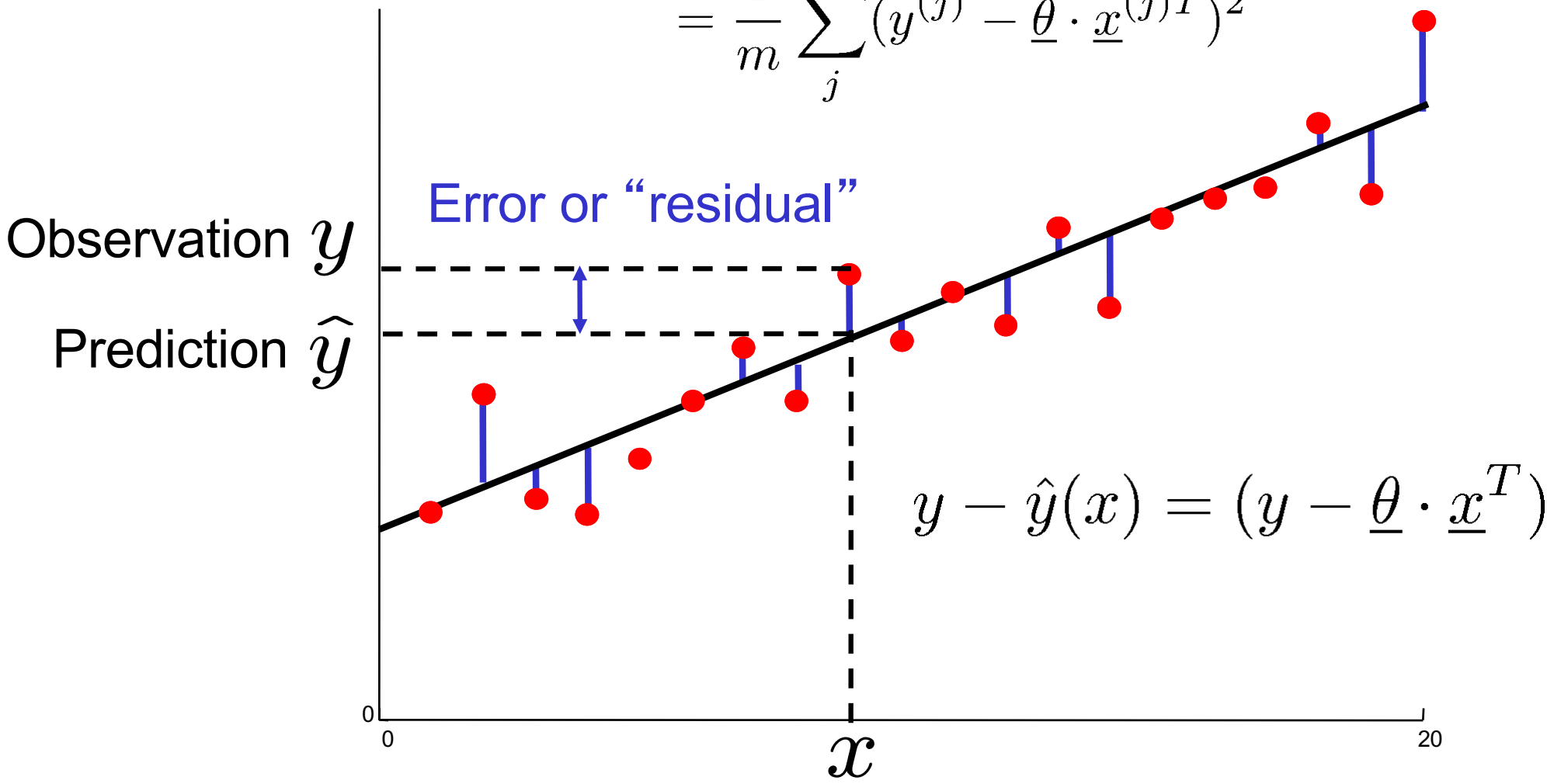
Linear Regression

Neural Networks

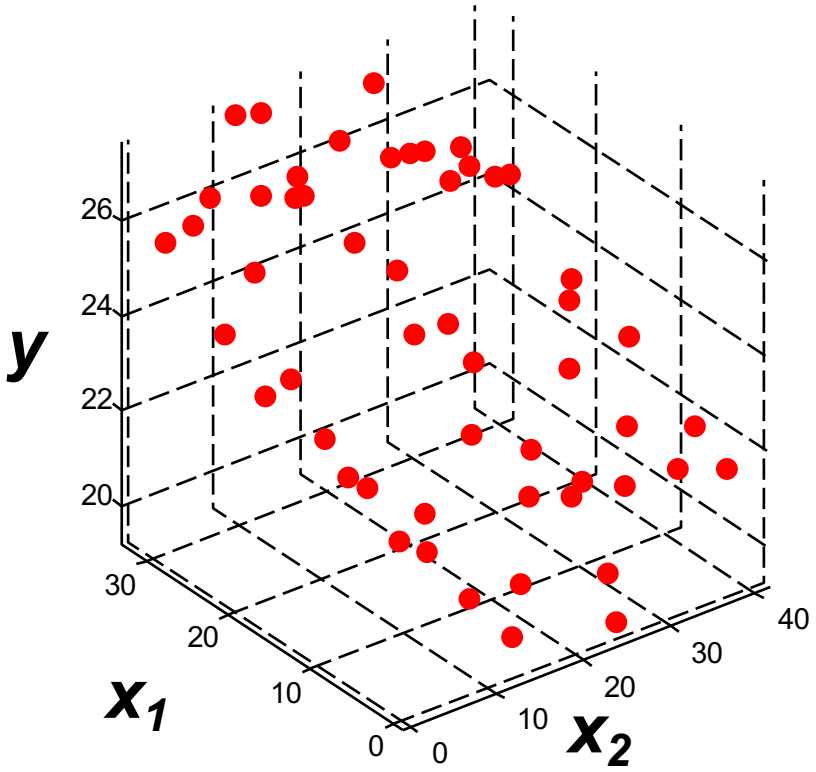
Regression & Mean Squared Error

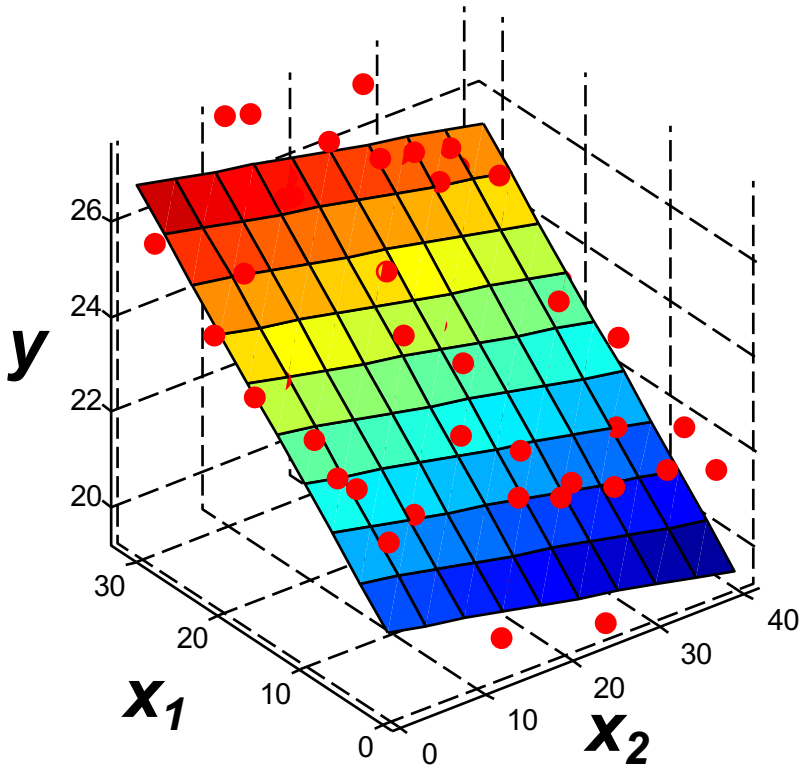
$$\text{MSE, } J(\underline{\theta}) = \frac{1}{m}\sum_j (y^{(j)} - \hat{y}(x^{(j)}))^2$$

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



Regression in Higher Dimensions



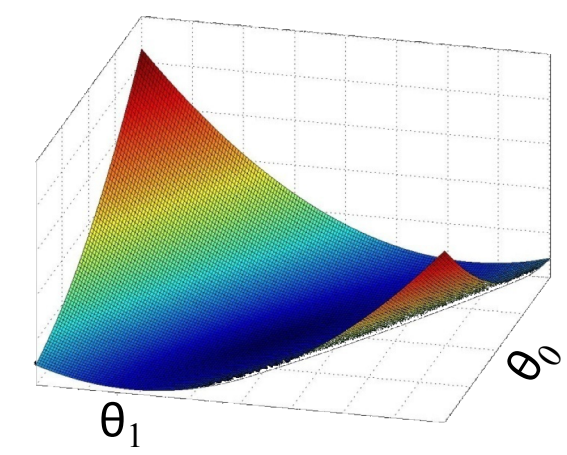


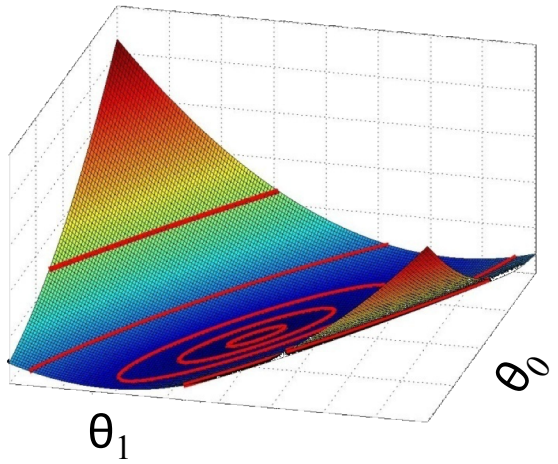
$$\hat{y}(x) = \underline{\theta} \cdot \underline{x}^T$$

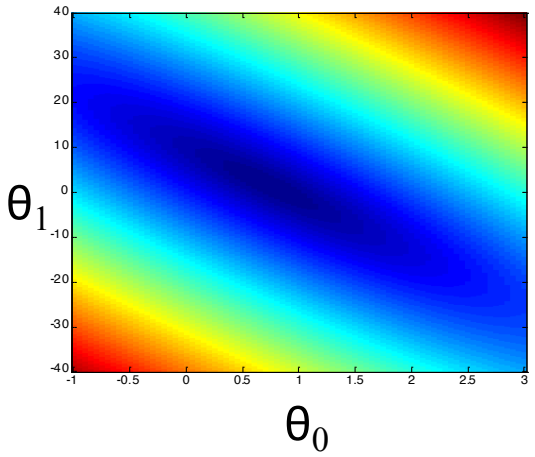
$$\underline{\theta} = [\theta_0 \ \theta_1 \ \theta_2]$$

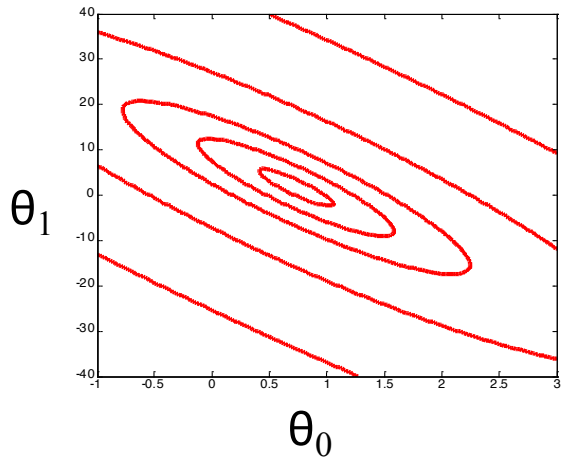
$$\underline{x} = [1 \ x_1 \ x_2]$$

Visualizing MSE loss function

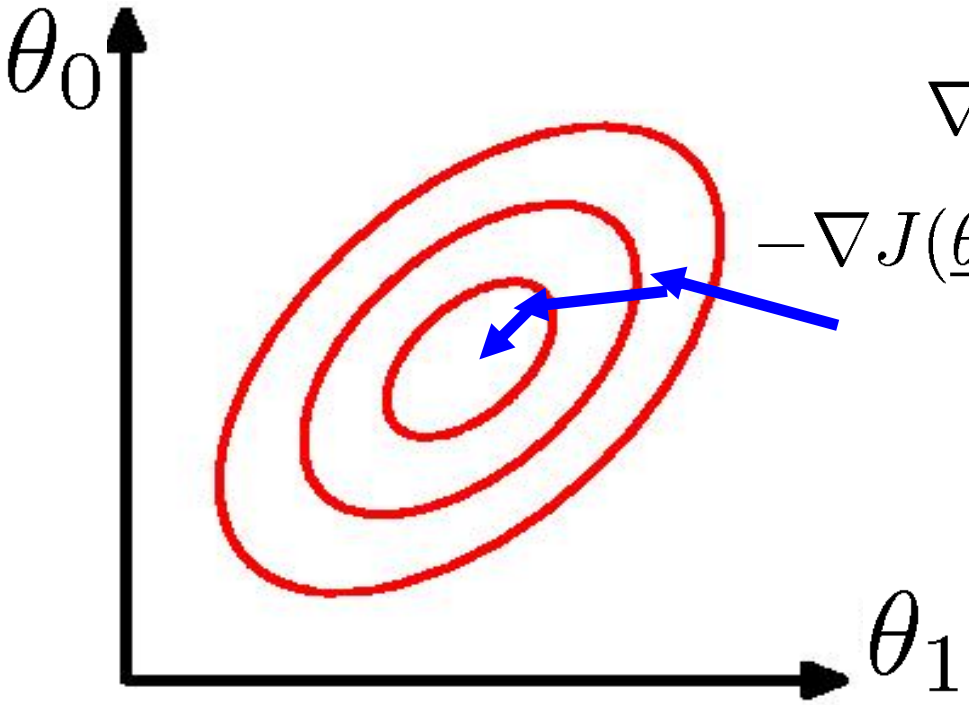








Gradient descent

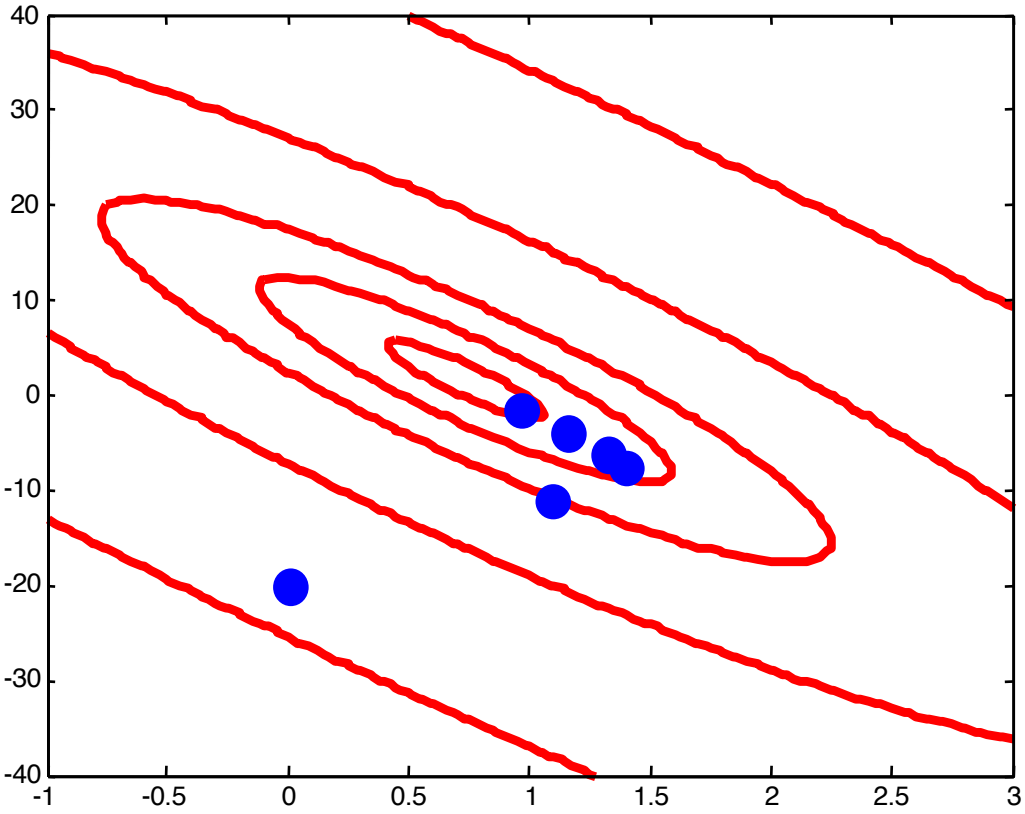


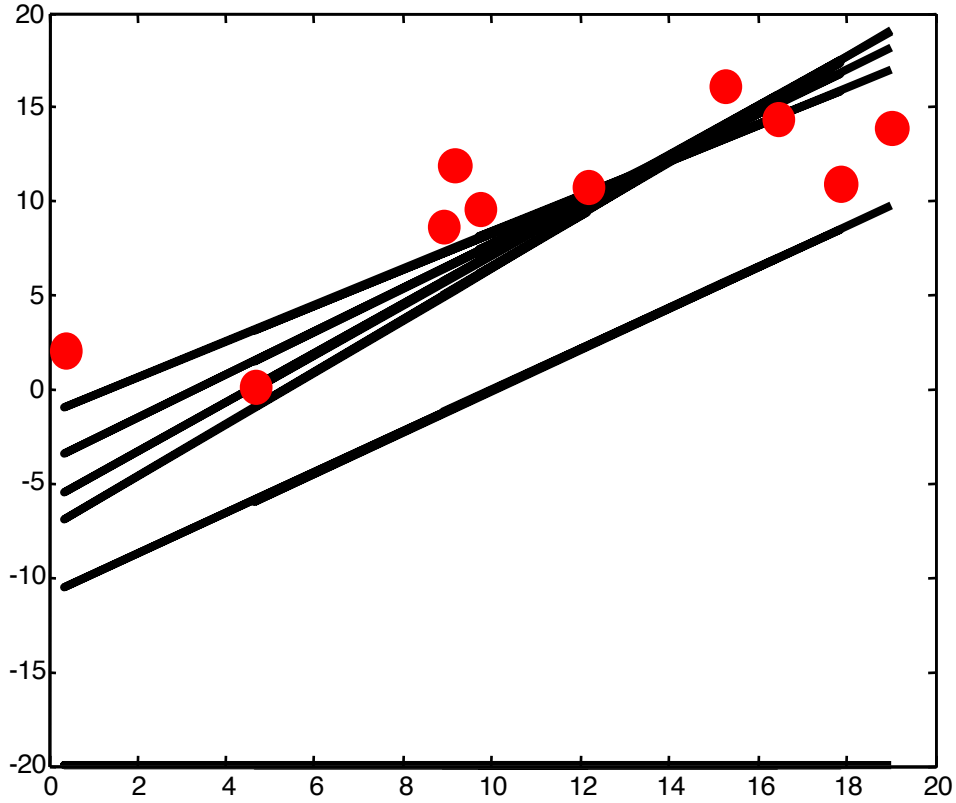
- **Gradient vector**

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

Indicates direction of steepest ascent (negative = steepest descent)

Gradient descent on cost function

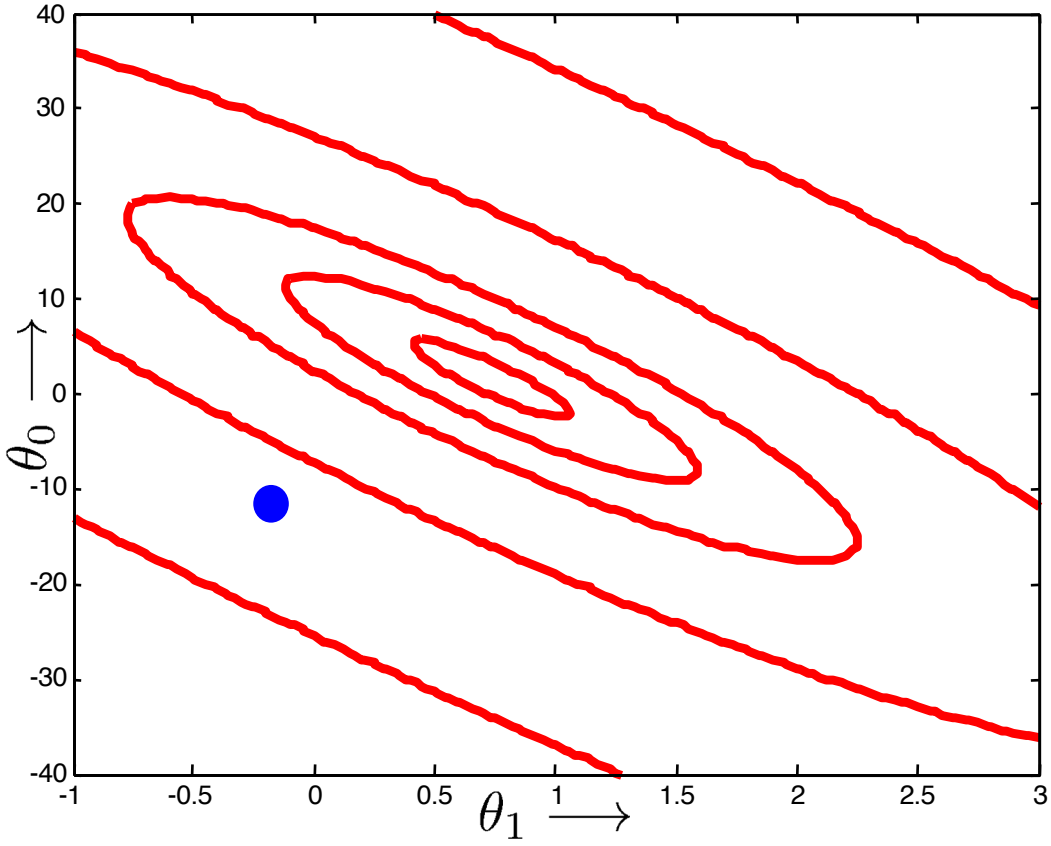


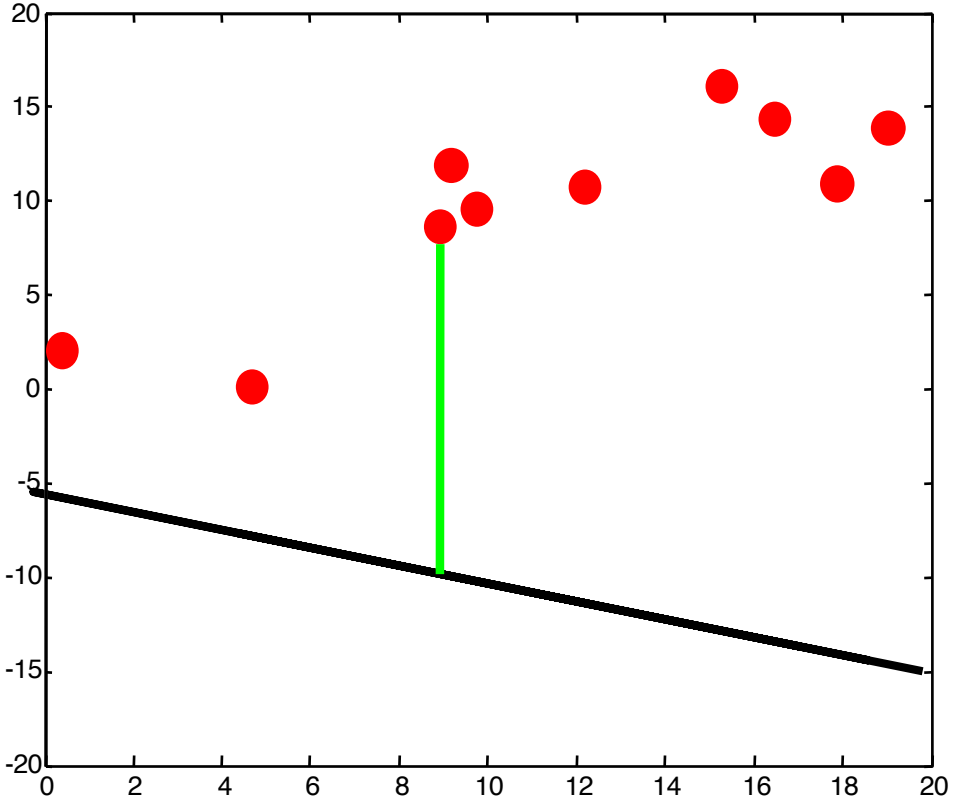


Online gradient descent

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

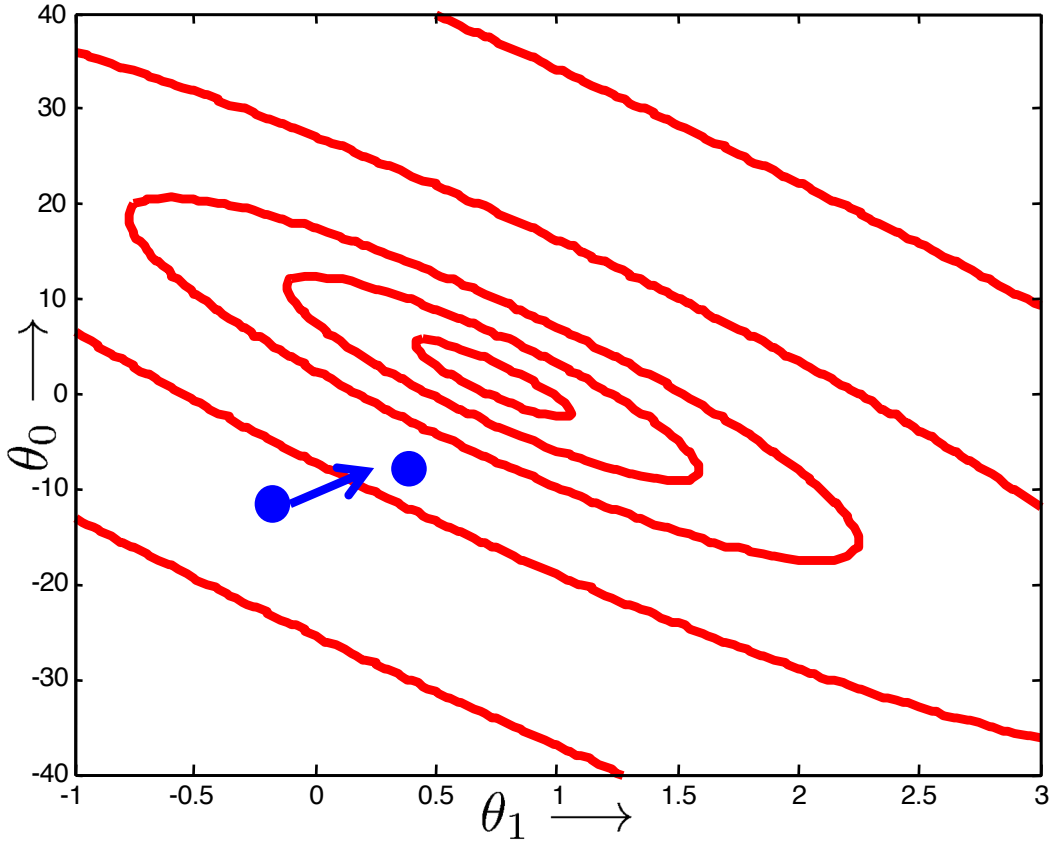
```
Initialize θ
Do {
 for j=1:m
   θ ← θ - α∇θJj(θ)
} while (not done)
```

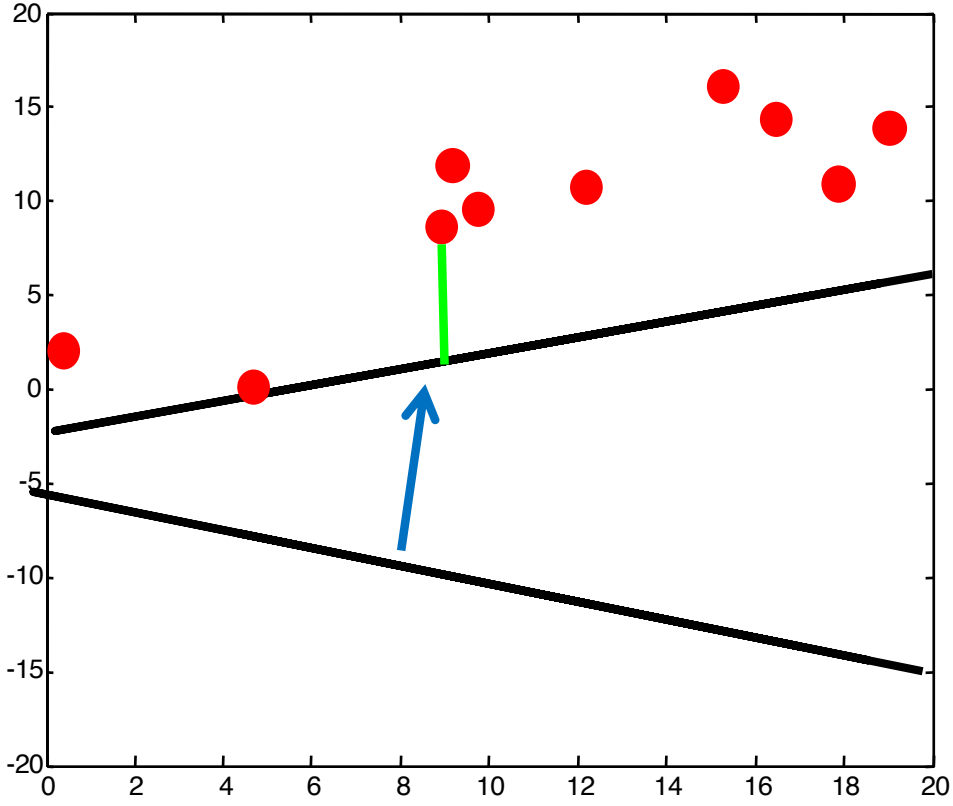




Online gradient descent

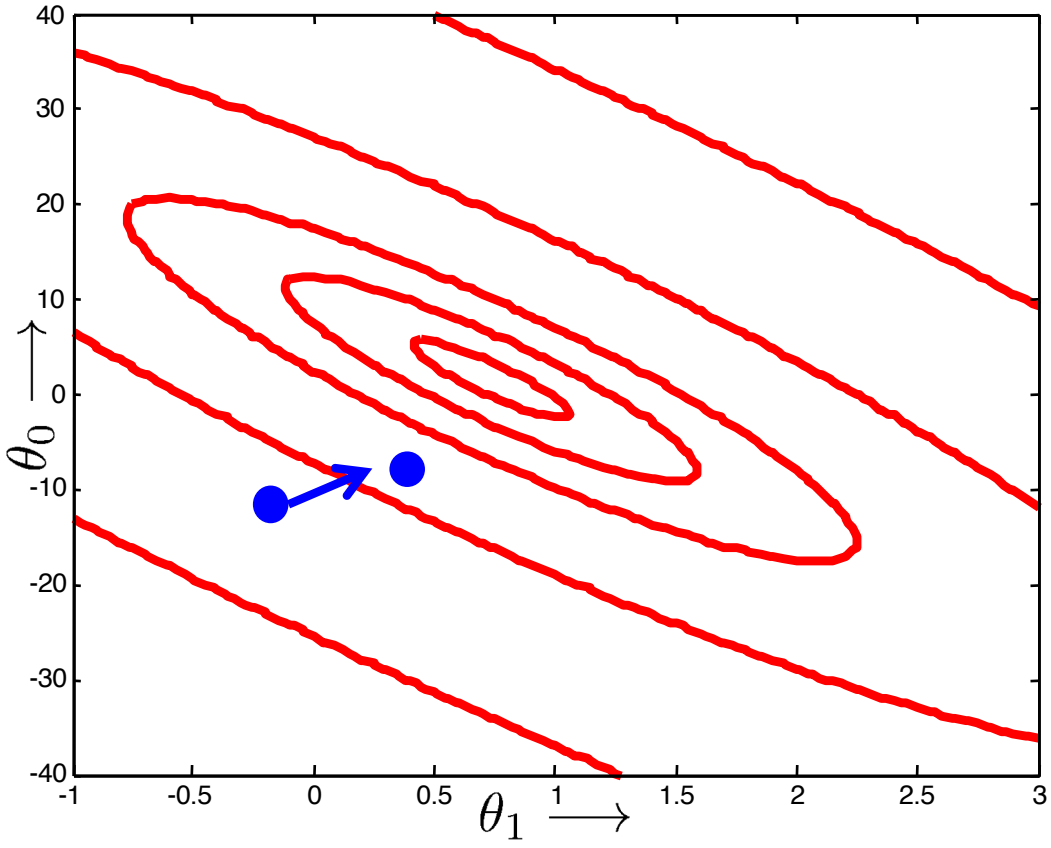
```
Initialize θ
Do {
 for j=1:m
   θ ← θ - α∇θJj(θ)
} while (not done)
```

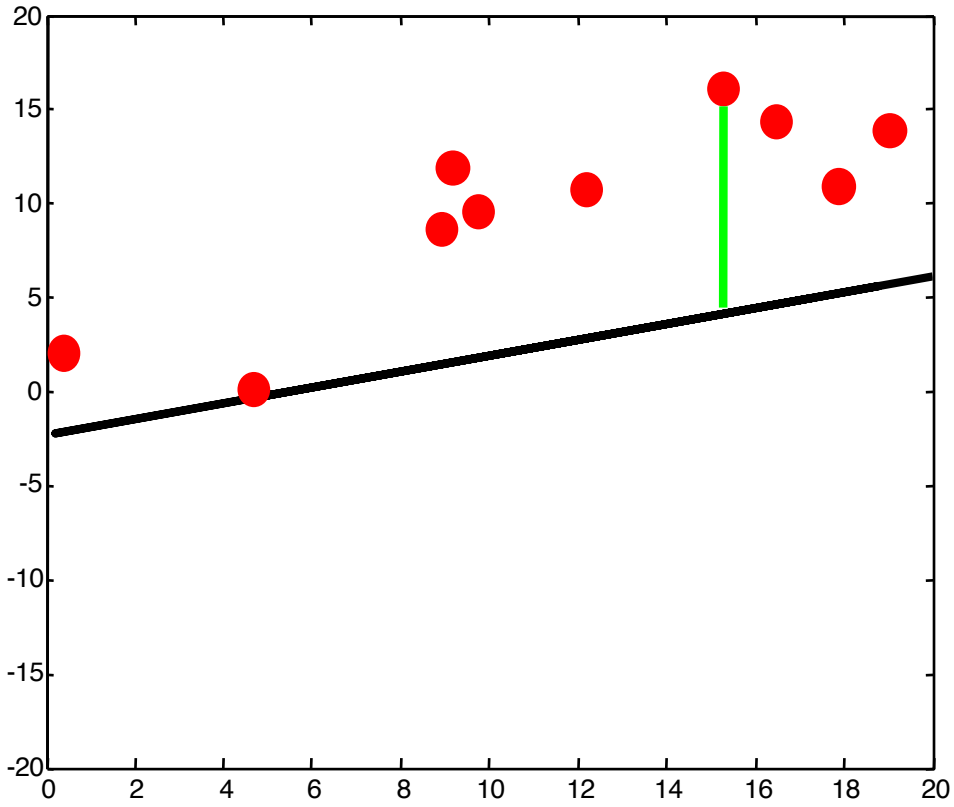




Online gradient descent

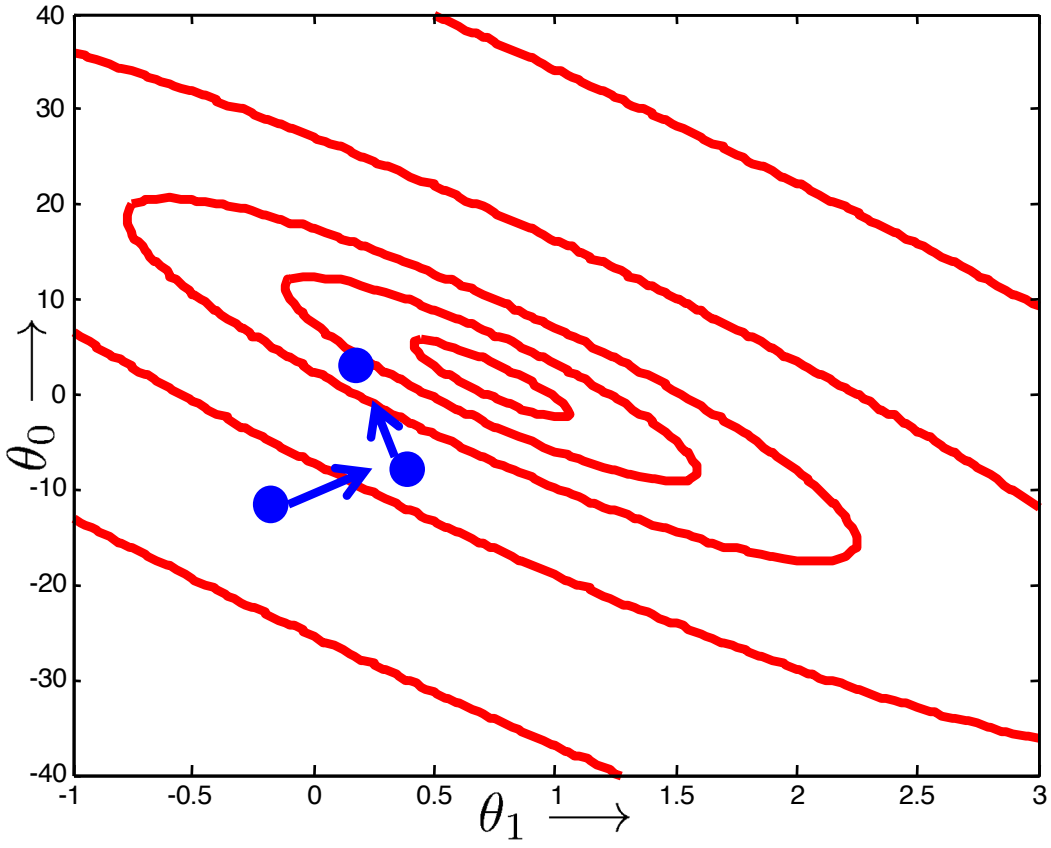
```
Initialize θ
Do {
 for j=1:m
   θ ← θ - α∇θJj(θ)
} while (not done)
```

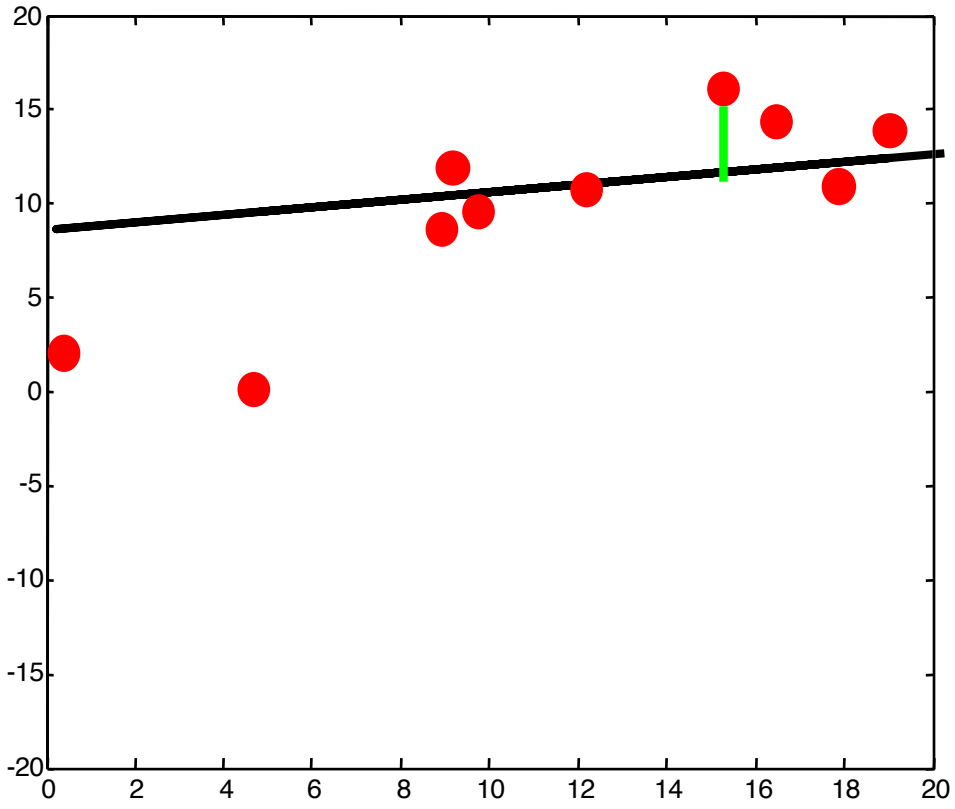




Online gradient descent

```
Initialize θ
Do {
 for j=1:m
   θ ← θ - α∇θJj(θ)
} while (not done)
```





Nonlinear functions

- Single feature x, predict target y:

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

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

Add features:

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

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

Linear regression in new features

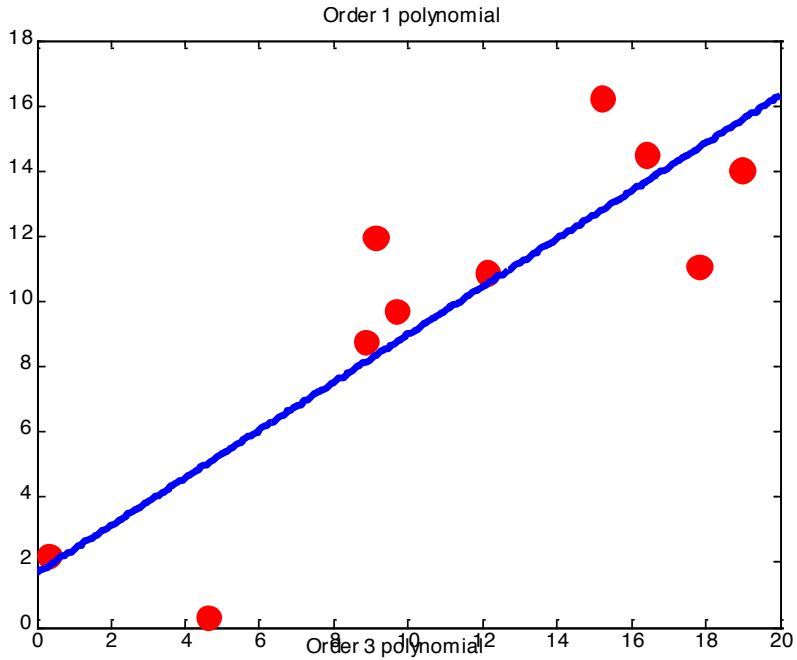
- Sometimes useful to think of "feature transform"

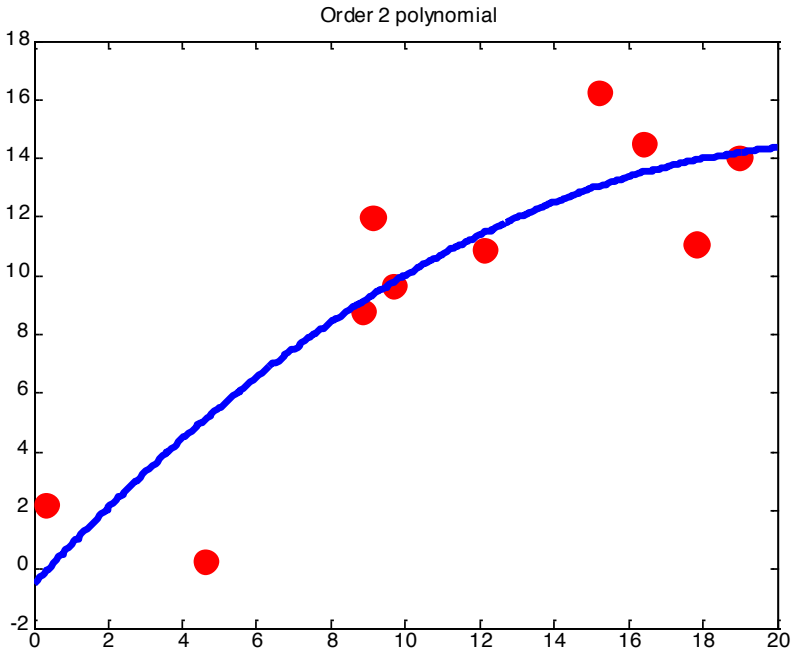
$$\Phi(x) = [1\,,\, x\,,\, x^2\,,\, x^3\,,\, \ldots]$$

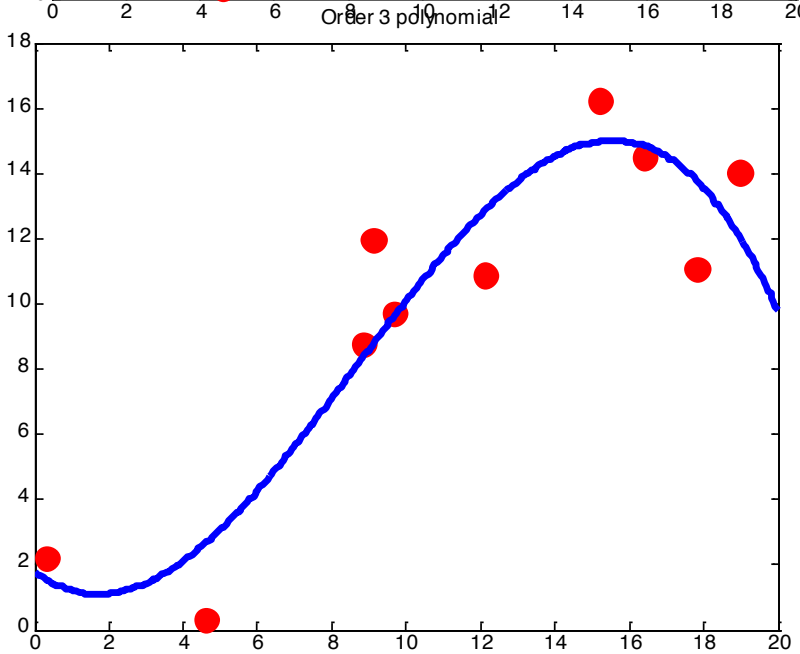
$$\hat{y}(x) = \underline{\theta} \cdot \Phi(x)$$

Higher-order polynomials

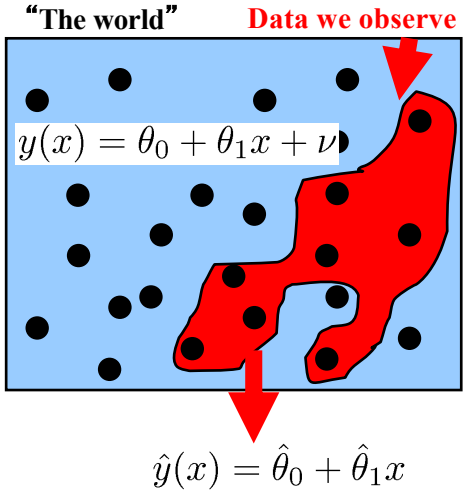
- Fit in the same way
- More “features”



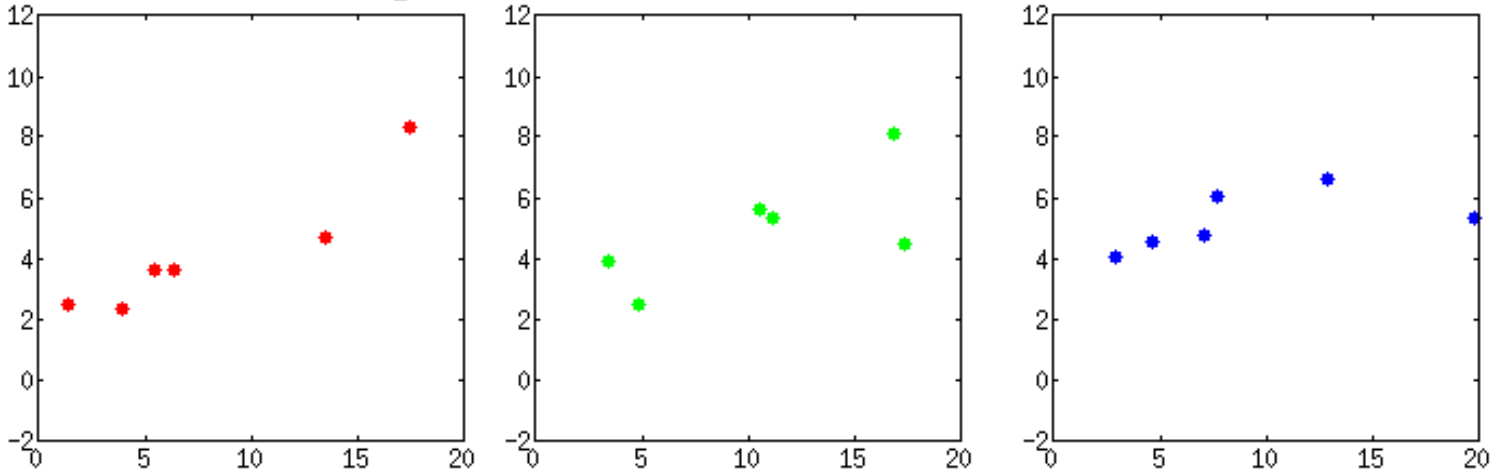




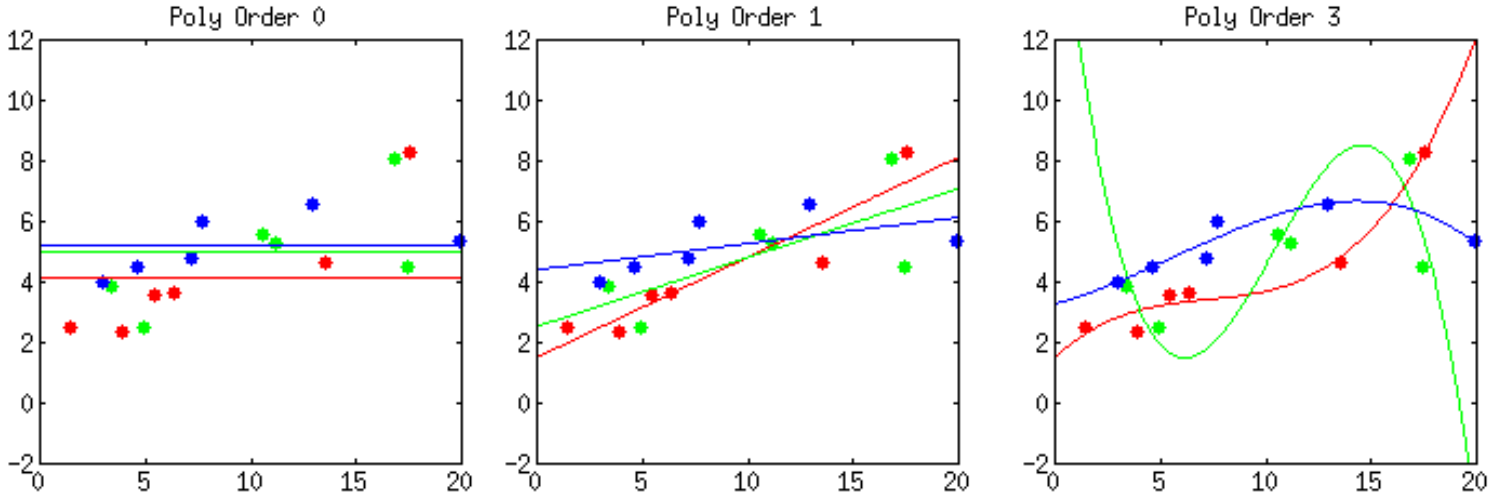
Bias & variance



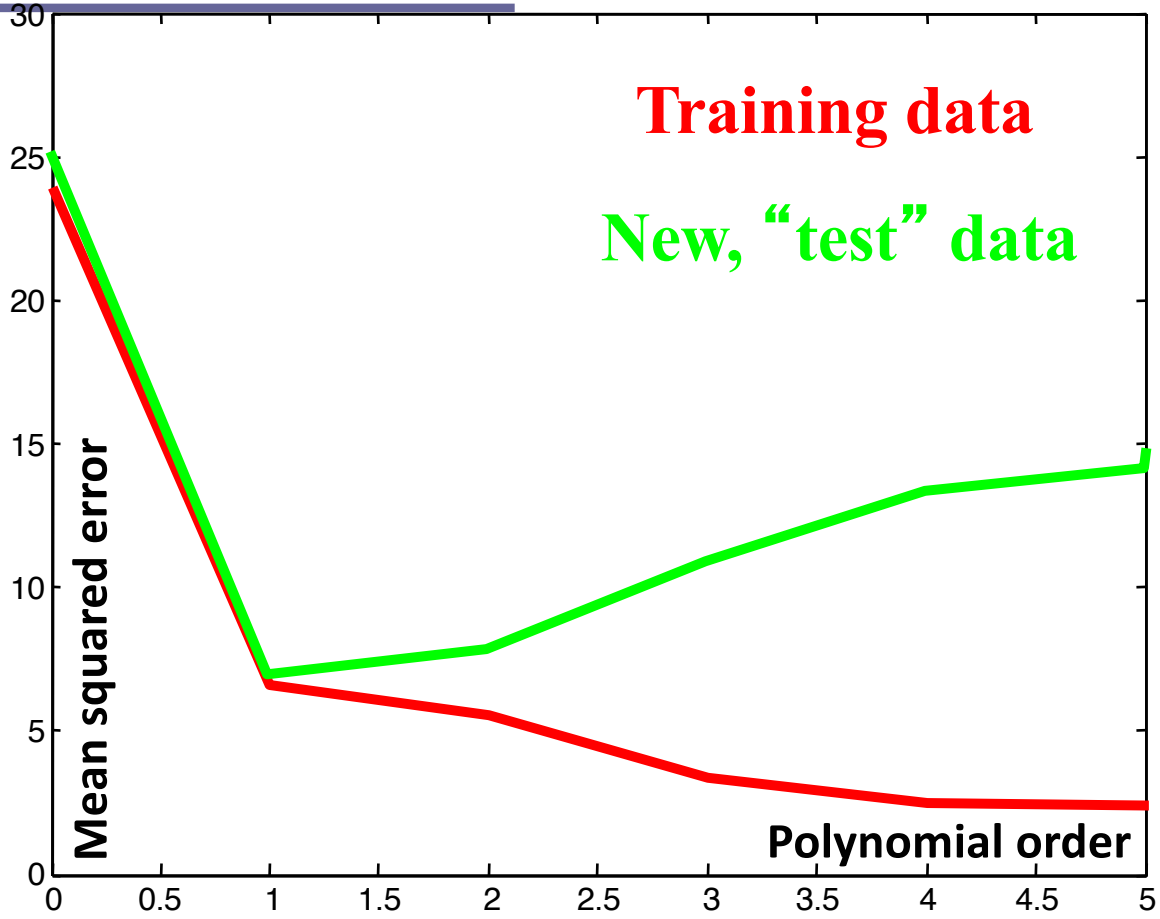
Three different possible data sets:

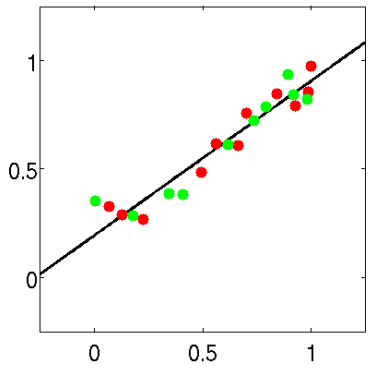


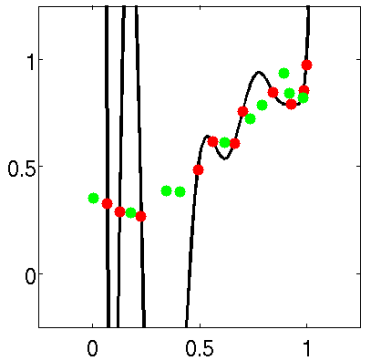
Each would give different predictors for any polynomial degree:



Training versus test error







Regularization

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

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

L_2 penalty:
"Ridge regression"

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

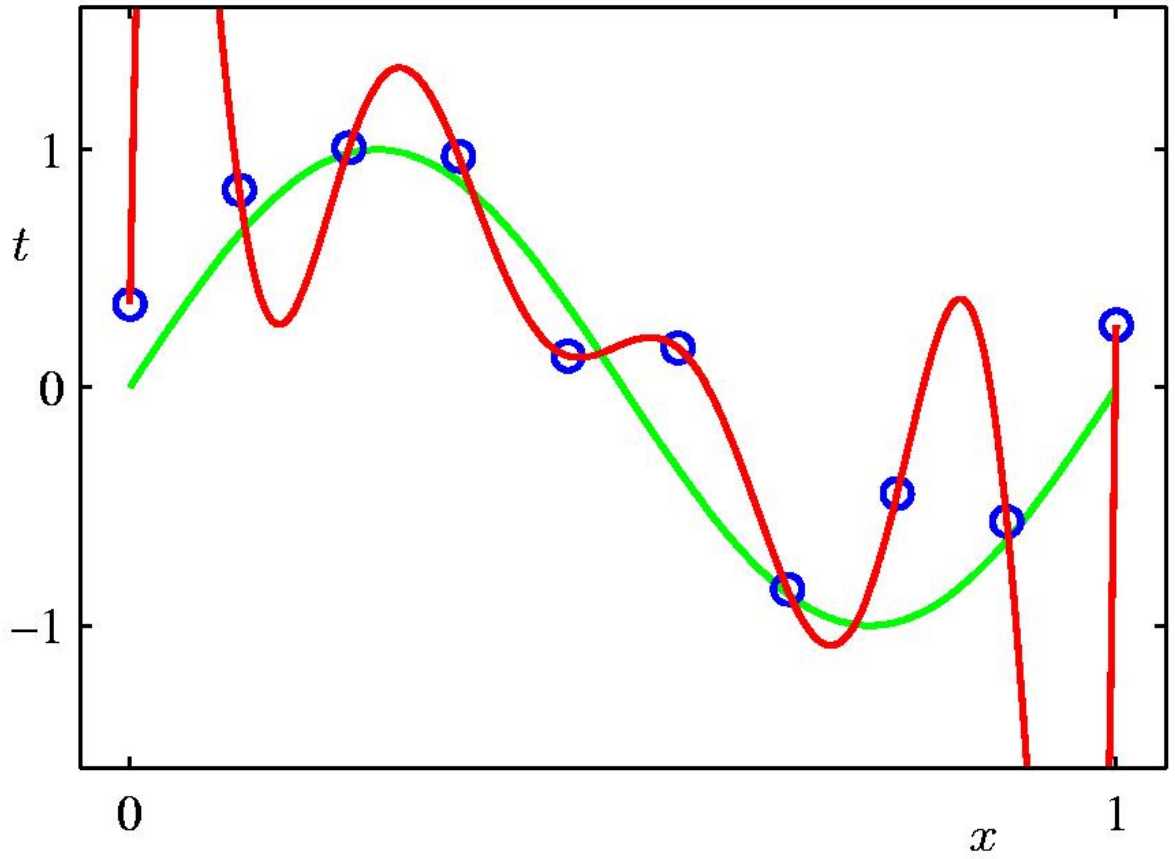
- New solution (derive the same way)

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

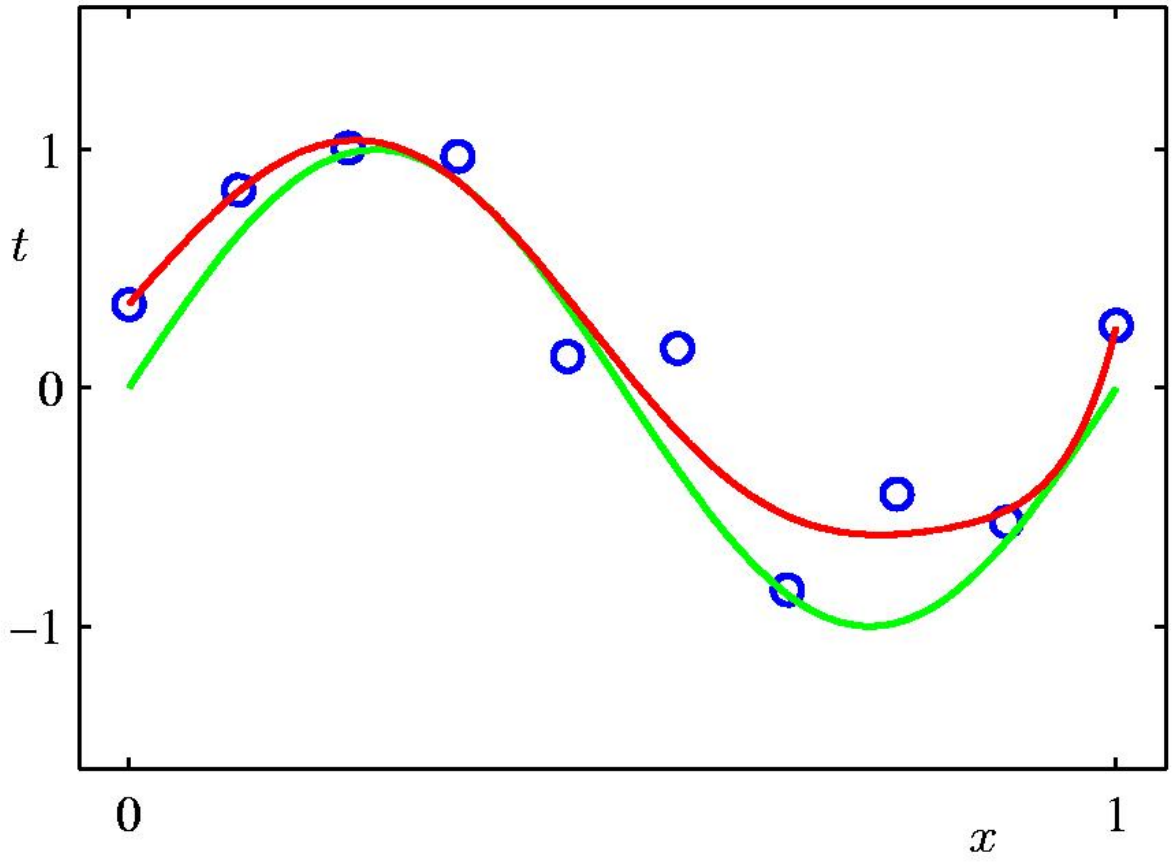
  - Problem is now well-posed for any degree

- 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

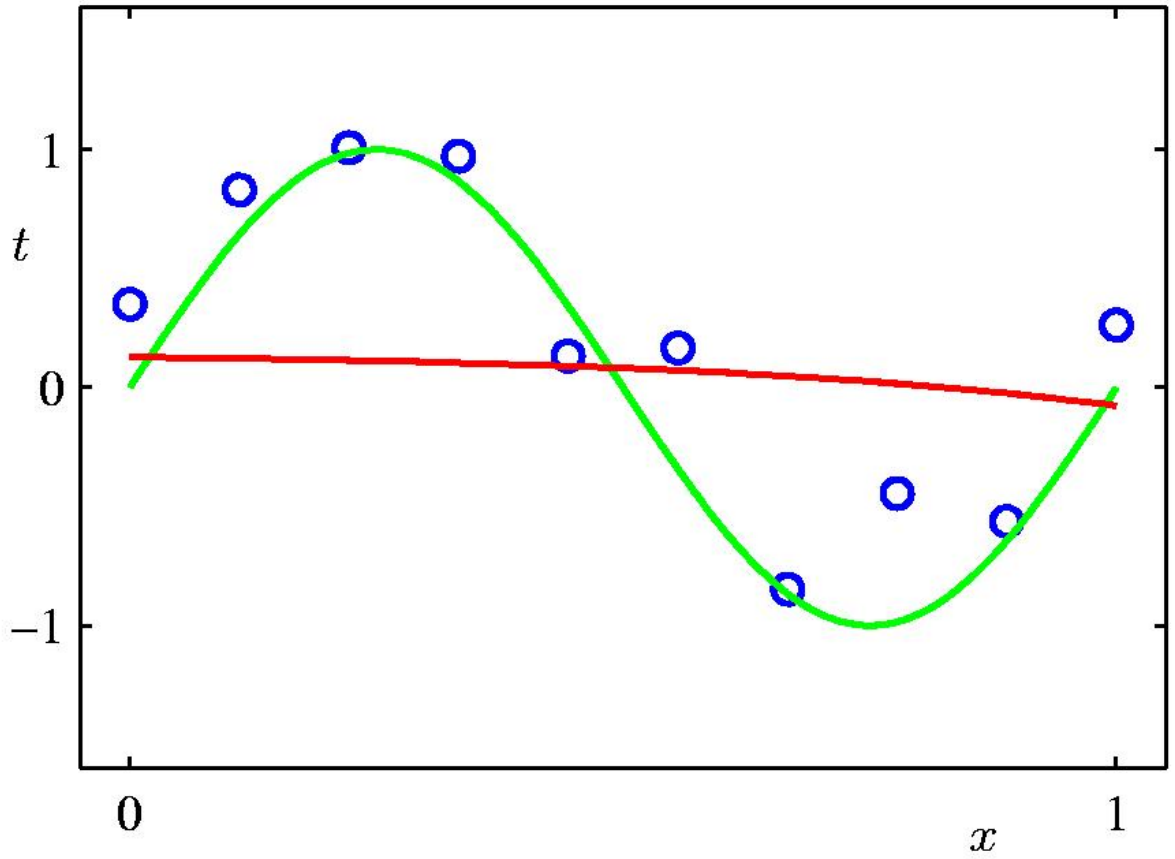
Regression: Zero Regularization



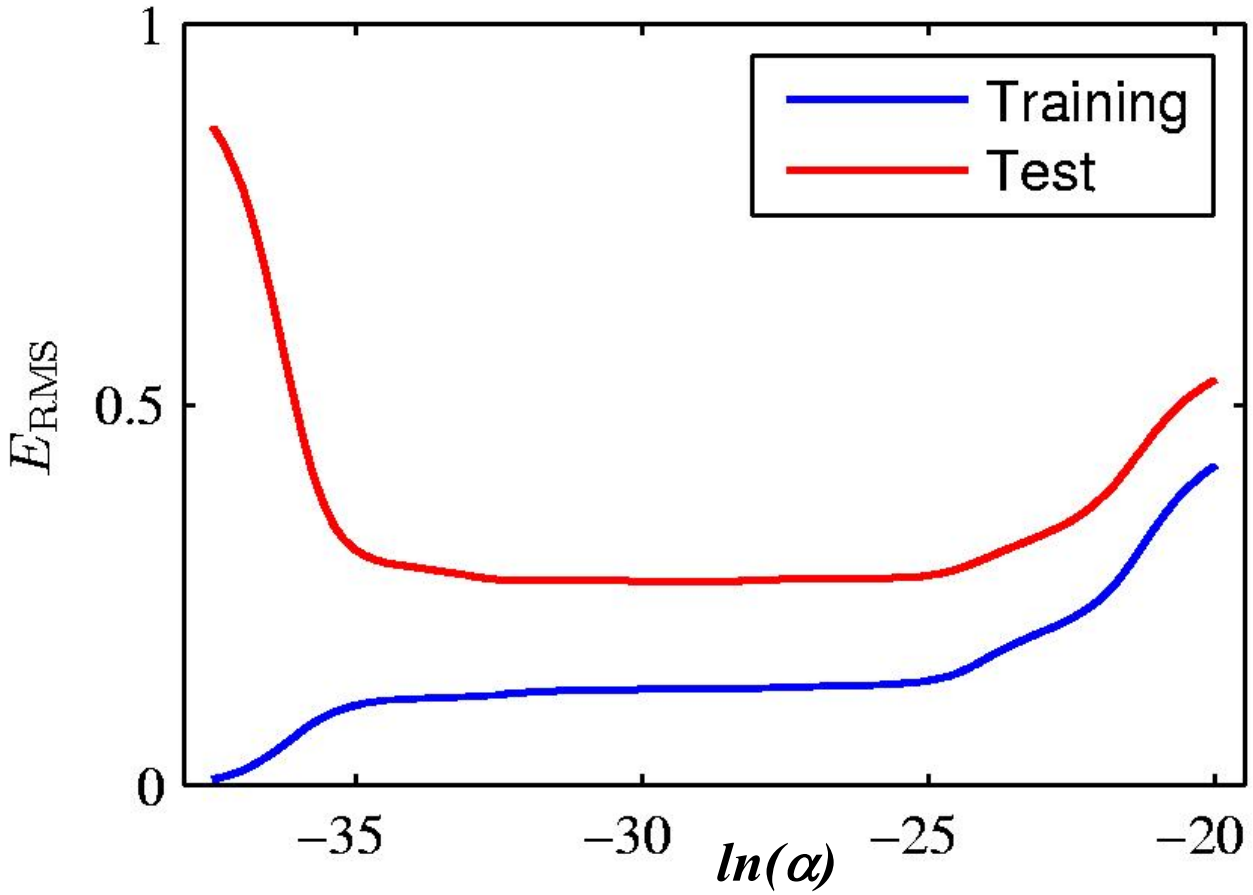
Regression: Moderate Regularization



Regression: Big Regularization



Impact of Regularization Parameter



Machine Learning

Overfitting & Cross-Validation

Nearest Neighbors

Linear Classification

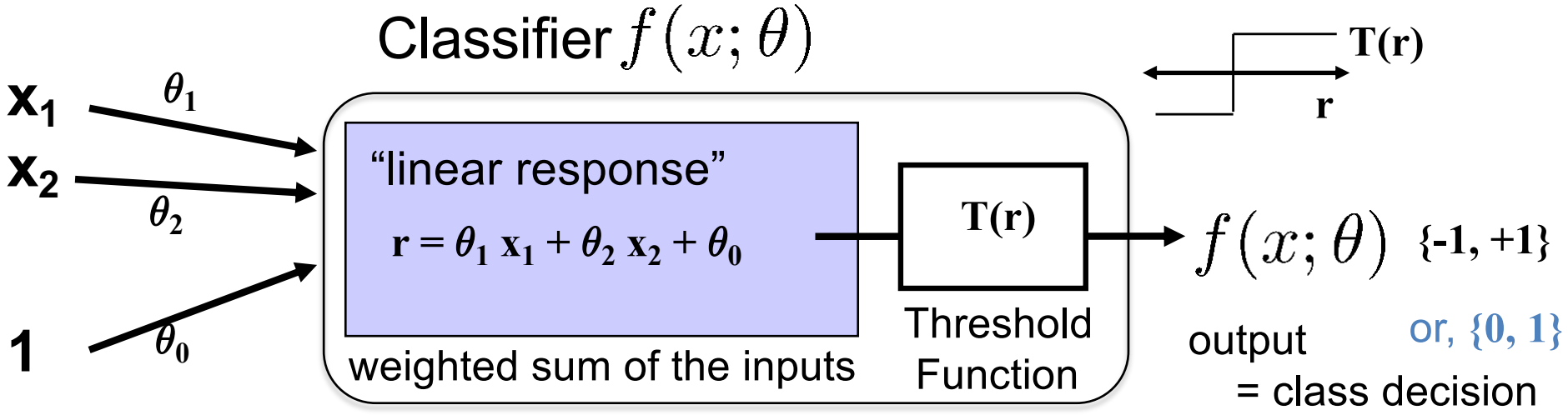
Naïve Bayes Classifiers

Support Vector Machines

Linear Regression

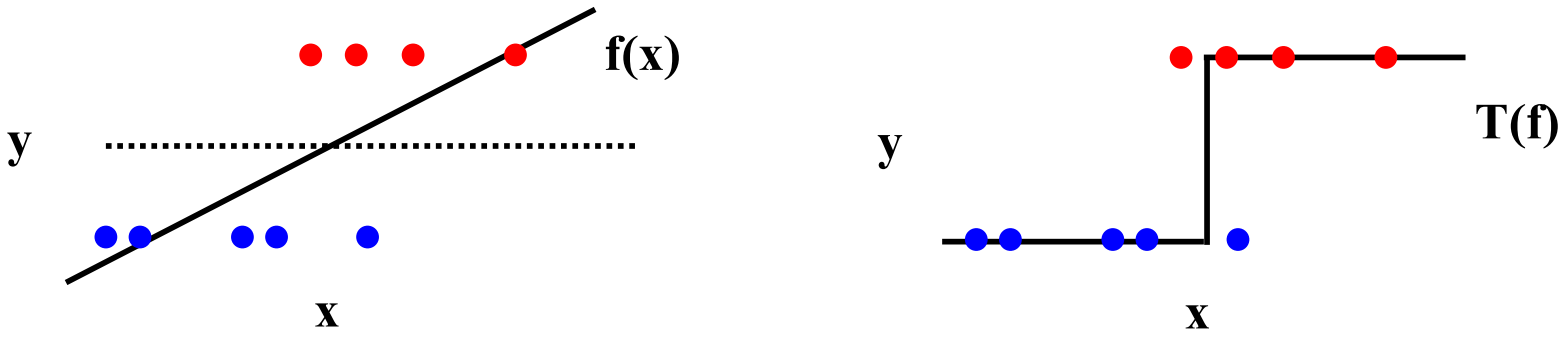
Neural Networks

Perceptron Classifier (2 features)



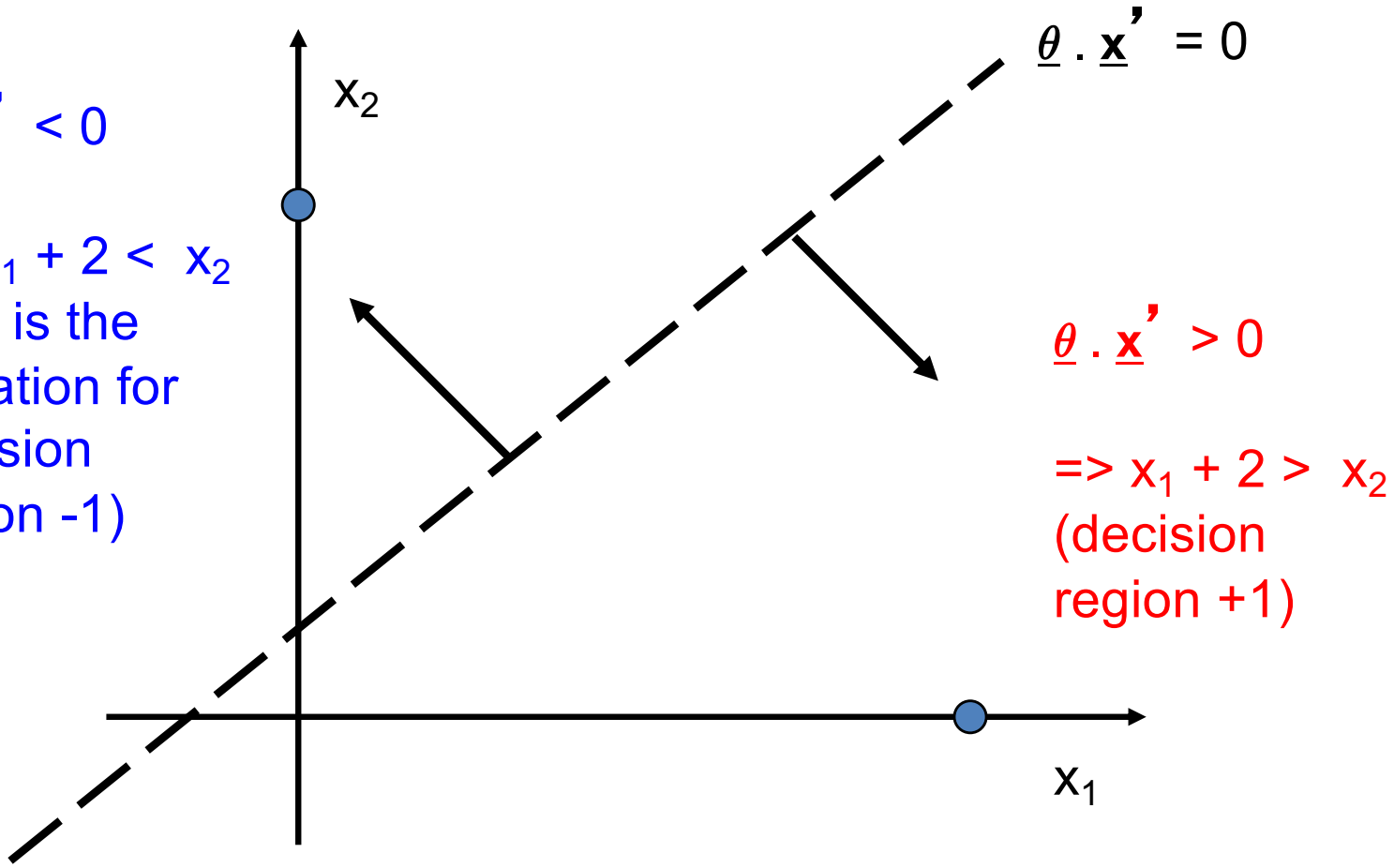

```
r = X.dot( theta.T )          # compute linear response
Yhat = (r > 0)                # predict class 1 vs 0
Yhat = 2*(r > 0)-1            # or "sign": predict +1 / -1
# Note: typically convert classes to "canonical" values 0,1,…
# then convert back ("learner.classes[c]") after prediction
```

Visualizing for one feature "x":



Example, Linear Decision Boundary

$\underline{\theta} = (\theta_0, \ \theta_1, \ \theta_2)$
$= (1, \ .5, \ -.5)$



From P. Smyth

Perceptron algorithm

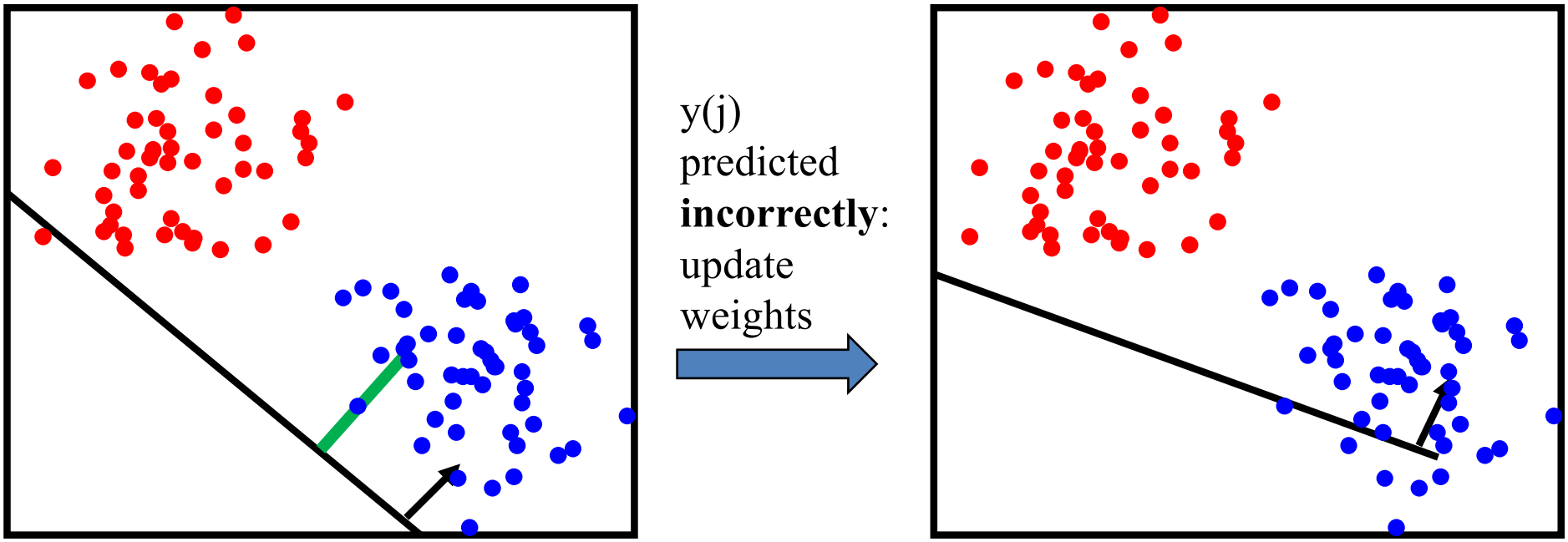
- Perceptron algorithm: an SGD-like algorithm

while $\neg$ done:

for each data point j:

$$\hat{y}^{(j)} = \text{sign}(\theta \cdot x^{(j)})$$ (predict output for point j)

$$\theta \leftarrow \theta + \alpha(y^{(j)} - \hat{y}^{(j)})x^{(j)}$$ ("gradient-like" step)



Perceptron algorithm

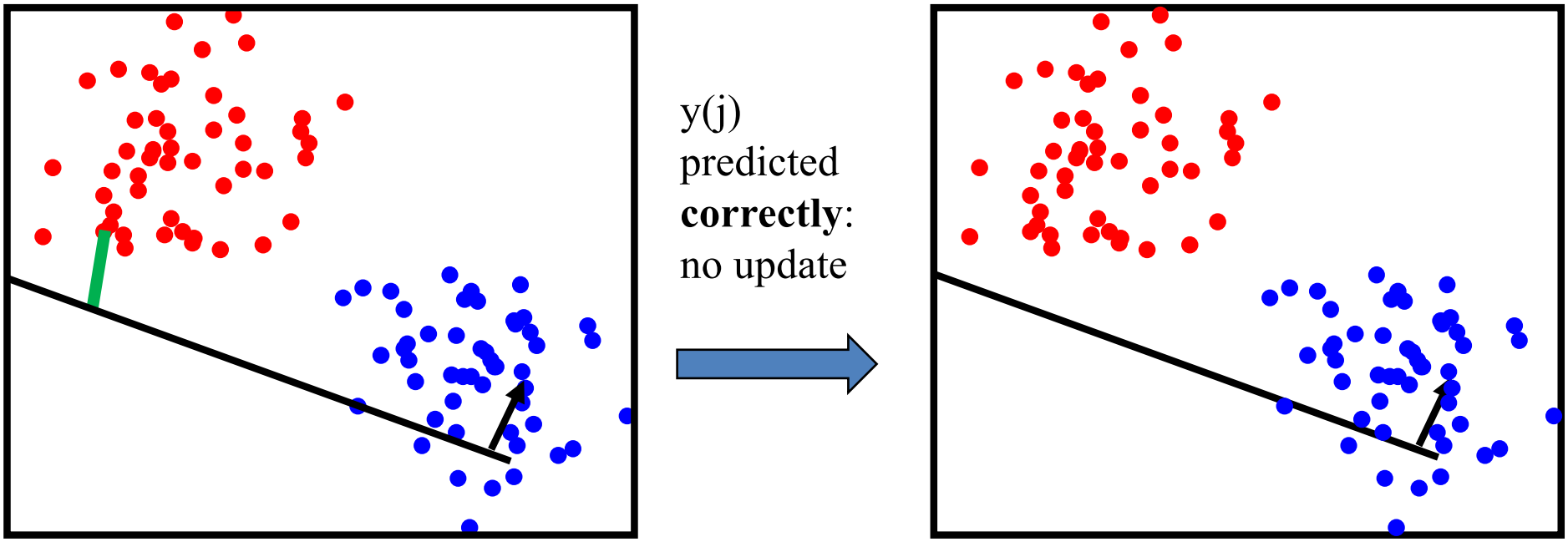
- Perceptron algorithm: an SGD-like algorithm

while $\neg$ done:

for each data point j:

$$\hat{y}^{(j)} = \text{sign}(\theta \cdot x^{(j)})$$ (predict output for point j)

$$\theta \leftarrow \theta + \alpha(y^{(j)} - \hat{y}^{(j)})x^{(j)}$$ ("gradient-like" step)



Perceptron algorithm

- Perceptron algorithm: an SGD-like algorithm

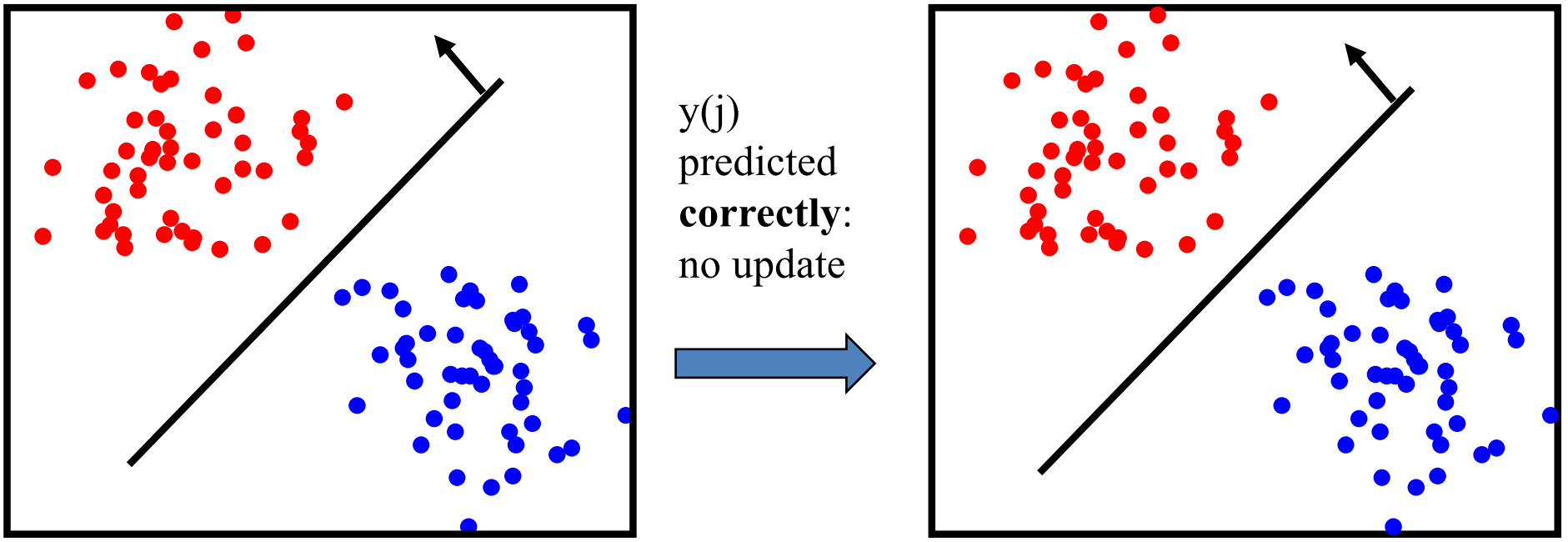
while $\neg$ done:

for each data point j:

$$\hat{y}^{(j)} = \text{sign}(\theta \cdot x^{(j)})$$ (predict output for point j)

$$\theta \leftarrow \theta + \alpha(y^{(j)} - \hat{y}^{(j)})x^{(j)}$$ ("gradient-like" step)

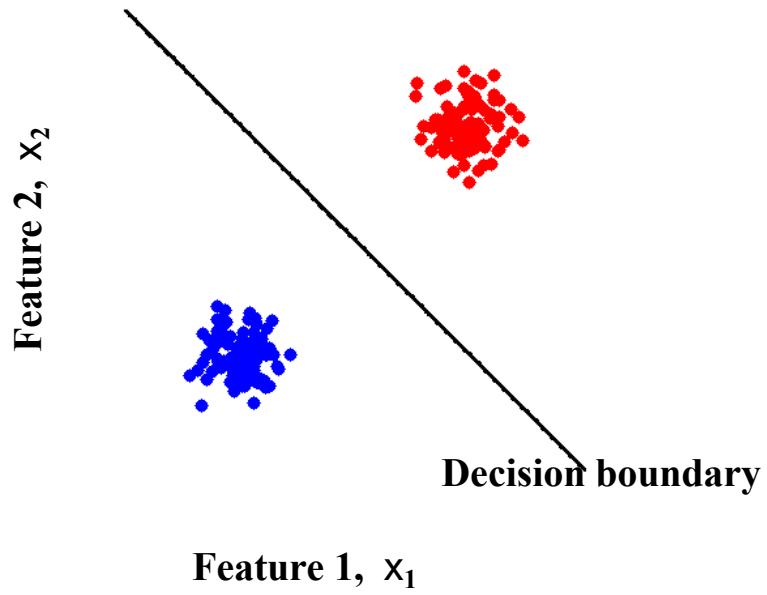
(Converges if data are linearly separable)



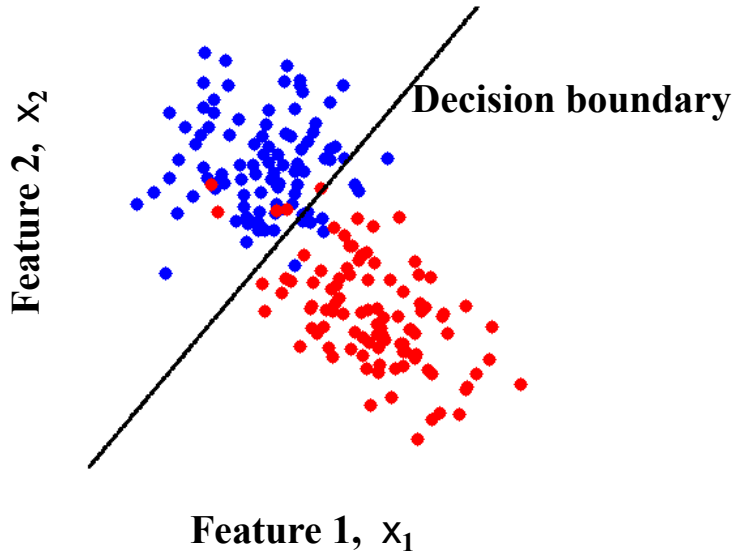
Separability

- A data set is separable by a learner if
  - There is some instance of that learner that correctly predicts all the data points
- Linearly separable data
  - Can separate the two classes using a straight line in feature space
  - in 2 dimensions the decision boundary is a straight line

Linearly separable data



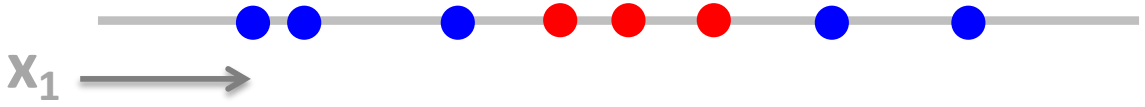
Linearly non-separable data



Adding features

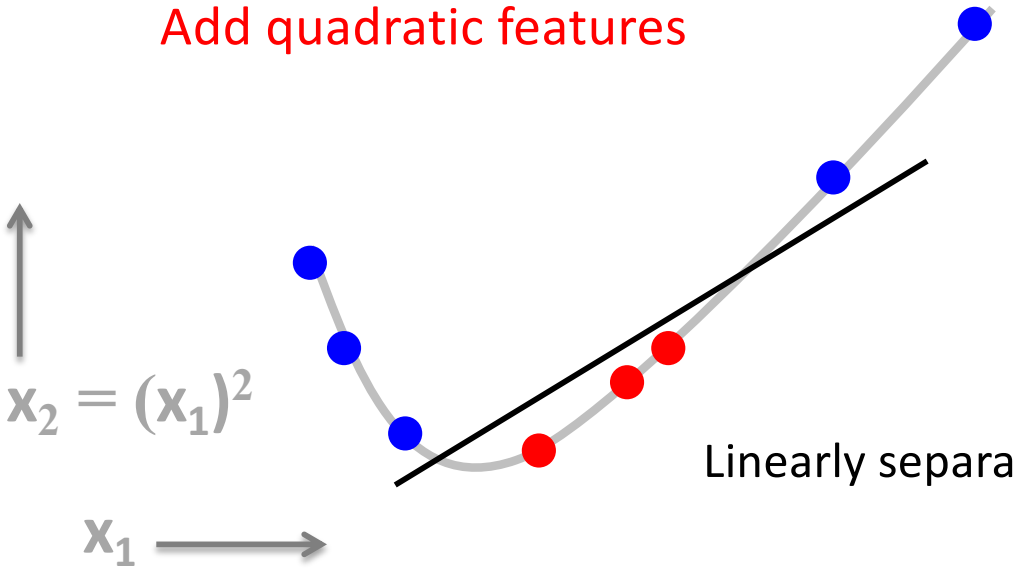
- Linear classifier can't learn some functions

1D example:



Not linearly separable

Add quadratic features

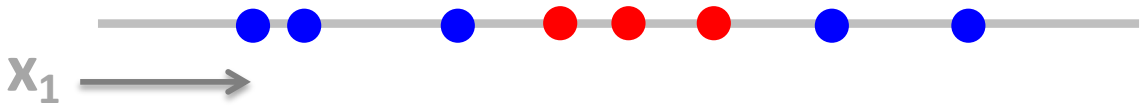


Linearly separable in new features…

Adding features

- Linear classifier can't learn some functions

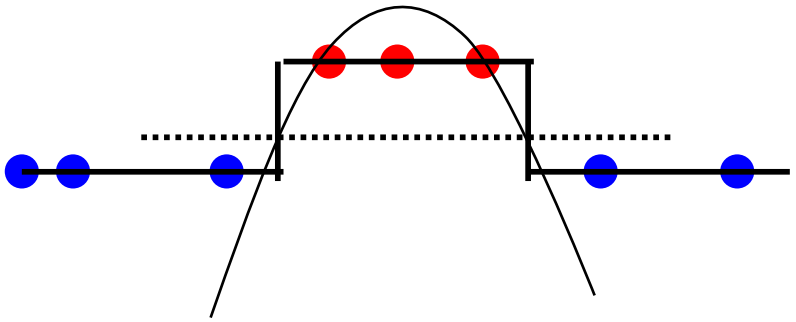
1D example:



Not linearly separable

Quadratic features, visualized in original feature space:

$$y = T(a x^2 + b x + c)$$



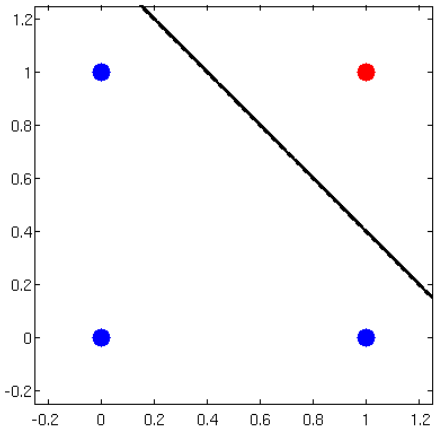
More complex decision boundary: $ax^2+bx+c = 0$

Representational Power of Perceptrons

- What mappings can a perceptron represent perfectly?
  - A perceptron is a linear classifier
  - thus it can represent any mapping that is linearly separable
  - some Boolean functions like AND (on left)
  - but not Boolean functions like XOR (on right)

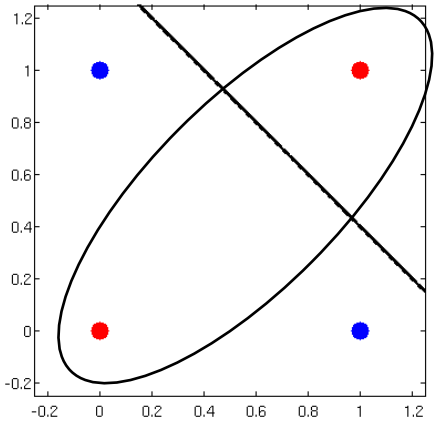
"AND"

x_1	x_2	y
0	0	-1
0	1	-1
1	0	-1
1	1	1



"XOR"

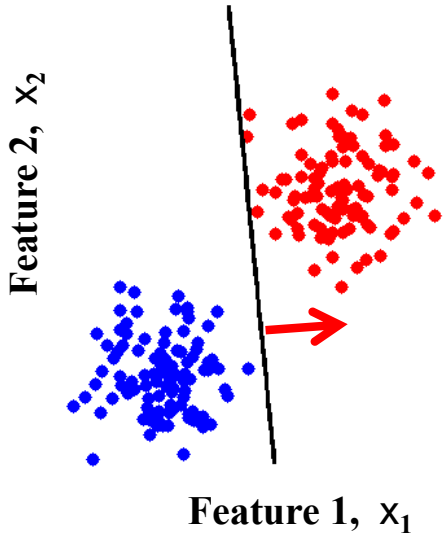
x_1	x_2	y
0	0	1
0	1	-1
1	0	-1
1	1	1

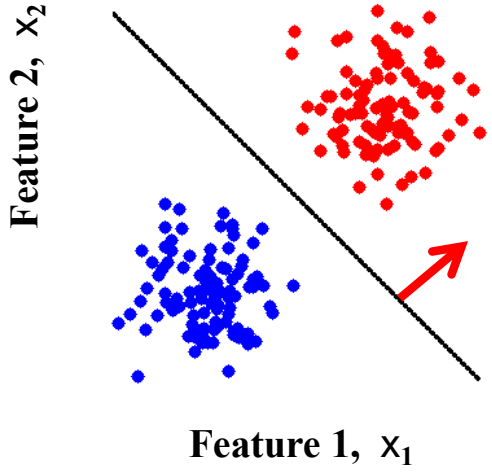


What kinds of functions would we need to learn the data on the right?
Ellipsiodal decision boundary: $a\,x_1^2 + b\,x_1 + c\,x_2^2 + d\,x_2 + e\,x_1x_2 + f = 0$

Beyond misclassification rate

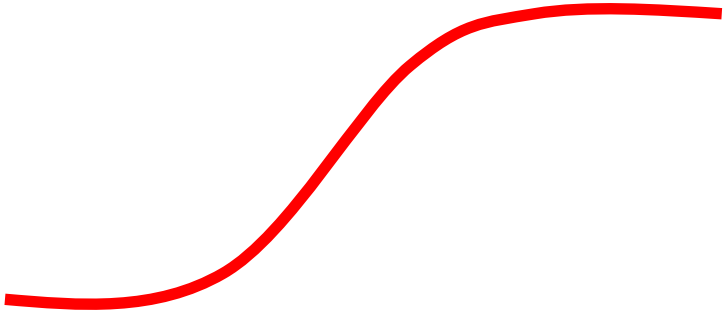
- Which decision boundary is “better”?
  - Both have zero training error (perfect training accuracy)
  - But, one of them seems intuitively better…





- Side benefit of many “smoothed” error functions
  - Encourages data to be far from the decision boundary
  - See more examples of this principle later...

Saturating Functions



- Many possible "saturating" functions
- "Logistic" sigmoid (scaled for range [0,1]) is
  $\sigma(z) = 1 / (1 + \exp(-z))$

  (z = linear response, $x^T\theta$)
- Derivative (slope of the function at a point z) is
  $\partial\sigma(z) = \sigma(z)\,(1-\sigma(z))$

  (to predict:
  threshold z at 0 or
  threshold σ (z) at ½)
- Python Implementation:

```
def sig(z):                              # logistic sigmoid
    return  1.0 / (1.0 + np.exp(-z) ) #   in [0,1]

def dsig(z):                             # its derivative at z
    return  sig(z) * (1-sig(z))
```

For range [-1 , +1]:

$\rho(z) = 2\,\sigma(z) - 1$

$\partial\rho(z) = 2\,\sigma(z)\,(1-\sigma(z))$

Predict: threshold z or ρ at zero

Regularized logistic regression

- Intepret $\sigma(\underline{\theta}\, x^T)$ as a probability that y = 1
- Use a negative log-likelihood loss function
  - If y = 1, cost is - log Pr[y=1] = - log $\sigma(\underline{\theta}\, x^T)$
  - If y = 0, cost is - log Pr[y=0] = - log (1 - $\sigma(\underline{\theta}\, x^T)$)

- Minimize weighted sum of negative log-likelihood and a regularizer that encourages small weights:

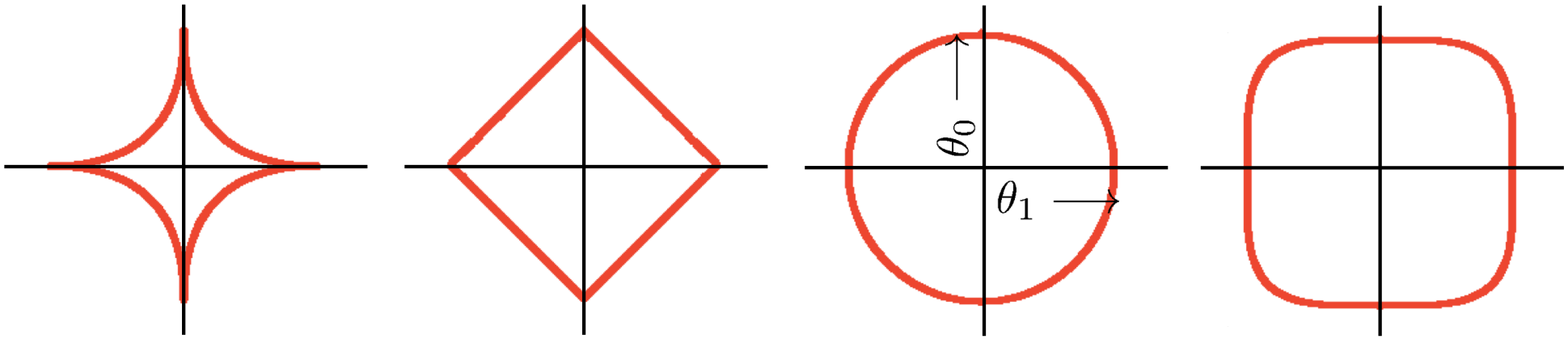
$$J(\underline{\theta}) = -\frac{1}{m}\left(\sum_i \underbrace{y^{(i)} \log \sigma(\theta \cdot x^{(i)})}_{\text{Nonzero only if y=1}} + \underbrace{(1-y^{(i)}) \log(1-\sigma(\theta \cdot x^{(i)}))}_{\text{Nonzero only if y=0}}\right)$$

$$+\alpha ||\theta||_p$$

Different regularization functions

- In general, for the L_p regularizer: $\left(\sum_i |\theta_i|^p \right)^{\frac{1}{p}} = ||\theta||_p$

Isosurfaces: $\|\theta\|_p = \text{constant}$



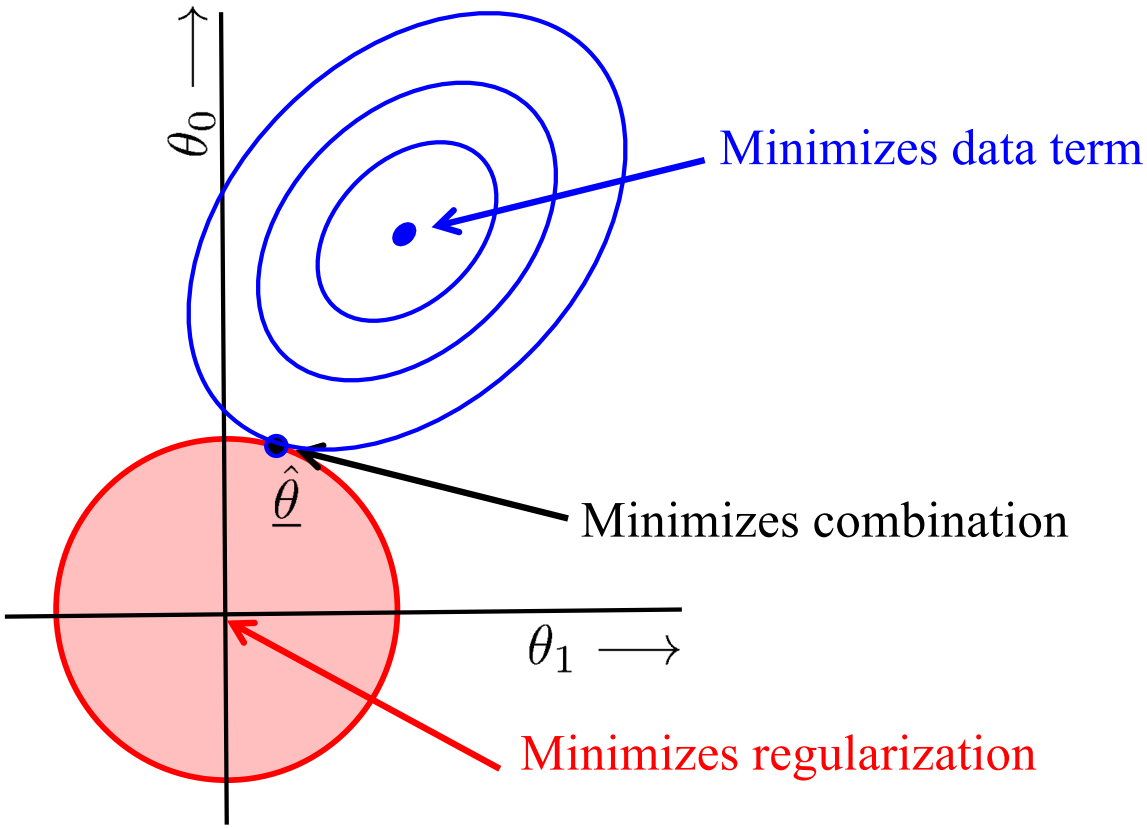
p=0.5 p=1 p=2 p=4

Lasso Quadratic

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

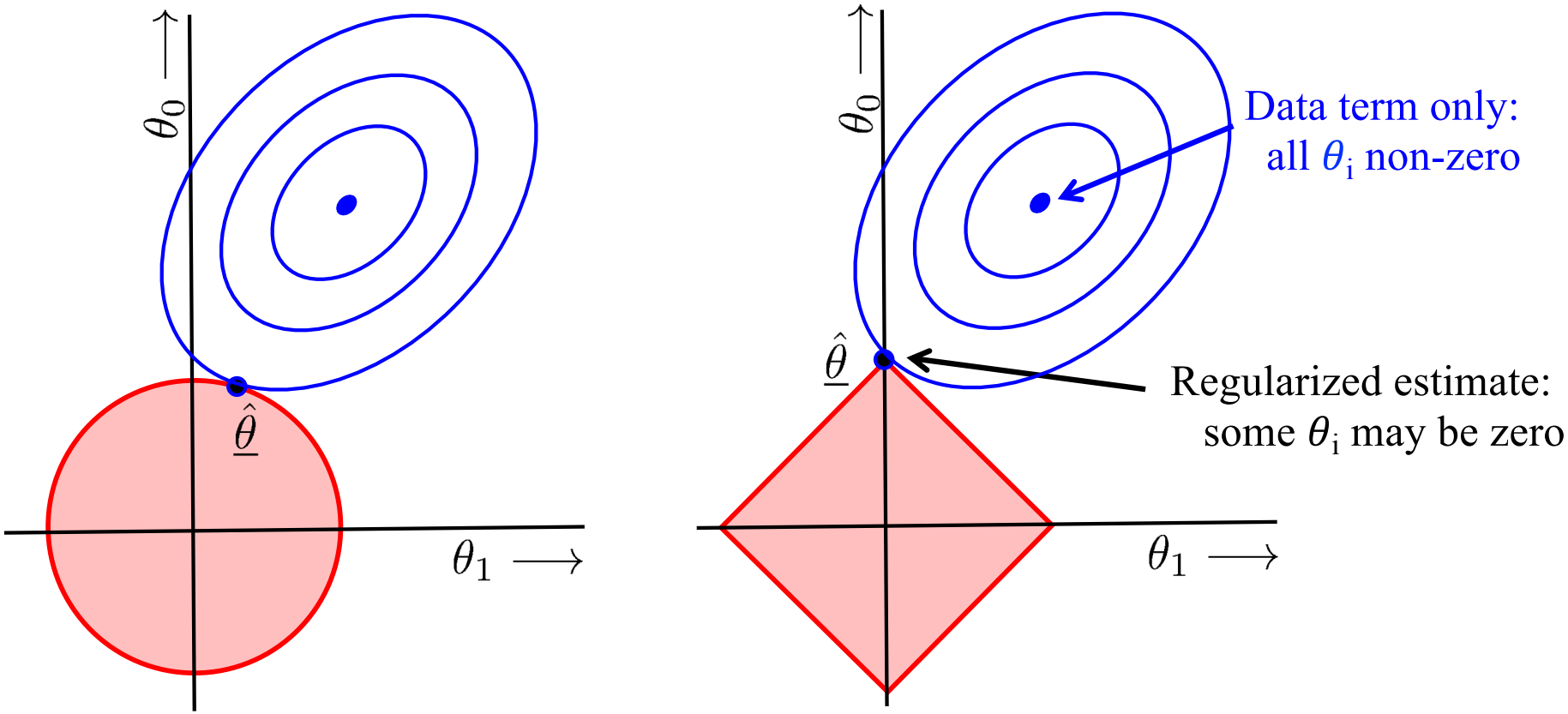
Regularization: L_2 vs L_1

- 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.



Machine Learning

Overfitting & Cross-Validation

Nearest Neighbors

Linear Classification

Naïve Bayes Classifiers

Support Vector Machines

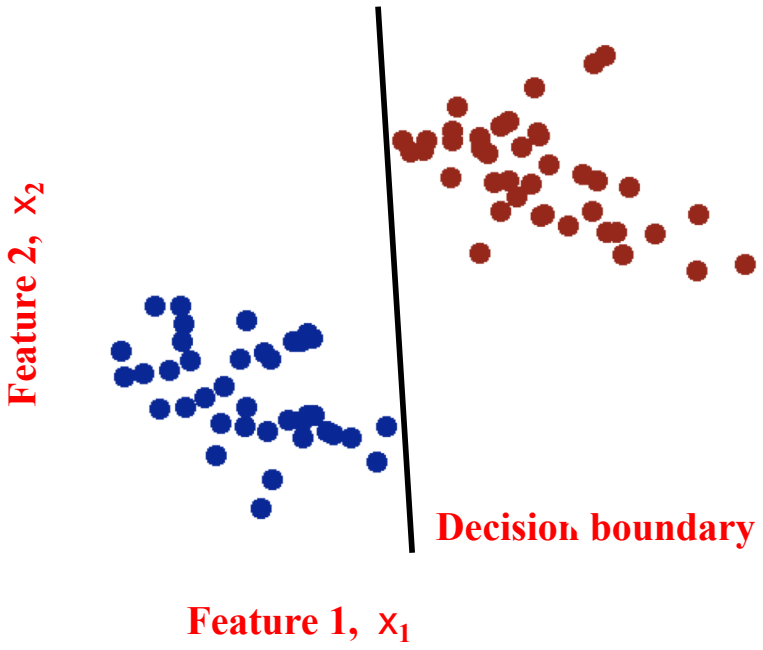
Linear Regression

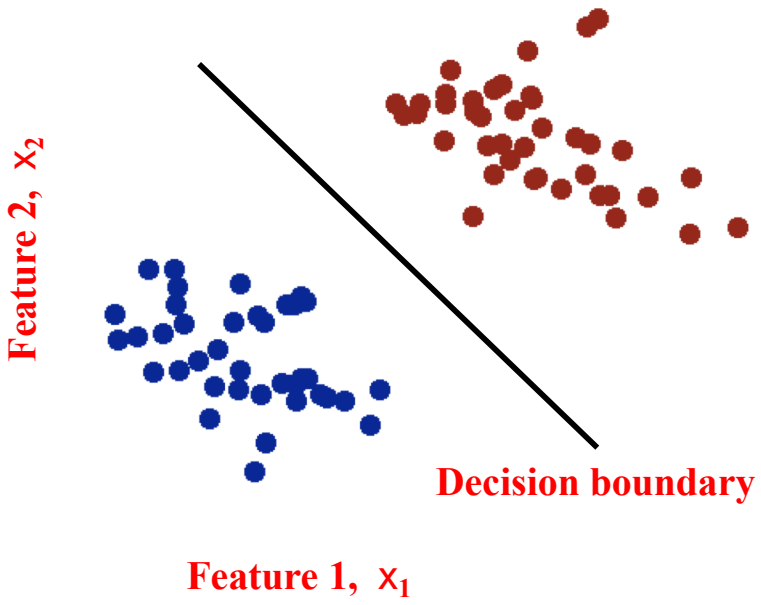
Neural Networks

Linear classifiers

- Which decision boundary is "better"?
  - Both have zero training error (perfect training accuracy)
  - But, one of them seems intuitively better…

- How can we quantify "better",
  and learn the "best" parameter settings?





One possible answer…

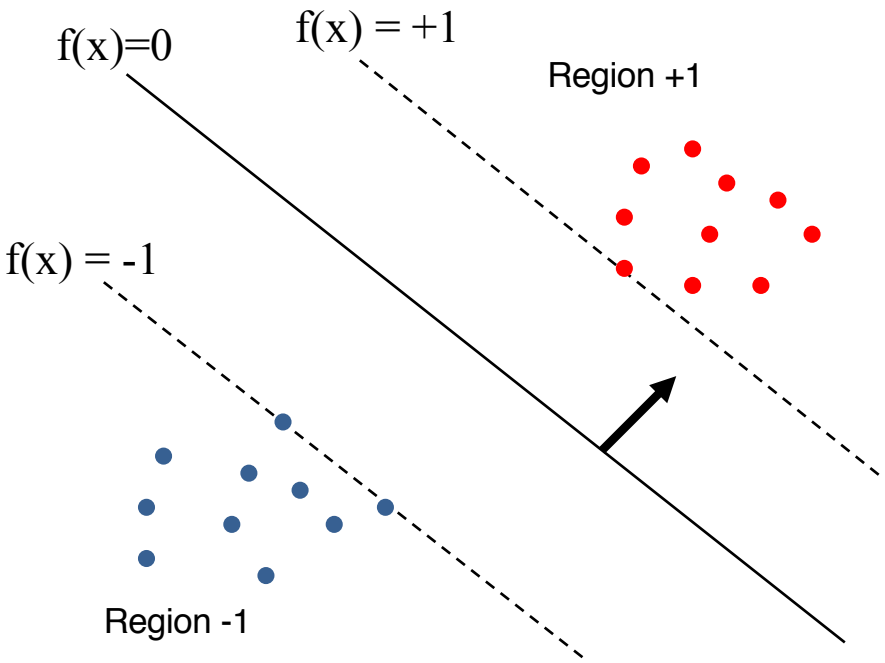
- Maybe we want to maximize our "margin"
- To optimize, relate to model parameters
- Remove "scale invariance"
  - Define class +1 in some region, class –1 in another
  - Make those regions as far apart as possible

Notation change!

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots$$

$$\Downarrow$$

$$b + w_1 x_1 + w_2 x_2 + \ldots$$



We could define such a function:

$f(x) = w*x' + b$

$f(x) > +1$ in region +1
$f(x) < -1$ in region –1

Passes through zero in center…

"Support vectors" – data points on margin

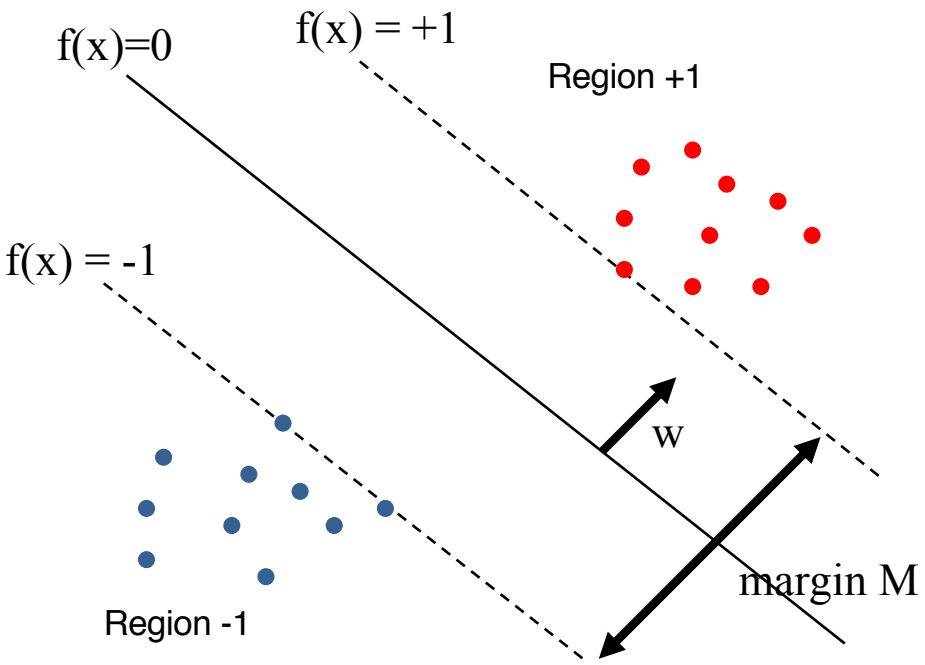
Maximum margin classifier

- Constrained optimization
  - Get all data points correct
  - Maximize the margin

$$w^* = \arg\max_w \frac{2}{\sqrt{w^T w}}$$

such that "all data on the correct side of the margin"

This is an example of a quadratic program: quadratic cost function, linear constraints



Primal problem:

$$w^* = \arg\min_w \sum_j w_j^2$$

s.t.

$$y^{(i)}(\, w \cdot x^{(i)} + b\,) \geq +1$$

(m constraints)

A 1D Example

- Suppose we have three data points

  x = -3, y = -1

  x = -1, y = -1

  x = 2, y = 1

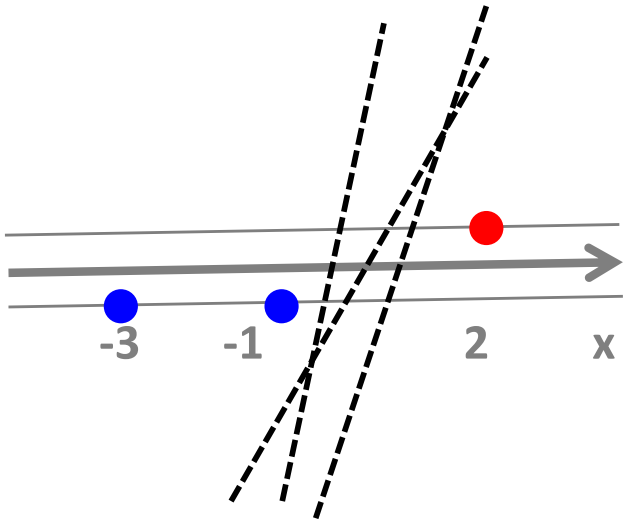
- Many separating perceptrons, T[ax+b]
  - Anything with ax+b = 0 between -1 and 2

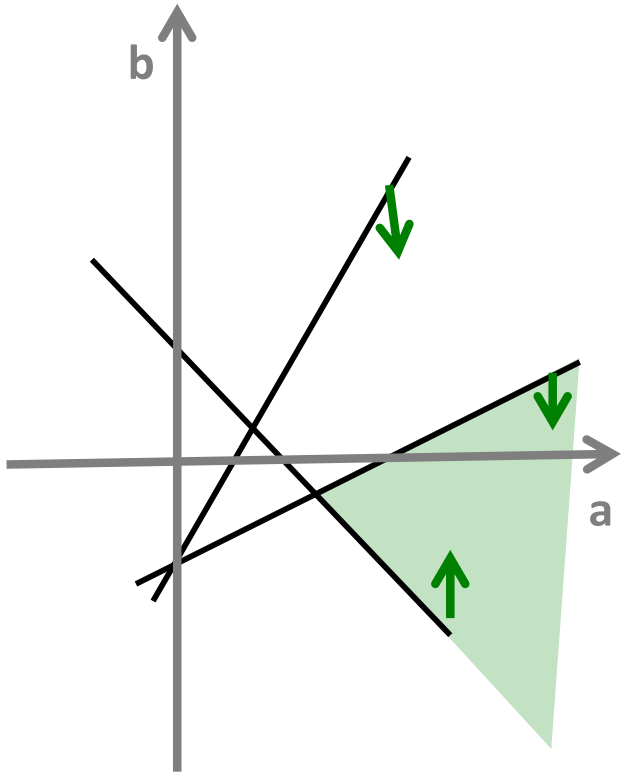
- We can write the margin constraints

  a (-3) + b < -1 => b < 3a - 1

  a (-1) + b < -1 => b < a - 1

  a (2) + b > +1 => b > -2a + 1





A 1D Example

- Suppose we have three data points

  x = -3, y = -1

  x = -1, y = -1

  x = 2, y = 1

- Many separating perceptrons, T[ax+b]
  - Anything with ax+b = 0 between -1 and 2

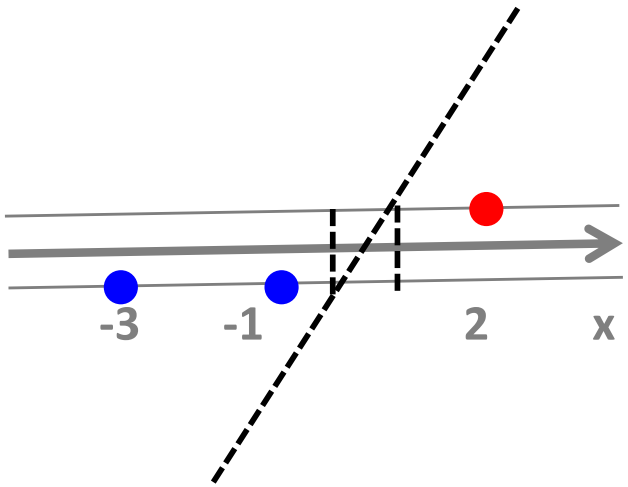
- We can write the margin constraints

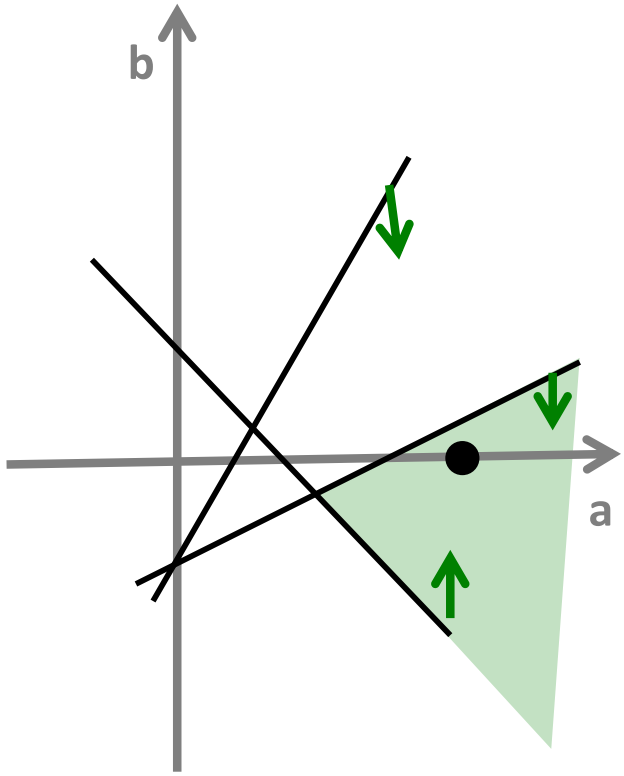
  a (-3) + b < -1 => b < 3a - 1

  a (-1) + b < -1 => b < a - 1

  a (2) + b > +1 => b > -2a + 1

- Ex: a = 1, b = 0





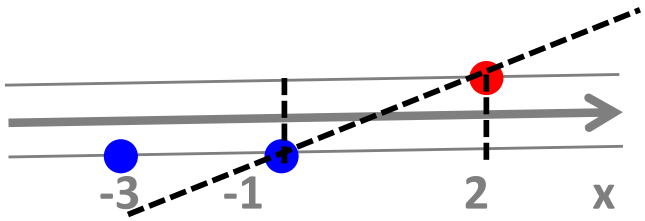
A 1D Example

- Suppose we have three data points

  x = -3, y = -1

  x = -1, y = -1

  x = 2, y = 1



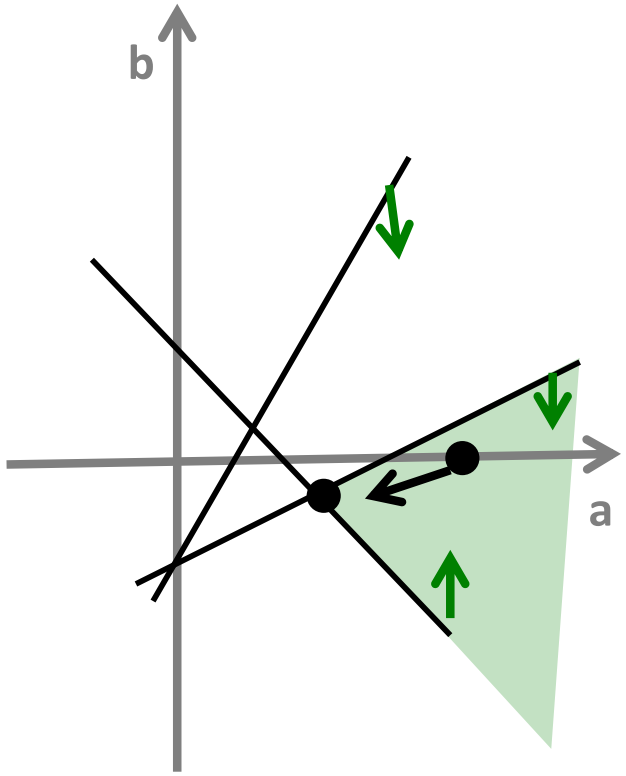
- Many separating perceptrons, T[ax+b]
  - Anything with ax+b = 0 between -1 and 2

- We can write the margin constraints

  a (-3) + b < -1 => b < 3a - 1

  a (-1) + b < -1 => b < a - 1

  a (2) + b > +1 => b > -2a + 1

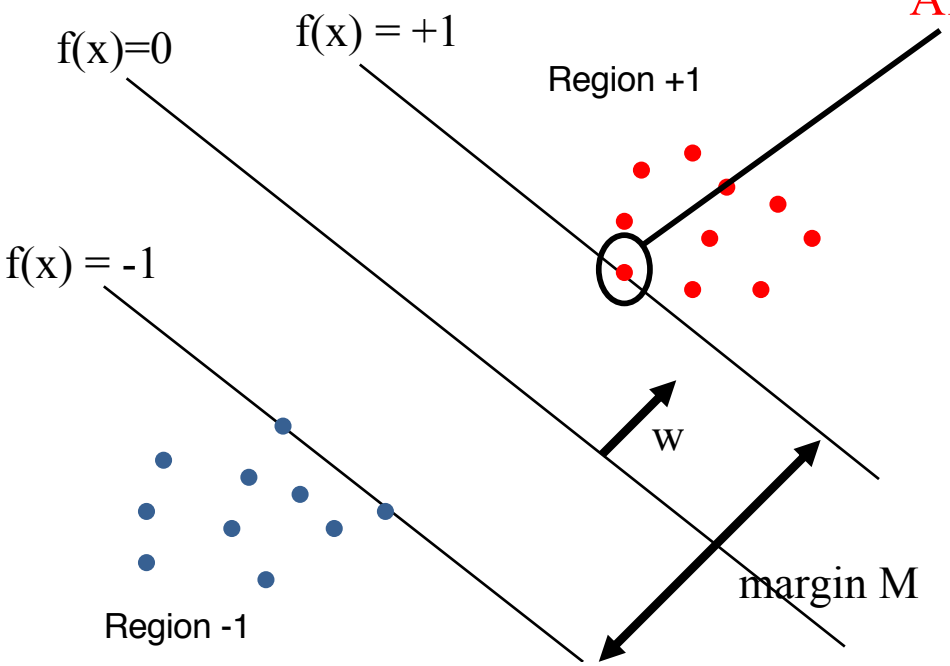


- Ex: a = 1, b = 0
- Minimize ||a|| => a = .66, b = -.33
  - Two data on the margin; constraints "tight"

Optimization

- Use Lagrange multipliers
  - Enforce inequality constraints

$$w^* = \arg\min_w \max_{\alpha \geq 0} \frac{1}{2} \sum_j w_j^2 + \sum_i \alpha_i (1 - y^{(i)} (w \cdot x^{(i)} + b))$$



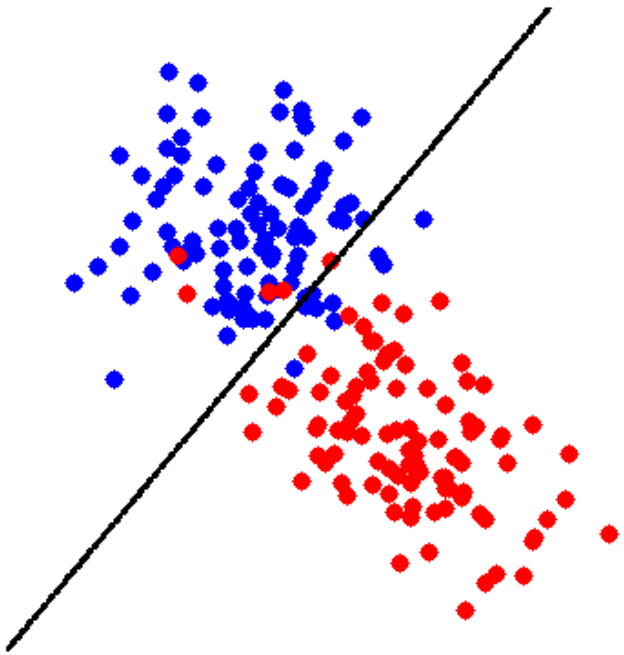
Alphas > 0 only on the margin:
"support vectors"

Stationary conditions wrt w:

$$w^* = \sum_i \alpha_i y^{(i)} x^{(i)}$$

Maximum margin classifier

- What if the data are not linearly separable?
  - Want a large "margin":
    $$\min_w \sum_j w_j^2$$
  - Want low error:
    $$\min_w \sum_i J(y^{(i)}\ ,\ w \cdot x^{(i)} + b)$$
  - "Soft margin" : introduce slack variables for violated constraints



$$w^* = \arg\min_{w,\epsilon} \sum_j w_j^2 + R \sum_i \epsilon^{(i)}$$

s.t.

$$y^{(i)}(\, w^T x^{(i)} + b\,) \geq +1 - \epsilon^{(i)} \quad \text{(violate margin by } \epsilon\text{)}$$

$$\epsilon^{(i)} \geq 0$$

Assigns "cost" R proportional to distance from margin
Another quadratic program!

Soft margin SVM

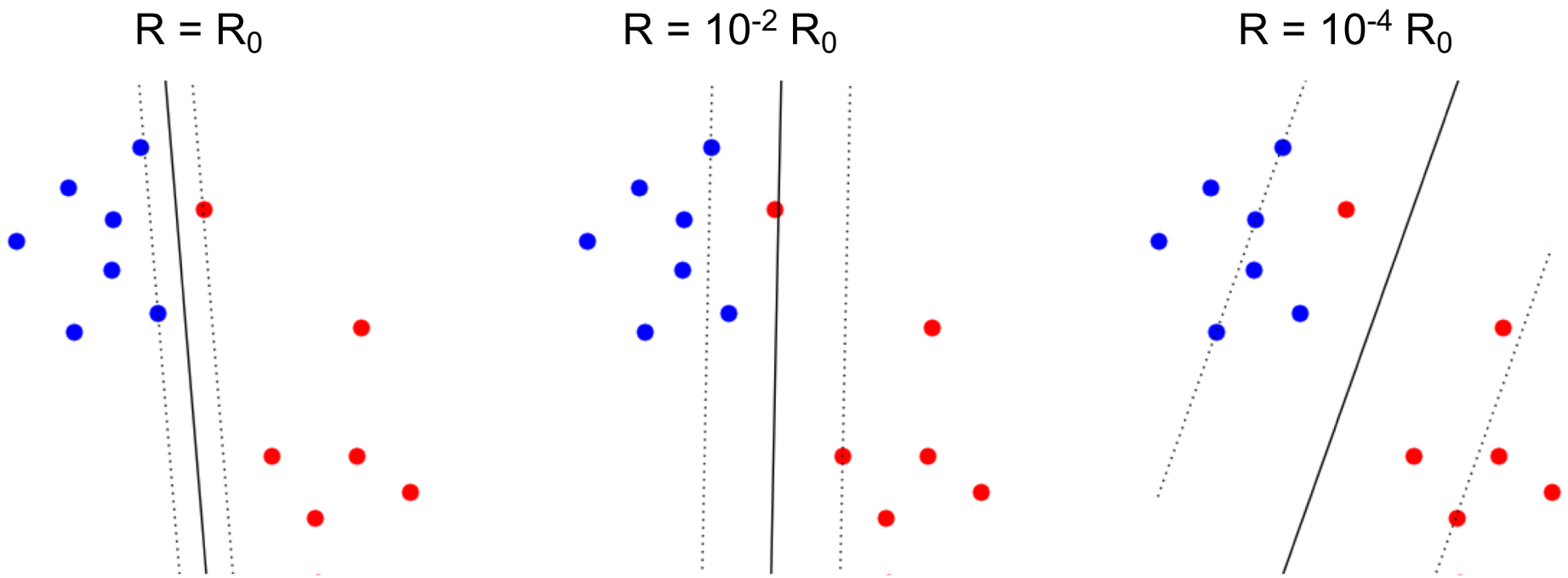
$$w^* = \arg\min_{w,\epsilon} \sum_j w_j^2 + R \sum_i \epsilon^{(i)}$$

$$s.t.$$

$$y^{(i)}(w^T x^{(i)} + b) \geq +1 - \epsilon^{(i)}$$

$$\epsilon^{(i)} \geq 0$$

- Large margin vs. Slack variables
- R large = hard margin
- R smaller
  - A few wrong predictions; boundary farther from rest

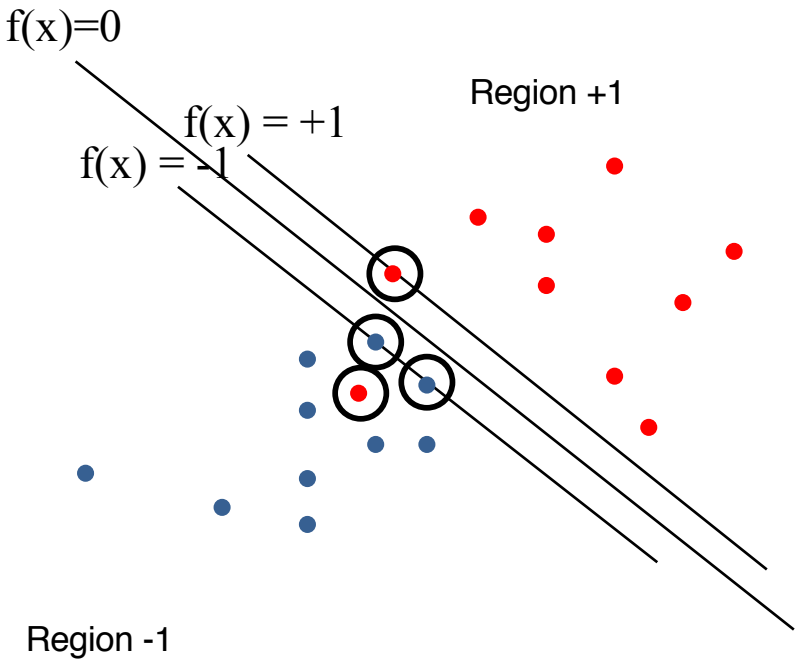


Support Vectors

The *support vectors* are data points i with non-zero weight α_i:

- Points with minimum margin (on optimized boundary)
- Points which violate margin constraint, but are still correctly classified
- Points which are misclassified

For all other training data, features have *no impact* on learned weight vector



Support vectors now data on or past margin…

Prediction:

$$\hat{y} = w^* \cdot x + b = \sum_i \alpha_i y^{(i)} x^{(i)} \cdot x + b$$

$$w^* = \sum_i \alpha_i y^{(i)} x^{(i)}$$

Machine Learning

Overfitting & Cross-Validation

Nearest Neighbors

Linear Classification

Naïve Bayes Classifiers

Support Vector Machines

Linear Regression

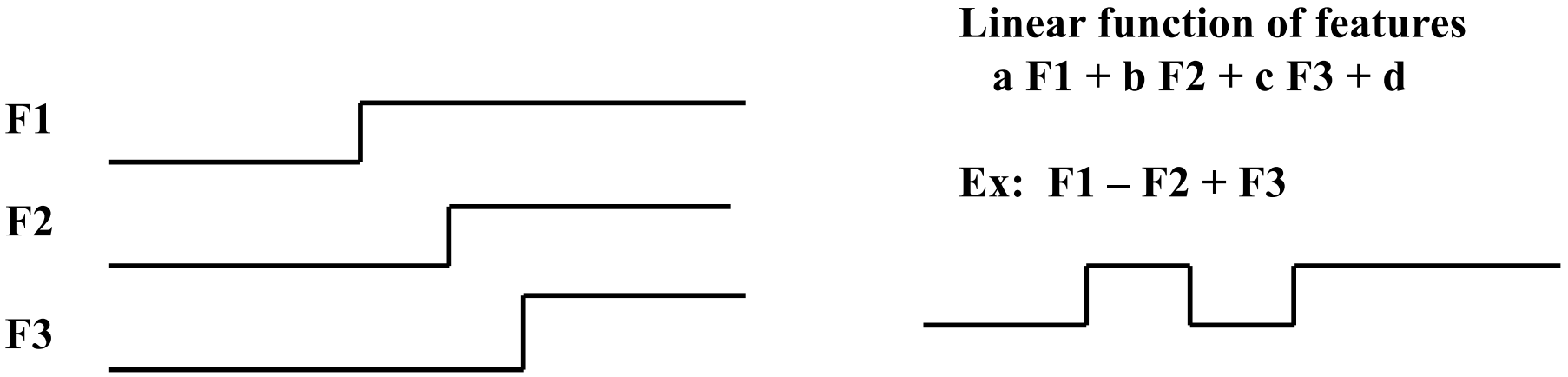
Neural Networks

Features and perceptrons

- Recall the role of features
  - We can create extra features that allow more complex decision boundaries
  - For example, polynomial features

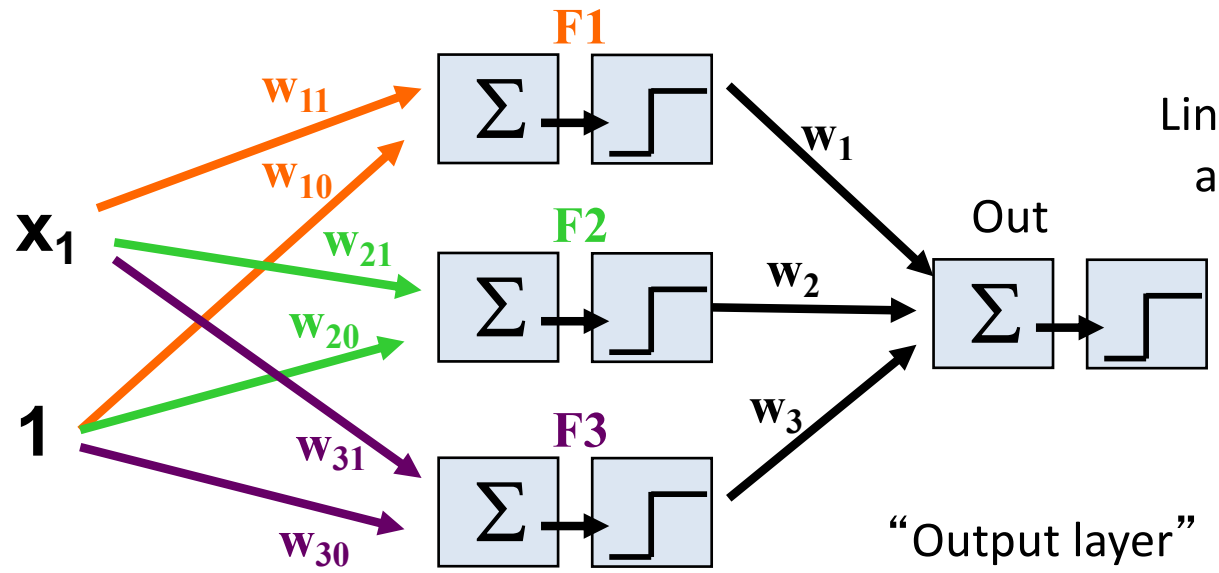
$$\Phi(x) = [1 \;\; x \;\; x^2 \;\; x^3 \;\; \ldots]$$

- What other kinds of features could we choose?
  - Step functions?



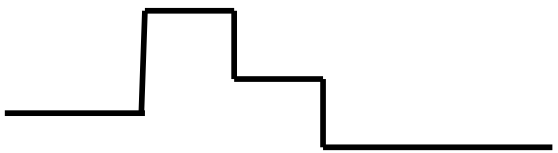
Multi-layer perceptron model

- Step functions are just perceptrons!
  - "Features" are outputs of a perceptron
  - Combination of features output of another



Linear function of features:
a F1 + b F2 + c F3 + d

Ex: F1 – F2 + F3



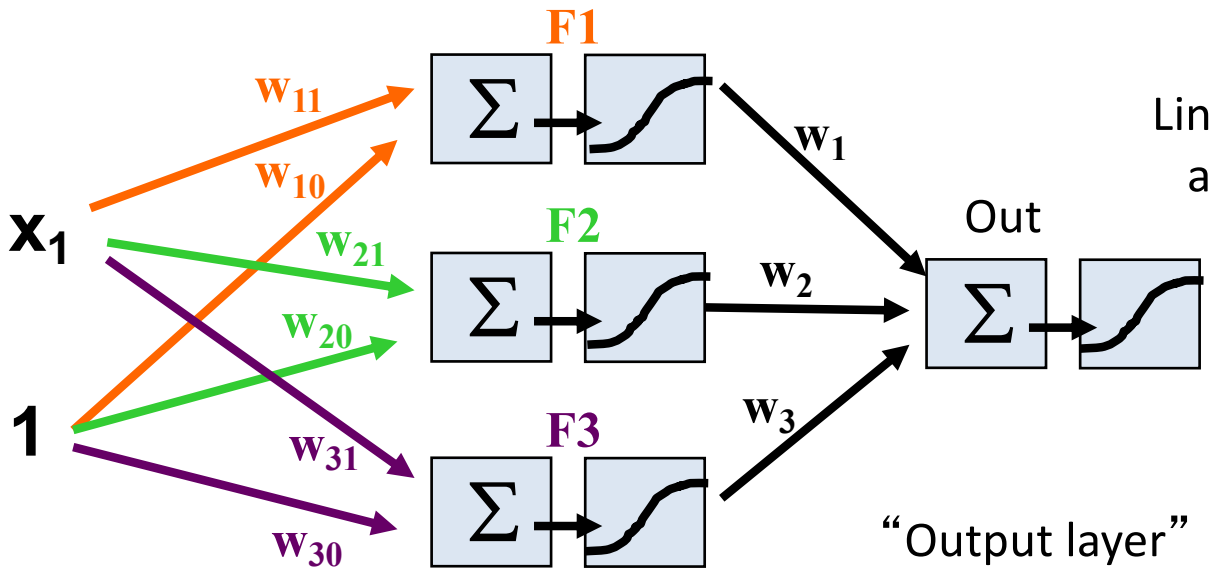
$$W^1 = \begin{matrix} w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{matrix}$$

$$W^2 = w_1 \; w_2 \; w_3$$

Multi-layer perceptron model

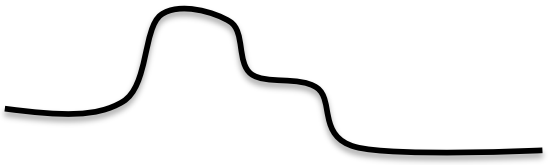
- Step functions are just perceptrons!
  - "Features" are outputs of a perceptron
  - Combination of features output of another

Regression version:
Remove activation function from output



Linear function of features:
a F1 + b F2 + c F3 + d

Ex: F1 – F2 + F3

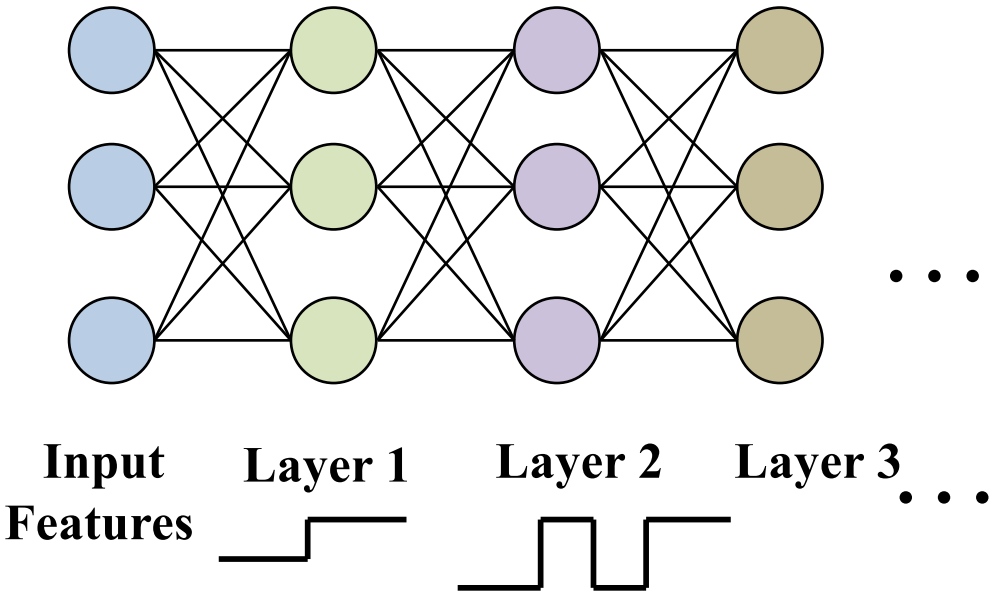


$$W^1 = \begin{matrix} w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{matrix}$$

$$W^2 = w_1 \; w_2 \; w_3$$

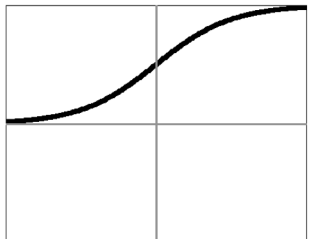
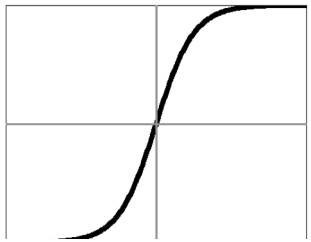
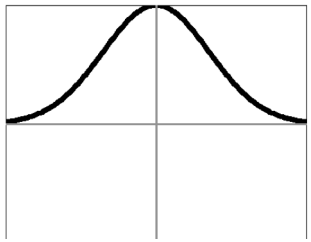
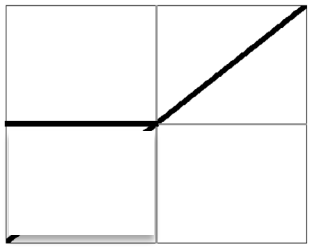
Features of MLPs

- Simple building blocks
  - Each element is just a perceptron function
- Can build upwards



Current research:
"Deep" architectures
(many layers)

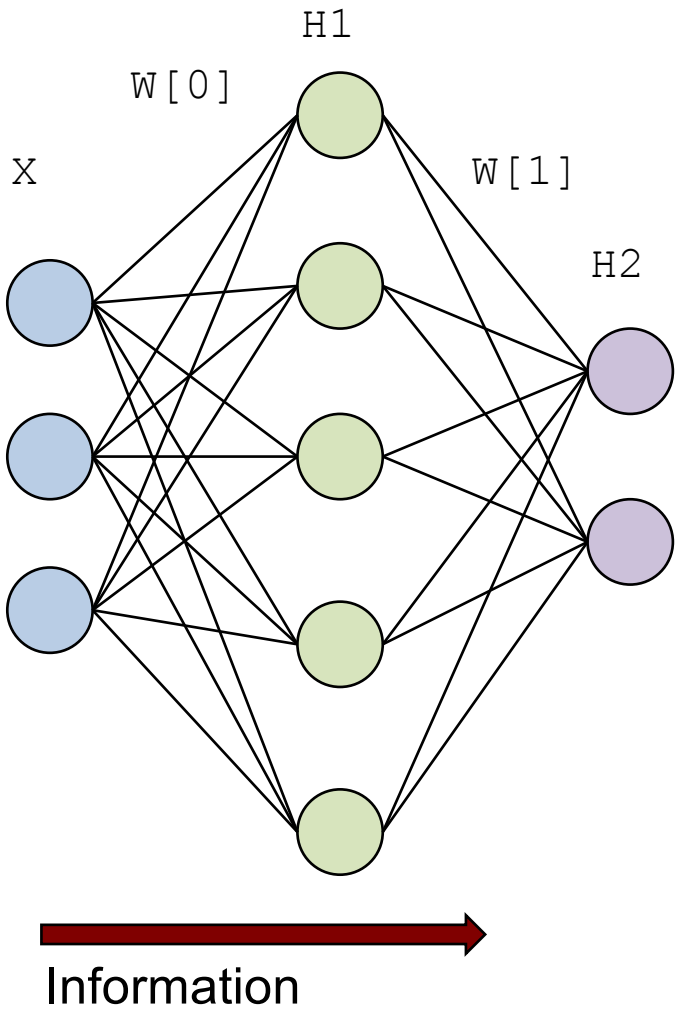
Activation functions

Logistic	$\sigma(z) = \frac{1}{1+\exp(-z)}$		$\frac{\partial \sigma}{\partial z}(z) = \sigma(z)(1-\sigma(z))$
Hyperbolic Tangent	$\sigma(z) = \frac{1-\exp(-2z)}{1+\exp(-2z)}$		$\frac{\partial \sigma}{\partial z}(z) = 1-(\sigma(z))^2$
Gaussian	$\sigma(z) = \exp(-z^2/2)$		$\frac{\partial \sigma}{\partial z}(z) = -z\sigma(z)$
ReLU (rectified linear)	$\sigma(z) = \max(0, z)$		$\frac{\partial \sigma}{\partial z}(z) = \mathbb{1}[z > 0]$
Linear	$\sigma(z) = z$		and many others…

Feed-forward networks

- Information flows left-to-right
  - Input observed features
  - Compute hidden nodes (parallel)
  - Compute next layer…

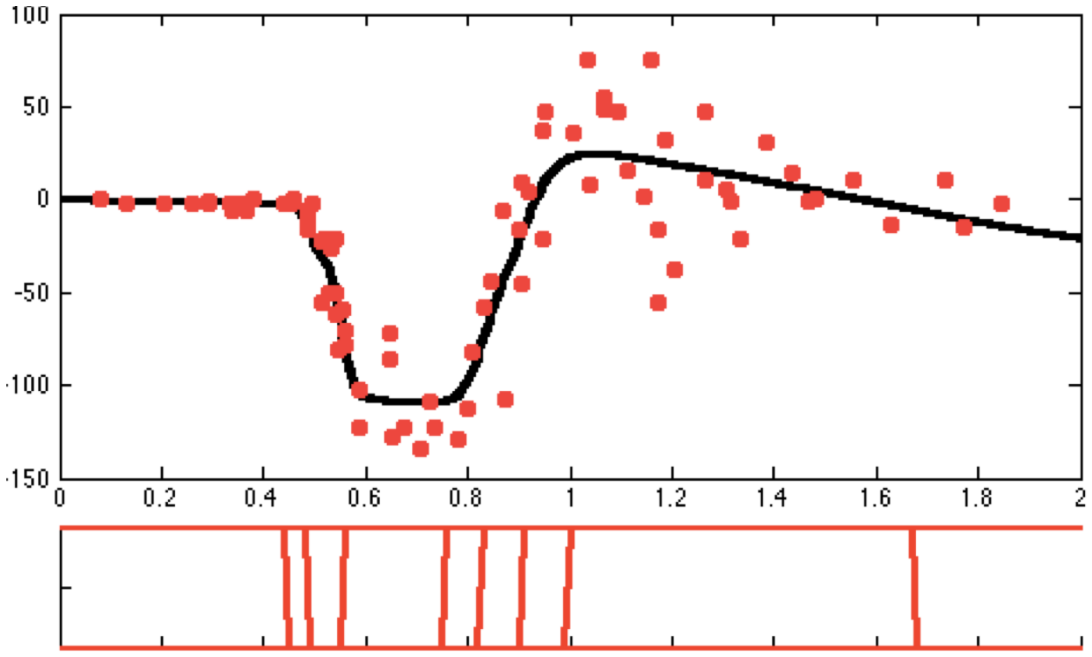
```
R = X.dot(W[0])+B[0] # linear response
H1= Sig( R )         # activation f'n

S = H1.dot(W[1])+B[1] # linear response
H2 = Sig( S )         # activation f'n
```



Example: Regression, MCycle data

- Train NN model, 2 layer
  - 1 input features => 1 input units
  - 10 hidden units
  - 1 target => 1 output units
  - Logistic sigmoid activation for hidden layer, linear for output layer

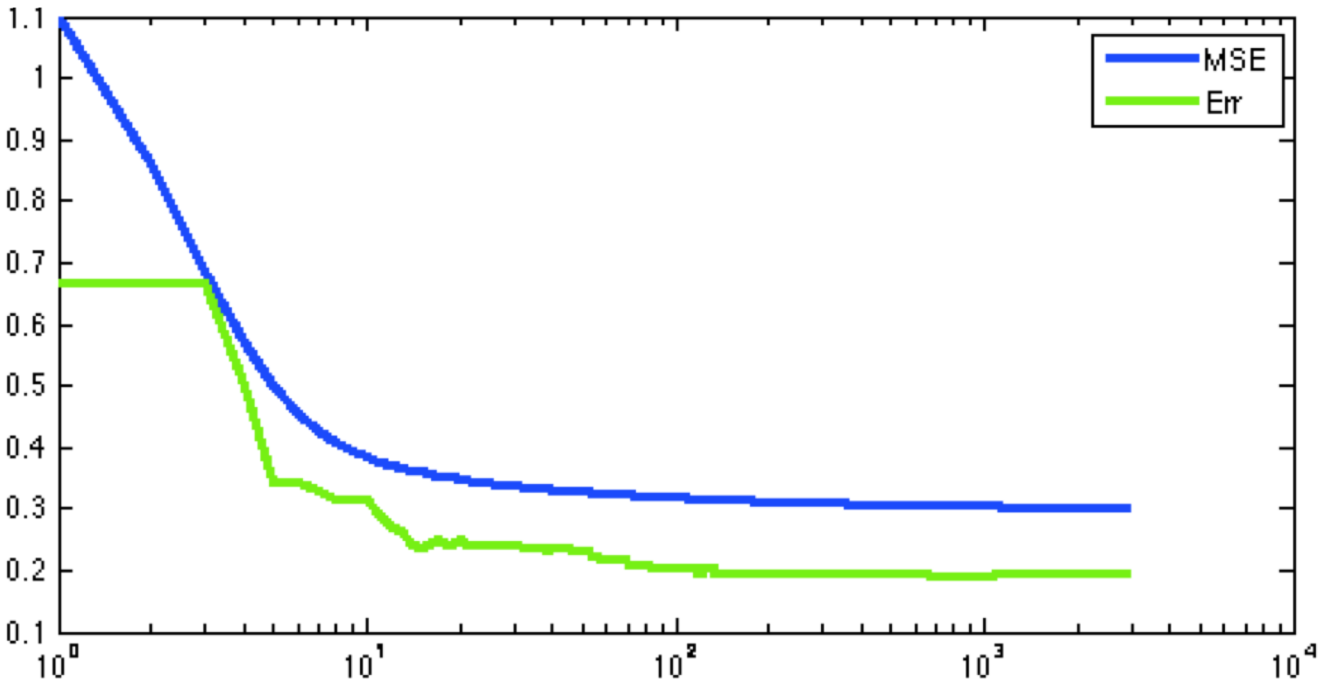
Data:
+
learned prediction f'n:

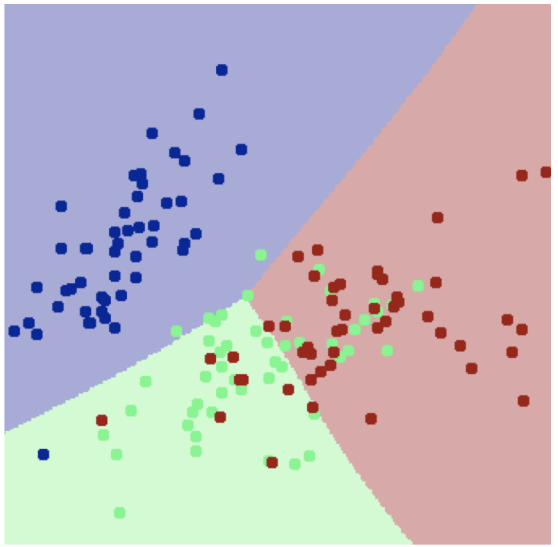


Responses of hidden nodes
(= features of linear regression):
select out useful regions of "x"

Example: Classification, Iris data

- Train NN model, 2 layer
  - 2 input features => 2 input units
  - 10 hidden units
  - 3 classes => 3 output units (y = [0 0 1], etc.)
  - Logistic sigmoid activation functions
  - Optimize MSE of predictions using stochastic gradient

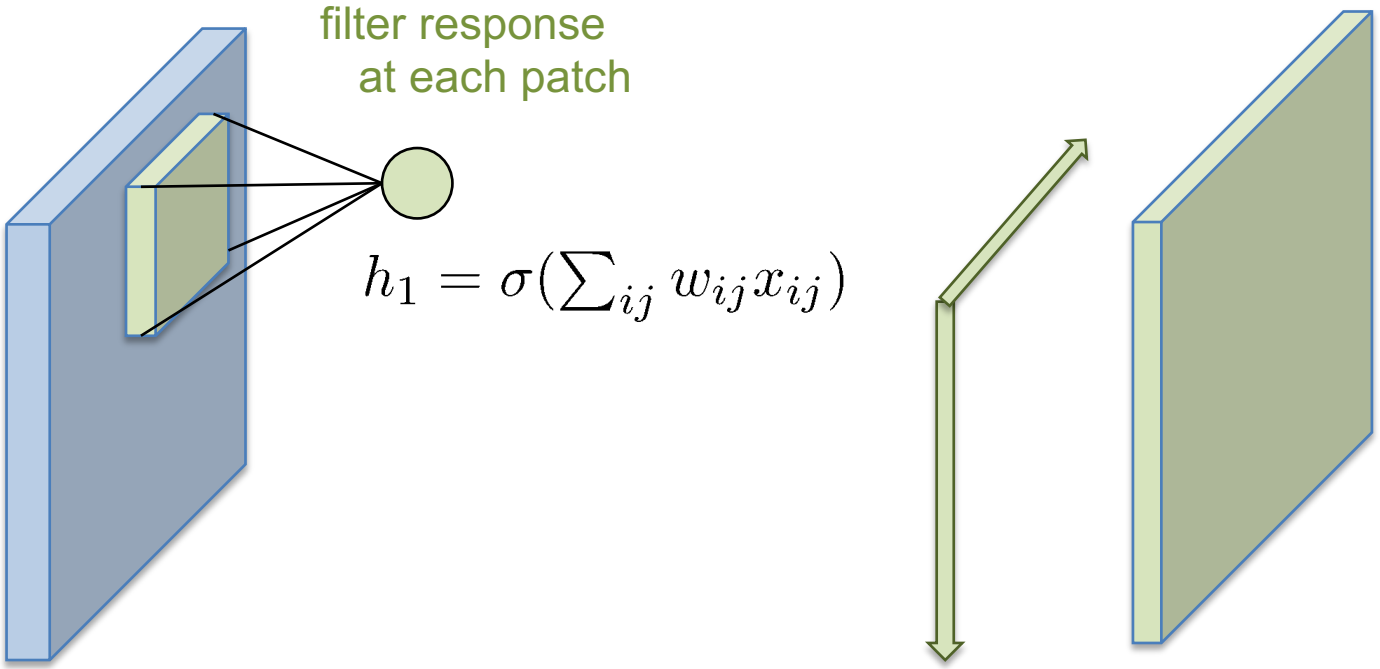




Convolutional networks

- Organize & share the NN's weights (vs "dense")
- Group weights into "filters" & convolve across input image

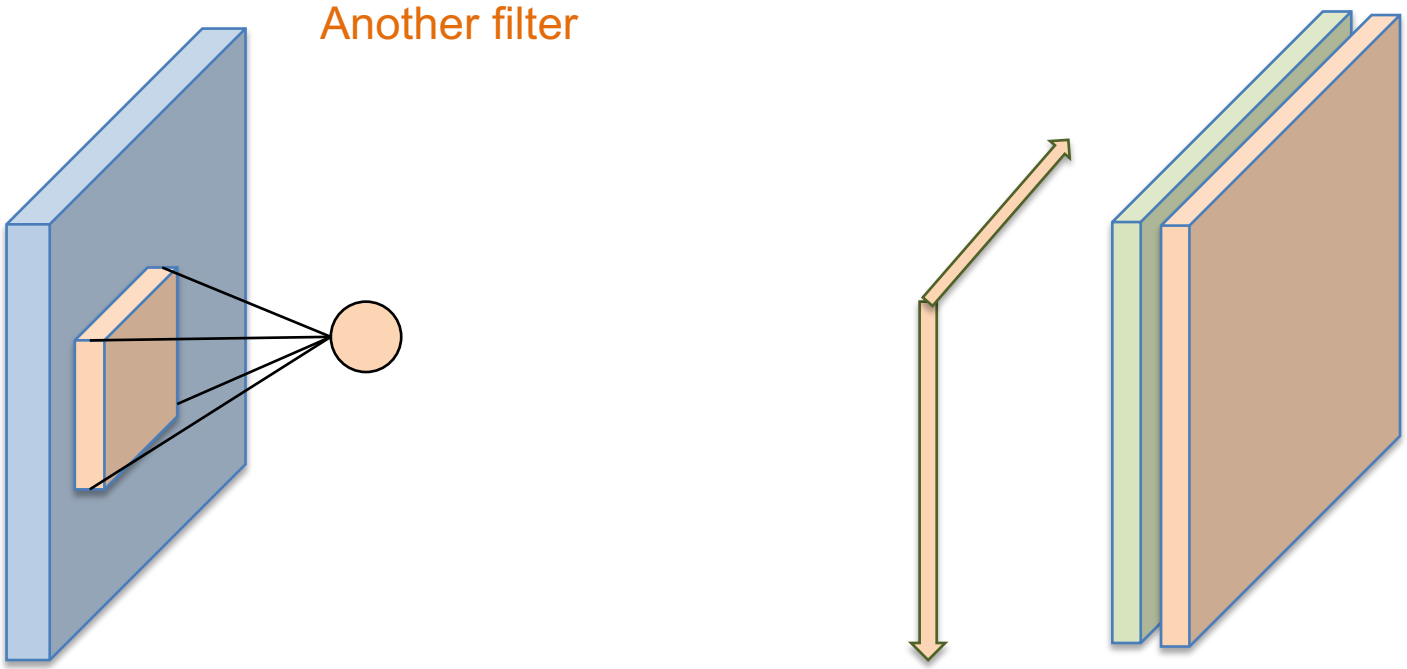
Input: 28x28 image Weights: 5x5 24x24 image



Convolutional networks

- Organize & share the NN's weights (vs "dense")
- Group weights into "filters" & convolve across input image

Input: 28x28 image Weights: 5x5



Convolutional networks

- Organize & share the NN's weights (vs "dense")
- Group weights into "filters" & convolve across input image
- Many hidden nodes, but few parameters!

Input: 28x28 image Weights: 5x5 Hidden layer 1