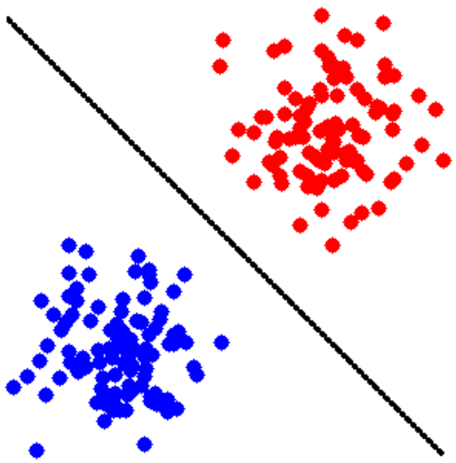
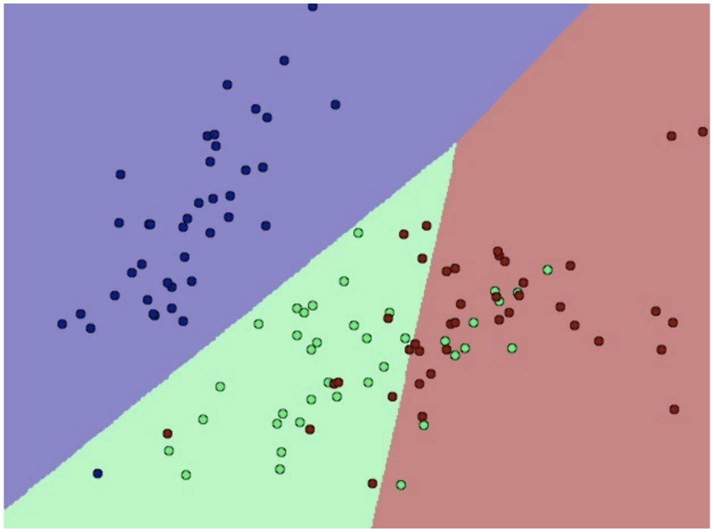
CS273A:
Linear Classifiers





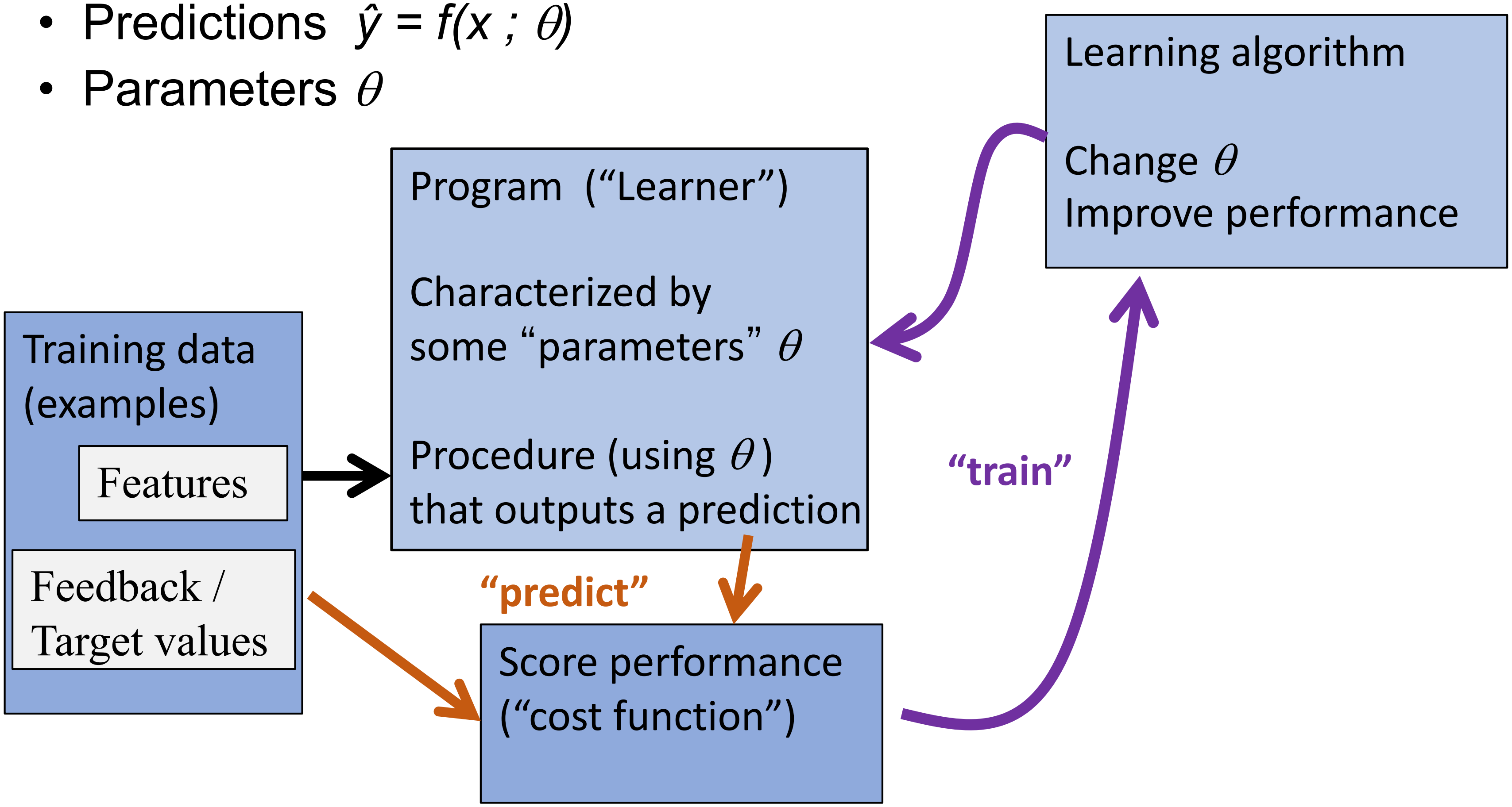
Prof. Alexander Ihler

Fall 2024

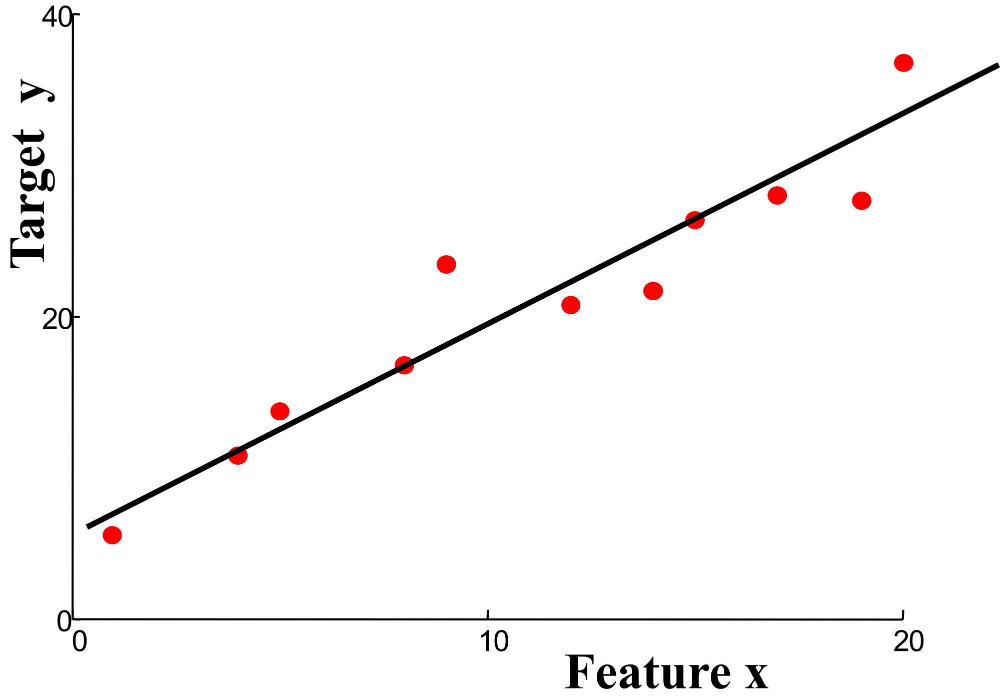
Linear Classifiers

Supervised Learning

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



Linear Regresson



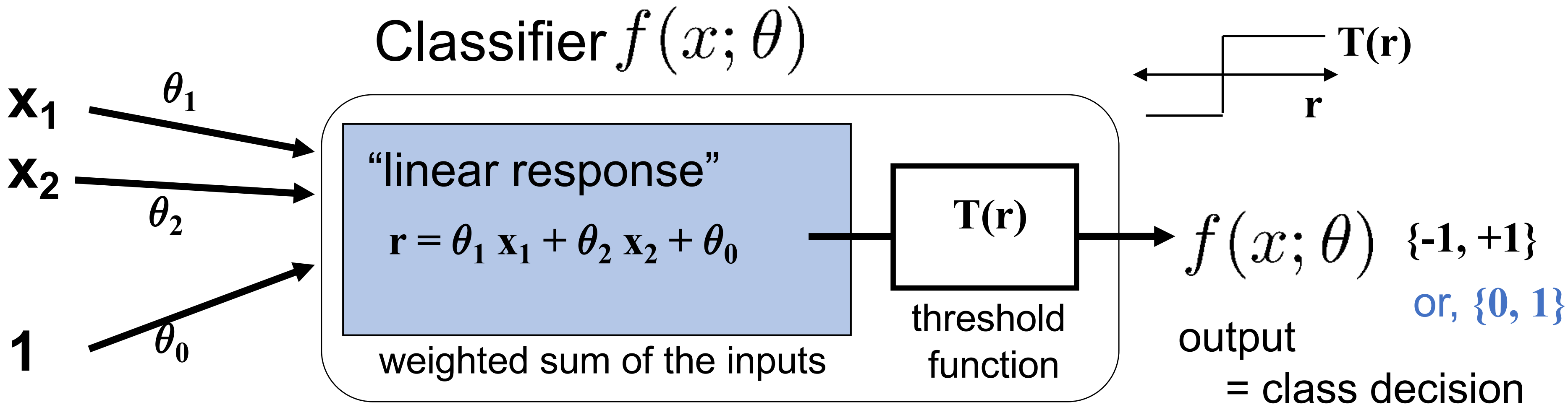
"Predictor":
Evaluate line:

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

return r

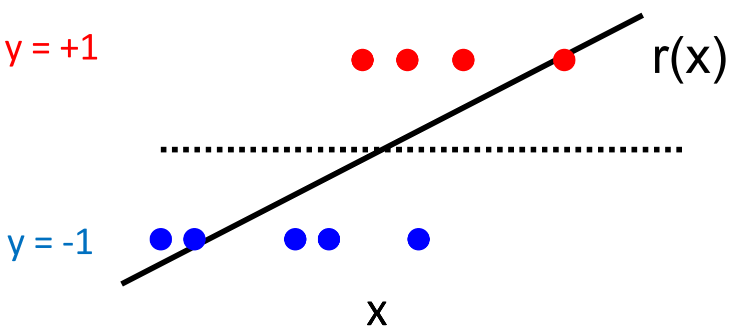
- Contrast with classification
  - Classify: predict discrete-valued target y
  - Initially: "classic" binary { -1, +1} classes; generalize later

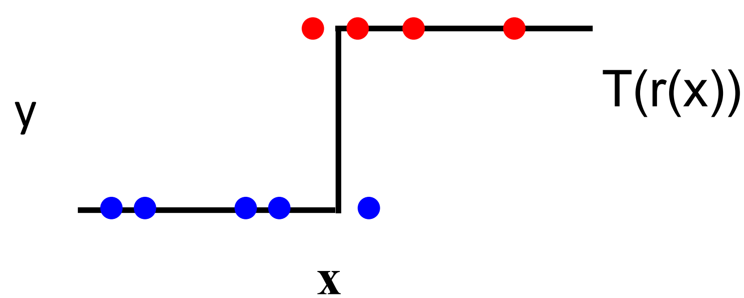
Perceptron Classifier (2 Features)




```
r = X @ theta.T         # compute linear response
Yhat = 1*(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":





Perceptrons

- Perceptron = a linear classifier
  - The parameters θ are sometimes called weights ("w")
    - real-valued constants (can be positive or negative)
  - Input features $x_1 \ldots x_n$ are arbitrary numbers
  - Define an additional constant input feature $x_0=1$

- A perceptron calculates:
  1. A weighted sum of the input features ("linear response")
  2. The sum is then thresholded by the T(.) function ("decision")

- Perceptron: a simple artificial model of human neurons
  - weights = "synapses"
  - threshold = "neuron firing"

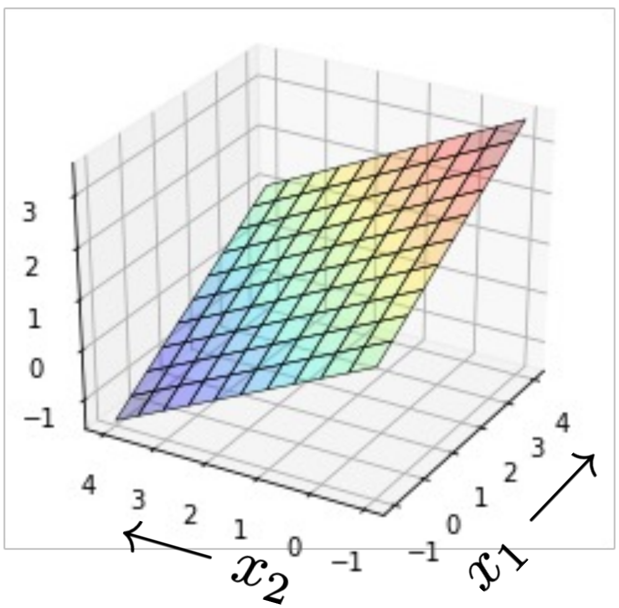
Perceptron Decision Boundary

- The perceptron is defined by the decision algorithm:

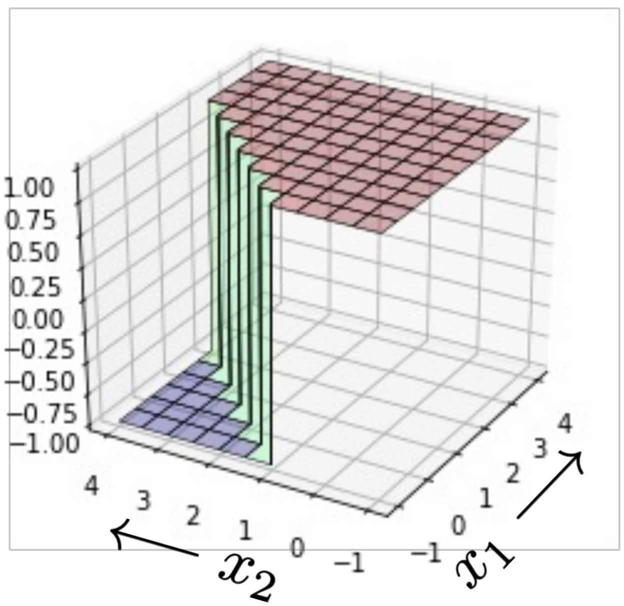
$$f(x;\theta) = \begin{cases} +1 & \text{if } \theta \cdot x^T > 0 \\ -1 & \text{otherwise} \end{cases}$$

Decision boundary:

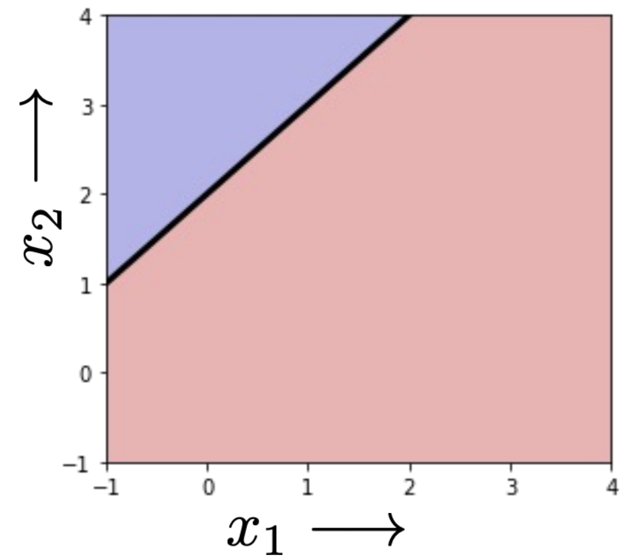
$$\theta \cdot x^T = 0$$



Linear response r(x)

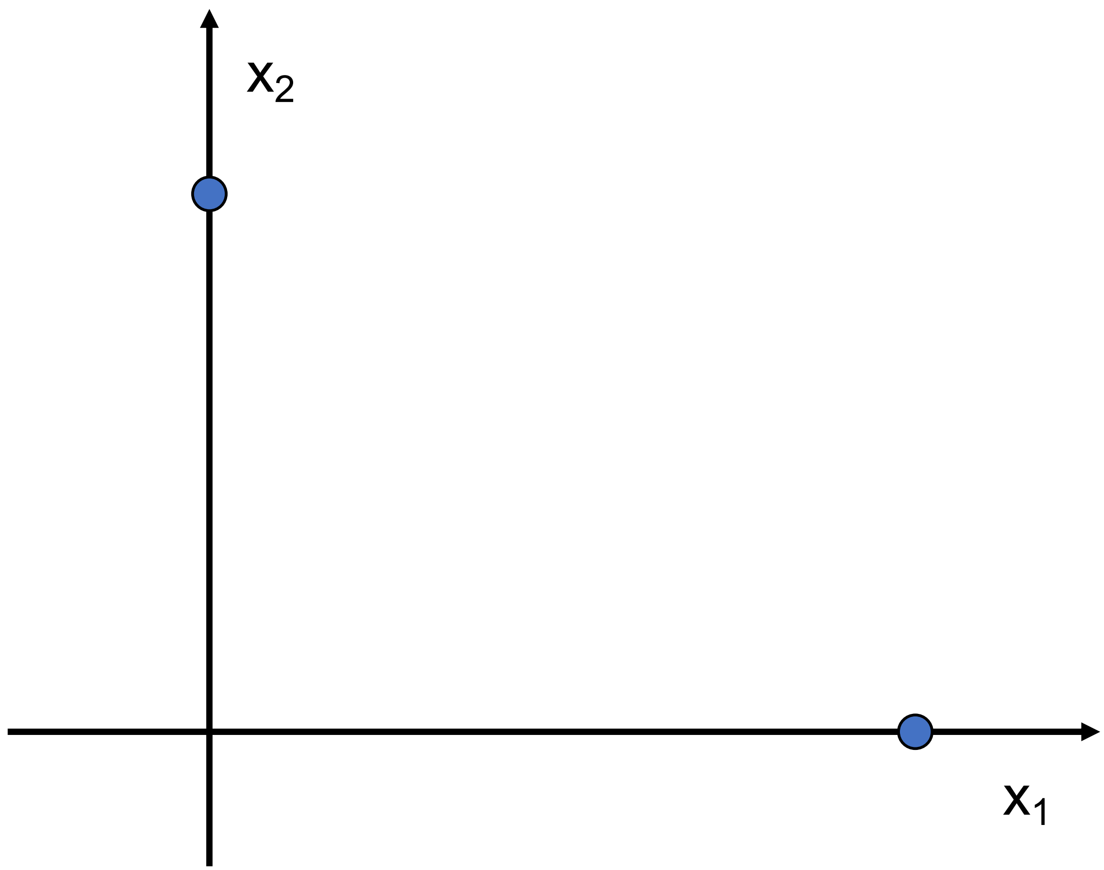


Decision f(x) = T(r(x))



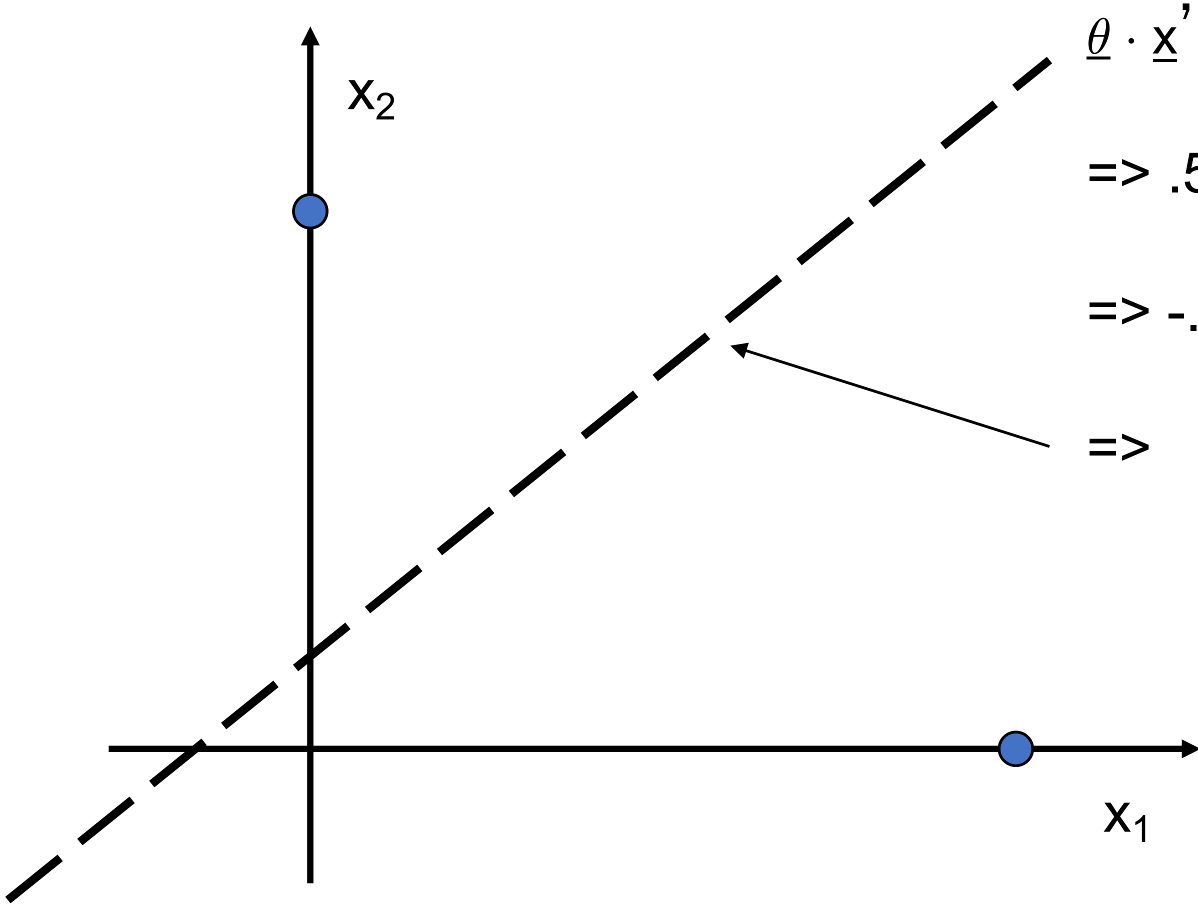
Example, Linear Decision Boundary

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



Example, Linear Decision Boundary

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



$\underline{\theta} \cdot \underline{x}' = 0$

$\Rightarrow .5 \cdot x_1 - .5 \cdot x_2 + 1 \cdot 1 = 0$

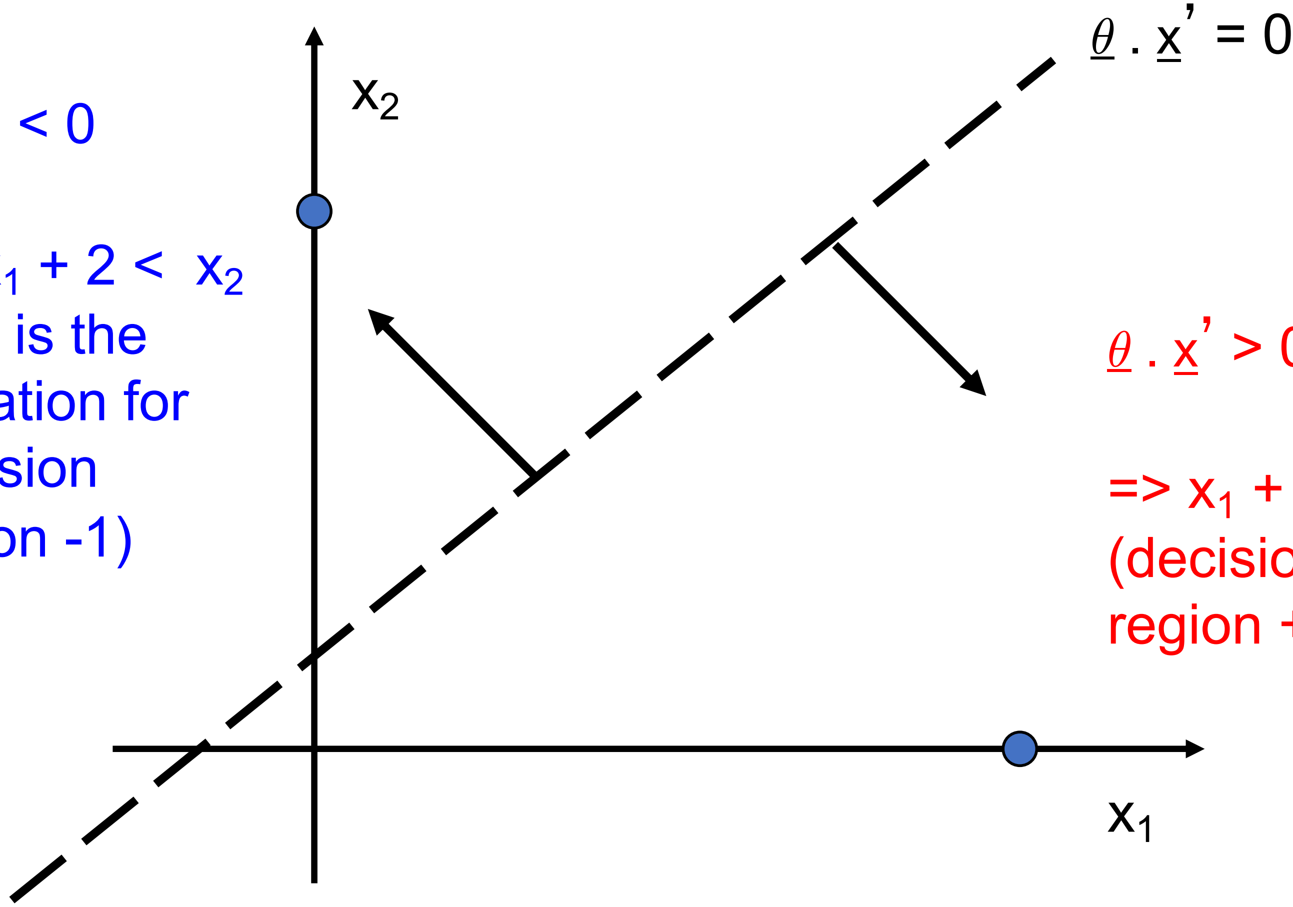
$\Rightarrow -.5\, x_2 = -.5\, x_1 - 1$

$\Rightarrow \quad x_2 = x_1 + 2$

From P. Smyth

Example, Linear Decision Boundary

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



$\underline{\theta} \,.\, \underline{x}' < 0$

$\Rightarrow x_1 + 2 < x_2$
(this is the equation for decision region -1)

$\underline{\theta} \,.\, \underline{x}' > 0$

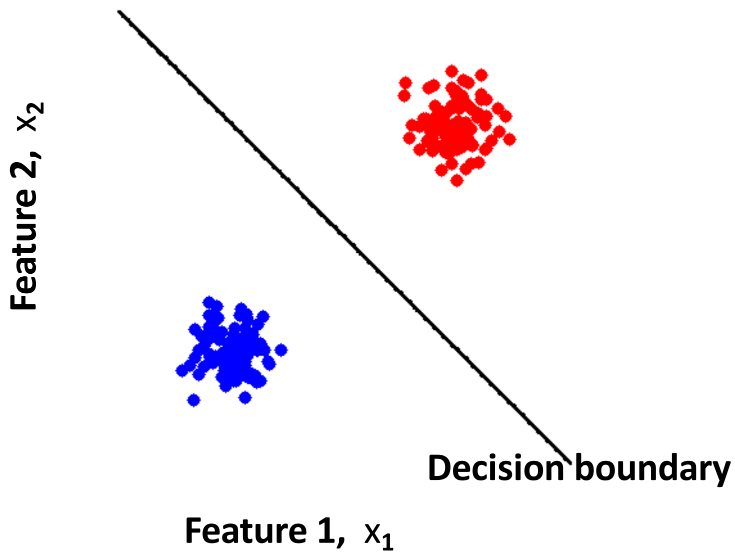
$\Rightarrow x_1 + 2 > x_2$
(decision region +1)

From P. Smyth

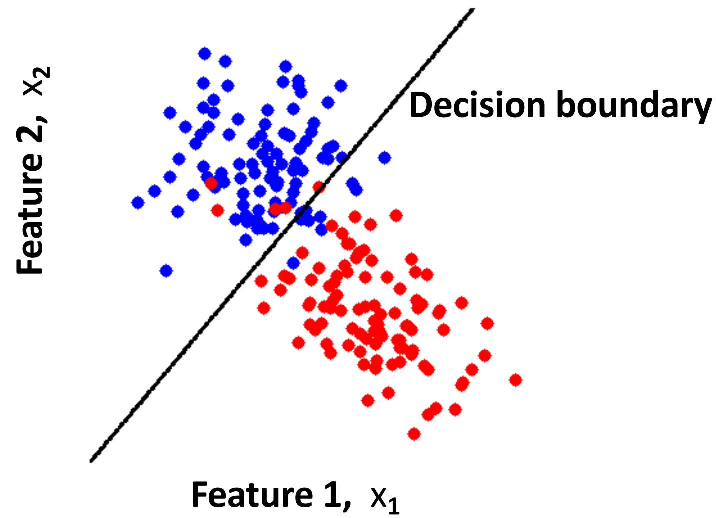
Separability

- 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 hyperplane in feature space
  - in 2 dimensions the decision boundary is a straight line

Linearly separable data

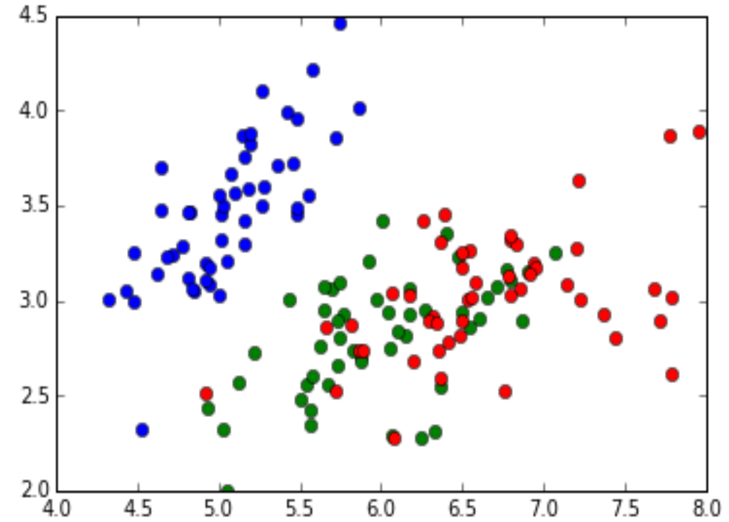


Linearly non-separable data



Class overlap

- Classes may not be well-separated
- Same observation values possible under both classes
  - High vs low risk; features {age, income}
  - Benign/malignant cells look similar
  - …

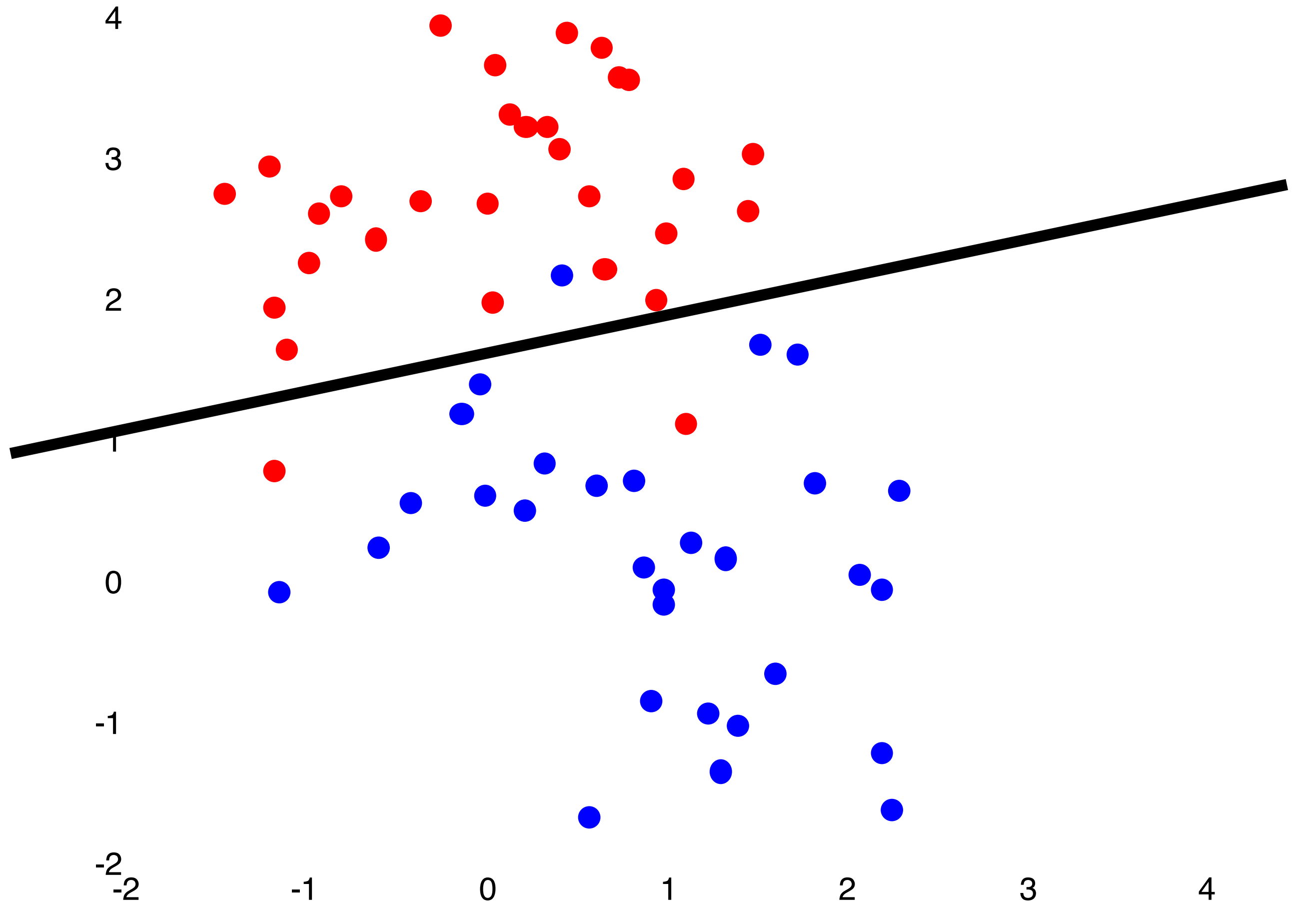


- Common in practice

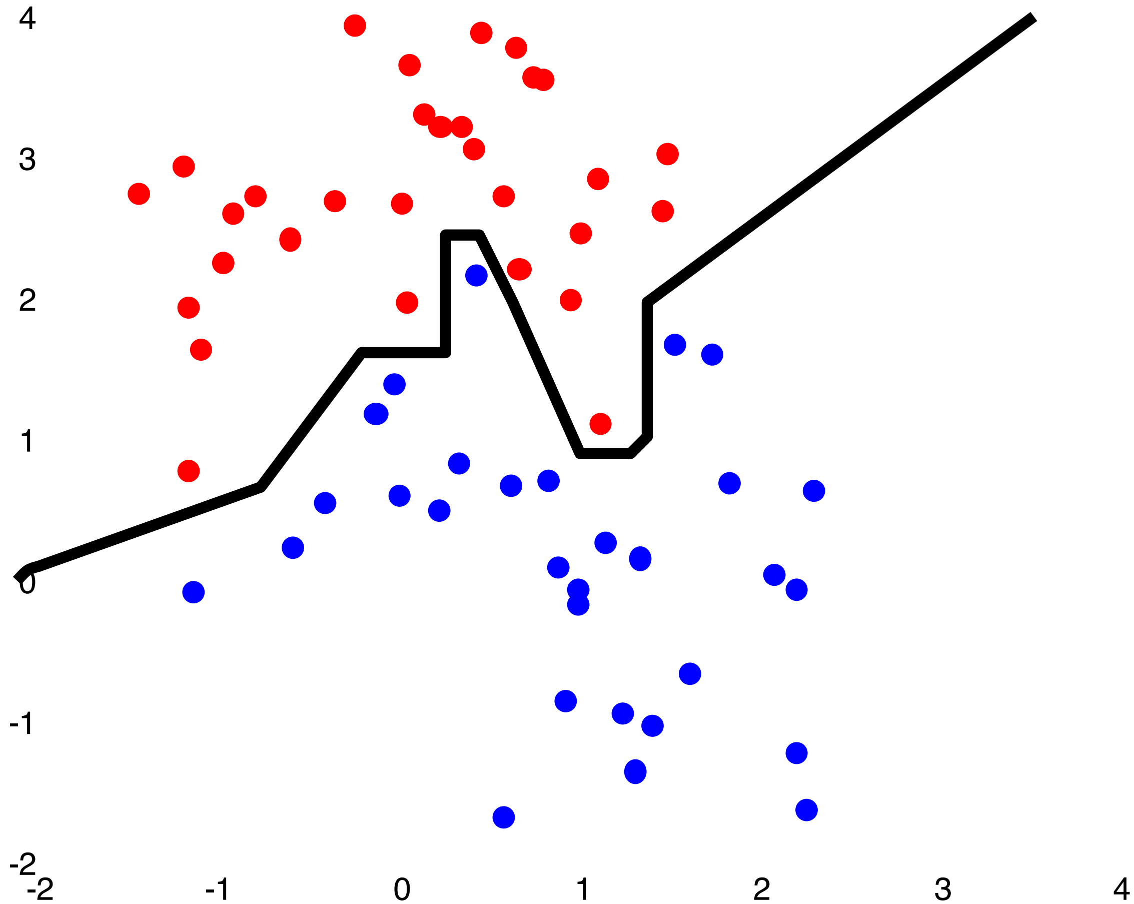
- May not be able to perfectly distinguish between classes
  - Maybe with more features?
  - Maybe with more complex classifier?

- Otherwise, may have to accept some errors

Another example



Non-linear decision boundary

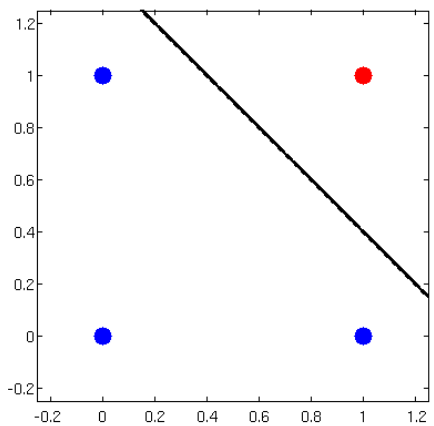


Representational Power of Perceptrons

- 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)

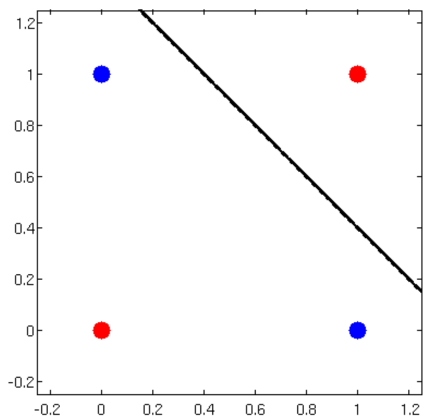
"AND"

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



"XOR"

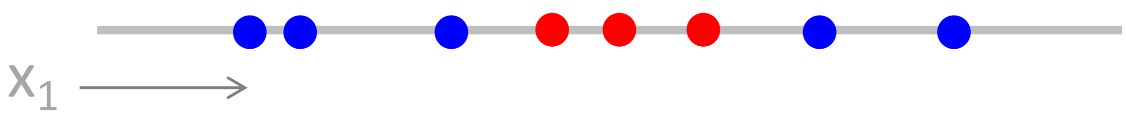
x_1	x_2	y
0	0	1
0	1	-1
1	0	-1
1	1	1



Adding features

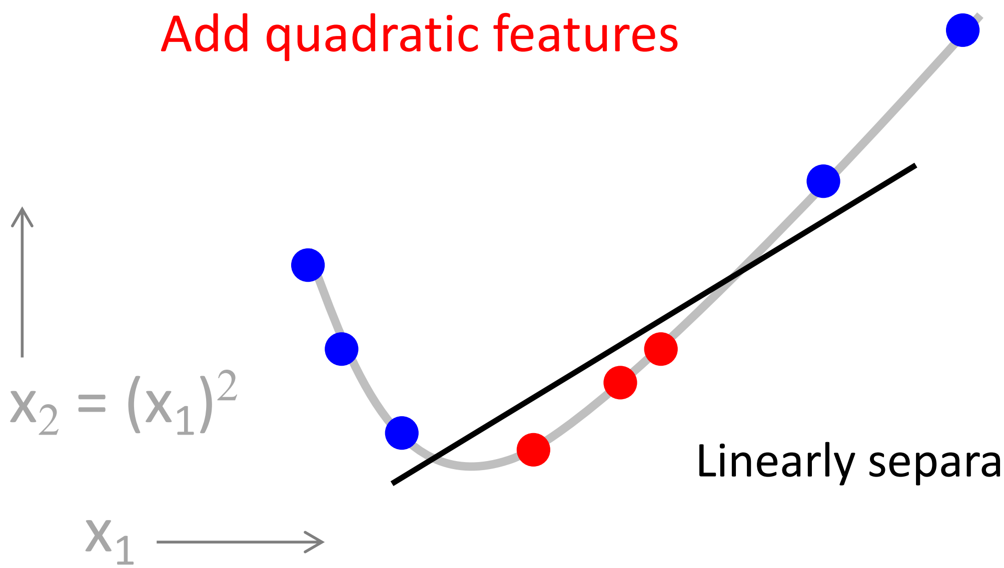
- Linear classifier can't learn some functions

1D example:



Not linearly separable

Add quadratic features

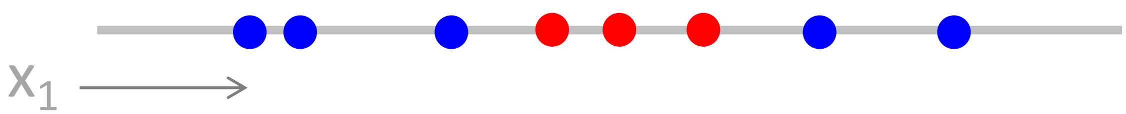


Linearly separable in new features…

Adding features

- Linear classifier can't learn some functions

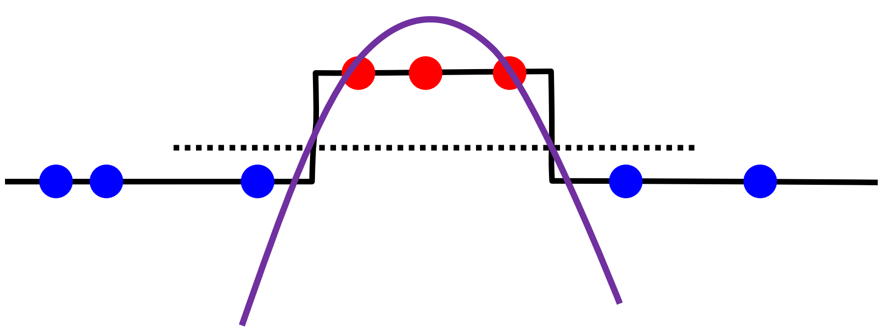
1D example:



Not linearly separable

Quadratic features, visualized in original feature space:

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



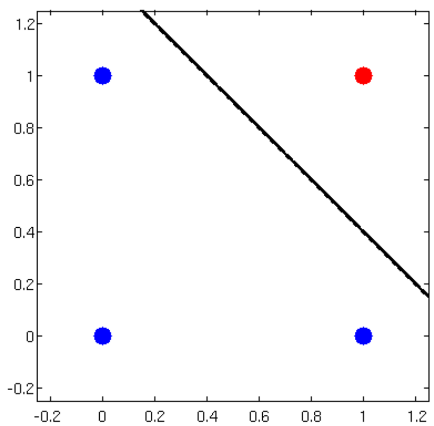
More complex decision boundary: $ax^2+bx+c = 0$

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)

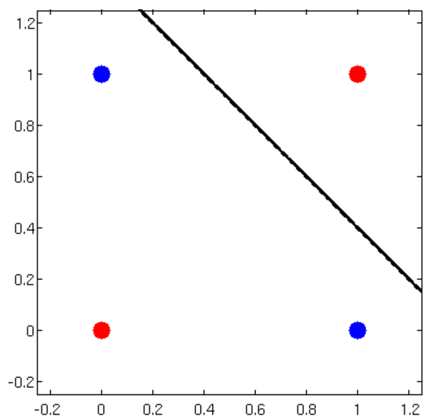
"AND"

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



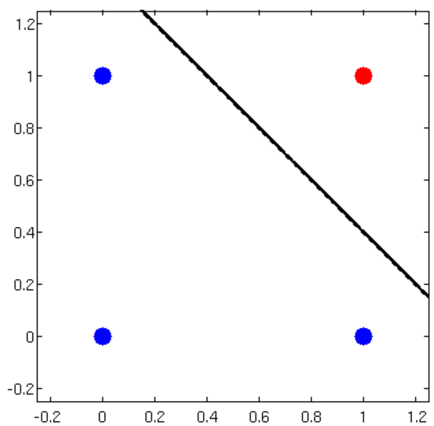
What kinds of functions could we use to learn the data on the right?

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)

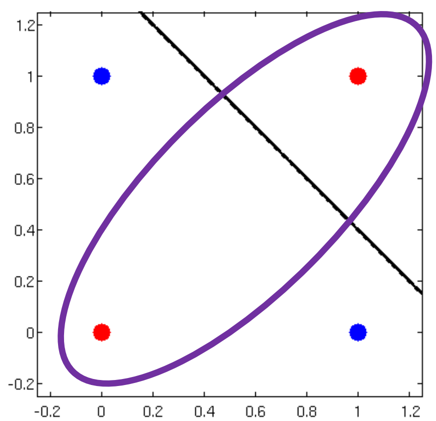
“AND”

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 could we use to learn the data on the right?

Ellipsiodal decision boundary: $a\,x_1^2 + b\,x_1 + c\,x_2^2 + d\,x_2 + e\,x_1x_2 + f = 0$

Feature representations

- Features are used in a linear way
- Learner is dependent on representation

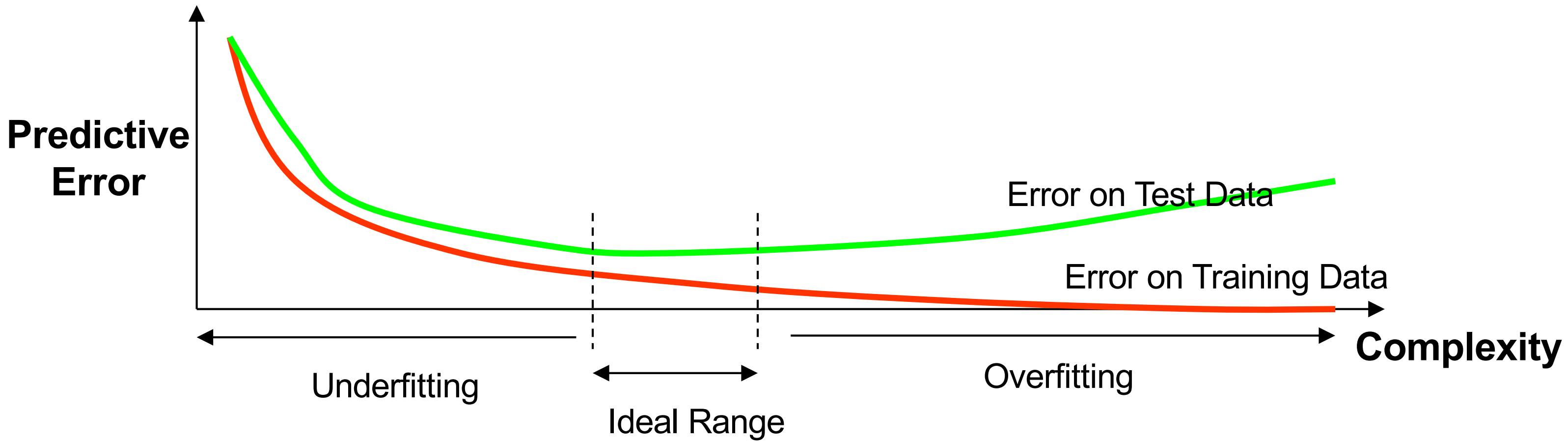
- Ex: discrete features
  - Mushroom surface: {fibrous, grooves, scaly, smooth}
  - Probably not useful to use x = {1, 2, 3, 4}
  - Better: 1-of-K,   x = { [1000], [0100], [0010], [0001] }
  - Introduces more parameters, but a more flexible relationship

Effect of dimensionality

- Data are increasingly separable in high dimension – is this a good thing?

- "Good"
  - Separation is easier in higher dimensions (for fixed # of data m)
  - Increase the number of features, and even a linear classifier will eventually be able to separate all the training examples!

- "Bad"
  - Remember training vs. test error? Remember overfitting?
  - Increasingly complex decision boundaries can eventually get all the training data right, but it doesn't necessarily bode well for test data…



Summary

- Linear classifier ⇔ perceptron
- Linear decision boundary
  - Computing and visualizing
- Separability
  - Limits of the representational power of a perceptron
- Adding features
  - Interpretations
  - Effect on separability
  - Potential for overfitting

Linear Classifiers

- Linear Classification with Perceptrons
- **Perceptron Learning**
- Gradient-Based Classifier Learning
- Multi-Class Classification
- Regularization for Linear Classifiers

Learning the Classifier Parameters

- Learning from Training Data:
  - training data = labeled feature vectors
  - Find parameter values that predict well (low error)
    - error is estimated on the training data
    - "true" error will be on future test data

- Define a loss function $J(\underline{\theta})$:
  - Classifier error rate (for a given set of weights $\underline{\theta}$ and labeled data)

- Minimize this loss function (or, maximize accuracy)
  - An optimization or search problem over the vector $(\theta_0, \theta_1, \theta_2, \ldots)$

Training a linear classifier

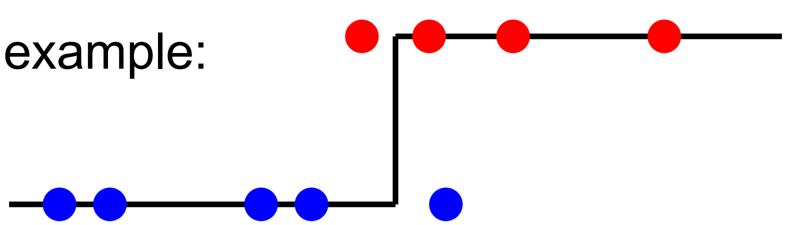
- How should we measure error?
  - Natural measure = "fraction we get wrong" (error rate)

$$\mathrm{err}(\theta) = \frac{1}{m} \sum_i \mathbb{1}\left[y^{(i)} \neq f(x^{(i)}; \theta) \right] \quad \text{where} \quad \mathbb{1}\left[y \neq \hat{y} \right] = \begin{cases} 1 & y \neq \hat{y} \\ 0 & \text{o.w.} \end{cases}$$

```
Yhat = np.sign( X @ theta.T  )       # predict class  (+1/-1)
err  = np.mean( Y != Yhat )          # count errors: empirical error rate
```

- But, hard to train via gradient descent
  - Not continuous
  - As decision boundary moves, errors change abruptly

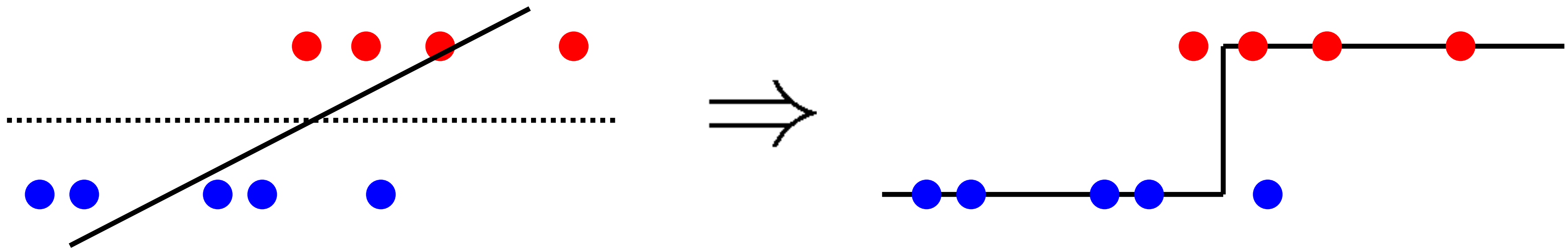
1D example:



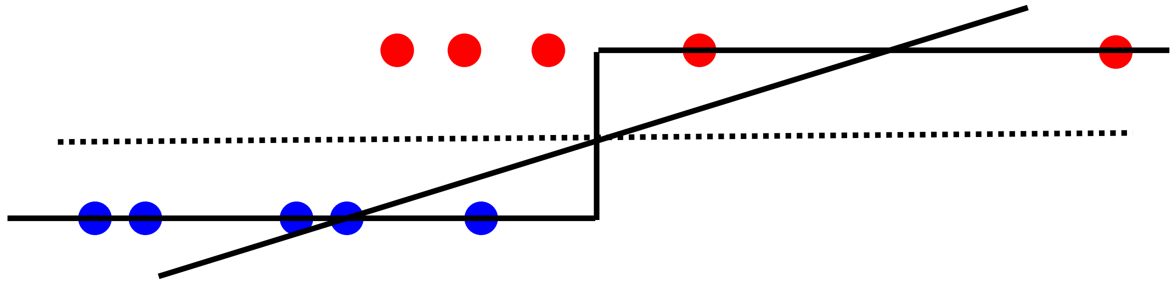
T(r(x)) = -1 if r(x) < 0
T(r(x)) = +1 if r(x) > 0

Linear regression?

- Simple option: set θ using linear regression



- In practice, this often doesn't work so well…
  - Consider adding a distant but "easy" point
  - MSE distorts the solution



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)

- Compare to linear regression + MSE cost
  - Identical update to SGD for MSE except error uses thresholded
    ŷ(j) instead of linear response $\underline{\theta}$ x':
  - (1) For correct predictions, $y^{(j)} - \hat{y}^{(j)} = 0$
  - (2) For incorrect predictions, $y^{(j)} - \hat{y}^{(j)} = \pm 2$

"adaptive" linear regression: correct predictions stop contributing

Perceptron algorithm

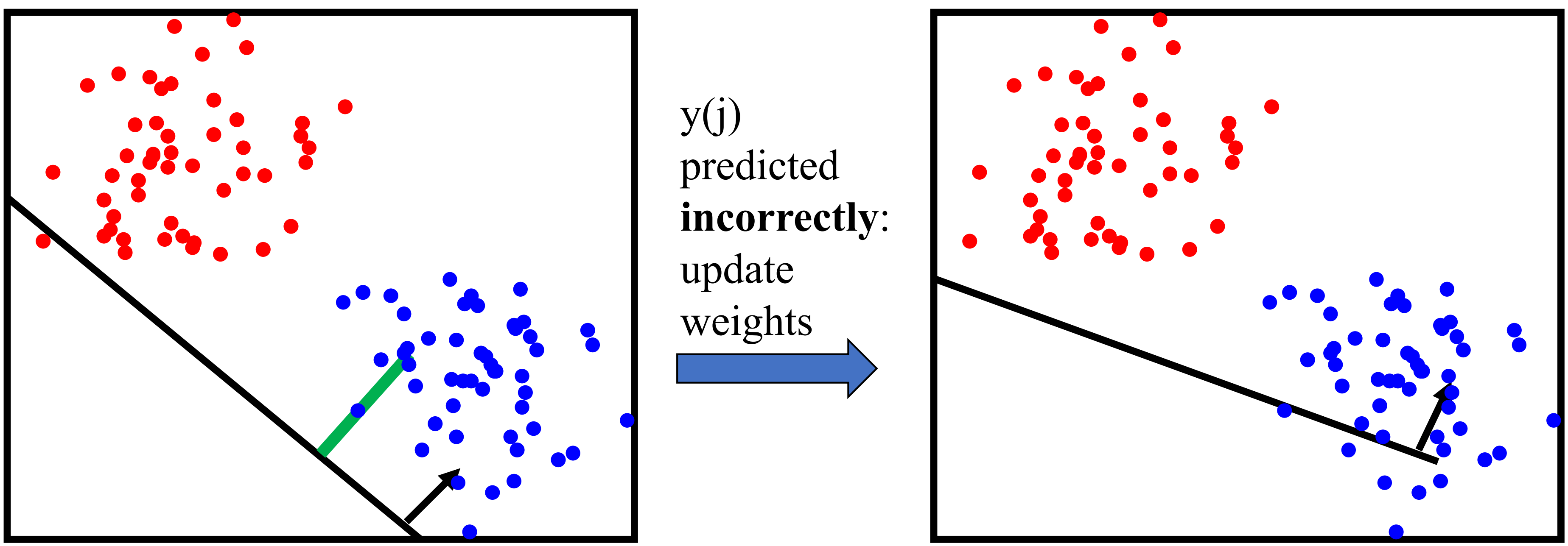
- Perceptron algorithm: an SGD-like algorithm

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

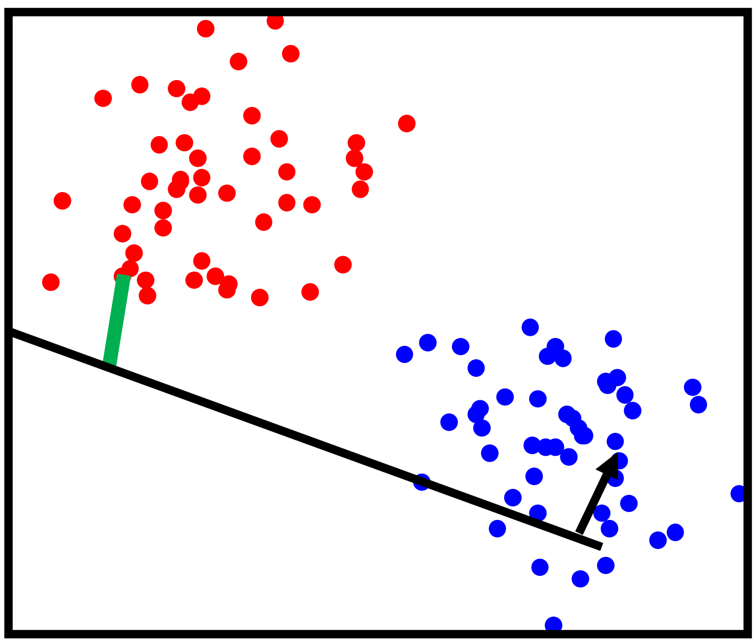
- Perceptron algorithm: an SGD-like algorithm

while $\neg$ done:

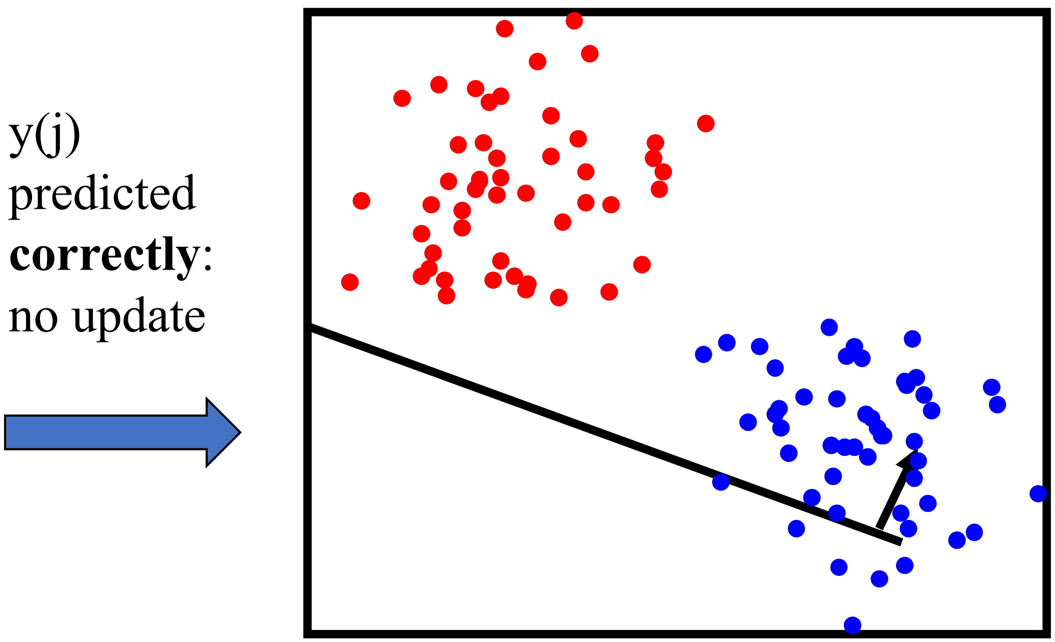
for each data point j:

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

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



y(j) predicted **correctly**: no update



Perceptron algorithm

- Perceptron algorithm: an SGD-like algorithm

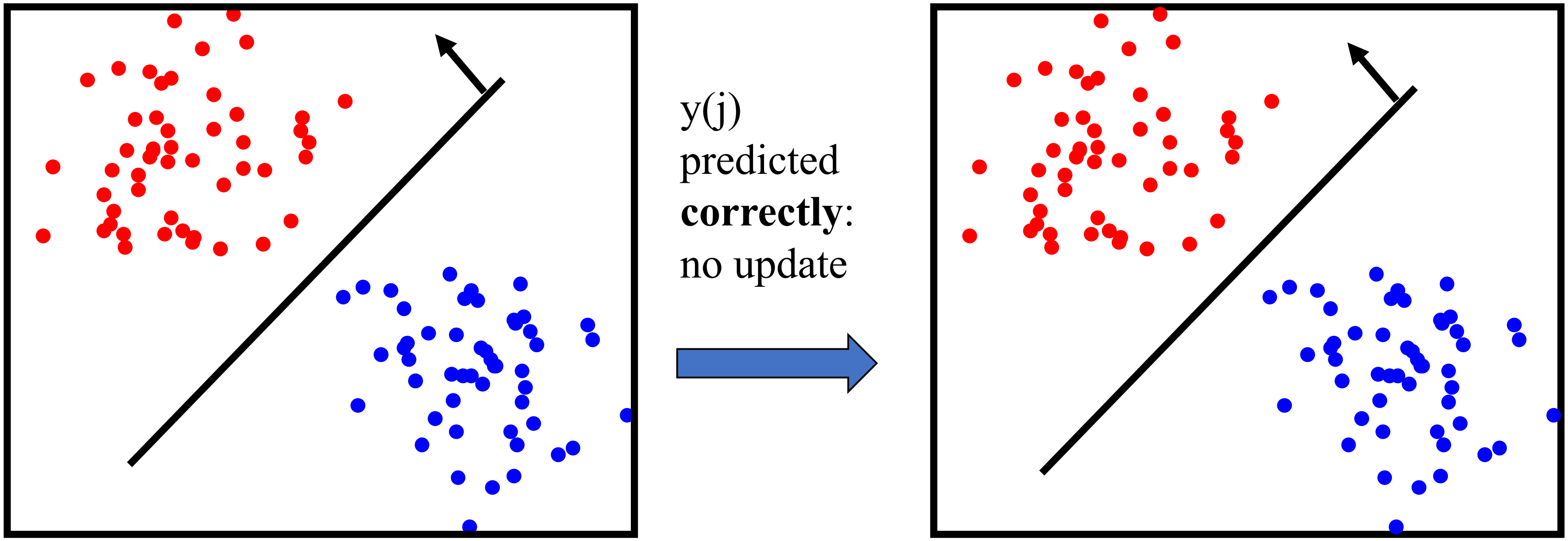
$$\text{while } \neg \text{ done:}$$

$$\quad \text{for each data point } j:$$

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

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

(Converges if data are linearly separable)



Perceptron MARK 1 Computer

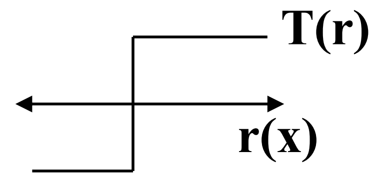
Frank Rosenblatt, late 1950s

Linear Classifiers

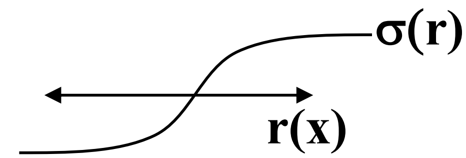
- Linear Classification with Perceptrons
- Perceptron Learning
- **Gradient-Based Classifier Learning**
- Multi-Class Classification
- Regularization for Linear Classifiers

Surrogate loss functions

- Another solution: use a “smooth” loss
  - e.g., approximate the threshold function



  - Usually some smooth function of distance
    - Example: logistic “sigmoid”, looks like an “S”



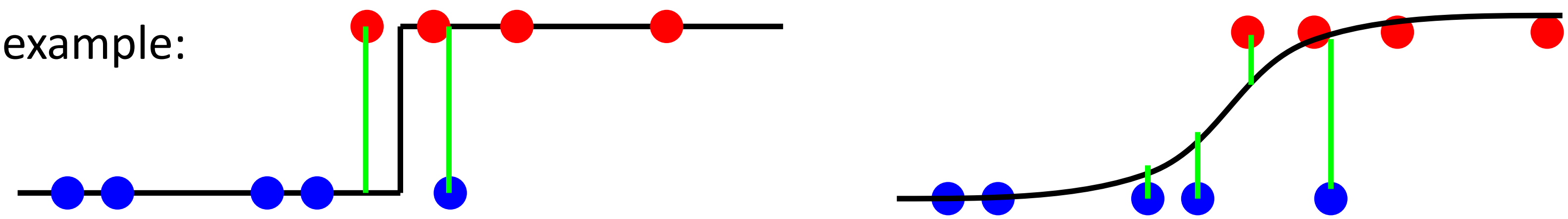
  - Now, measure e.g. MSE

Class y = {0, 1} …

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

  - Far from the decision boundary: |f(.)| large, small error
  - Nearby the boundary: |f(.)| near 1/2, larger error

1D example:



Classification error = 2/9

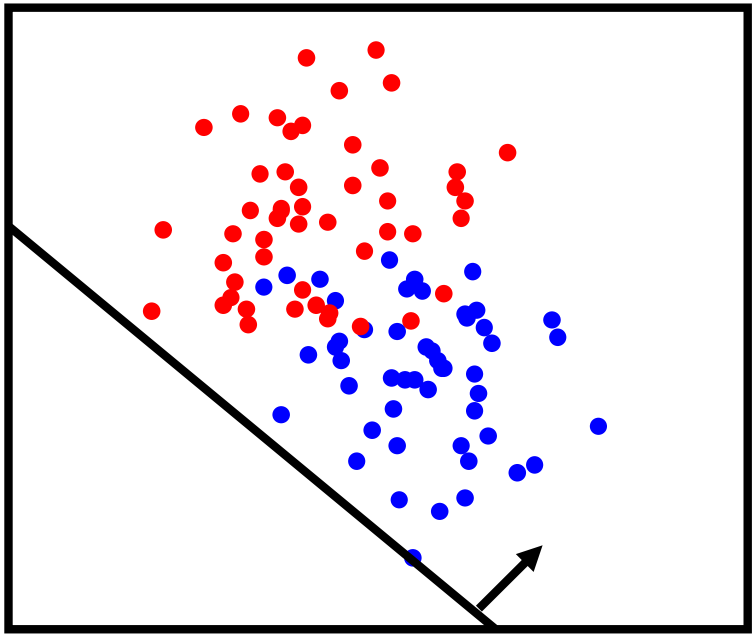
MSE = $(0^2 + .1^2 + .2^2 + .25^2 + .05^2 + \ldots)/9$

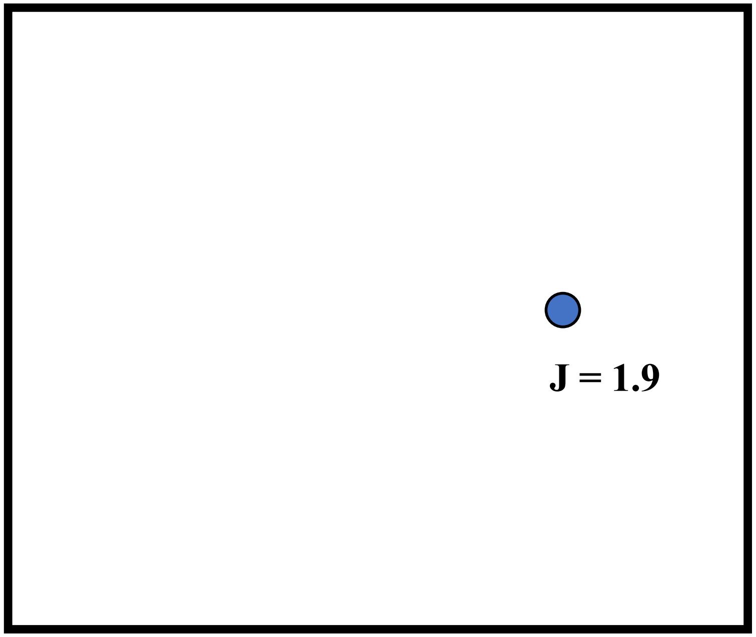
Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of

$$r(x_1,x_2) = a*x_1 + b*x_2 + c$$

- Example: 2D feature space ⇔ parameter space



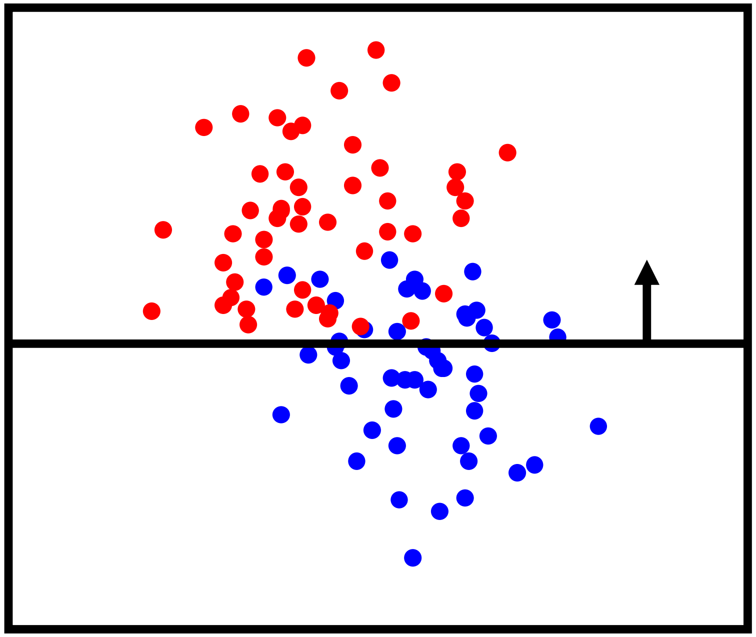


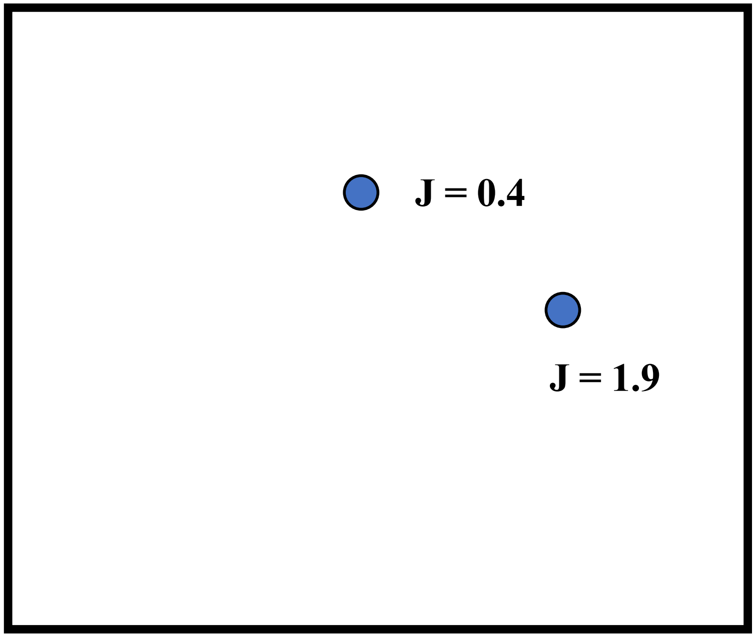
Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of

$$r(x_1,x_2) = a^*x_1 + b^*x_2 + c$$

- Example: 2D feature space ⬄ parameter space



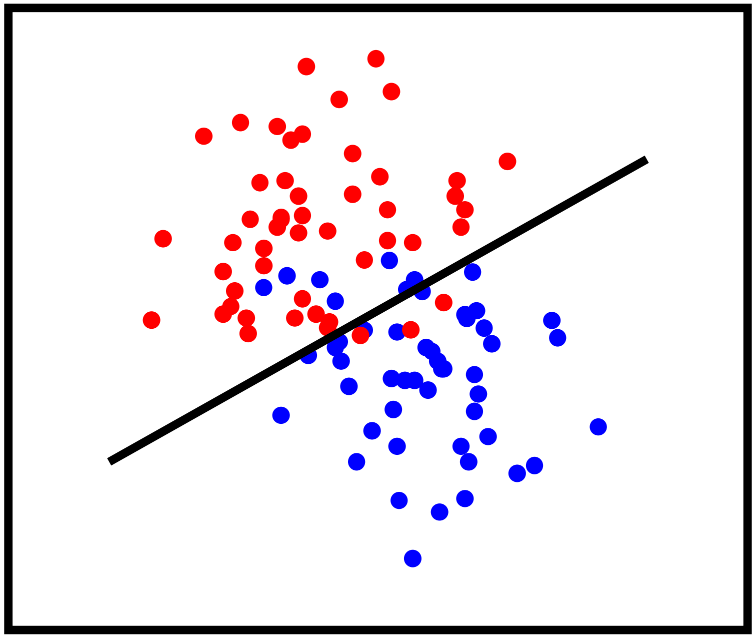


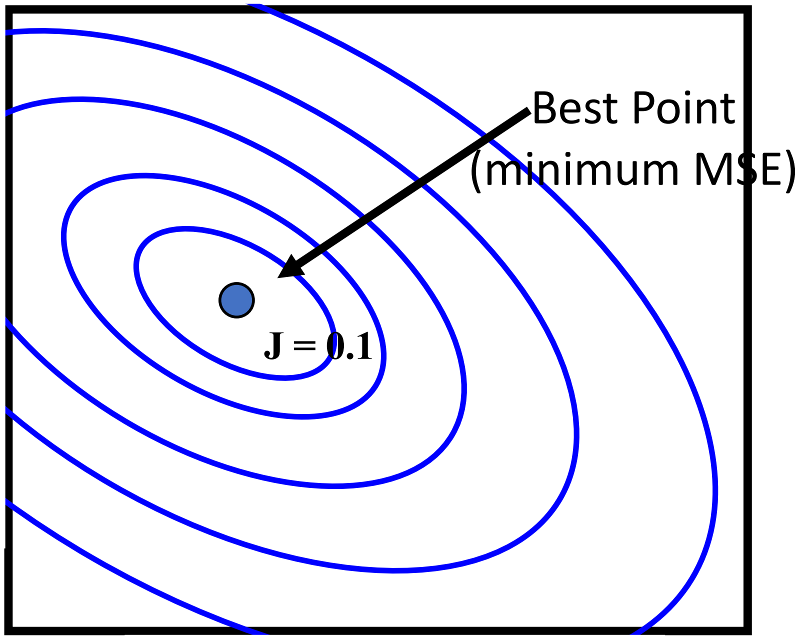
Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of

$$r(x_1,x_2) = a*x_1 + b*x_2 + c$$

- Example: 2D feature space ⇔ parameter space





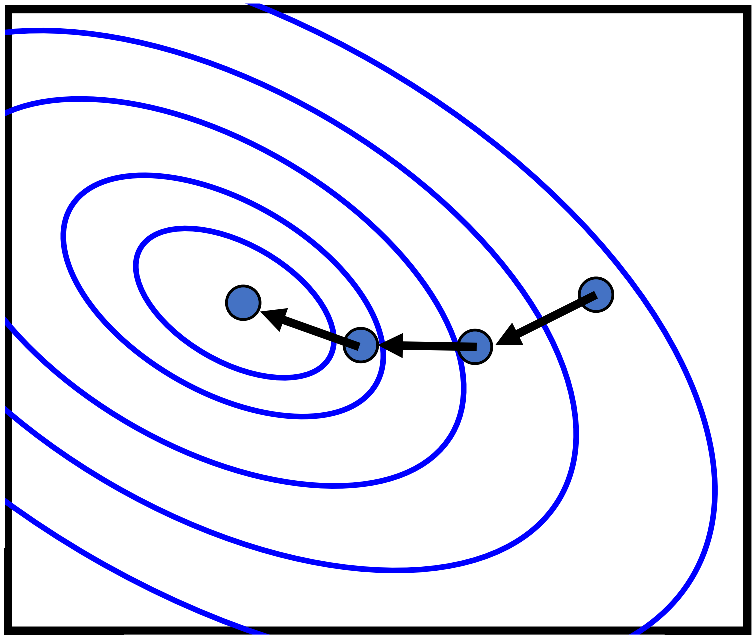
Finding the Best MSE

- As in linear regression, this is now just optimization

- Methods:
  - Gradient descent
    - Improve loss by small changes in parameters ("small" = learning rate)

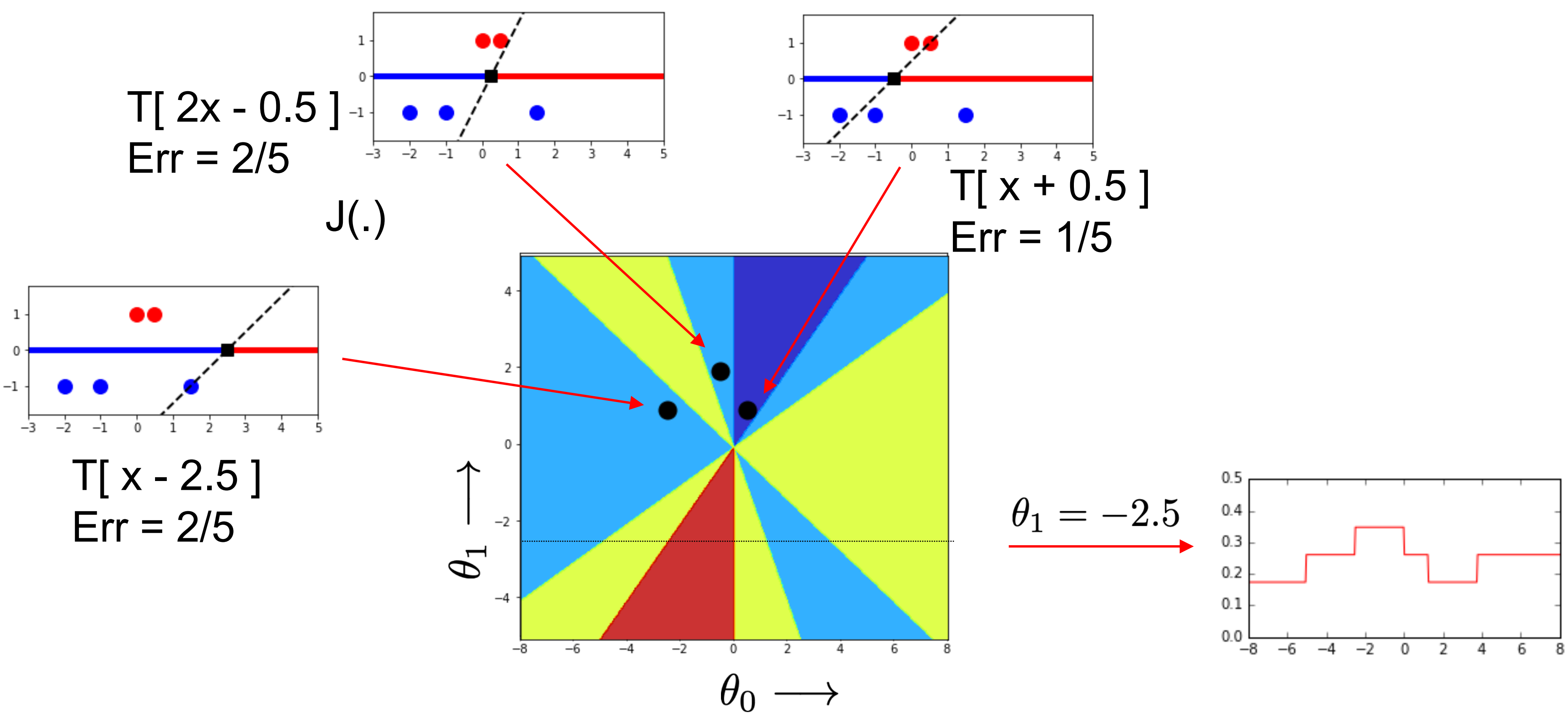
  - Or, substitute your favorite optimization algorithm…
    - Coordinate descent
    - Stochastic search

Gradient Descent



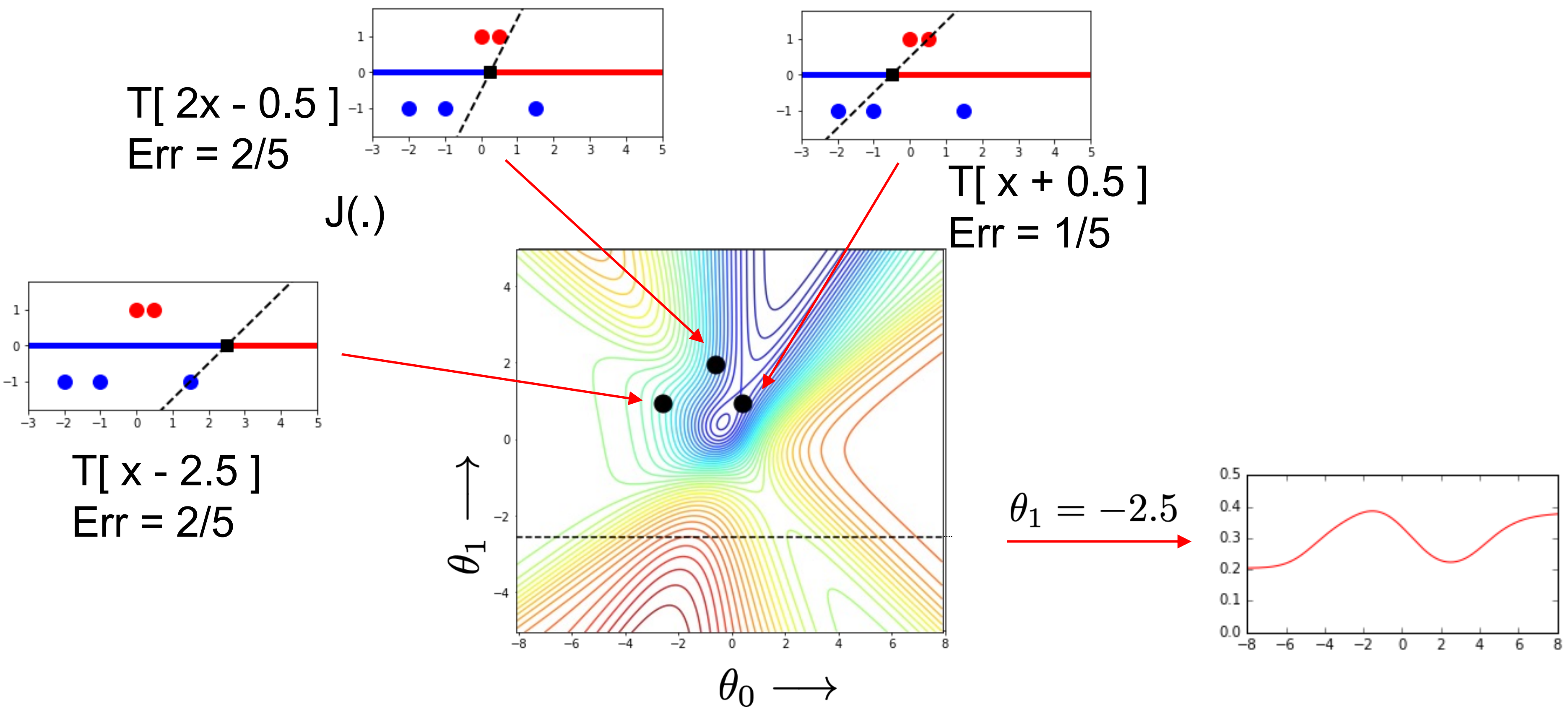
0/1 (Error Rate) Loss Example

- Data set: one feature, 5 data points



Logistic MSE Surrogate

- Smoother version of 0/1 Loss



Gradient Equations

- MSE (note, depends on function σ(.))

$$J(\underline{\theta} = [a, b, c]) = \frac{1}{m} \sum_i \left(\sigma(a x_1^{(i)} + b x_2^{(i)} + c) - y^{(i)} \right)^2$$

- What's the derivative with respect to one of the parameters?
  - Recall the chain rule of calculus:

$$\frac{\partial}{\partial a} f(\, g(\, h(a)\,)\,) = f'(\, g(h(a))\,)\; g'(\, h(a)\,)\; h'(a)$$

$$f(g) = (\,g\,)^2 \qquad \Rightarrow \qquad f'(g) = 2(g)$$

$$g(h) = \sigma(h) - y \qquad \Rightarrow \qquad g'(h) = \sigma'(h)$$

$$h(a) = a x_1^{(i)} + b x_2^{(i)} + c \qquad \Rightarrow \qquad h'(a) = x_1^{(i)}$$

w.r.t. b,c : similar; replace x_1 with x_2 or 1

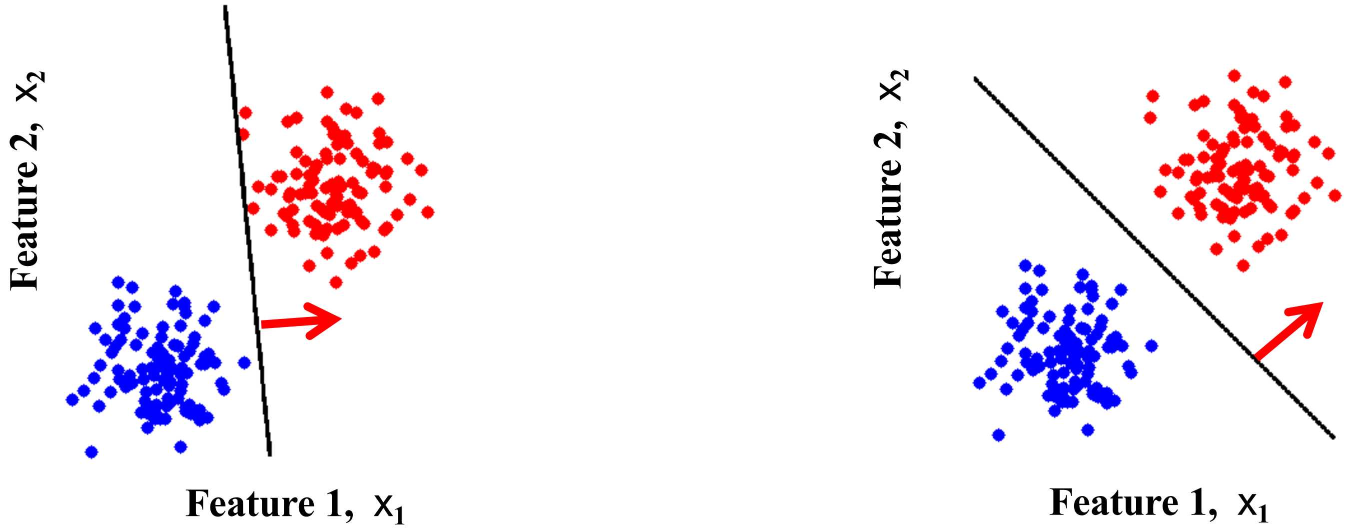
$$\frac{\partial J}{\partial a} = \frac{1}{m} \sum_i 2 \boxed{(\sigma(\theta \cdot x^{(i)}) - y^{(i)})}\; \boxed{\partial\sigma(\theta \cdot x^{(i)})\;\; x_1^{(i)}}$$

Error between class and prediction

Sensitivity of prediction to changes in parameter "a"

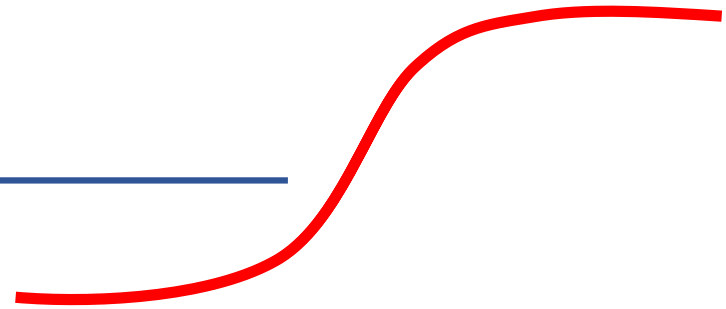
Beyond misclassification rate

- 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

(z = linear response, $x^T\theta$)

$$\sigma(z) = 1 / (1 + \exp(-z))$$

- 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]:

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

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

Predict: threshold z or ρ at zero

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 $(1 - \sigma(\underline{\theta}\, x^T))$

(**Note:** Some software calls this "cross entropy")

- Can write this succinctly:

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

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 - $\sigma(\underline{\theta}\, x^T)$)

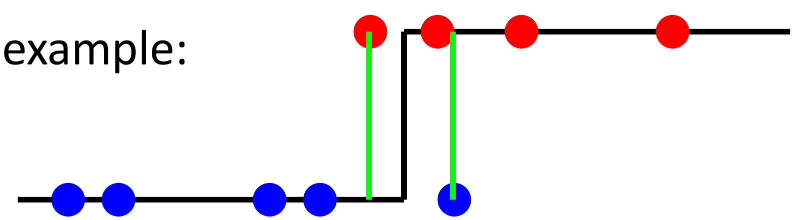
(**Note:** Some software calls this "cross entropy")

- Can write this succinctly:

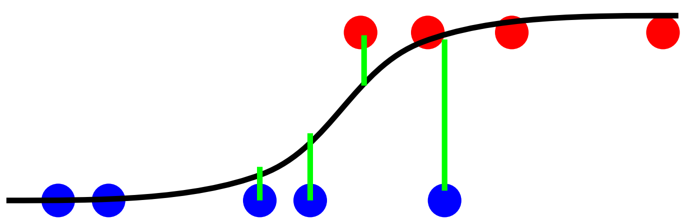
$$J(\underline{\theta}) = -\frac{1}{m}\Big(\sum_i y^{(i)} \log \sigma(\theta \cdot x^{(i)}) + (1 - y^{(i)}) \log(1 - \sigma(\theta \cdot x^{(i)}))\Big)$$

  - Convex! Otherwise similar: optimize J(θ) via …

1D example:



Classification error = 2/9



NLL = - (log(.99) + log(.97) + …)/9

Gradient Equations

- Logistic neg-log likelihood loss:

$$J(\underline{\theta}) = -\frac{1}{m}\Big(\sum_i y^{(i)} \log \sigma(\theta \cdot x^{(i)}) + (1 - y^{(i)}) \log(1 - \sigma(\theta \cdot x^{(i)}))\Big)$$

- What's the derivative with respect to one of the parameters?

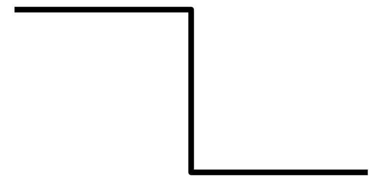
$$\frac{\partial J}{\partial a} = -\frac{1}{m}\Big(\sum_i y^{(i)} \frac{1}{\sigma(\theta \cdot x^{(i)})}\ \partial\sigma(\theta \cdot x^{(i)})\ x_1^{(i)} + (1 - y(i)) \ldots\Big)$$

$$= -\frac{1}{m}\Big(\sum_i y^{(i)} (1 - \sigma(\theta \cdot x^{(i)}))\ x_1^{(i)} + (1 - y^{(i)}) \ldots\Big)$$

Surrogate loss functions

- Replace 0/1 loss $\Delta_i(\theta) = \mathbb{1}\left[T(\theta x^{(i)}) \neq y^{(i)}\right]$

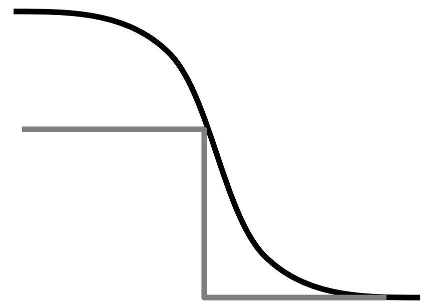
  with something easier:



0 / 1 Loss

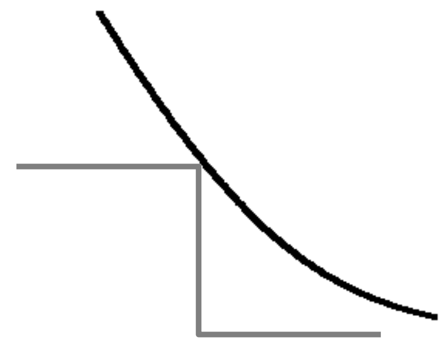
- Logistic MSE

$$J_i(\theta) = 4\big(\sigma(\theta x^{(i)}) - y^{(i)}\big)^2$$



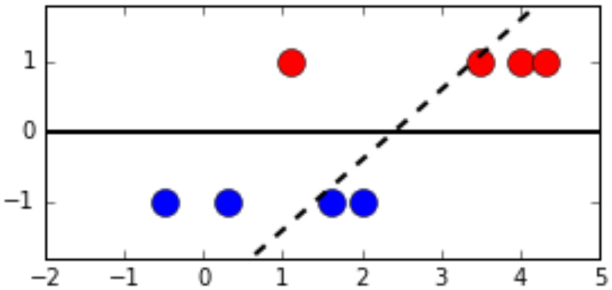
- Logistic Neg Log Likelihood

$$J_i(\underline{\theta}) = -\frac{y^{(i)}}{\log 2}\log\sigma(\theta \cdot x^{(i)}) + \dots$$

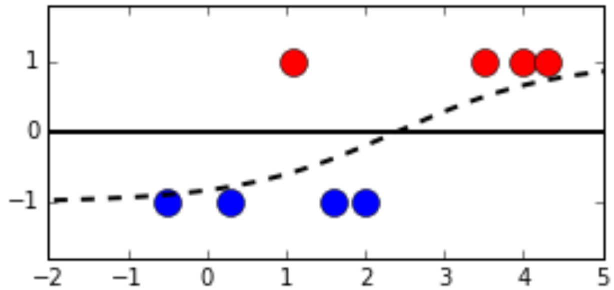


Surrogate loss functions

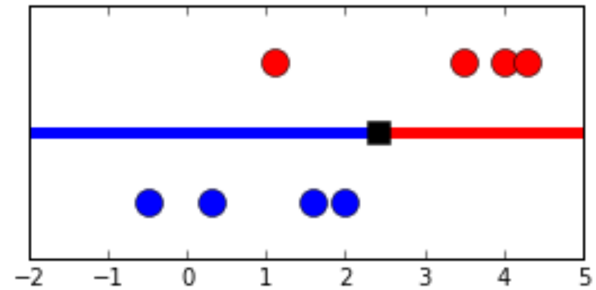
θ = [-2.4, 1.0]



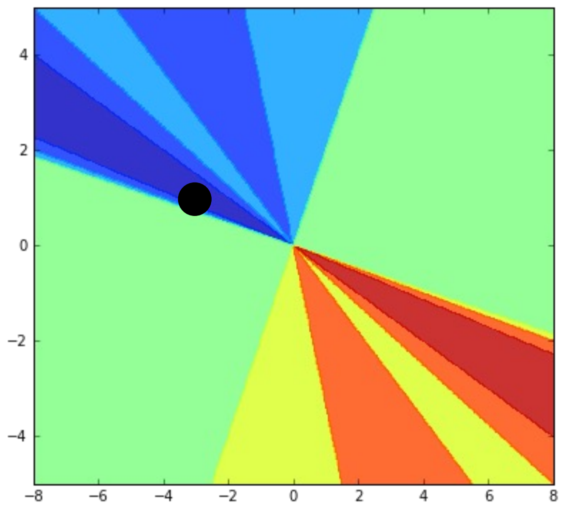
Linear response: r(x)



Saturating response: ρ(r(x))

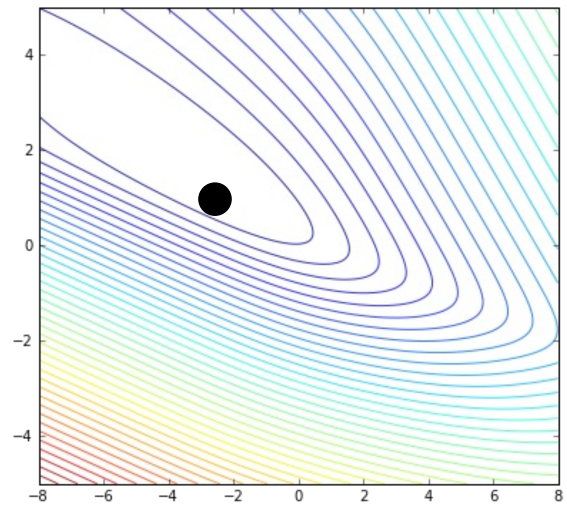


Decision function: T(r(x))

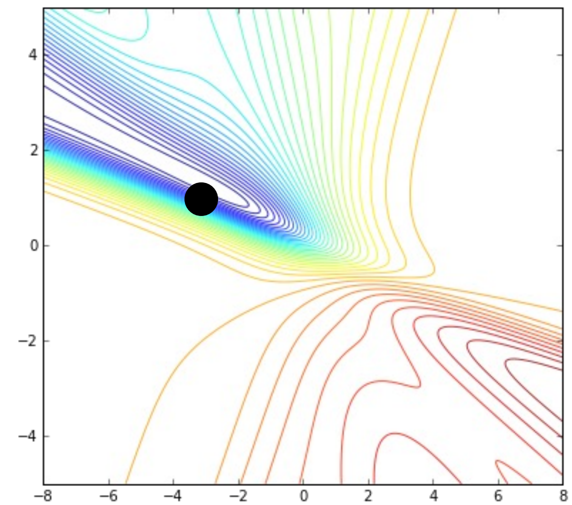


J(θ)

0/1 Loss



Logistic NLL



Logistic MSE

Summary

- Linear classifier ⇔ perceptron

- Measuring quality of a decision boundary
  - Error rate (0/1 loss)
  - Logistic sigmoid + MSE criterion
  - Logistic Regression

- Learning the weights of a linear classifer from data
  - Reduces to an optimization problem
  - Perceptron algorithm
  - For MSE or Logistic NLL, we can do gradient descent
  - Gradient equations & update rules

Linear Classifiers

Linear Classification with Perceptrons

Perceptron Learning

Gradient-Based Classifier Learning

Multi-Class Classification

Regularization for Linear Classifiers

Multi-class linear models

- What about multiple classes? One option:
  - Define one linear response per class
  - Choose class with the largest response

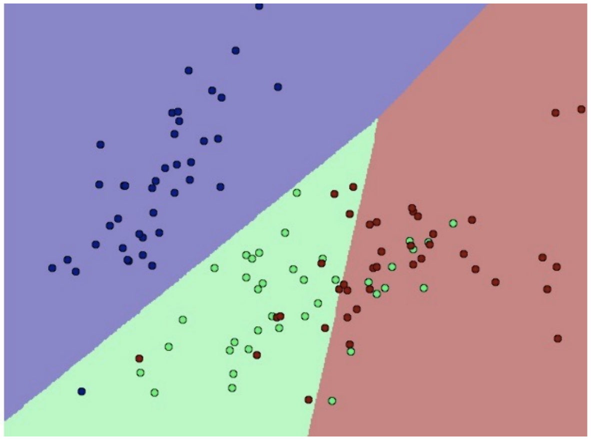
$$\theta = \begin{bmatrix} \theta_{00} & \dots & \theta_{0n} \\ \vdots & \ddots & \vdots \\ \theta_{C0} & \dots & \theta_{Cn} \end{bmatrix}$$

$$f(x;\theta) = \arg\max_c \ \theta_c \cdot x^T$$

  - Boundary between two classes, c vs. c'?

$$= \begin{cases} c & \text{if } \ \theta_c \cdot x^T > \theta_{c'} x^T \qquad \Leftrightarrow (\theta_c - \theta_{c'}) x^T > 0 \\ c' & \text{otherwise} \end{cases}$$

  - Linear boundary: $(\theta_c - \theta_{c'})\, x^T = 0$



Multiclass perceptron algorithm

- Perceptron algorithm:
  - Make prediction f(x)
  - Increase linear response of true target y; decrease for prediction f

$$
\begin{aligned}
&\text{while} \neg \text{ done:} \\
&\quad \text{for each data point } j\text{:} \\
&\qquad f^{(j)} = \arg\max_c(\theta_c \cdot x^{(j)}) \\
&\qquad \theta_{f^{(j)}} \leftarrow \theta_{f^{(j)}} - \alpha\, x^{(j)} \\
&\qquad \theta_{y^{(j)}} \leftarrow \theta_{y^{(j)}} + \alpha\, x^{(j)}
\end{aligned}
$$

Predict output for data point j

Decrease response of class f on data $x^{(j)}$

Increase response of class y on data $x^{(j)}$

If prediction "f" matches true class "y", no update

Otherwise, makes it more likely that "y" will have the largest response next time

Multilogit regression

- Define the probability of each class:

$$p(Y = y|X = x) = \frac{\exp(\theta_y \cdot x^T)}{\sum_c \exp(\theta_c \cdot x^T)}$$

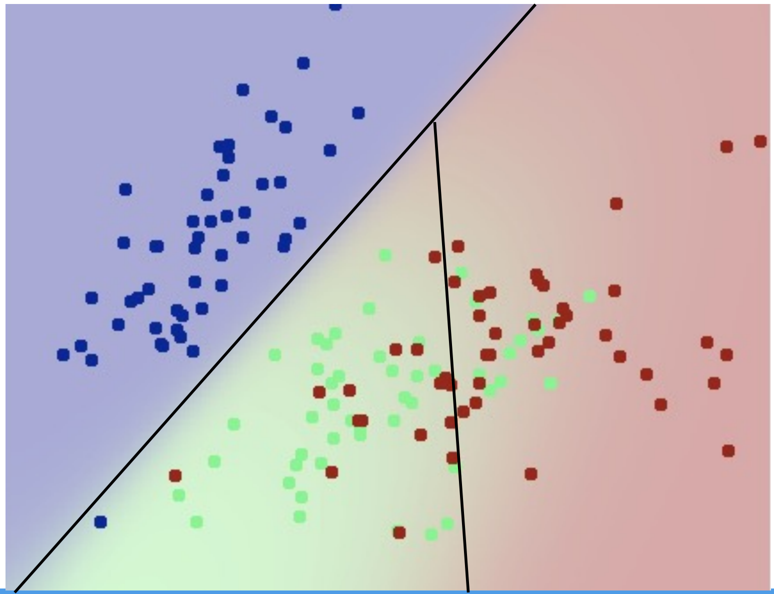
(**Note:** Some software calls this function "softmax")

(Y binary = logistic regression)

- Then, the NLL loss function is:

$$J(\theta) = -\frac{1}{m}\sum_i \log p(y^{(i)}|x^{(i)}) = -\frac{1}{m}\sum_i \left[\theta_{y^{(i)}} \cdot x^{(i)} - \log \sum_c \exp(\theta_c \cdot x^{(i)})\right]$$

- P: "confidence" of each class
  - Soft decision value
- Decision: predict most probable
  - Linear decision boundary
- Convex loss function



Linear Classifiers

Linear Classification with Perceptrons

Perceptron Learning

Gradient-Based Classifier Learning

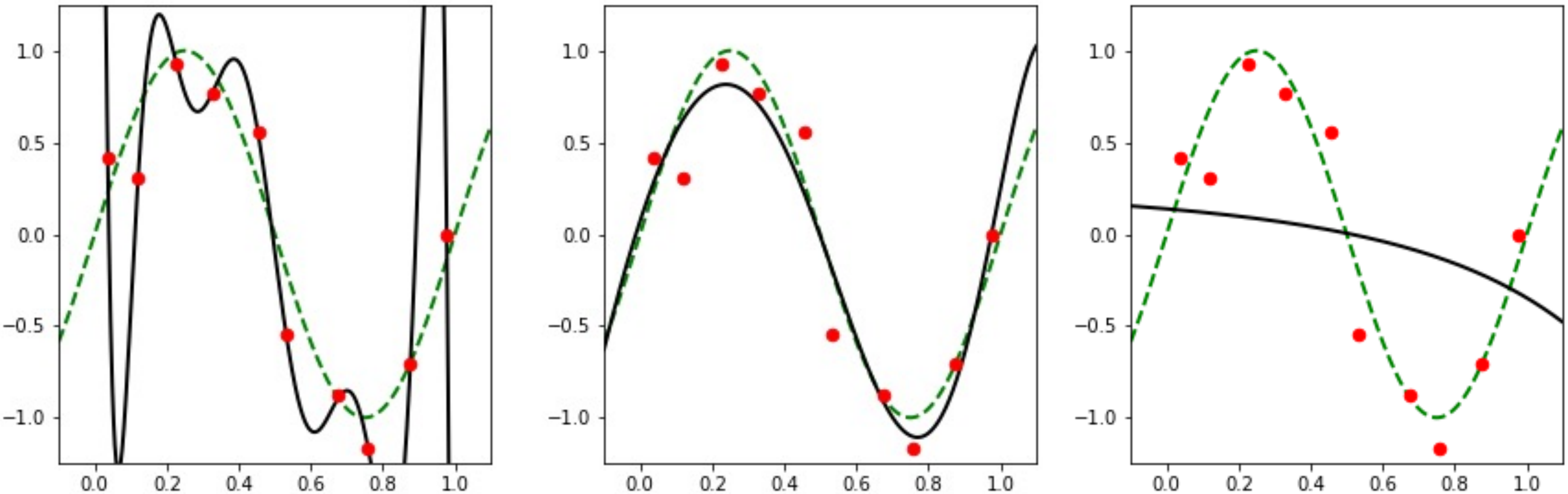
Multi-Class Classification

Regularization for Linear Classifiers

Regularization

- Reminder: Regularization for linear regression
  - Encourage models with small parameter values
  - Can reduce overfitting (by reducing variance)

Degree-9 polynomial fit



No regularization | Moderate regularization | Strong regularization

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:

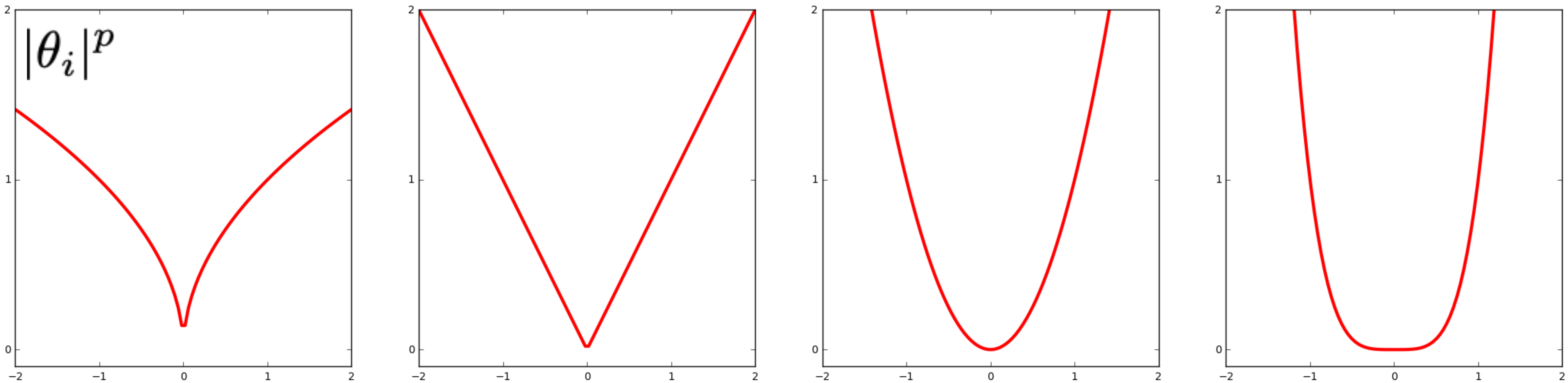
$$J(\underline{\theta}) = -\frac{1}{m}\Big(\sum_i \underbrace{y^{(i)} \log \sigma(\theta \cdot x^{(i)})}_{\text{Nonzero only if y=1}} + \underbrace{(1-y^{(i)}) \log(1-\sigma(\theta \cdot x^{(i)}))}_{\text{Nonzero only if y=0}}\Big)$$

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

(Also equivalent: minimize NLL subject to Lp norm < R)

Different regularization functions

- More generally, for the L_p regularizer: $$\left(\sum_i |\theta_i|^p \right)^{\frac{1}{p}}$$



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

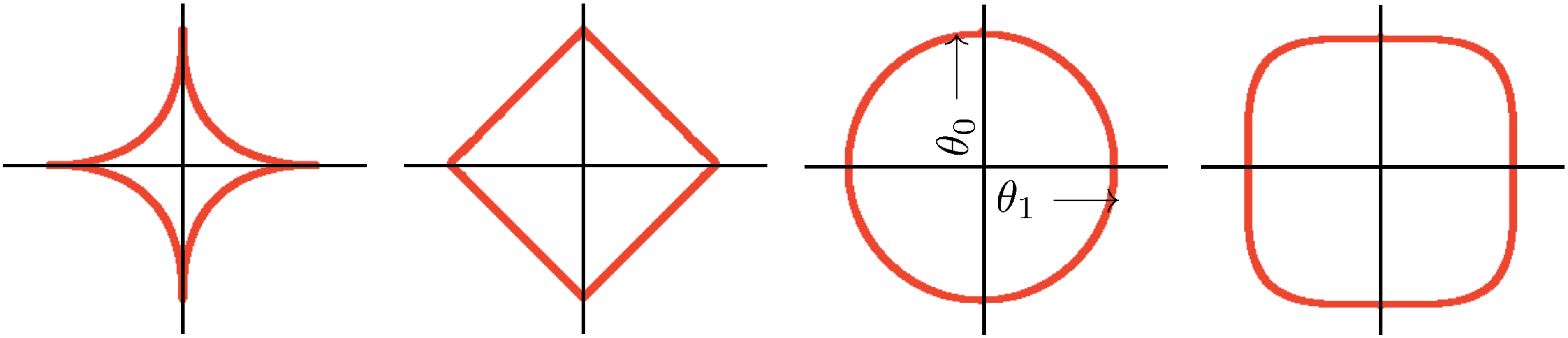
Penalize small non-zero parameters more
Prefer some parameters exactly zero;
a few big parameters are OK

Penalize small non-zero parameters less,
but big parameter values a lot
Prefer lots of small weights, no big weights

Different regularization functions

- More generally, for the L_p regularizer: $\left(\sum_i |\theta_i|^p \right)^{\frac{1}{p}}$

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



p=0.5

p=1
Lasso

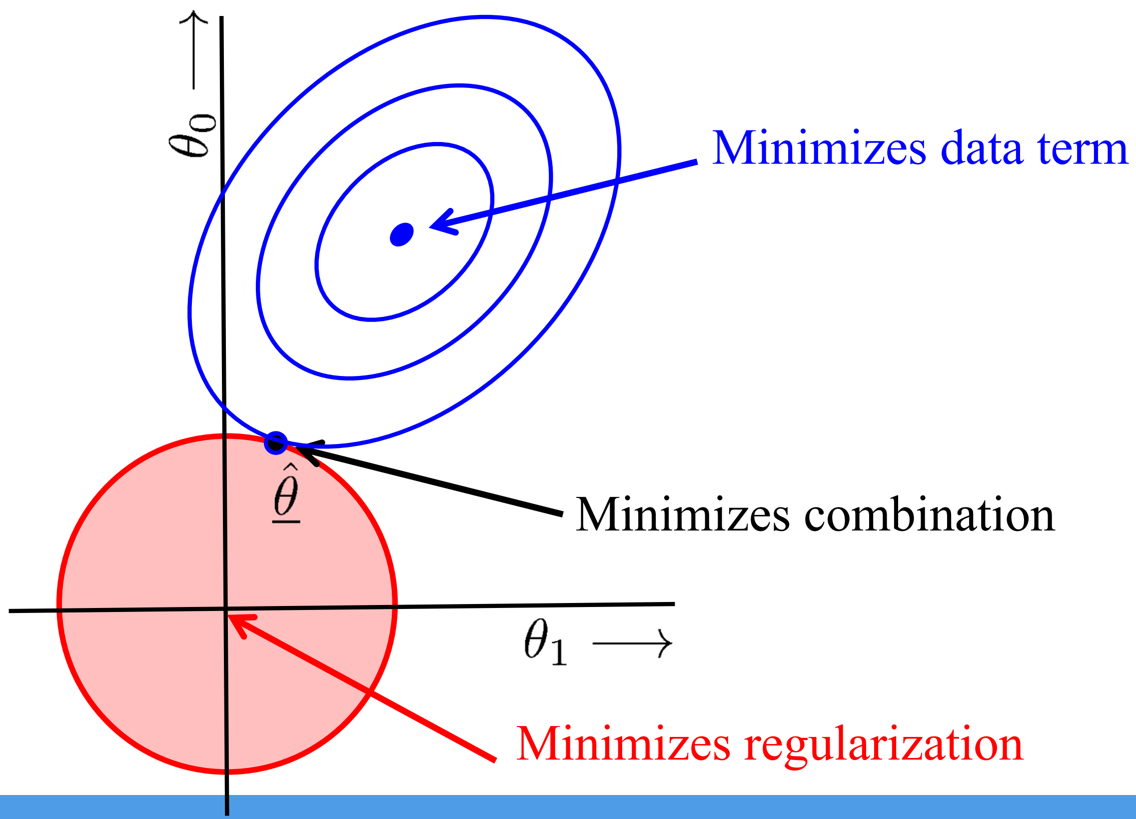
p=2
Quadratic

p=4

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

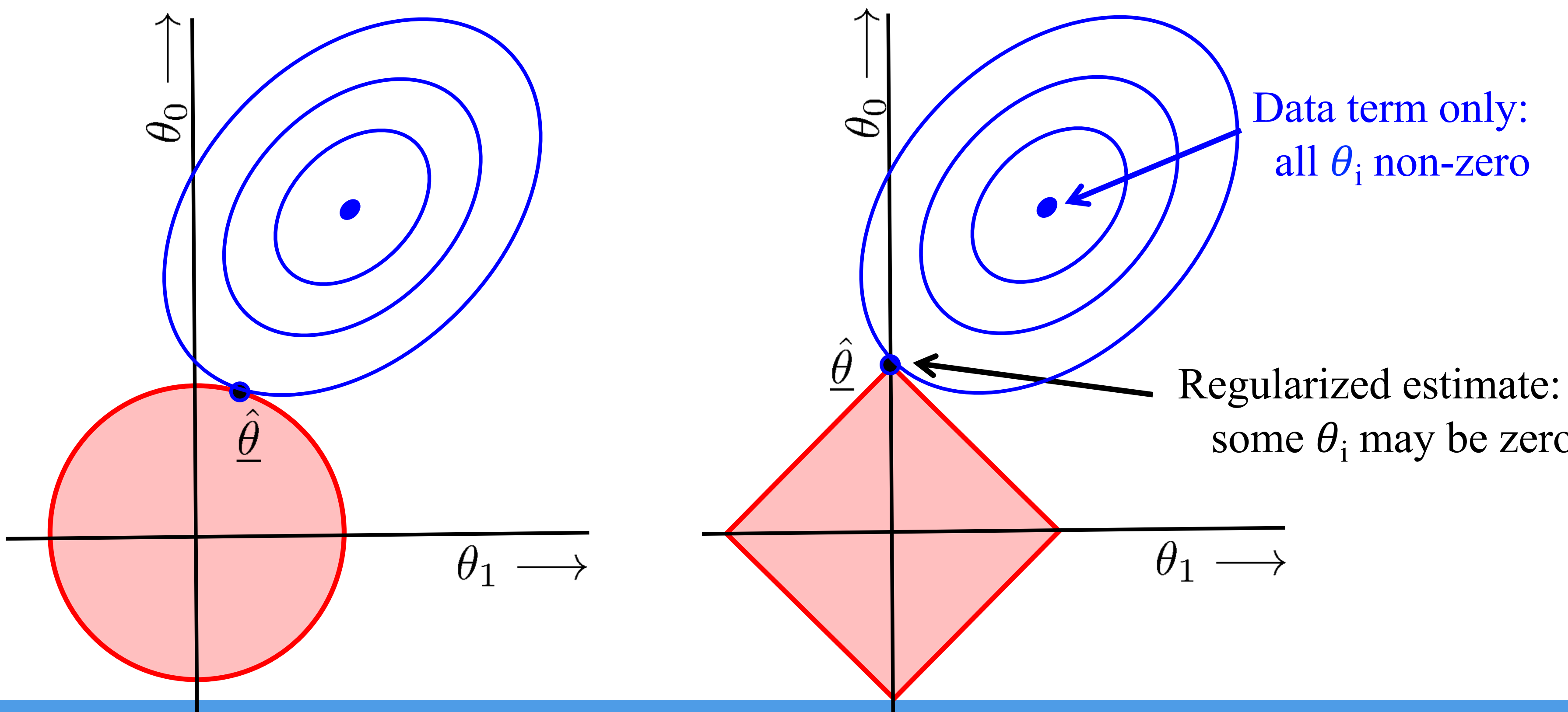
Regularization: L2 vs L1

- Estimate balances data term & regularization term



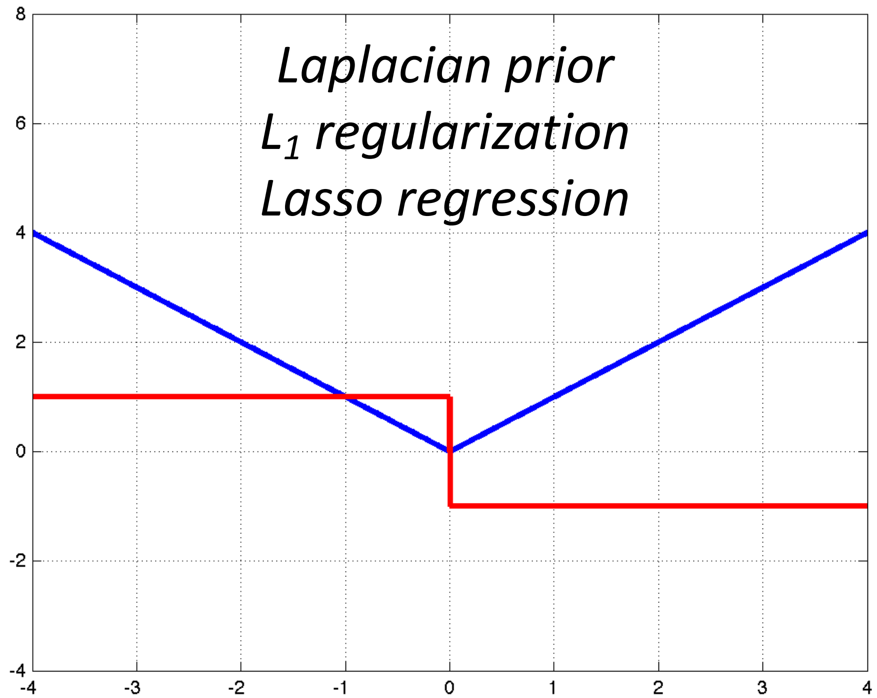
Regularization: L2 vs L1

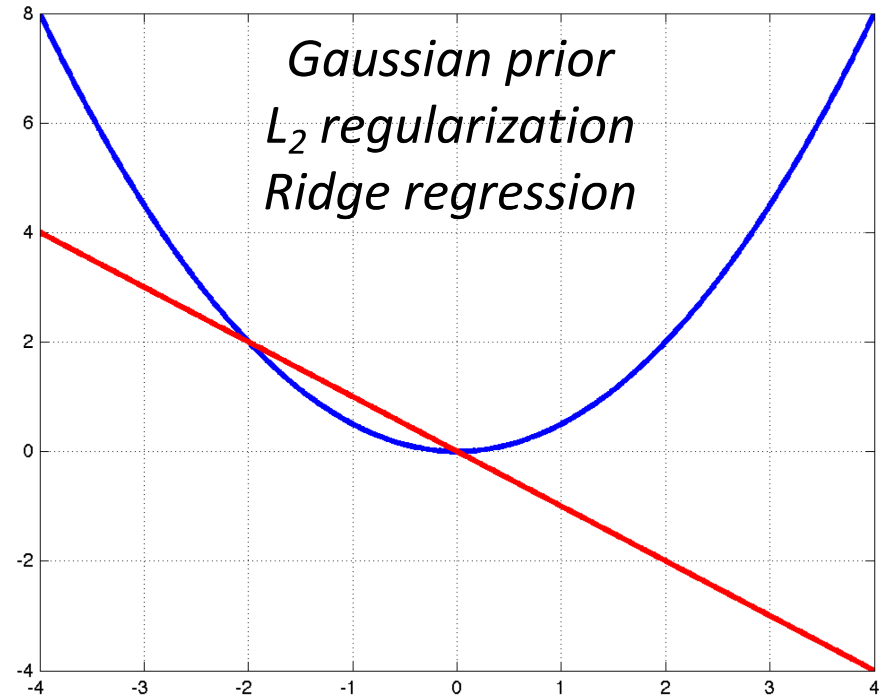
- Estimate balances data term & regularization term
- Lasso tends to generate sparser solutions than a quadratic regularizer.



Gradient-Based Optimization

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





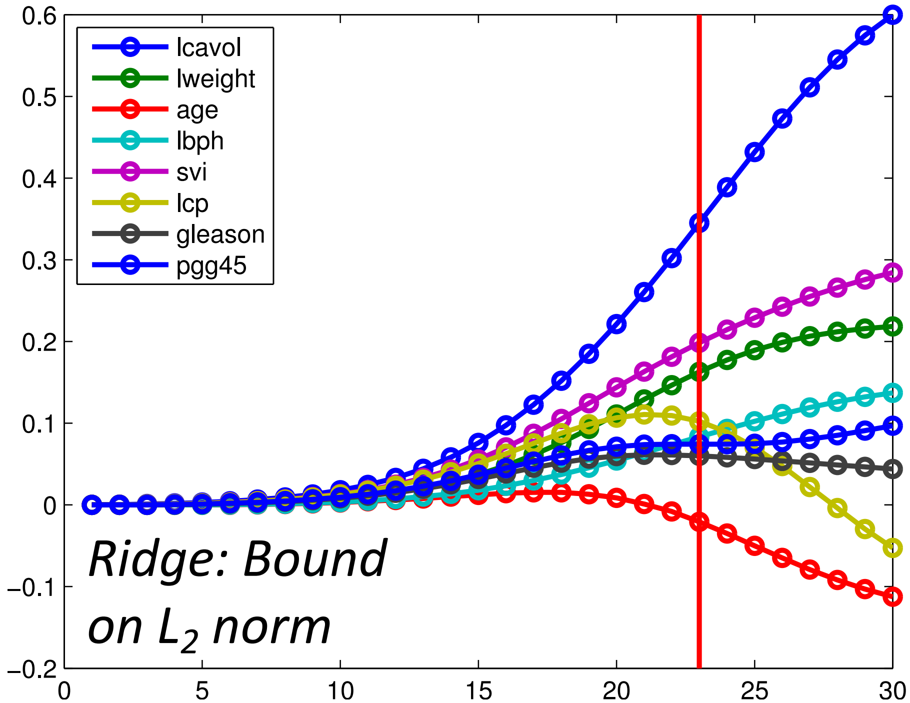
Objective Function: $f(\theta_i) = |\theta_i|^p$

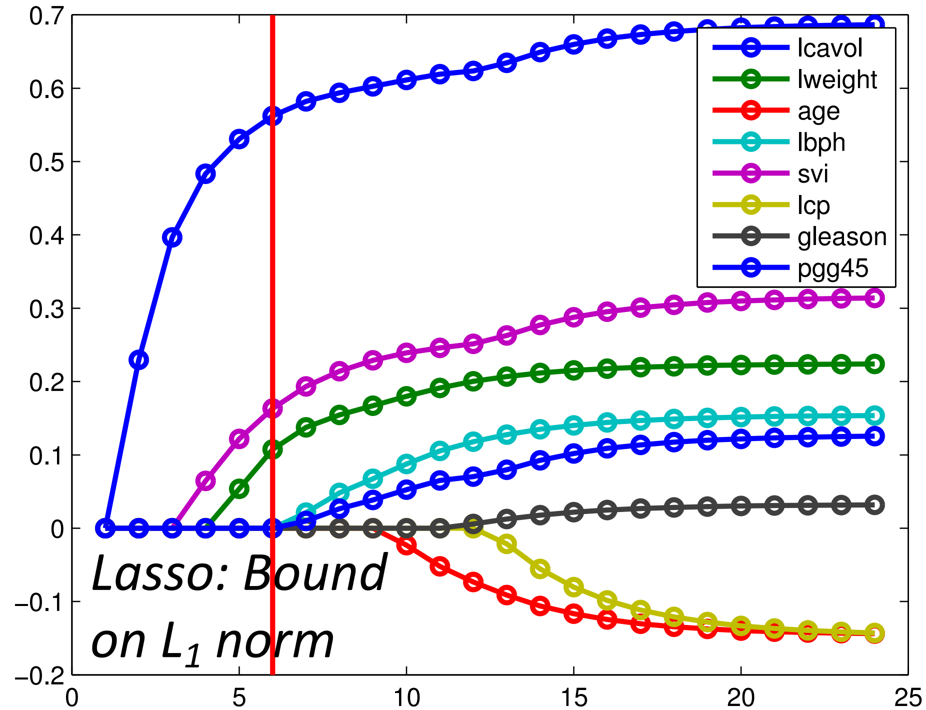
Negative Gradient: $-f'(\theta_i)$

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

Regularization Paths

Prostate Cancer Dataset with M=67, N=8





- Horizontal axis increases bound on weights (less regularization, smaller α)
- For each bound, plot values of estimated feature weights
- Vertical lines are models chosen by cross-validation