Linear Classification

PROF XIAOHUI XIE
SPRING 2019

CS 273P Machine Learning and Data Mining

Slides courtesy of Alex Ihler

Machine Learning

Linear Classification with Perceptrons

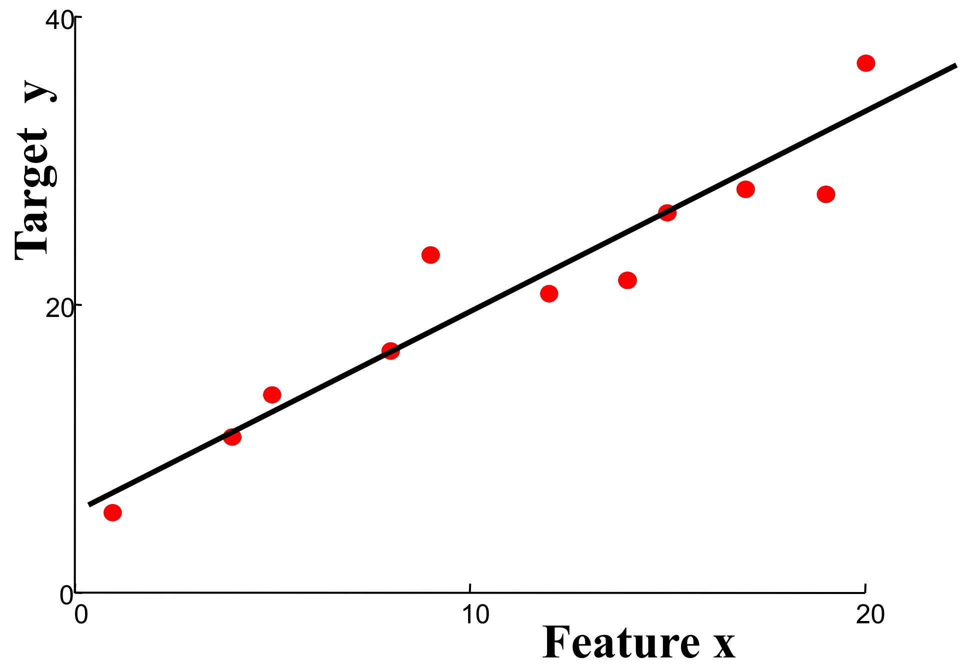
Perceptron Learning

Gradient-Based Classifier Learning

Multi-Class Classification

Regularization for Linear Classification

Linear regression



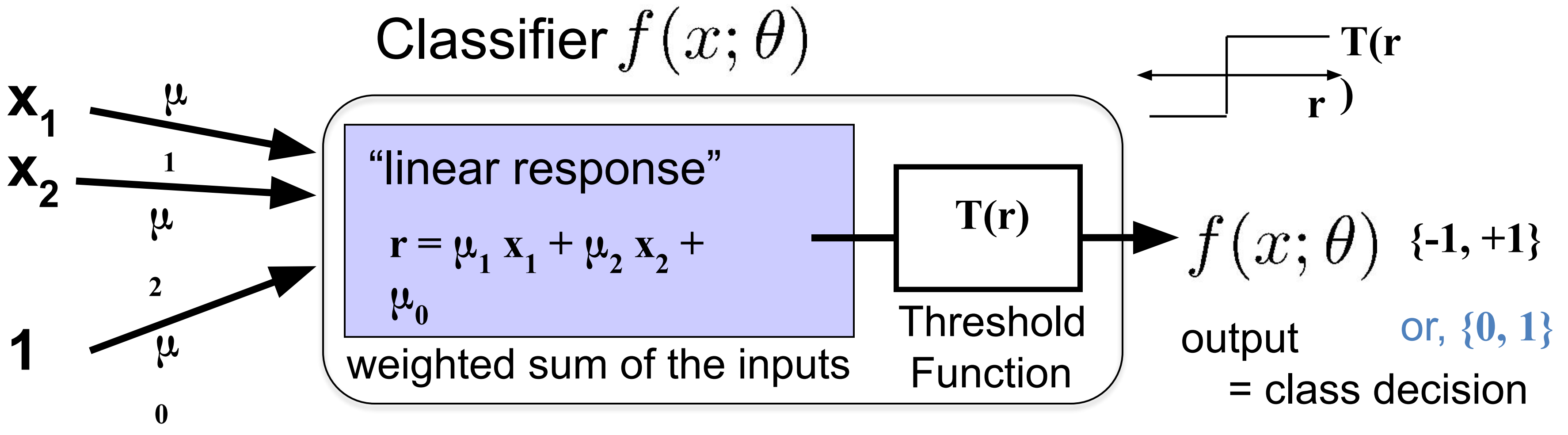
"Predictor":
Evaluate line:

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

return r

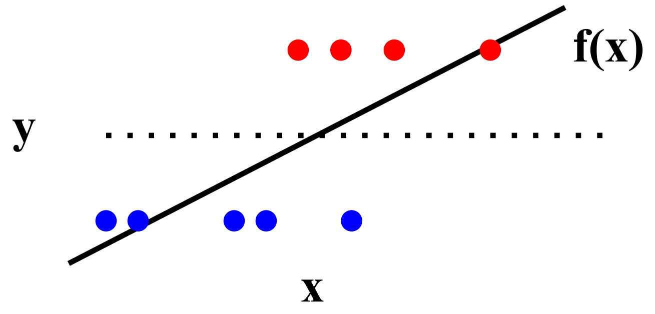
- Contrast with classification
  - Classify: predict discrete-valued target y
  - Initially: "classic" binary { -1, +1} classes; generalize later

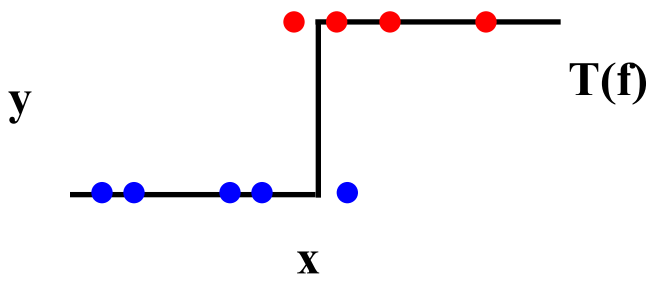
Perceptron Classifier (2 features)



```
r = X.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”:





Perceptrons

- Perceptron = a linear classifier
  - The parameters μ 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 2 quantities:
  - 1. A weighted sum of the input features
  - 2. This sum is then thresholded by the T(.) function

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

Notation

- Inputs:
  - $x_0, x_1, x_2, \ldots\ldots\ldots\ldots, x_n,$
  - $x_1, x_2, \ldots\ldots\ldots\ldots, x_{n-1}, x_n$ are the values of the n features
  - $x_0 = 1$ (a constant input)
  - $\underline{x} = [[x_0, x_1, x_2, \ldots\ldots\ldots\ldots, x_n]]$: feature vector (row vector)
- Weights (parameters):
  - $\mu_0, \mu_1, \mu_2, \ldots\ldots\ldots\ldots, \mu_n,$
  - we have n+1 weights: one for each feature + one for the constant
  - $\underline{\mu} = [[\mu_0, \mu_1, \mu_2, \ldots\ldots\ldots\ldots, \mu_n]]$: parameter vector (row vector)
- Linear response
  - $\mu_0 x_0 + \mu_1 x_1 + \ldots\ \mu_n x_n = \underline{x} \cdot \underline{\mu}'$ then threshold

```
F = X.dot( theta.T );        # compute linear response
Yhat = np.sign(F)            # predict class +1 or -1
Yhat = 2*(F>0)-1             # manual "sign" of F
```

Perceptron Decision Boundary

- The perceptron is defined by the decision algorithm:

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

- The perceptron represents a hyperplane decision surface in d-dimensional space
  - A line in 2D, a plane in 3D, etc.

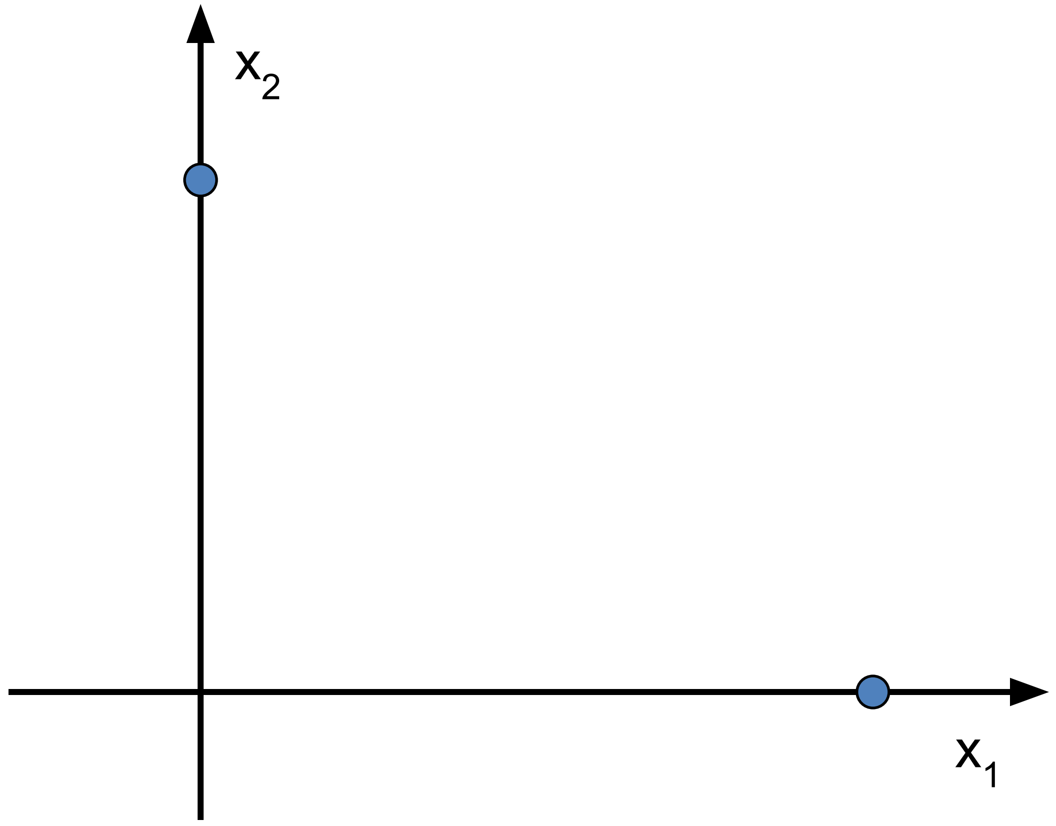
- The equation of the hyperplane is given by

$$\underline{\mu} \cdot \underline{\mathbf{x}}^{\mathbf{T}} = 0$$

This defines the set of points that are on the boundary.

Example, Linear Decision Boundary

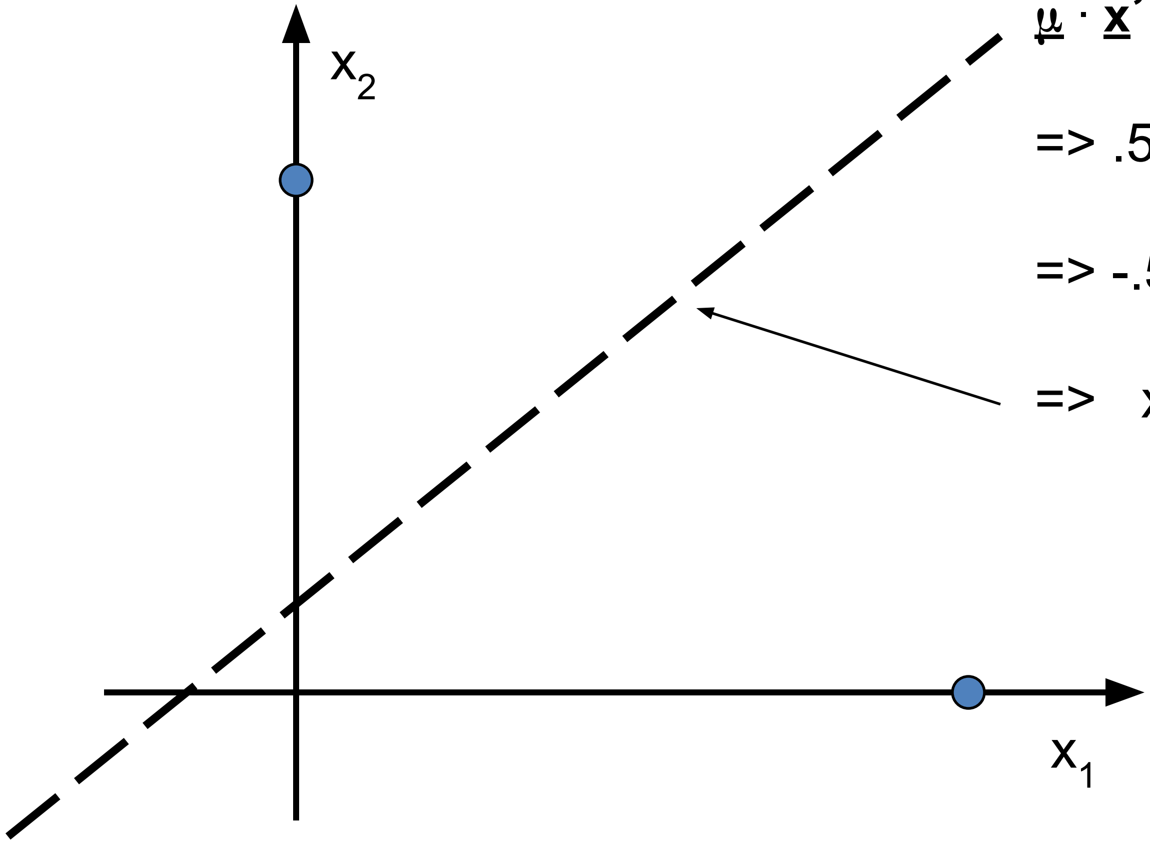
$$\underline{\mu} = (\mu_0, \quad \mu_1, \quad \mu_2)$$
$$= (1, \quad .5, \quad -.5\ )$$



From P. Smyth

Example, Linear Decision Boundary

$\underline{\mu} = (\mu_0, \quad \mu_1, \quad \mu_2)$
$= (1, \quad .5, \quad -.5\ )$



$\underline{\mu} \cdot \underline{\mathbf{x}}' = 0$

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

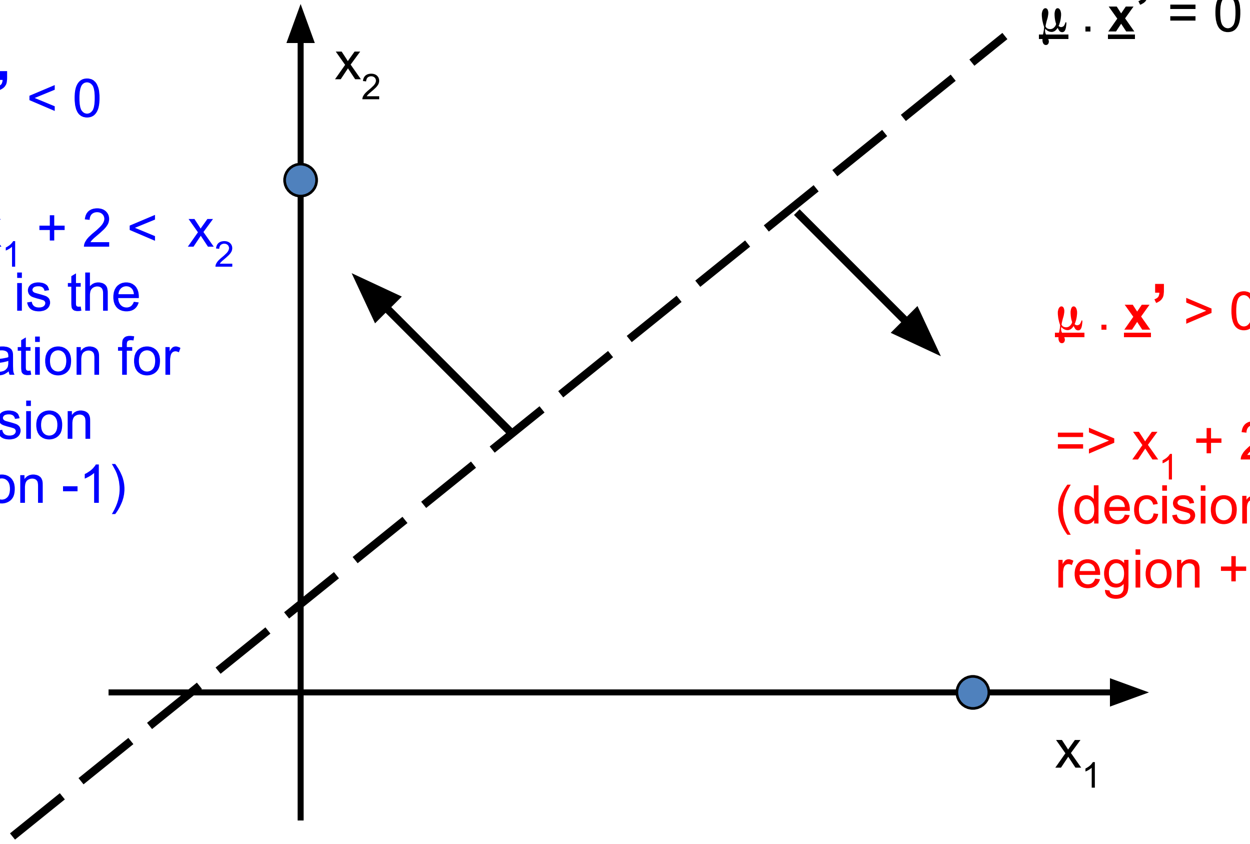
$\Rightarrow -.5\ x_2 = -.5\ x_1 - 1$

$\Rightarrow \quad x_2 = x_1 + 2$

From P. Smyth

Example, Linear Decision Boundary

$\underline{\mu} = (\mu_0, \ \mu_1, \ \mu_2)$
$= (1, \ .5, \ -.5)$



$\underline{\mu} \cdot \underline{\mathbf{x}}' < 0$

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

$\underline{\mu} \cdot \underline{\mathbf{x}}' > 0$

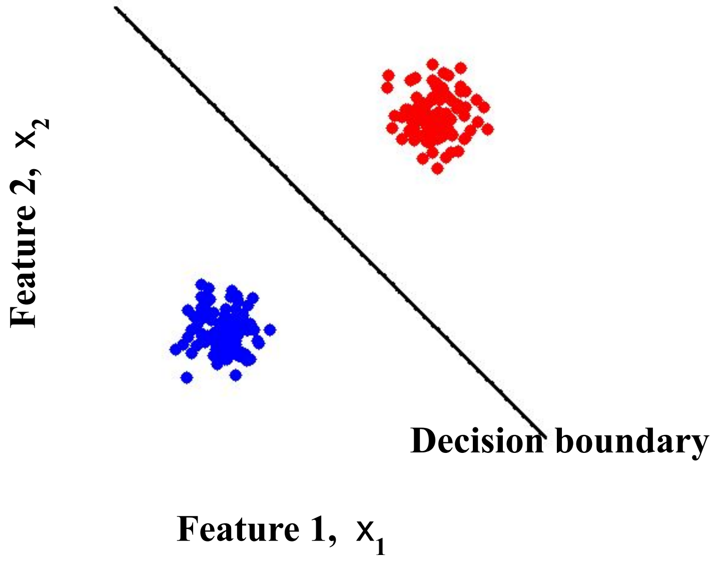
$\Rightarrow x_1 + 2 > x_2$
(decision region +1)

From P. Smyth

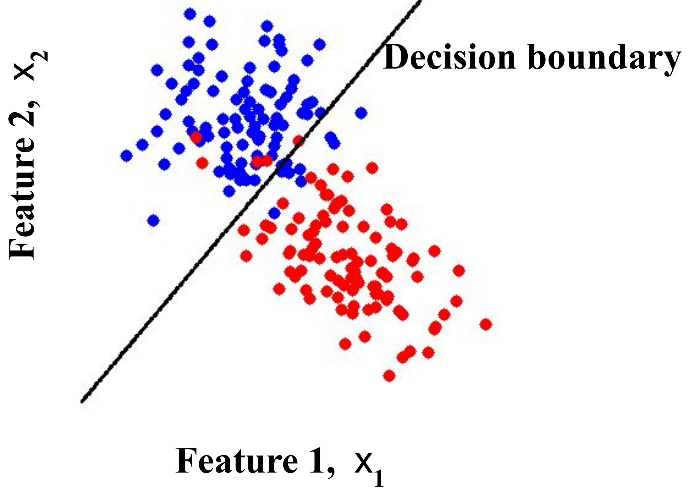
Separability

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

Linearly separable data

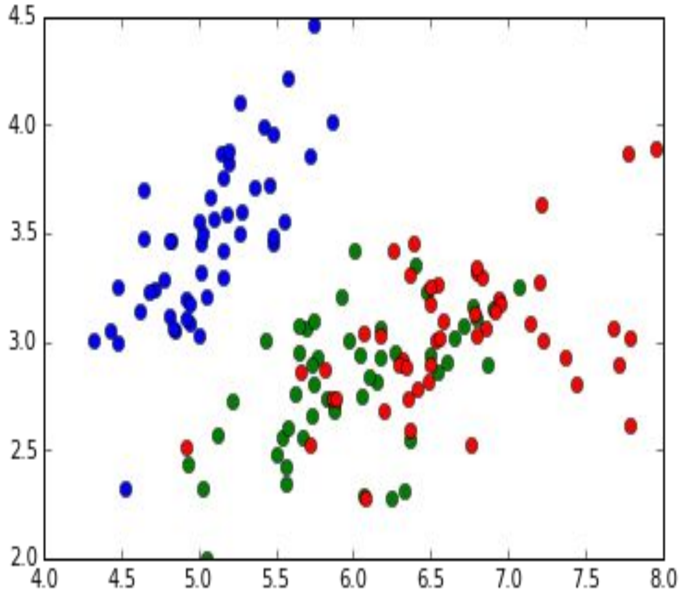


Linearly non-separable data

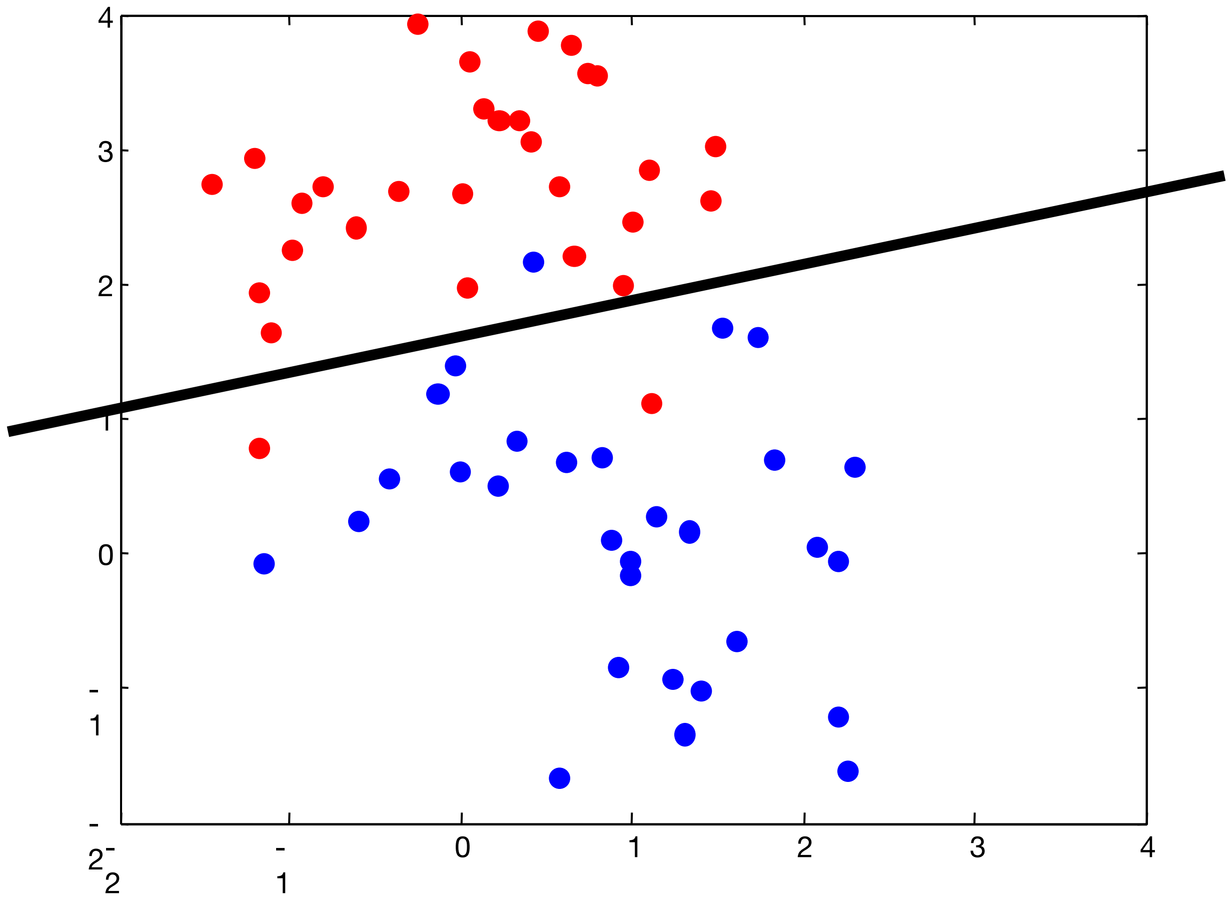


Class overlap

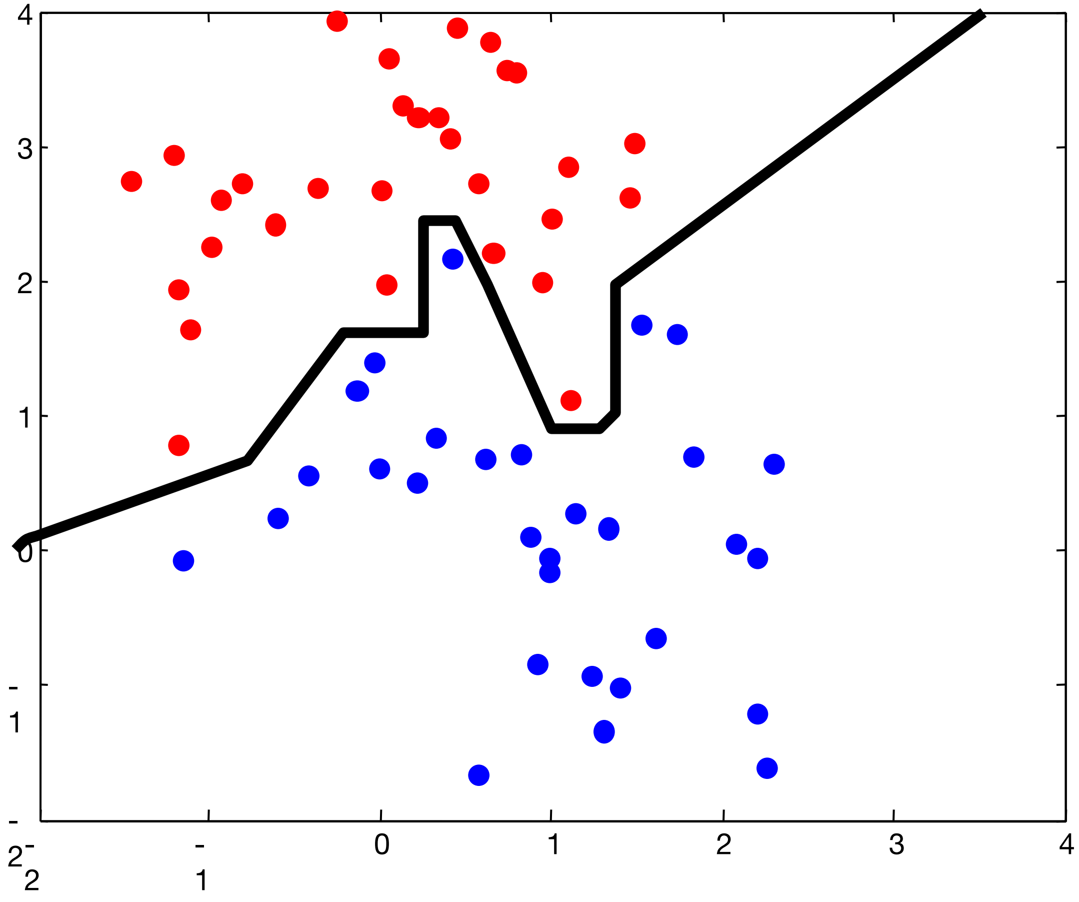
- Classes may not be well-separated
- Same observation values possible under both classes
  - High vs low risk; features {age, income}
  - Benign/malignant cells look similar
  - …
- Common in practice
- May not be able to perfectly distinguish between classes
  - Maybe with more features?
  - Maybe with more complex classifier?
- Otherwise, may have to accept some errors



Another example



Non-linear decision boundary

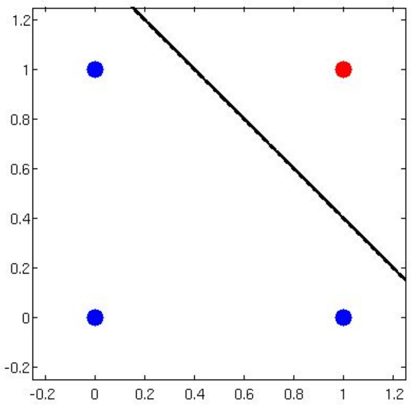


Representational Power of Perceptrons

- 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)

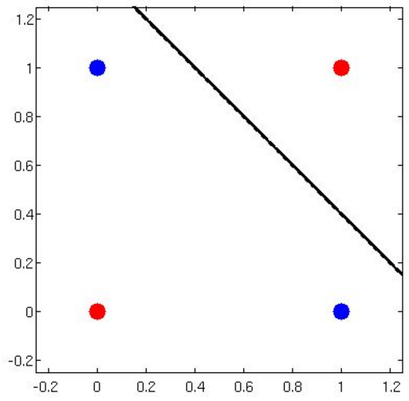
“AND”

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



“XOR”

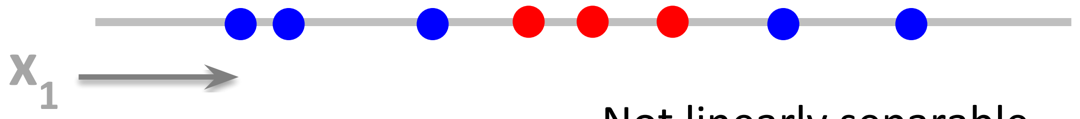
x_1	x_2	y
0	0	1
0	1	-1
1	0	-1
1	1	1



Adding features

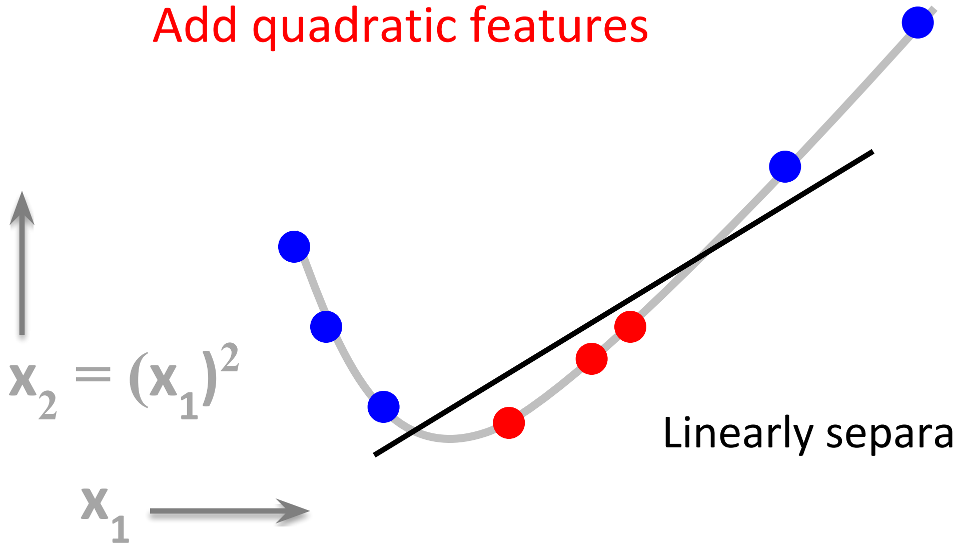
- Linear classifier can't learn some functions

1D example:



Not linearly separable

Add quadratic features

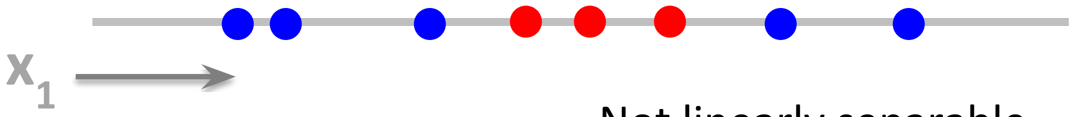


Linearly separable in new features…

Adding features

- Linear classifier can't learn some functions

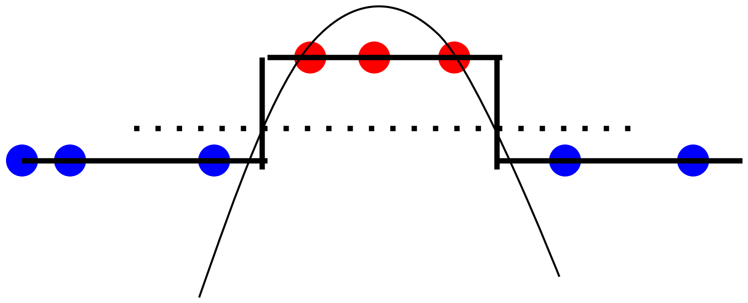
1D example:



Not linearly separable

Quadratic features, visualized in original feature space:

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



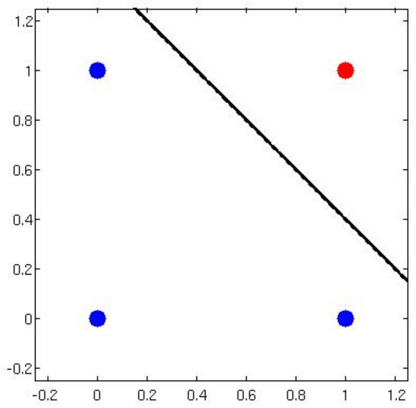
More complex decision boundary: $ax^2+bx+c = 0$

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)

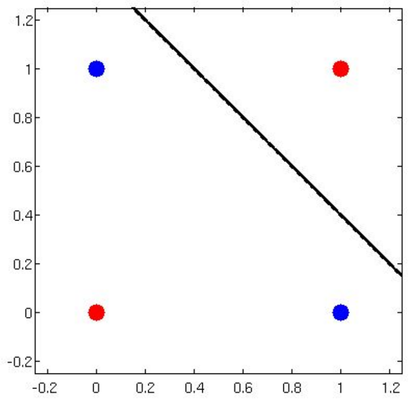
"AND"

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



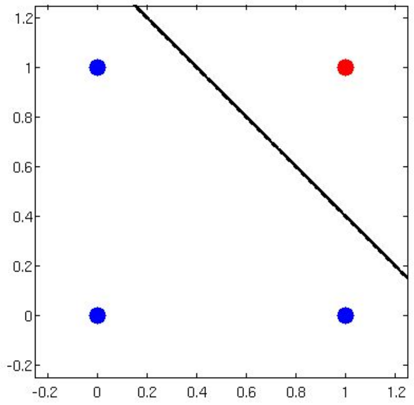
What kinds of functions would we need to learn the data on the right?

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)

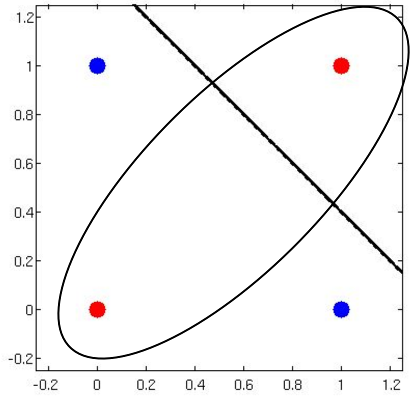
"AND"

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 would we need to learn the data on the right?
Ellipsoidal 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

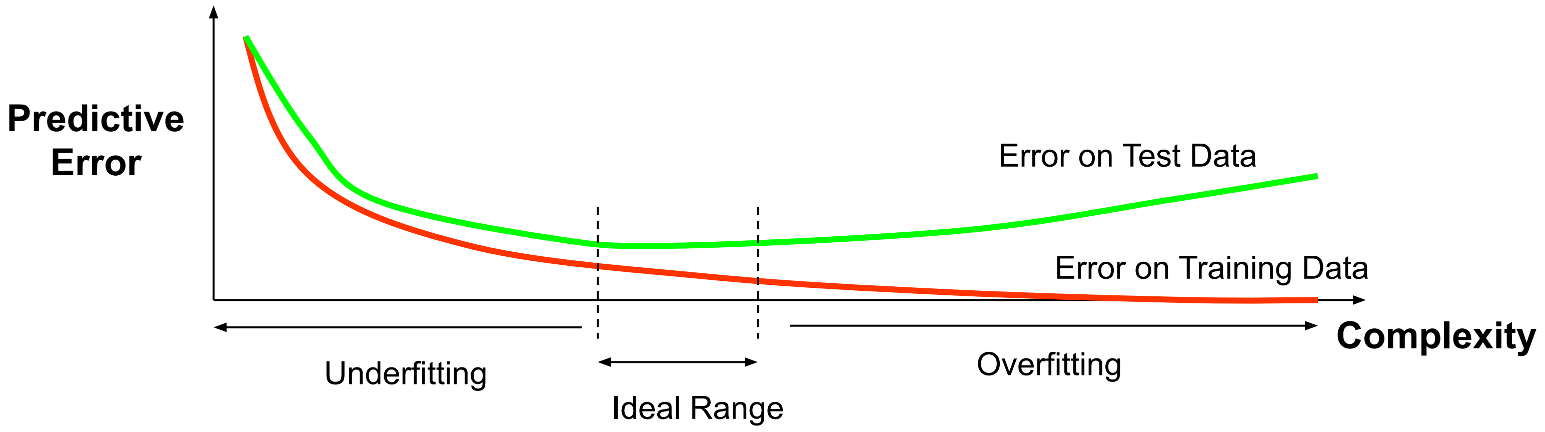
- Ex: discrete features
  - Mushroom surface: {fibrous, grooves, scaly, smooth}
  - Probably not useful to use x = {1, 2, 3, 4}
  - Better: 1-of-K,   x = { [1000], [0100], [0010], [0001] }
  - Introduces more parameters, but a more flexible relationship

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
  - Complex features => Complex decision boundaries
  - Effect on separability
  - Potential for overfitting

Machine Learning

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

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 $J(\boldsymbol{\theta})$:
  - Classifier error rate (for a given set of weights $\boldsymbol{\theta}$ 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

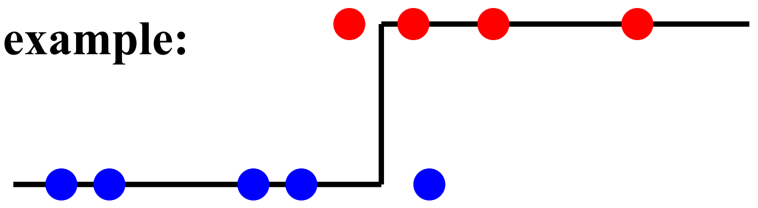
- How should we measure error?
  - Natural measure = "fraction we get wrong" (error rate)

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

```
Yhat = np.sign( X.dot( 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

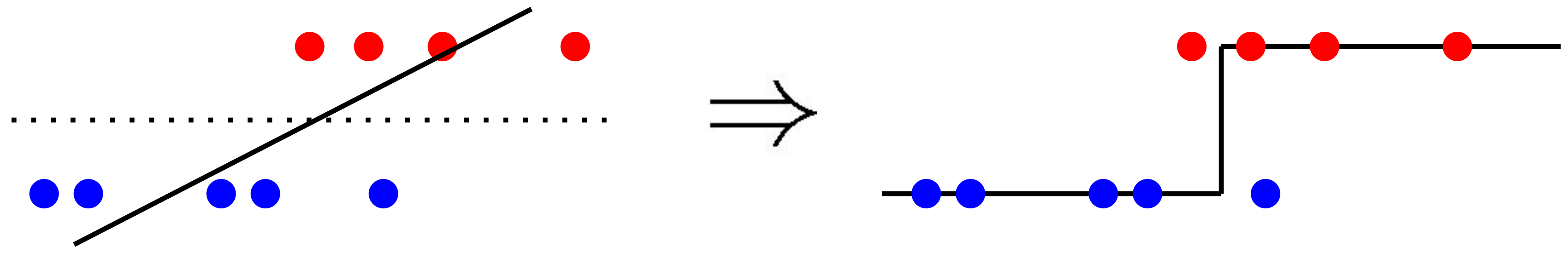
1D example:



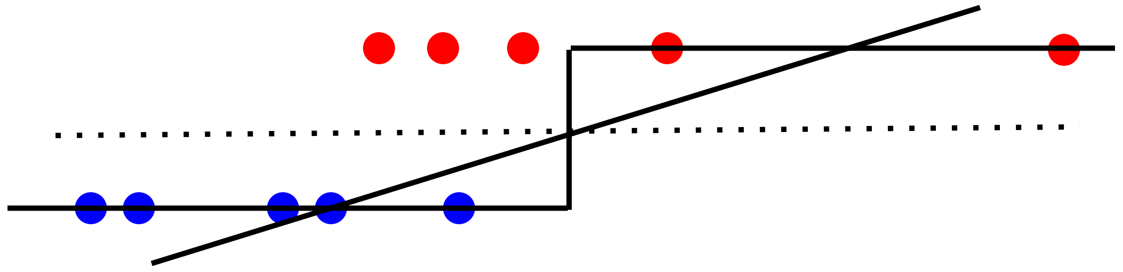
T(f) = -1 if f < 0
T(f) = +1 if f > 0

Linear regression?

- Simple option: set $\boldsymbol{\theta}$ 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 **θ**•x:

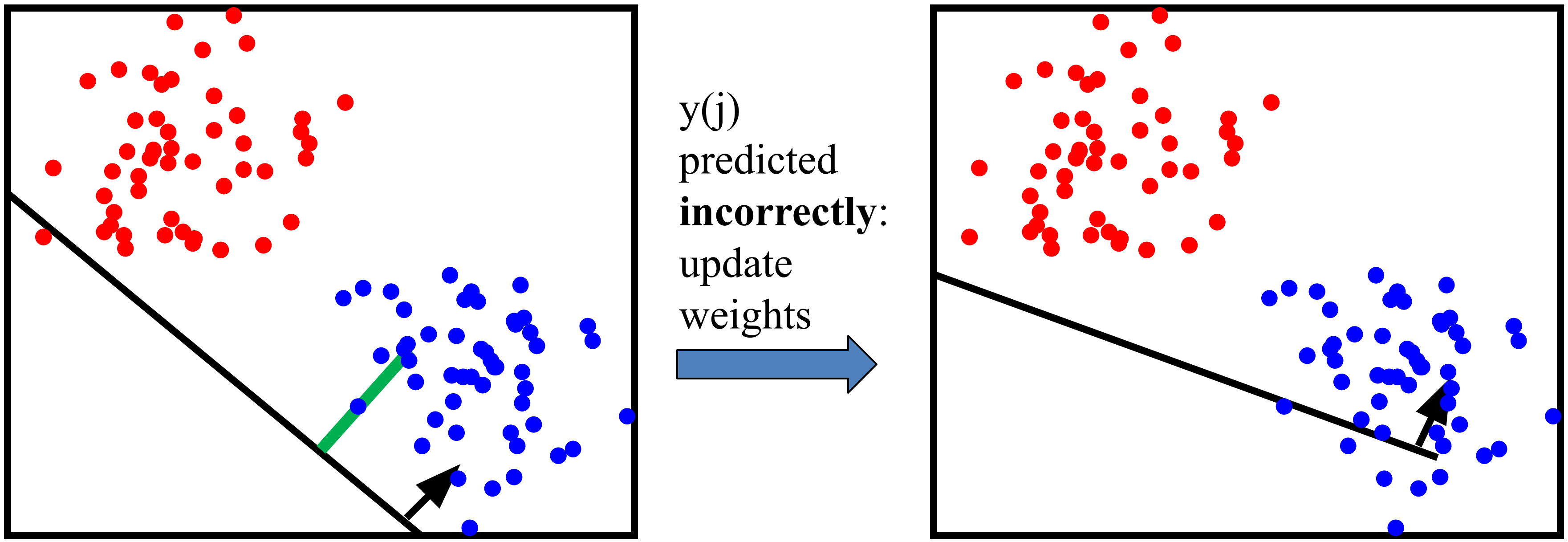
(1) For correct predictions, y(j) - ŷ(j) = 0
(2) For incorrect predictions, y(j) - ŷ(j) = ± 2

"adaptive" linear regression: correct predictions stop contributing

Perceptron algorithm

- Perceptron algorithm: an SGD-like algorithm

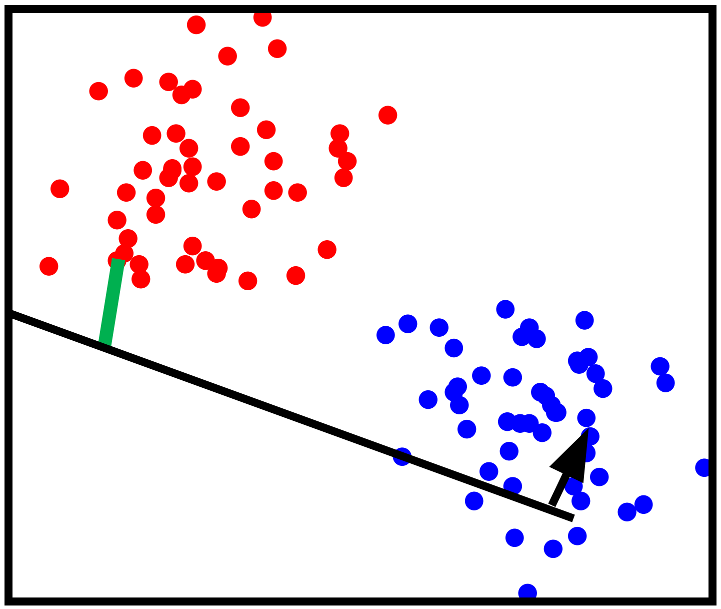
```
while $\neg$ done:
    for each data point $j$:
        $\hat{y}^{(j)} = \text{sign}(\theta \cdot x^{(j)})$    (predict output for point j)
        $\theta \leftarrow \theta + \alpha(y^{(j)} - \hat{y}^{(j)})x^{(j)}$    ("gradient-like" step)
```


y(j) predicted **incorrectly**: update weights

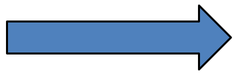
Perceptron algorithm

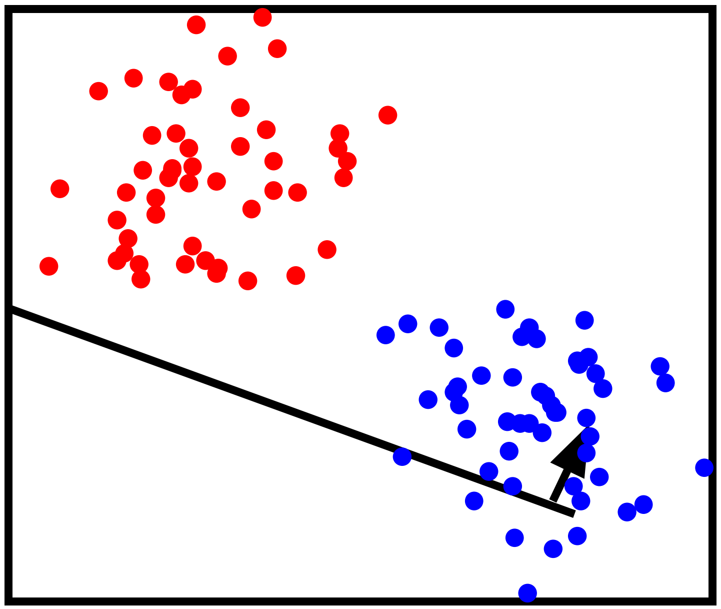
- Perceptron algorithm: an SGD-like algorithm

```
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)
```



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



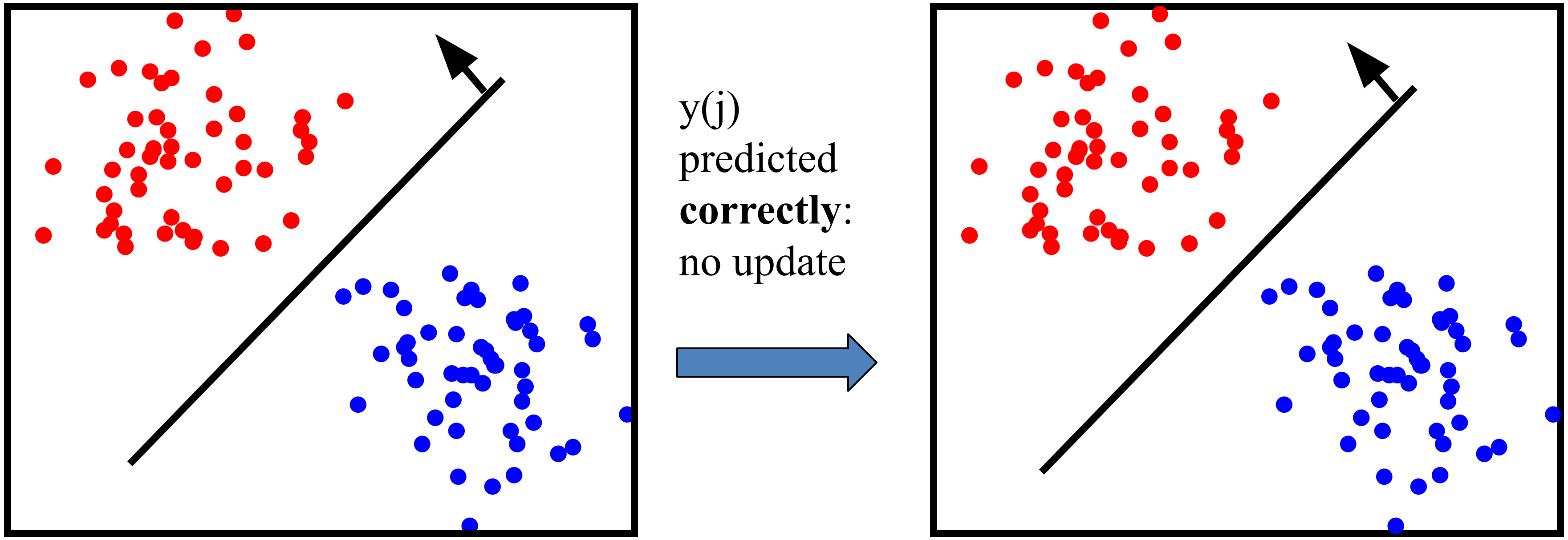


Perceptron algorithm

- Perceptron algorithm: an SGD-like algorithm

```
while ¬ 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)



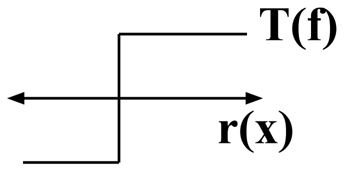
Perceptron MARK 1 Computer

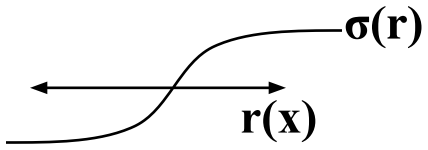
Frank Rosenblatt, late 1950s

Machine Learning

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



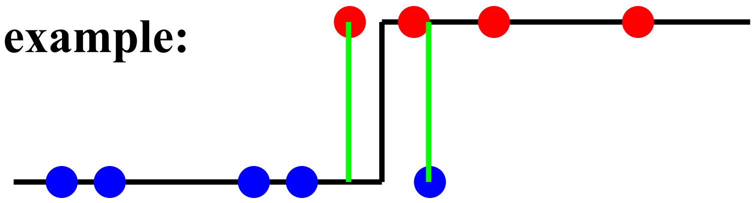


$$J(\underline{\theta}) = \frac{1}{m}\sum_j \left(\sigma(\, r(x^{(j)})\,) - y^{(j)} \right)^2$$

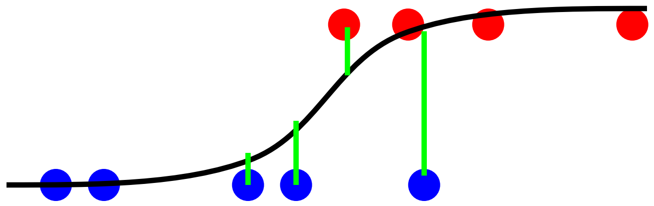
  - Far nall error
  - Nearby the boundary: |f(.)| near 1/2, larger error

Class y = {0, 1} …

1D example:



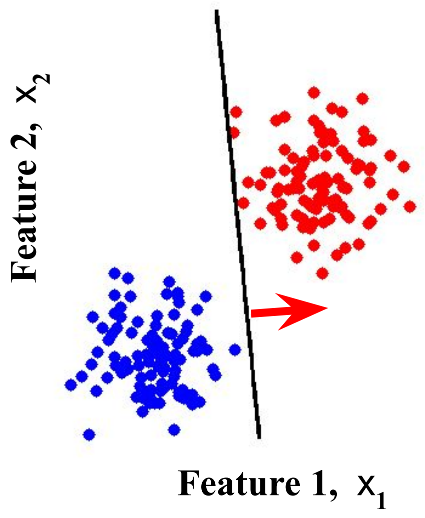
Classification error = 2/9

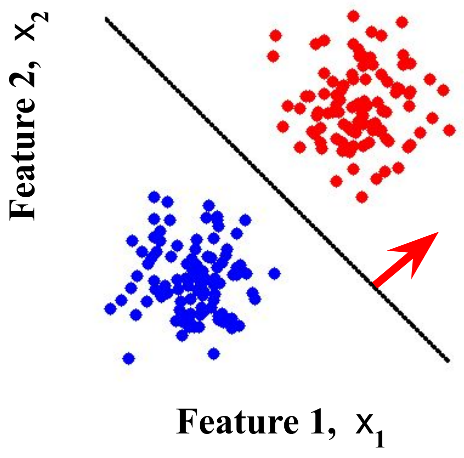


MSE = ($0^2 + 1^2 + .2^2 + .25^2 + .05^2 + \ldots$)/9

Beyond misclassification rate

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





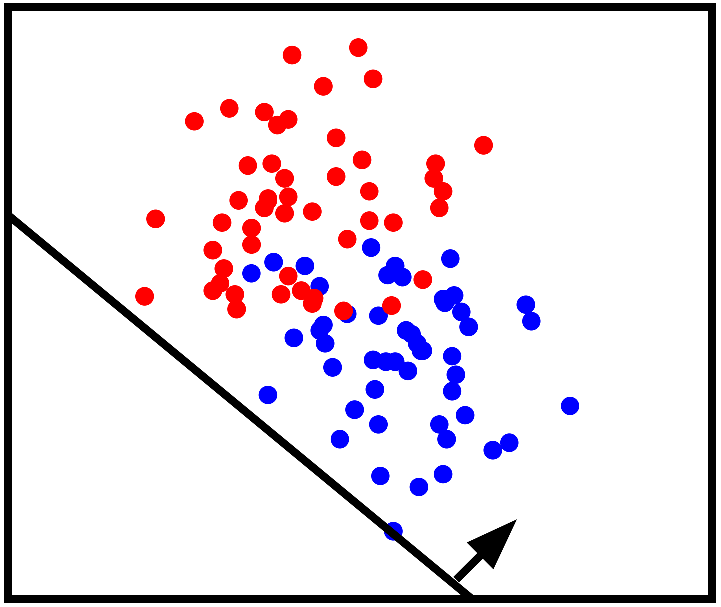
- Side benefit of many "smoothed" error functions
  - Encourages data to be far from the decision boundary
  - See more examples of this principle later...

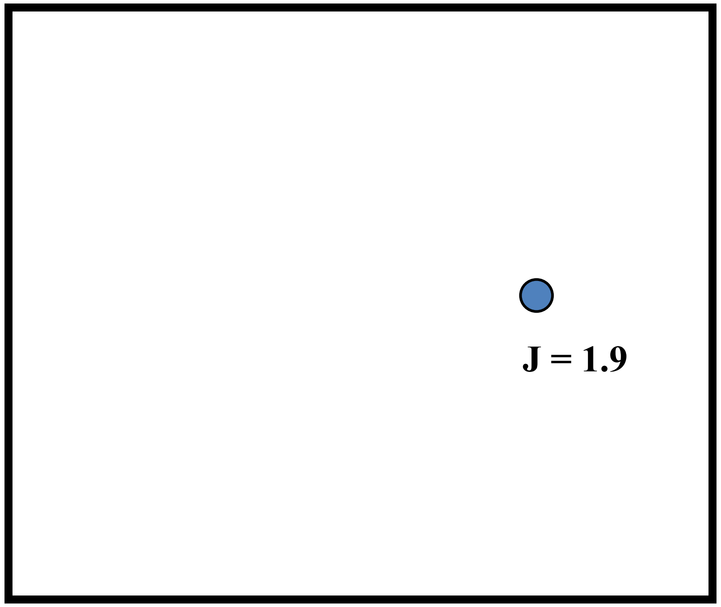
Training the Classifier

- 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



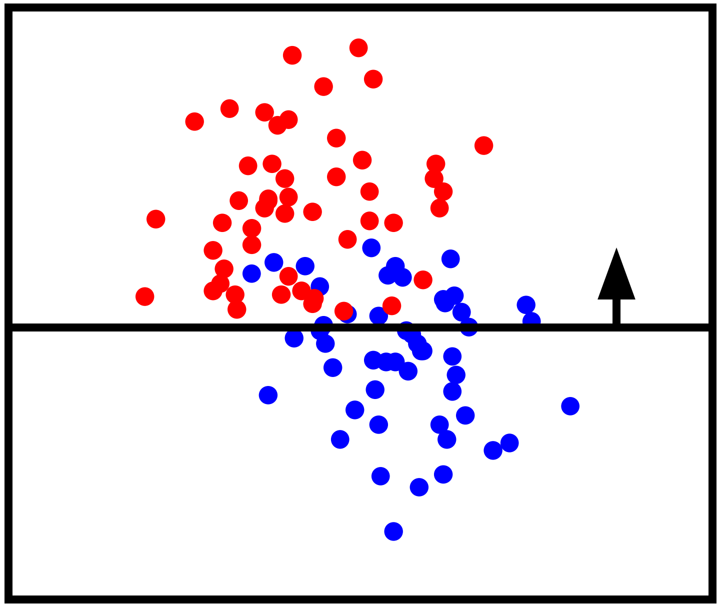


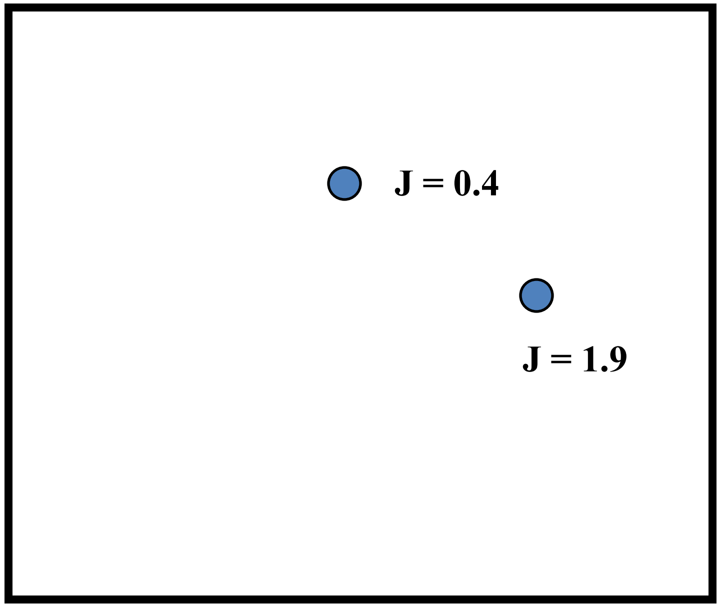
Training the Classifier

- 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



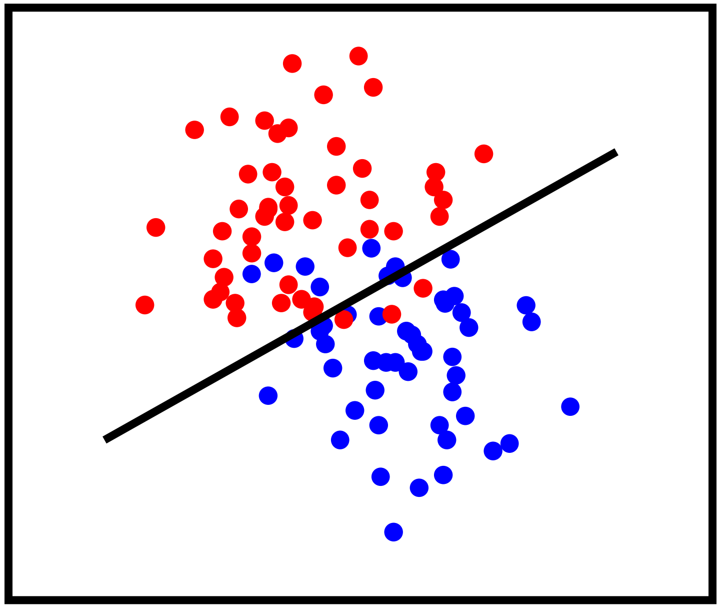


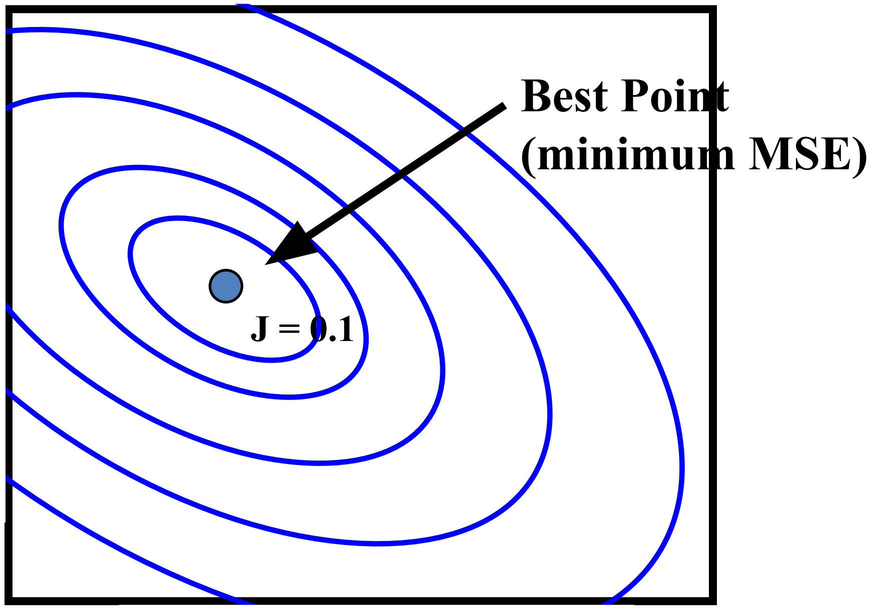
Training the Classifier

- 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



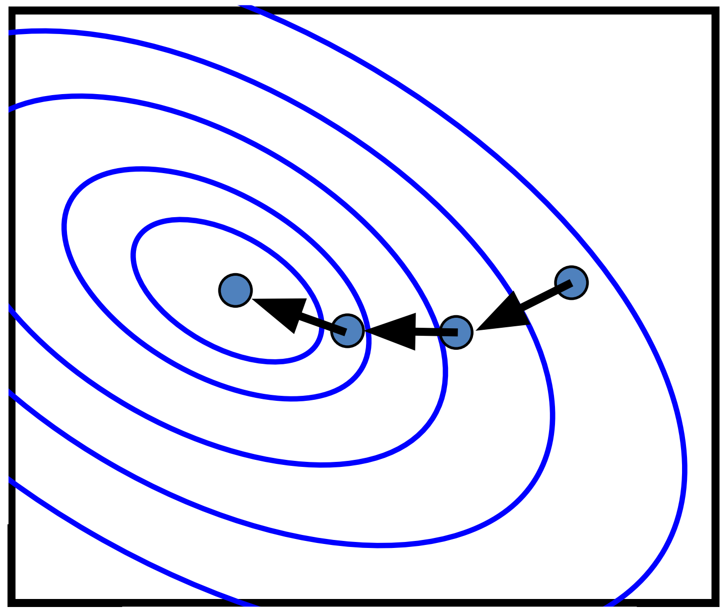


Finding the Best MSE

- 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



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?

$$\frac{\partial J}{\partial a} = \frac{1}{m} \sum_i 2 \underbrace{\boxed{(\sigma(\theta \cdot x^{(i)}) - y^{(i)})}}_{\text{Error between class and prediction}} \underbrace{\boxed{\partial \sigma(\theta \cdot x^{(i)}) \; x_1^{(i)}}}_{\text{Sensitivity of prediction to changes in parameter “a”}}$$

- Similar for parameters b, c [replace x_1 with x_2 or 1 (constant)]

Gradient Equations

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

$$J(\underline{\theta} = [a,b,c]) = \frac{1}{m}\sum_i (\ \sigma(ax_1^{(i)} + bx_2^{(i)} + c) - y^{(i)}\ )^2$$

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

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

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

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

$$h(a) = ax_1^{(i)} + bx_2^{(i)} + c \qquad \Rightarrow \qquad h'(a) = x_1^{(i)}$$

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"

Saturating Functions

- Many possible "saturating" functions

- "Logistic" sigmoid (scaled for range [0,1]) is

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

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

- 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)

- Matlab Implementation:

```
function s = sig(z)
% value of [0,1] sigmoid
   s = 1 ./ (1+exp(-z));

function ds = dsig(x)
% derivative of (scaled) sigmoid
   ds = sig(z) .* (1-sig(z));
```

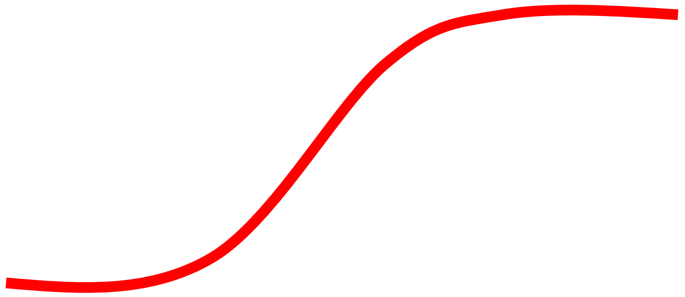
For range [-1 , +1]:

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

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

Predict: threshold z or ρ at zero

Saturating Functions



- Many possible "saturating" functions
- "Logistic" sigmoid (scaled for range [0,1]) is

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

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

- 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 $\boldsymbol{\sigma}$ (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

Class posterior probabilities

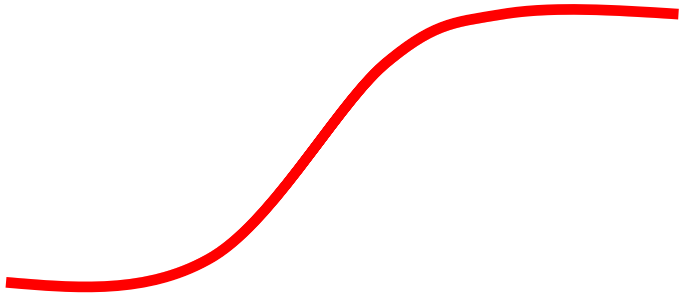
- Useful to also know class *probabilities*
- Some notation
  - p(y=0) , p(y=1) – class *prior* probabilities
    - How likely is each class in general?
  - p(x | y=c) – class conditional probabilities
    - How likely are observations “x” in that class?
  - p(y=c | x) – class posterior probability
    - How likely is class c *given* an observation x?

- We can compute posterior using Bayes’ rule
  - p(y=c | x) = p(x|y=c) p(y=c) / p(x)
- Compute p(x) using sum rule / law of total prob.
  - p(x) = p(x|y=0) p(y=0) + p(x|y=1)p(y=1)

Class posterior probabilities

- Consider comparing two classes
  - p(x | y=0) * p(y=0)   vs   p(x | y=1) * p(y=1)
  - Write probability of each class as
  - p(y=0 | x) = p(y=0, x) / p(x)
  -                          = p(y=0, x) / (p(y=0,x) + p(y=1,x))
  -       = 1 / (1 + exp(-a))   (**)

  - a = log [p(x|y=0) p(y=0) / p(x|y=1) p(y=1)]
  - (**) called the logistic function, or logistic sigmoid.



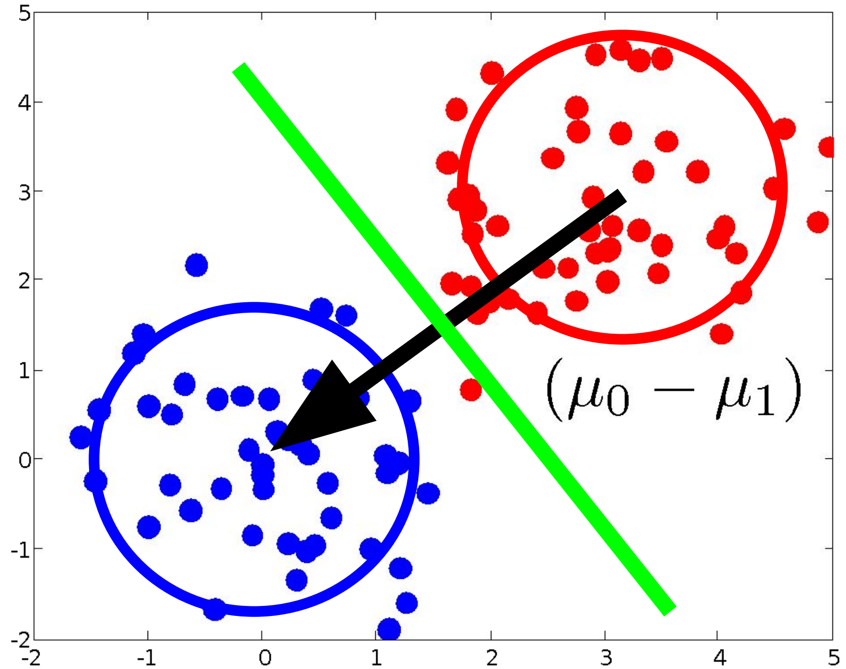
Gaussian models and Logistics

- For Gaussian models with equal covariances

$$\mathcal{N}(\underline{x}\ ;\ \underline{\mu}, \Sigma) = \frac{1}{(2\pi)^{d/2}} |\Sigma|^{-1/2} \exp\left\{ -\frac{1}{2} (\underline{x} - \underline{\mu})^T \Sigma^{-1} (\underline{x} - \underline{\mu}) \right\}$$

$$0 \begin{array}{c} < \\ > \end{array} \log \frac{p(x|y=0)}{p(x|y=1)} \frac{p(y=0)}{p(y=1)} = \quad (\mu_0 - \mu_1)^T \Sigma^{-1} x + constants$$

The probability of each class is given by:
p(y=0 | x) = Logistic(w^T x + b)



Logistic regression

- Interpret σ(**θ**•x) as a probability that y = 1
- Use a negative log-likelihood loss function
  - If y = 1, cost is - log Pr[y=1] = - log σ(**θ**•x)
  - If y = 0, cost is - log Pr[y=0] = - log (1 - σ(**θ**•x))

- Can write this succinctly:

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

Logistic regression

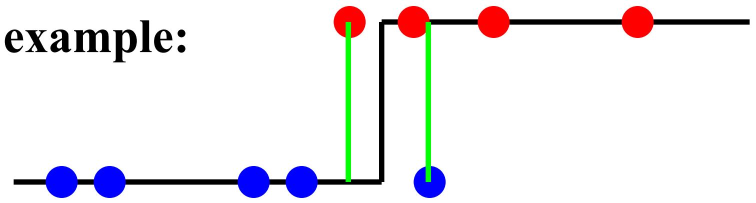
- Interpret σ(**θ**•x) as a probability that y = 1
- Use a negative log-likelihood loss function
  - If y = 1, cost is - log Pr[y=1] = - log σ(**θ**•x)
- If y = 0, cost is - log Pr[y=0] = - log (1 - σ(**θ**•x))

- Can write this succinctly:

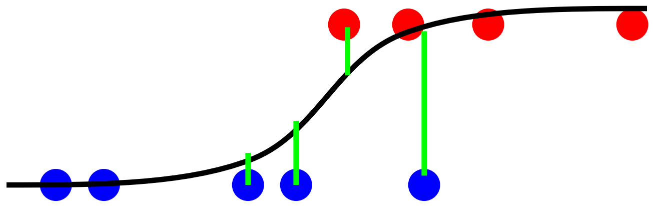
$$J(\underline{\theta}) = -\frac{1}{m}\Big(\sum_i y^{(i)} \log \sigma(\theta \cdot x^{(i)}) + (1-y^{(i)}) \log(1-\sigma(\theta \cdot x^{(i)}))\Big)$$

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

1D example:



Classification error = MSE = 2/9



NLL = - (log(.99) + log(.97) + …)/9

Gradient Equations

- Logistic neg-log likelihood loss:

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

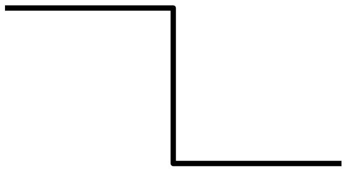
- What's the derivative with respect to one of the parameters?

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

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

Surrogate loss functions

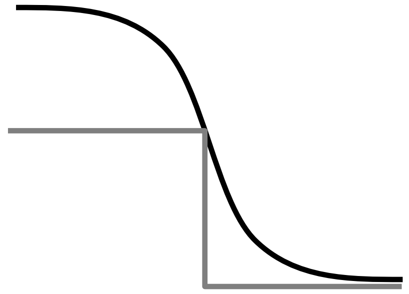
- Replace 0/1 loss with something easier: $\Delta_i(\theta) = \mathbb{1}\left[T(\theta x^{(i)}) \neq y^{(i)}\right]$



0 / 1 Loss

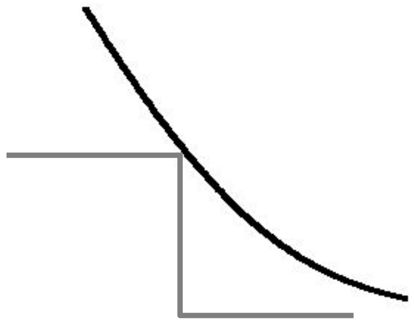
- Logistic MSE

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



- Logistic Neg Log Likelihood

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



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 classifier from data
  - Reduces to an optimization problem
  - Perceptron algorithm
  - For MSE or Logistic NLL, we can do gradient descent
  - Gradient equations & update rules

Machine Learning

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

Multi-class linear models

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

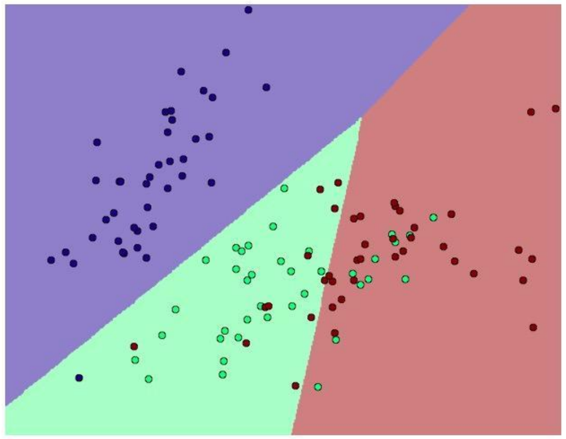
$$\theta = \begin{bmatrix} \theta_{00} & \dots & \theta_{0n} \\ \vdots & \ddots & \vdots \\ \theta_{C0} & \dots & \theta_{Cn} \end{bmatrix}$$

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

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

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

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



Multiclass linear models

- More generally, can define a generic linear classifier by

$$f(x\,;\theta) = \arg\max_y \; \theta \cdot \Phi(x,y)$$

- Example: y ∈ {-1, +1}

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

$$f(x\,;\theta) = \begin{cases} +1 & \theta \cdot [1\;\, x\;\, x^2 \ldots] > -\theta \cdot [1\;\, x\;\, x^2 \ldots] \\ -1 & \text{o.w.} \end{cases}$$

(Standard perceptron rule)

Multiclass linear models

- More generally, can define a generic linear classifier by

$$f(x\,;\theta) = \arg\max_y \;\theta \cdot \Phi(x,y)$$

- Example: y ∈ {0,1,2,…}

$$\begin{aligned}\Phi(x,y) &= [\;\mathbb{1}[y=0][1\;x\;x^2\;\;\ldots]\;\;\mathbb{1}[y=1][1\;x\;x^2\ldots]\ldots]\\ \theta &= [\;\;[\theta_{00}\;\;\theta_{01}\;\;\theta_{02}\ldots]\;\;\;\;[\theta_{10}\;\;\theta_{11}\;\;\theta_{12}\ldots]\;\;\;\ldots]\end{aligned}$$

(parameters for each class c)

$$f(x\,;\theta) = \arg\max_c \;\theta_{c\cdot} \cdot [1\;x\;x^2\;\;\ldots]$$

(predict class with largest linear response)

Multiclass perceptron algorithm

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

```
While (~done)
    For each data point j:
        $f^{(j)} = \arg\max ( \theta_c * \underline{x}^{(j)} )$        : predict output for data point j
        $\theta_f = \theta_f - \alpha \underline{x}^{(j)}$        : decrease response of class $f^{(j)}$ to $x^{(j)}$
        $\theta_y = \theta_y + \alpha \underline{x}^{(j)}$        : increase response of true class $y^{(j)}$
```

More general form update:

$$f(x\,;\theta) = \arg\max_y \ \theta \cdot \Phi(x, y)$$

$$\theta \leftarrow \theta + \alpha\big(\Phi(x, y) - \Phi(x, f(x))\big)$$

Multilogit regression

- Define the probability of each class:

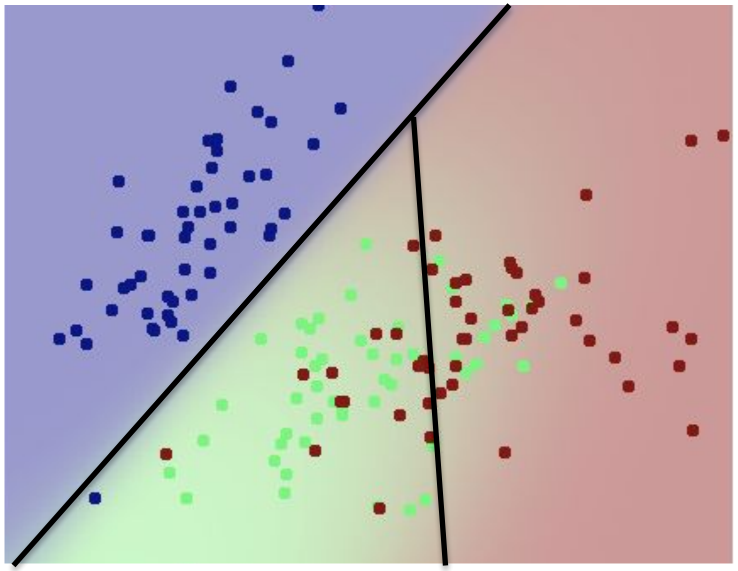
$$p(Y = y | X = x) = \frac{\exp(\theta_y \cdot x^T)}{\sum_c \exp(\theta_c \cdot x^T)}$$

(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



Machine Learning

Linear Classification with Perceptrons

Perceptron Learning

Gradient-Based Classifier Learning

Multi-Class Classifiction

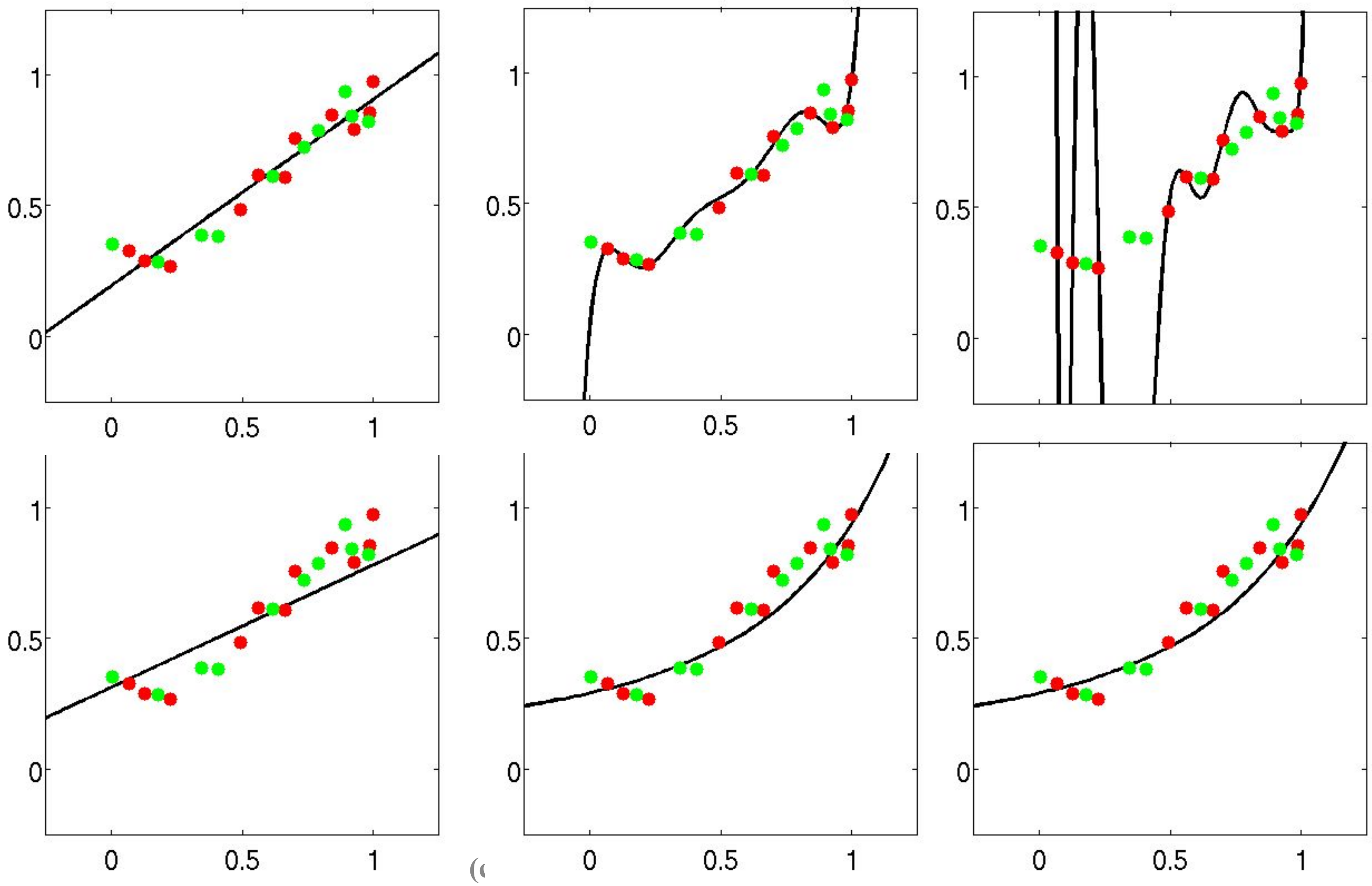
Regularization for Linear Classification

Regularization

- Reminder: Regularization for linear regression

Alpha =0
(Unregularized)

Alpha =1



Regularized logistic regression

- Intepret ¾($\underline{\mu}$ 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 ¾($\underline{\mu}$ x^T)
  - If y = 0, cost is - log Pr[y=0] = - log (1 - ¾($\underline{\mu}$ x^T))

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

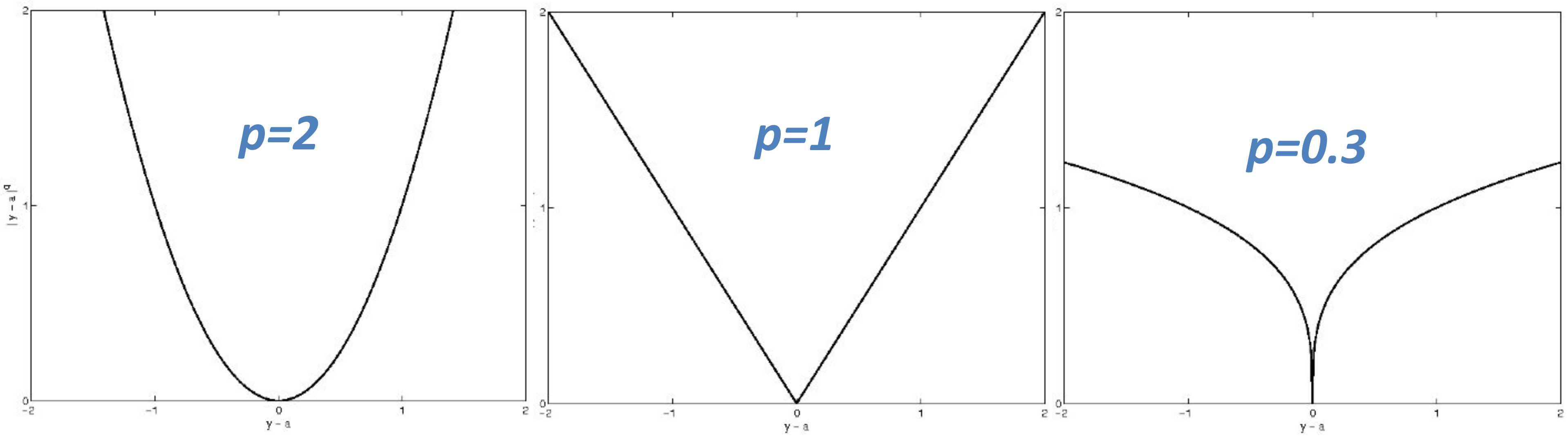
$$J(\underline{\theta}) = -\frac{1}{m}\left(\sum_i \underbrace{y^{(i)} \log \sigma(\theta \cdot x^{(i)})}_{\text{Nonzero only if y=1}} + \underbrace{(1-y^{(i)}) \log(1-\sigma(\theta \cdot x^{(i)}))}_{\text{Nonzero only if y=0}}\right)$$

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

Different regularization functions

- In general, for the L_p regularizer:

$$\left(\sum_i |\theta_i|^p \right)^{\frac{1}{p}} = ||\theta||_p$$

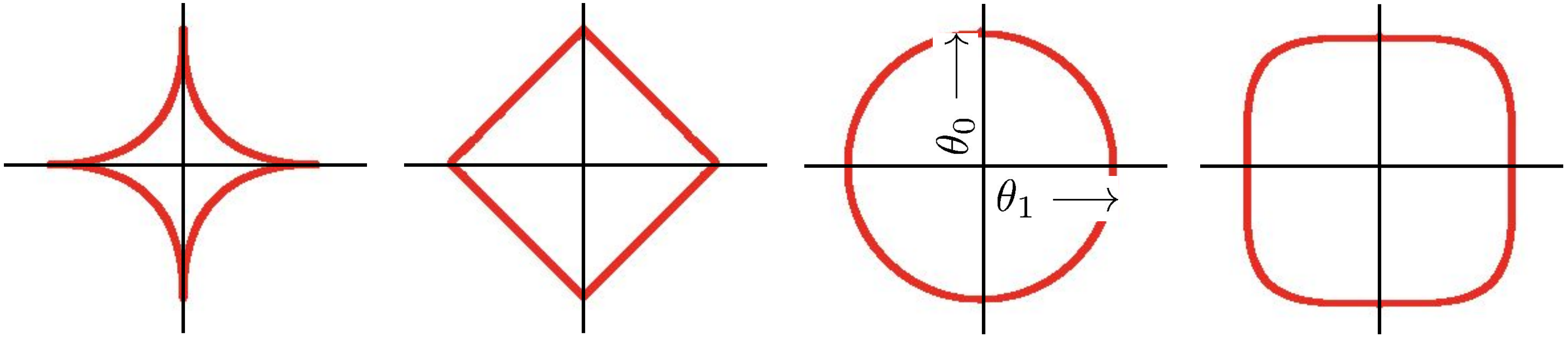


Different regularization functions

- 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
Lasso

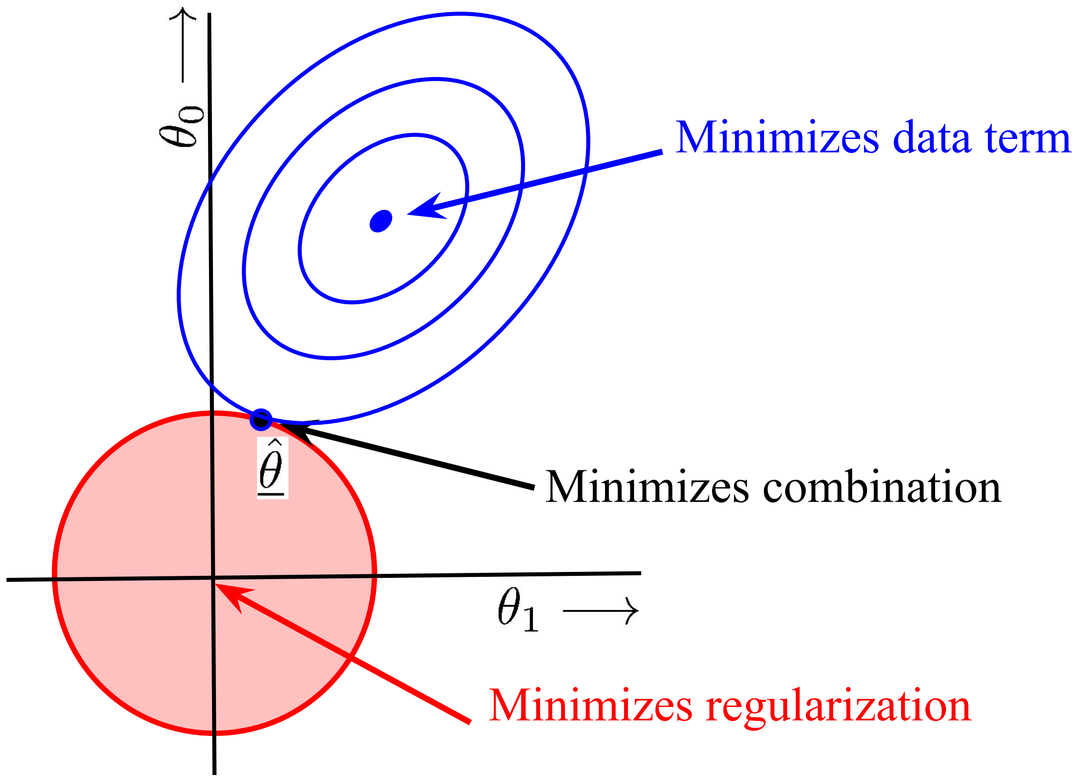
p=2
Quadratic

p=4

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

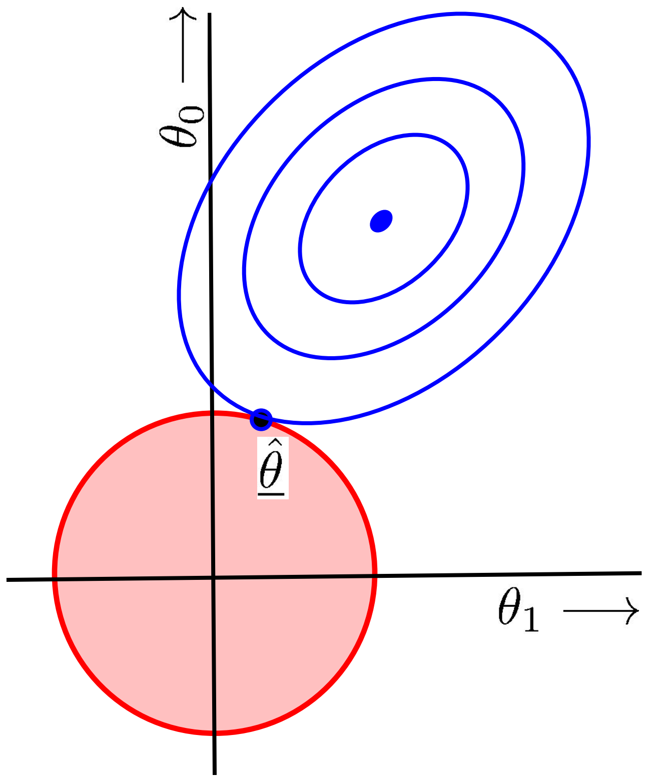
Regularization: L_2 vs L_1

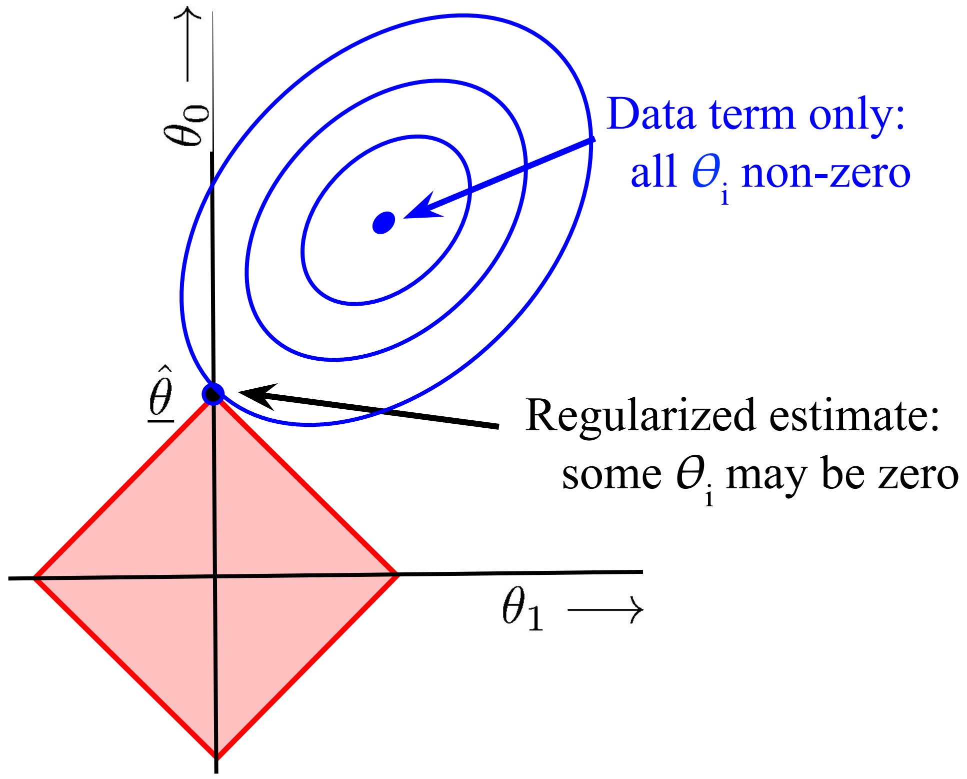
- Estimate balances data term & regularization term


$\theta_0 \longrightarrow$
Minimizes data term
$\hat{\underline{\theta}}$
Minimizes combination
$\theta_1 \longrightarrow$
Minimizes regularization

Regularization: L_2 vs L_1

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

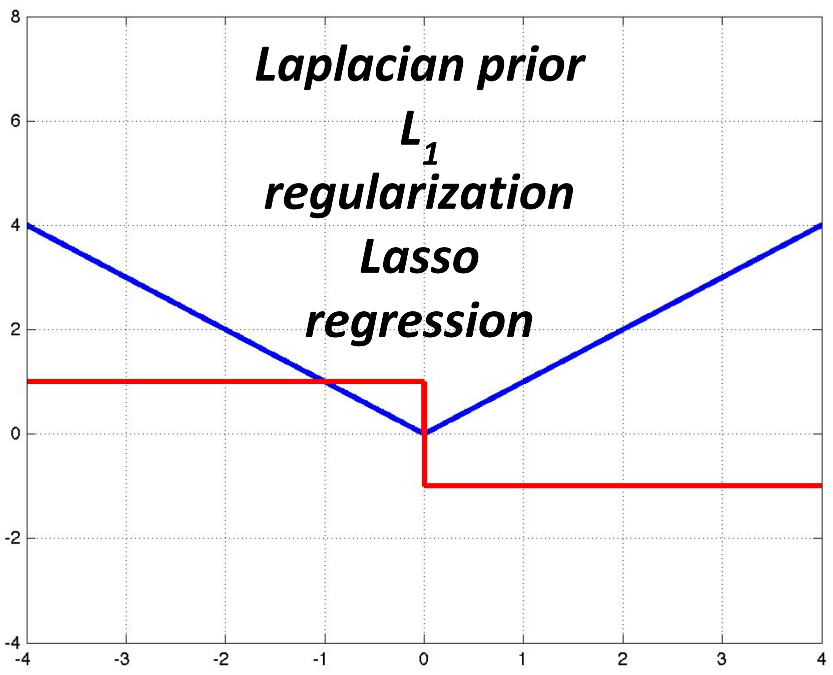


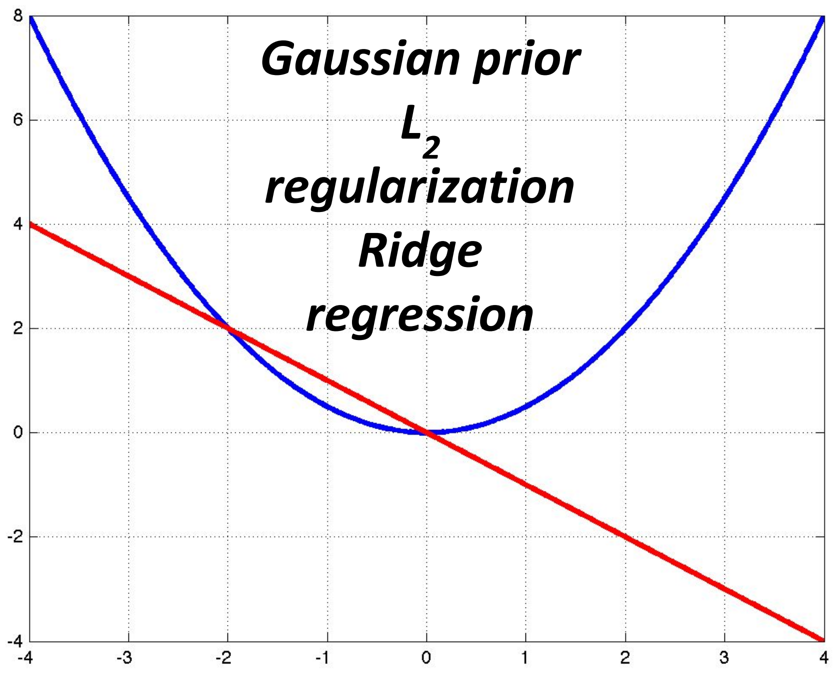


$\theta_0 \longrightarrow$
Data term only: all θ_i non-zero
$\hat{\theta}$
Regularized estimate: some θ_i may be zero
$\theta_1 \longrightarrow$

Gradient-Based Optimization

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





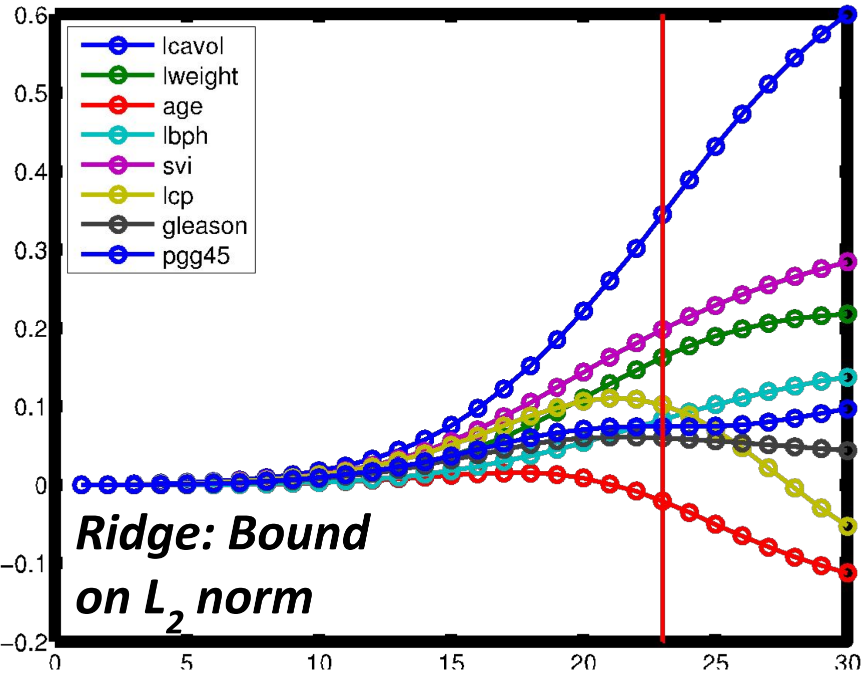
Objective Function: $f(\theta_i) = |\theta_i|^p$

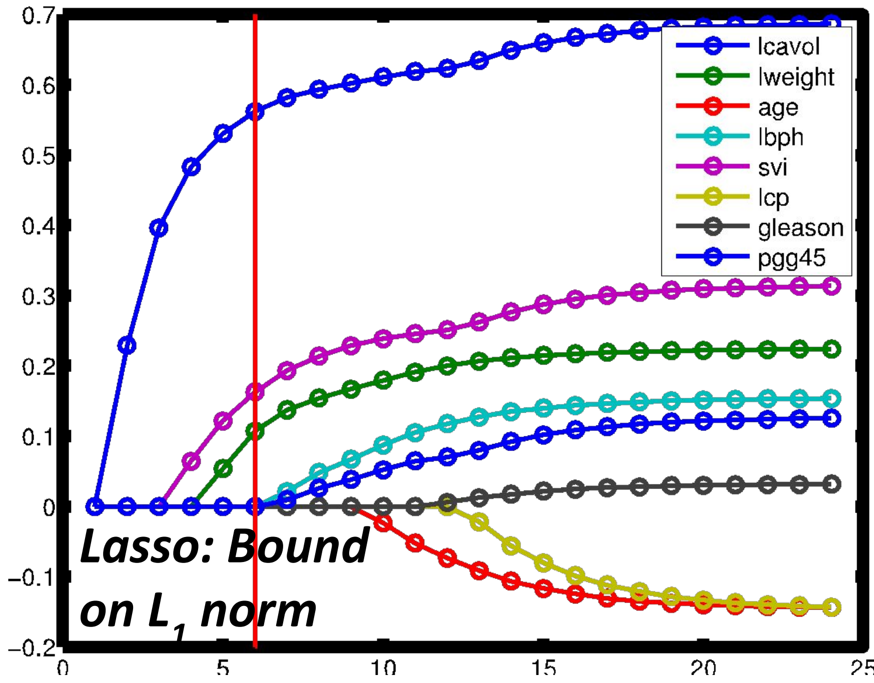
Negative Gradient: $-f'(\theta_i)$

(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

Acknowledgement

Based on slides by Alex Ihler