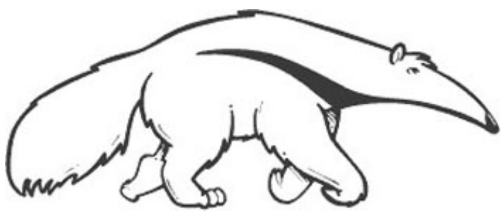
Machine Learning and Data Mining

Linear classification

Prof. Alexander Ihler

Fall 2012

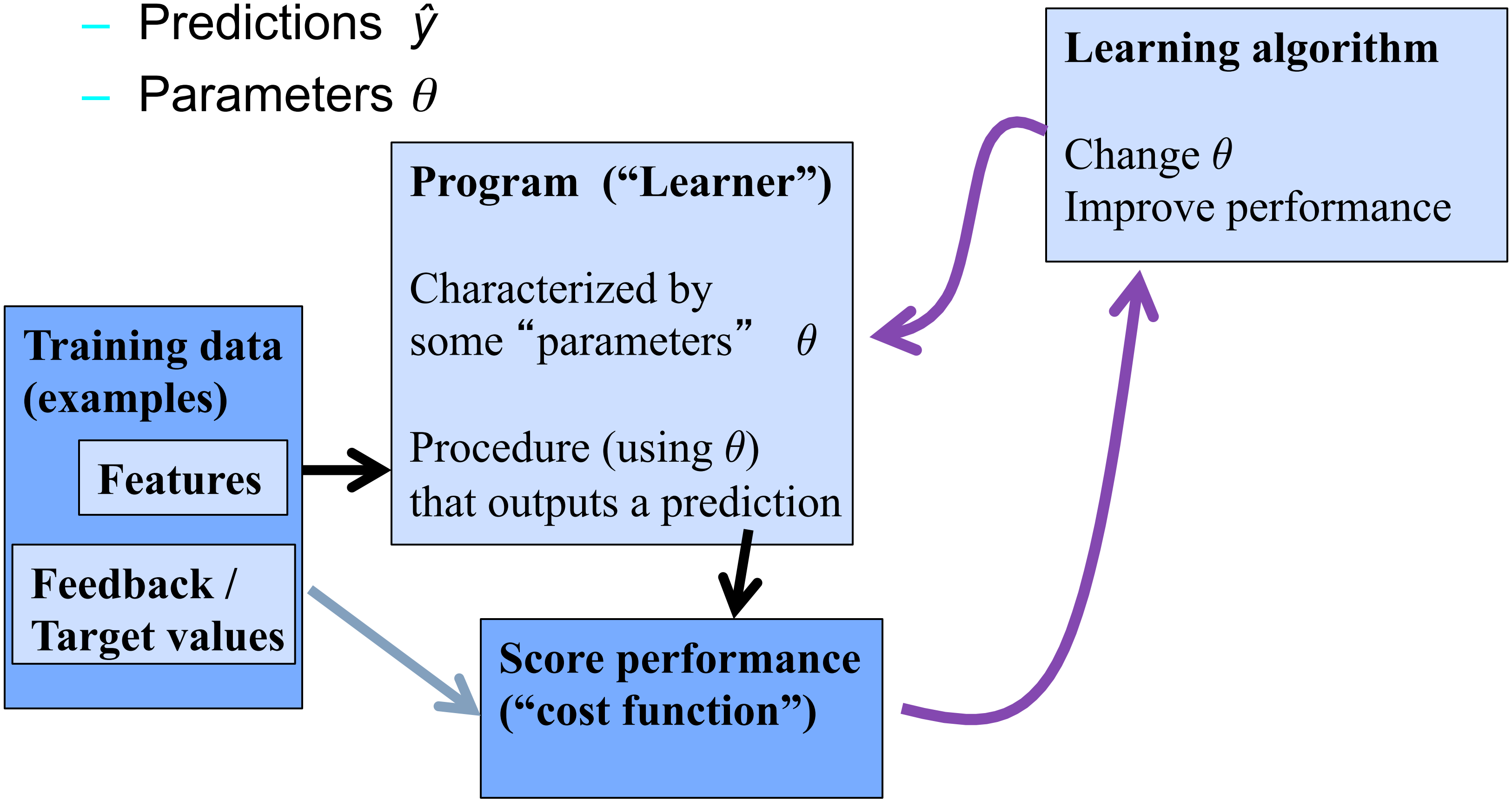


BREN:ICS INFORMATION AND COMPUTER SCIENCES

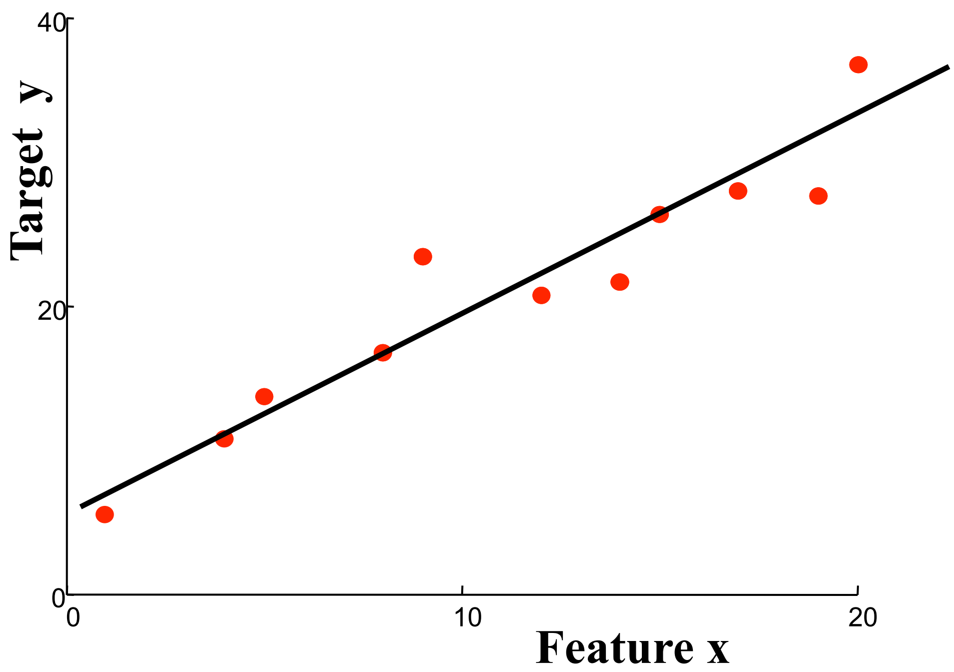
UNIVERSITY of CALIFORNIA IRVINE

Supervised learning

- Notation
  - Features x
  - Targets y
  - Predictions $\hat{y}$
  - Parameters θ



Linear regression



"Predictor":
Evaluate line:

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

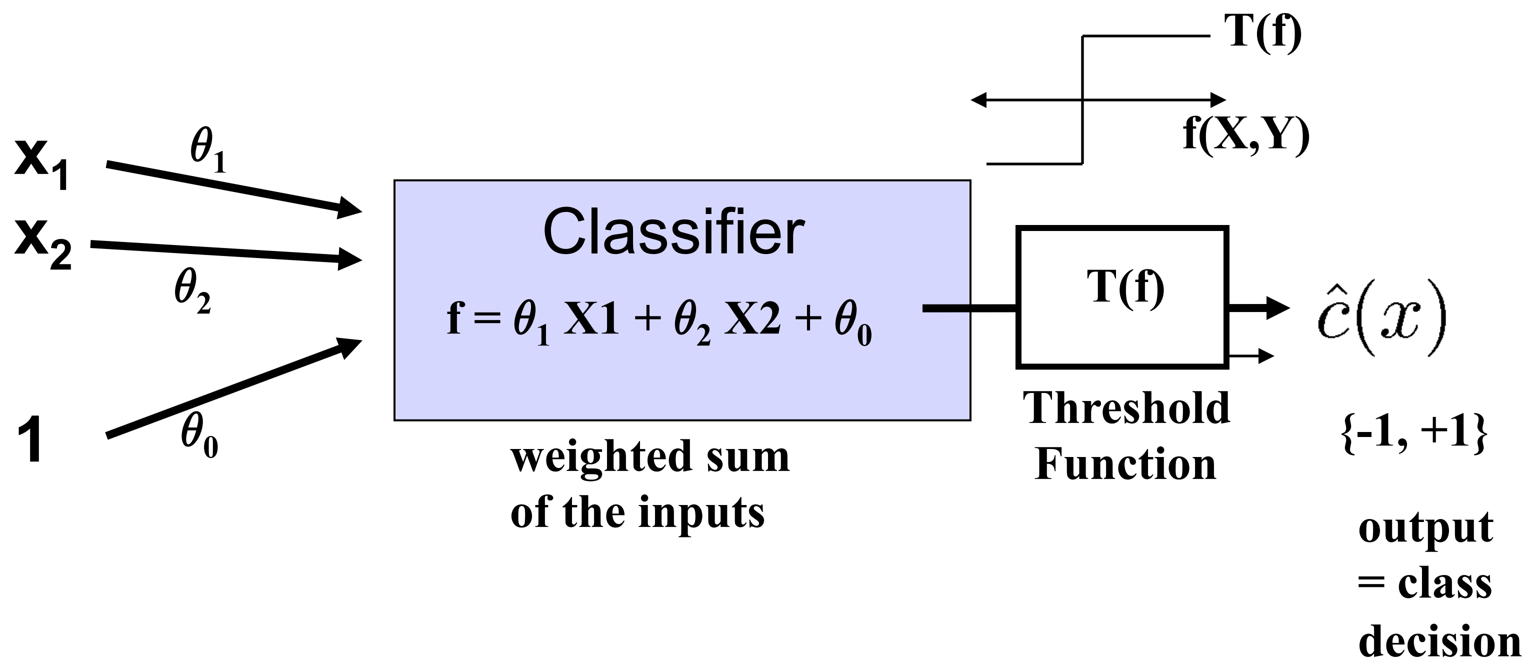
return r

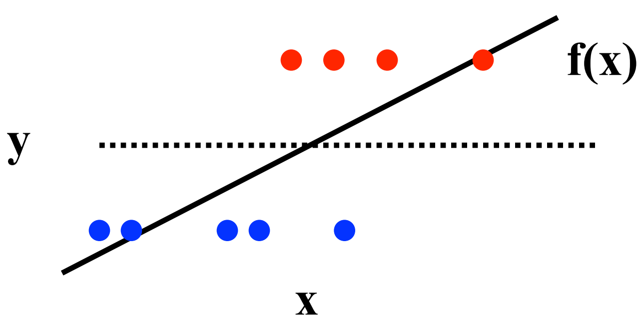
- Contrast with classification
  - Classify: predict discrete-valued target y

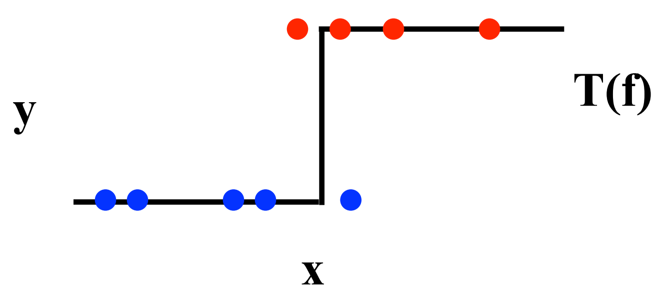
Linear Classifiers: Parametric Form

- Let: feature 1 = "X1", feature 2 = "X2"
- Linear classifier is a linear function of features X1 and X2, i.e.,
  - f(X1,X2) = a*X1 + b*X2 + c
  - Coefficients [a,b,c] are the "weights" / "parameters" of the classifier
  - In general, d + 1 coefficients (one for each feature, plus offset)
- Output of the classifier is a class, {-1, 1} :
  - T(f) = -1 if f < 0, T(f) = +1 if f > 0
- Decision boundary
  - Transition from one class decision to another at f(X1,X2) = 0
  - Decision boundary is: a*X1 + b*X2 + c =0 – Linear
- In higher dimensions, equation is a "hyperplane"

Perceptron Classifier (2 features)







Perceptrons

- Perceptron = a linear classifier
  - The w's are the weights (denoted as a, b,c, earlier)
    - real-valued constants (can be positive or negative)
  - Define an additional constant input "1" (allows an intercept in decision boundary)

- A perceptron calculates 2 quantities:
  - 1. A weighted sum of the input features
  - 2. This sum is then thresholded by the T function

- A simple artificial model of human neurons
    - weights = "synapses"
    - threshold = "neuron firing"

Notation

- Inputs:
  - $x_0, x_1, x_2, \ldots\ldots\ldots\ldots, x_d,$
  - $x_1, x_2, \ldots\ldots\ldots\ldots, x_{d-1}, x_d$ are the values of the d features
  - $x_0 = 1$ (a constant input)
  - $\underline{\mathbf{x}} = (x_0, x_1, x_2, \ldots\ldots\ldots\ldots, x_d)$
- Weights (parameters):
  - $\theta_0, \theta_1, \theta_2, \ldots\ldots\ldots\ldots, \theta_d,$
  - we have d+1 weights
  - one for each feature + one for the constant
  - $\underline{\theta} = (\theta_0, \theta_1, \theta_2, \ldots\ldots\ldots\ldots, \theta_d)$

Perceptron Operation

- Equations of operation:

$$o[x_1, x_2, \ldots, x_{d-1}, x_d] \begin{cases} = 1 & (\text{if } \theta_1 x_1 + \ldots \theta_d x_d + \theta_0 > 0) \\ = -1 & (\text{otherwise}) \end{cases}$$

Note that

$\theta = (\theta_0, \ldots.. \theta_d)$, the "weight vector" (row vector, 1 x d+1)

and $\underline{x} = (x_0, \ldots\ldots x_d)$, the "feature vector" (row vector, 1 x d+1)

$$\Rightarrow \quad \theta_0 x_0 + \theta_1 x_1 + \ldots \theta_d x_d = \underline{\theta} \,.\, \underline{\mathbf{x}}'$$

and $\underline{\theta} \,.\, \underline{\mathbf{x}}'$ is the vector inner product (θ * x' or "sum(θ .* x)" in MATLAB)

Perceptron Decision Boundary

- Equations of operation (in vector form):

$$o(x_1, x_2, \ldots, x_d, x_{d+1}) = \begin{cases} 1 & (\text{if } \underline{\theta} \cdot \underline{\mathbf{x}}' > 0) \\ -1 & (\text{otherwise}) \end{cases}$$

The perceptron represents a hyperplane decision surface in d-dimensional space

e.g., a line in 2d, a plane in 3d, etc

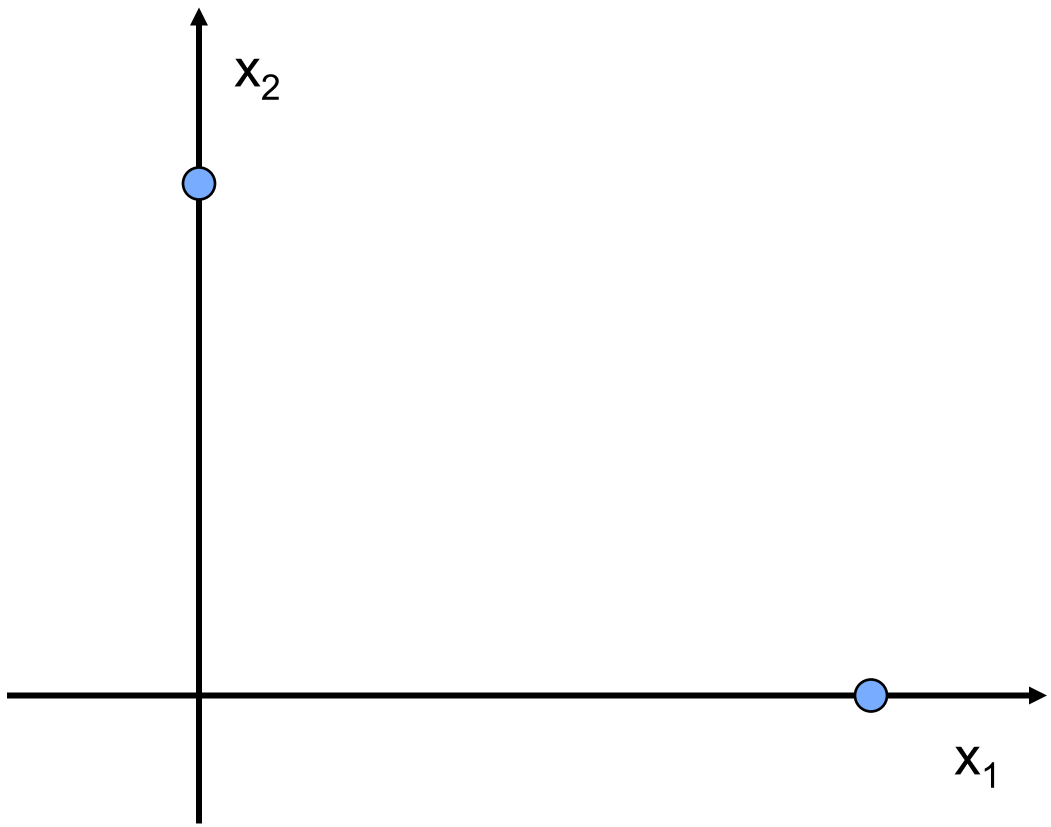
The equation of the hyperplane is

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

This is the equation for points in x-space that are on the boundary

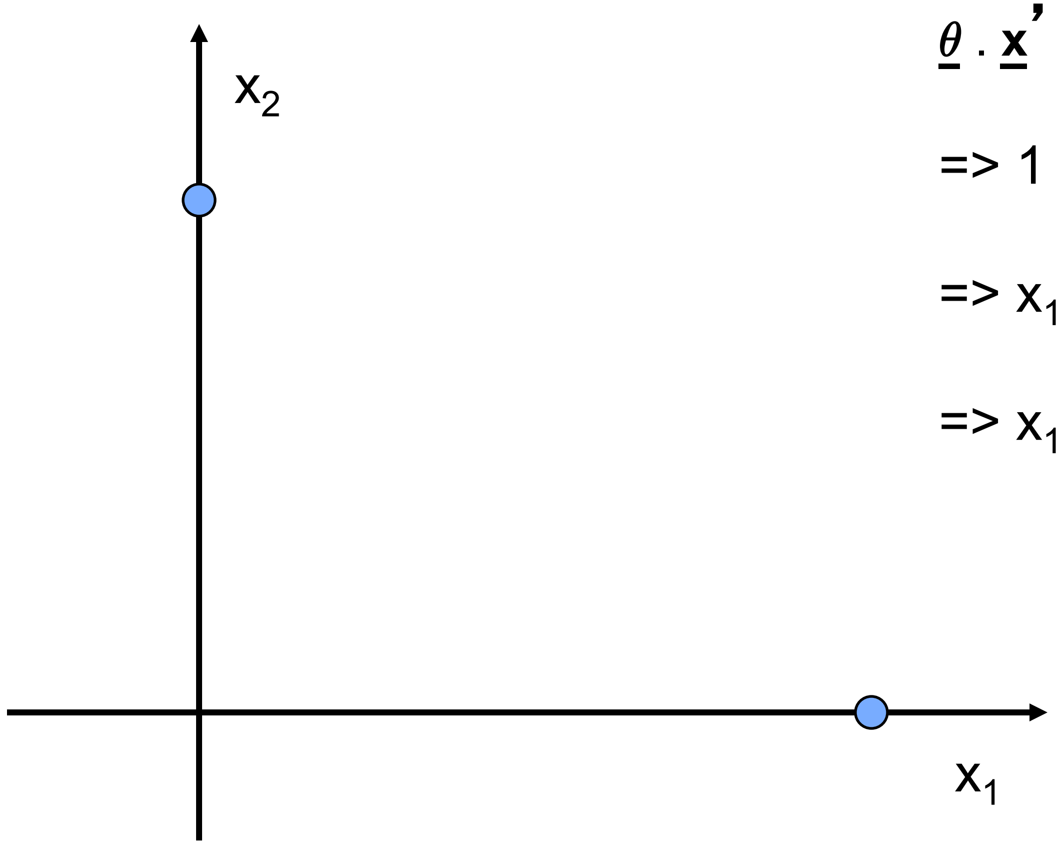
Example, Linear Decision Boundary

$$\underline{\theta} = (\theta_1, \theta_2, \theta_0)$$
$$= (1, -1, 0)$$



Example, Linear Decision Boundary

$\underline{\theta} = (\theta_1, \theta_2, \theta_0)$
$= (1, -1, 0)$



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

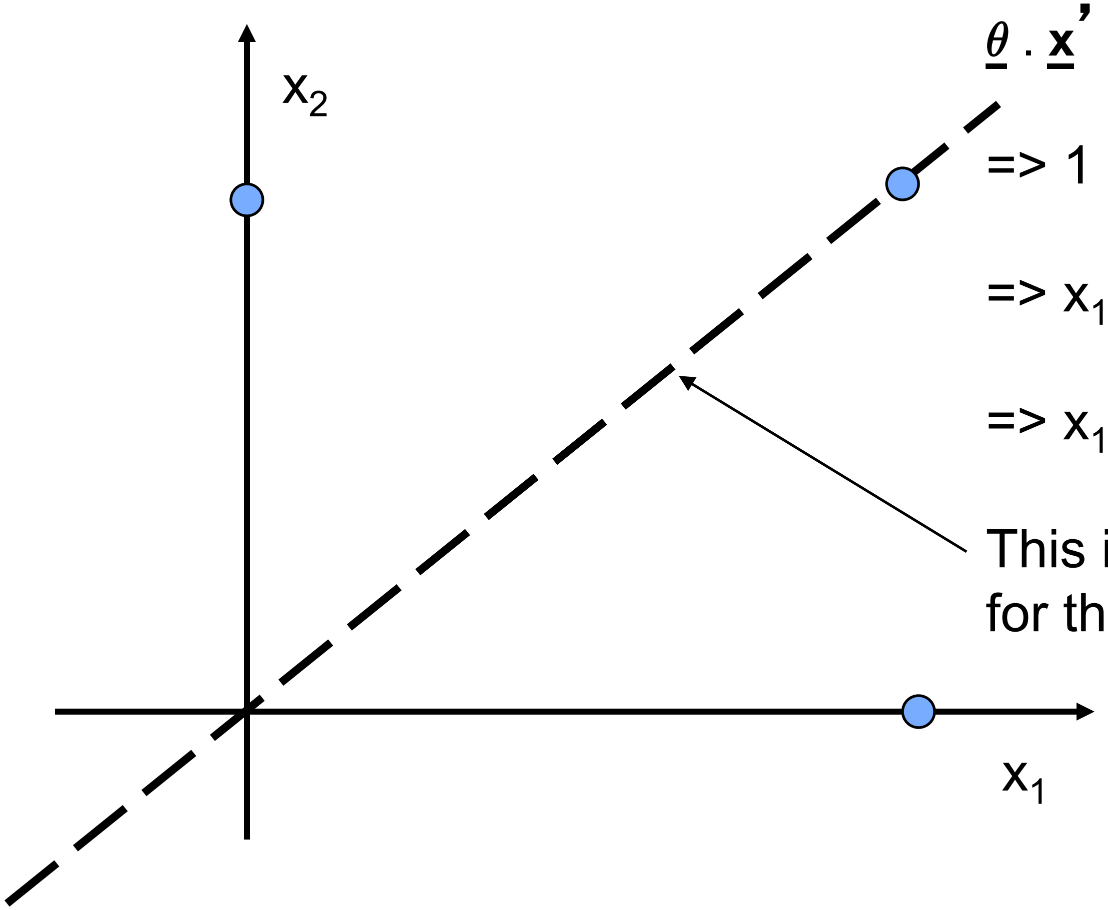
$\Rightarrow 1 \cdot x_1 - 1 \cdot x_2 + 0 \cdot 1 = 0$

$\Rightarrow x_1 - x_2 = 0$

$\Rightarrow x_1 = x_2$

Example, Linear Decision Boundary

$$\underline{\theta} = (\theta_1, \theta_2, \theta_0) = (1, -1, 0)$$



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

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

$\Rightarrow x_1 - x_2 = 0$

$\Rightarrow x_1 = x_2$

This is the equation
for the decision boundary

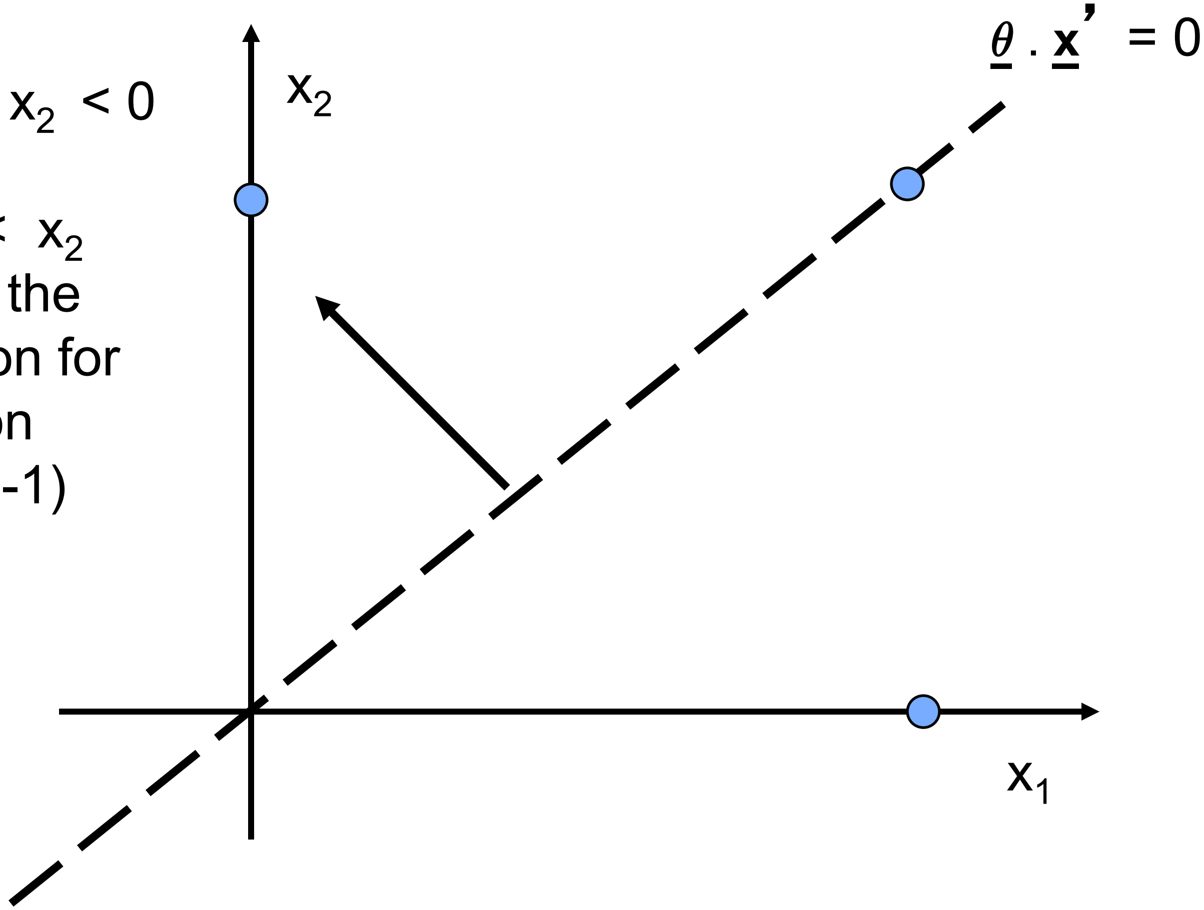
Example, Linear Decision Boundary

$\underline{\theta} = (\theta_1, \theta_2, \theta_0)$
$= (1, -1, 0)$

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

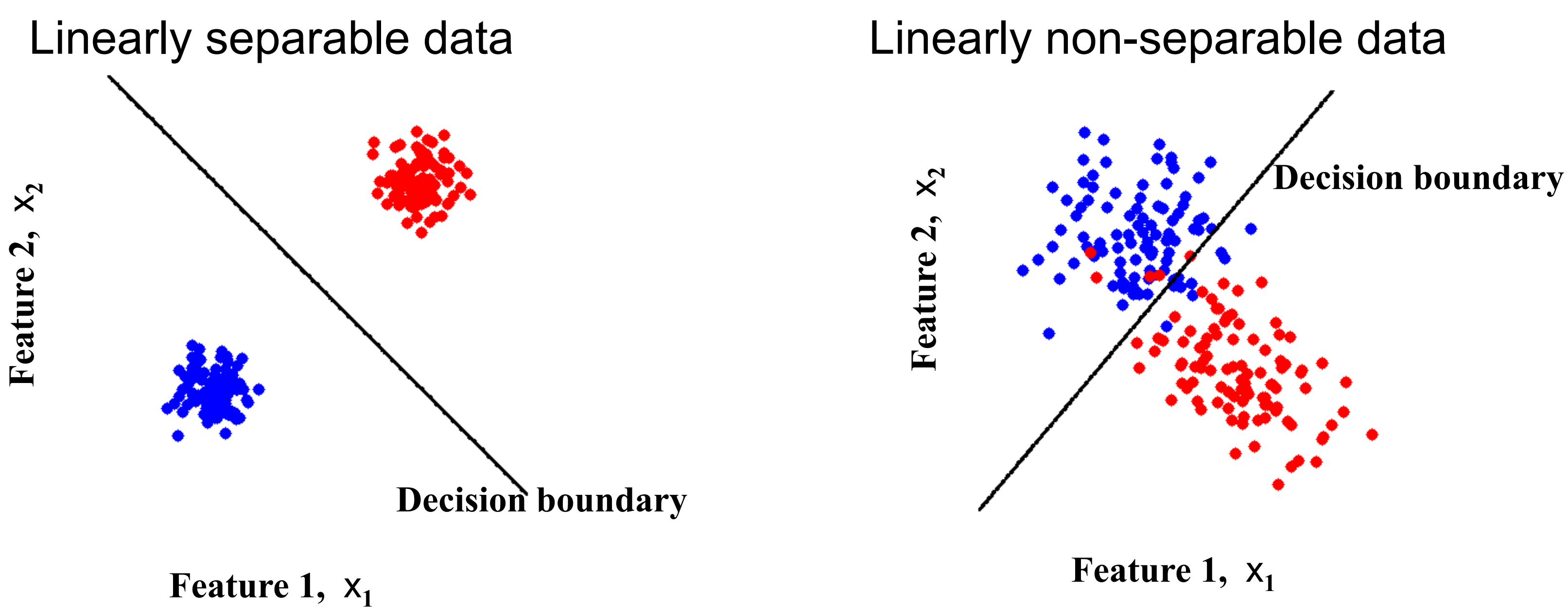
$\Rightarrow x_1 - x_2 < 0$

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

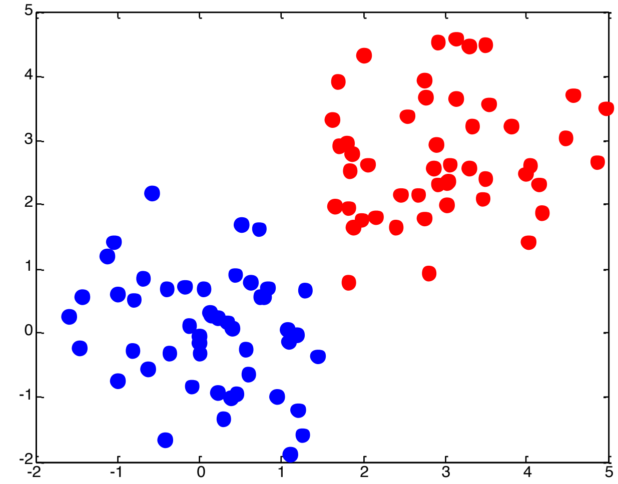


Separability

- 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

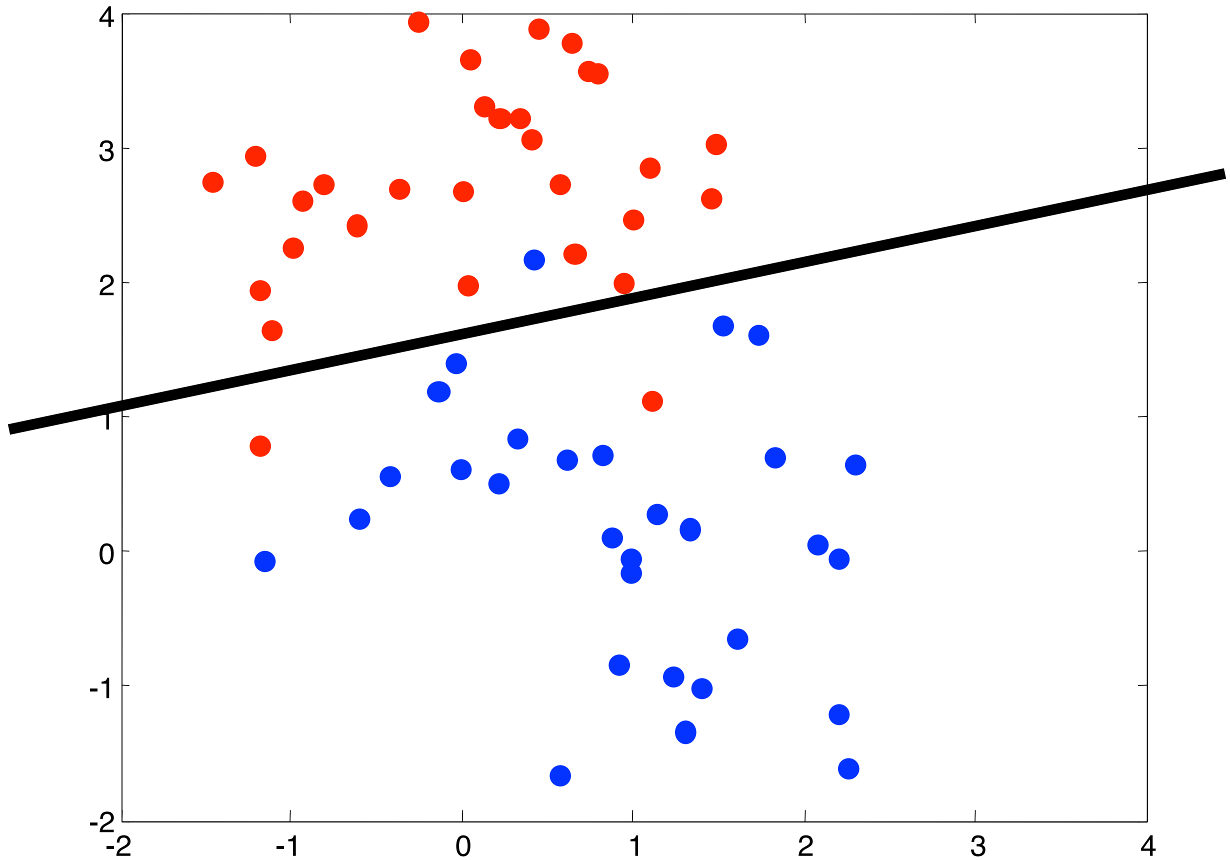


Class overlap

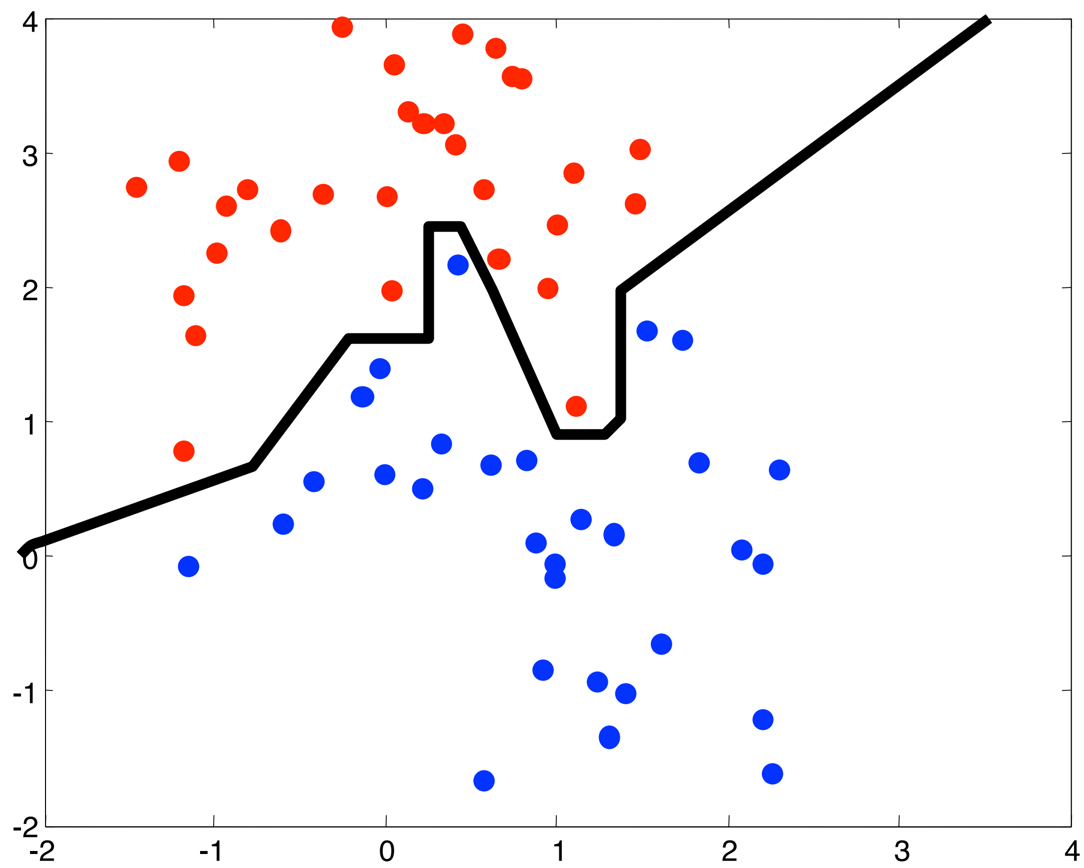


- Classes may not be well-separated
- Same observation values possible under both classes
  - High vs low risk; features {age, income}
  - Benign/malignant cells look similar
  - …
- Common in practice
- May not be able to perfectly distinguish between classes
  - Maybe with more features?
  - Maybe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

Another example

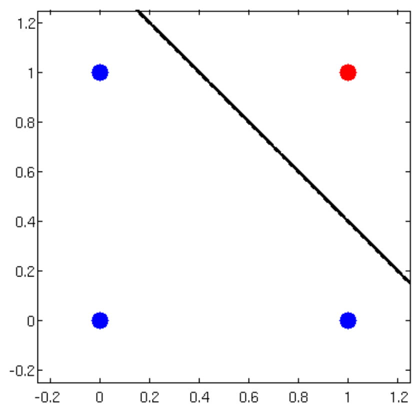


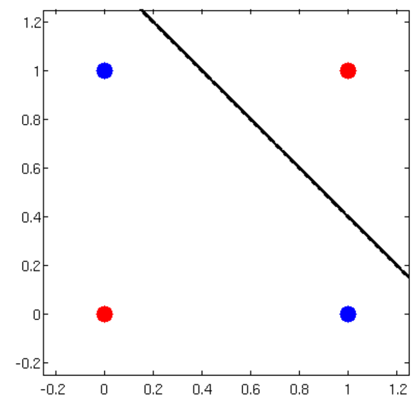
Non-linear decision boundary



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)

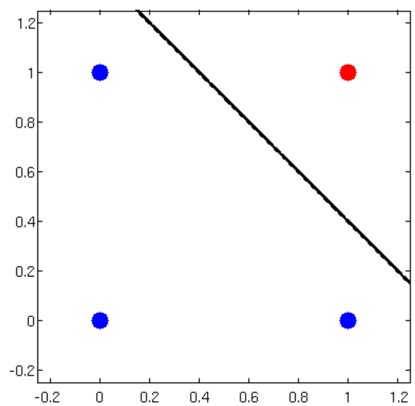


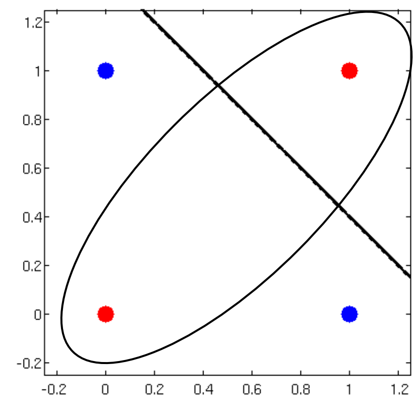


What kinds of functions would we need to learn the data on the right?

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)





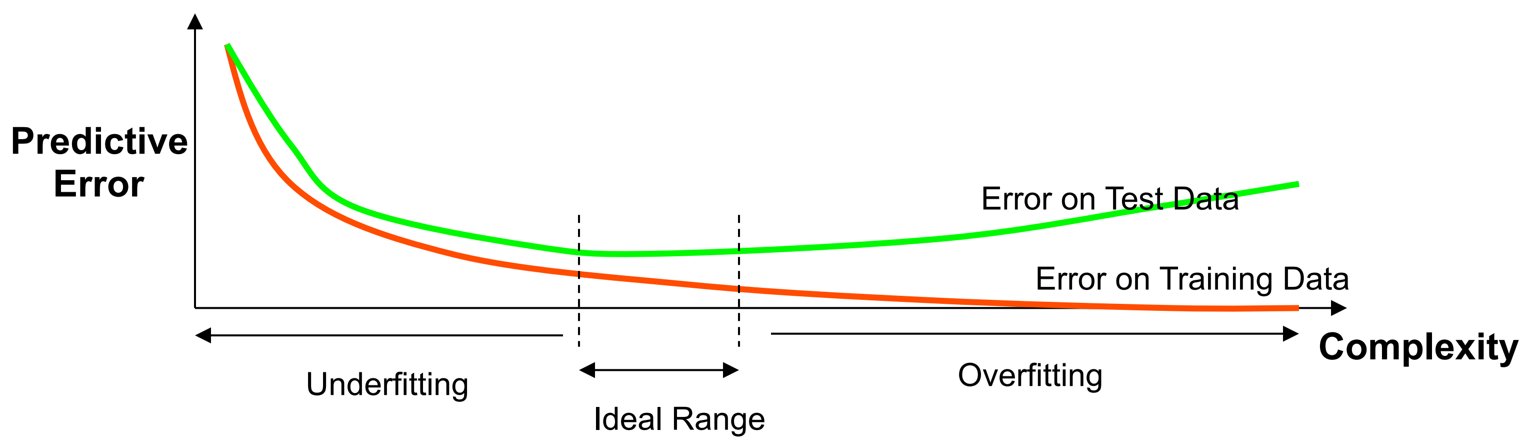
What kinds of functions would we need to learn the data on the right?

Effect of dimensionality

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

- “Good”
  - Separation is easier in higher dimensions (for fixed N)
  - 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…



Learning the Classifier Parameters

- Where do the parameters (weights) of the classifier come from?
  - If we know a lot about the problem, we could "design" them
  - Typically we don't know ahead of time what the values should be

- Learning from Training Data:
  - training data = labeled feature vectors
  - i.e., a set of N feature vectors each with a class label
  - we can use the training data to try to find good parameters
  - "good" parameters are ones which provide low error
    - error is estimated on the training data
    - "true" error will be on future test data
  - Statement of the Learning Problem:
    - given a classifier, and some training data, find the values for the classifier's parameters which maximize training accuracy

Learning the Weights from Data

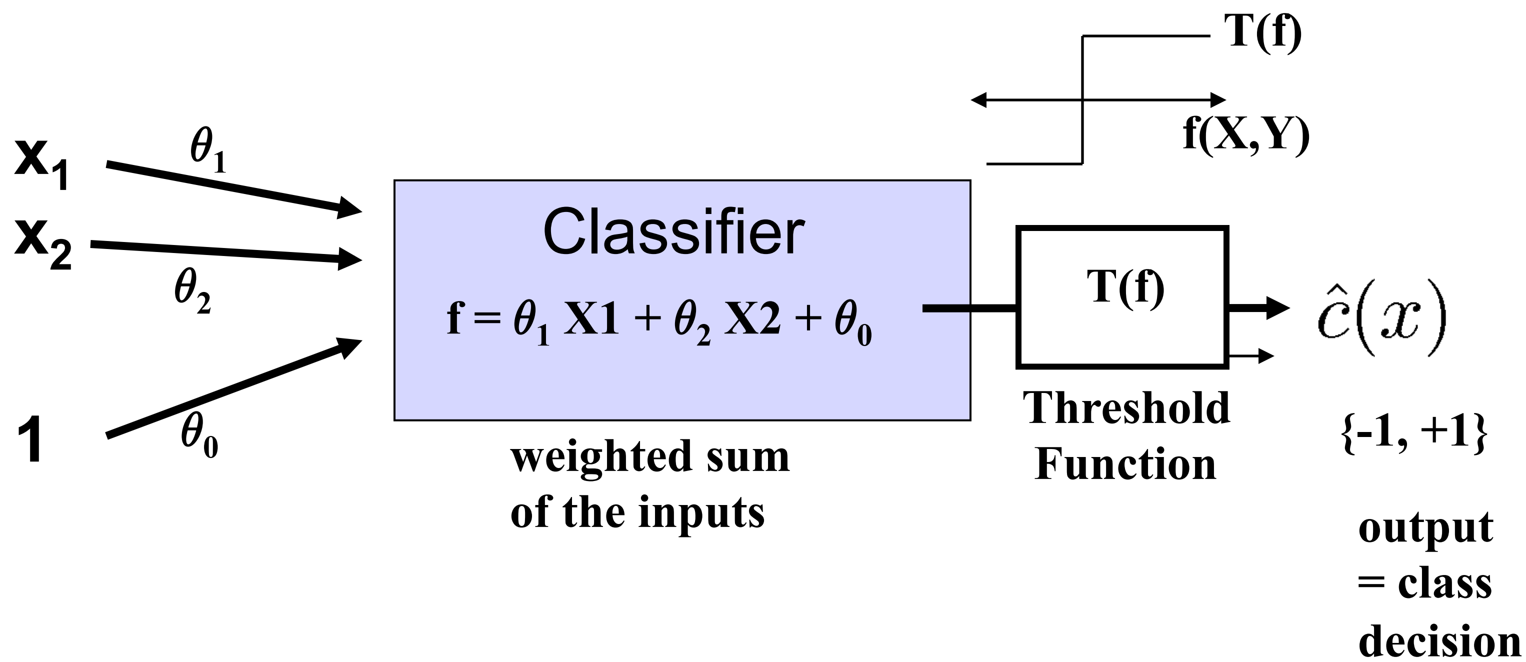
An Example of a Training Data Set

Example	x_1	x_2	….	x_d	true class label, y
$\underline{x}(1)$	3.4	-1.2	…..	7.1	1
$\underline{x}(2)$	4.1	-3.1	…..	4.6	-1
$\underline{x}(3)$	5.7	-1.0	…..	6.2	-1
$\underline{x}(4)$	2.2	4.1	…..	5.0	1
$\underline{x}(n)$	1.2	4.3	…..	6.1	1

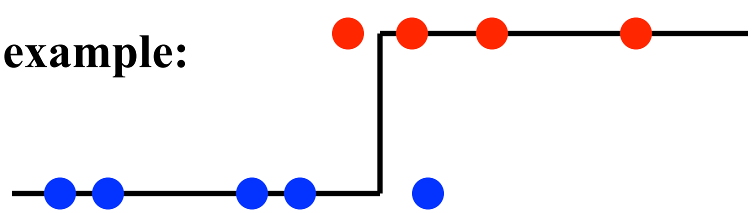
Learning as a Search Problem

- The objective function J($\underline{\theta}$) :
  - Classifier accuracy (for a given set of weights $\underline{\theta}$ and labeled data)
- Problem:
  - maximize this objective function (or, minimize error)
- Equivalent to an optimization or search problem
  - i.e., think of the vector (θ_1, θ_2, θ_0)
  - this defines a 3-dimensional "parameter space"
  - we want to find the value of (θ_1, θ_2, θ_0) which maximizes the objective
  - we could use hill-climbing, systematic search, etc., to search this parameter space
    - many learning algorithms = hill-climbing with random restarts

Perceptron Classifier (2 features)



1D example:



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

Decision boundary = "x such that T(θ_1 x + θ_0) transitions"

Training a linear classifier

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

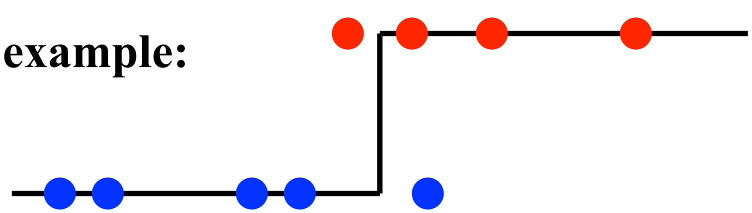
$$err(\underline{\theta}) = 1/N \sum \delta\big(\hat{y}(i) \neq y(i)\big)$$

where $\delta\big(\hat{y}(i) \neq y(i)\big) = 0$ if $\hat{y}(i) = y(i)$, and 1 otherwise

(Matlab)
```
>> yh = sign(th*X'); err = mean(y ~= yh);
```

- 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

Training a linear classifier

- "Online" gradient descent
  - Perform a gradient update one data point at a time
    - For each data point j, predict, calculate error, modify parameters; repeat

- Perceptron algorithm
  - For each data point j:

    $\hat{y}(j) = T(\underline{w} * \underline{x}(j))$: predict output for data point j

    $\underline{w} \leftarrow \underline{w} + \alpha\, (y(j) - \hat{y}(j))\, \underline{x}(j)$: "gradient-like" step

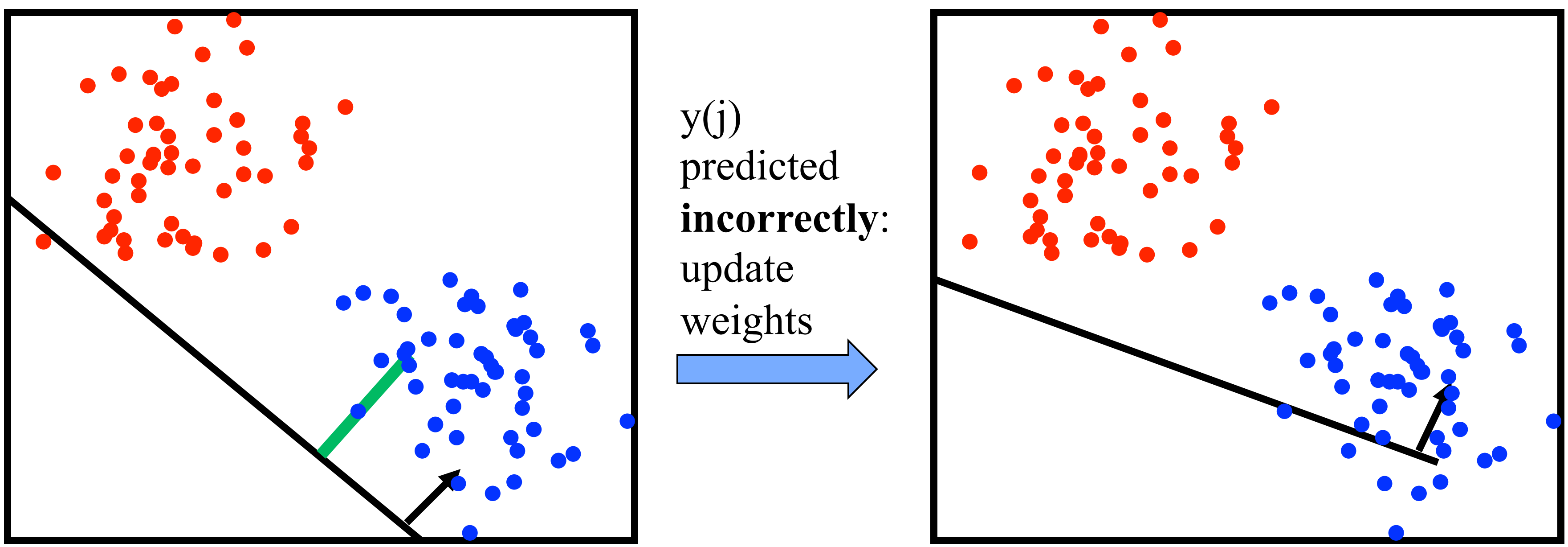
  - Converges if data are linearly separable

Perceptron algorithm

- Perceptron algorithm
  - For each data point j:

    $\hat{y}(j) = T(\, \underline{w} * \underline{x}(j) \,)$: predict output for data point j

    $\underline{w} \leftarrow \underline{w} + \alpha \, (\, y(j) - \hat{y}(j) \,)\, \underline{x}(j)$: "gradient-like" step

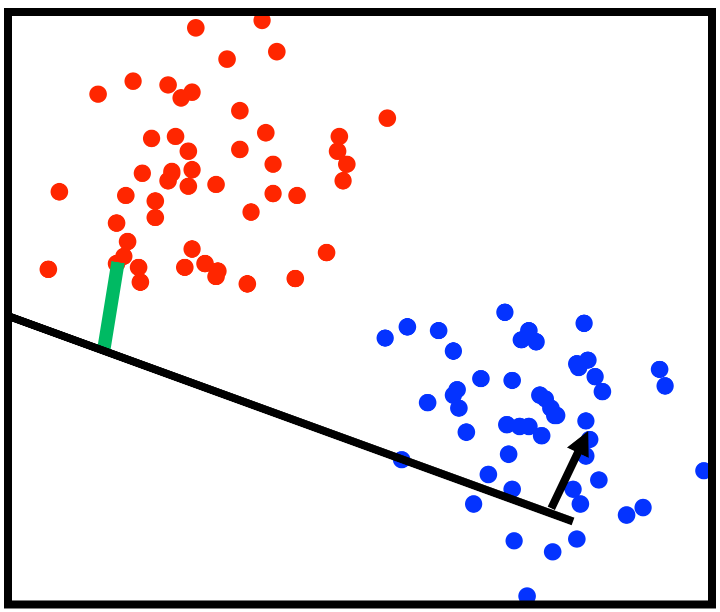


Perceptron algorithm

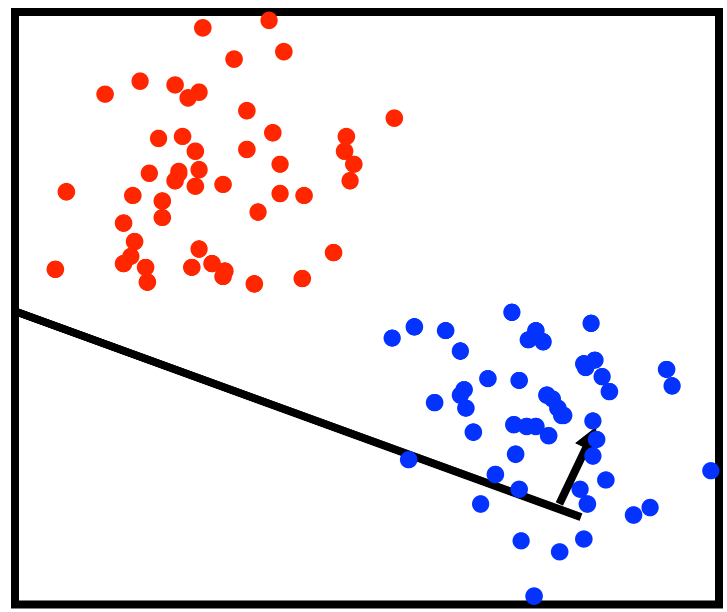
- Perceptron algorithm
  - For each data point j:

    $\hat{y}(j) = T(\ \underline{w} * \underline{x}(j)\ )$: predict output for data point j

    $\underline{w} \leftarrow \underline{w} + \alpha\ (\ y(j) - \hat{y}(j)\ )\ \underline{x}(j)$: "gradient-like" step



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



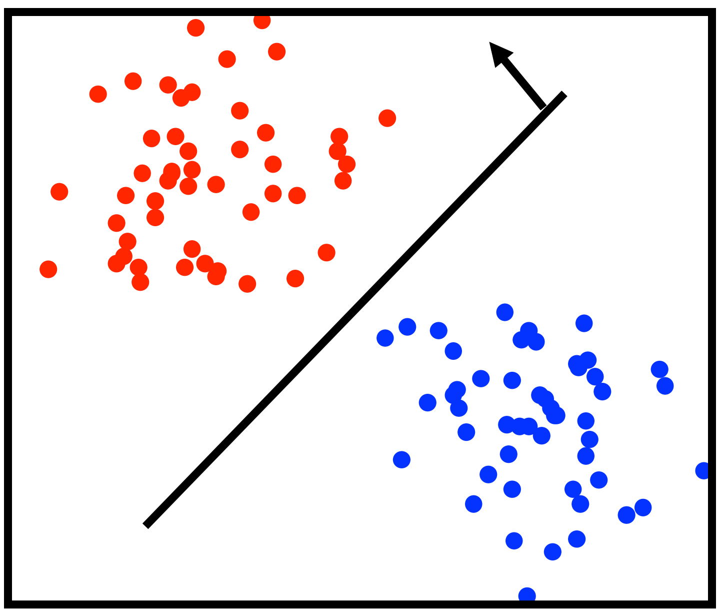
Perceptron algorithm

- Perceptron algorithm
  - For each data point j:

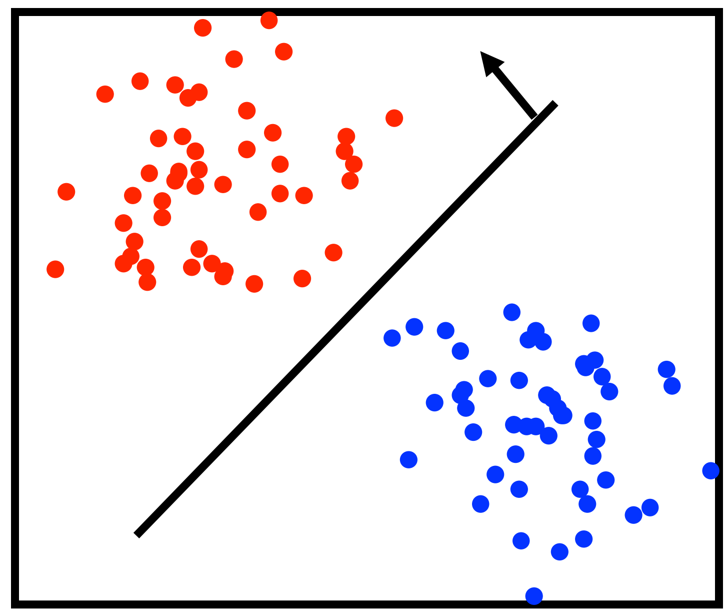
    $\hat{y}(j) = T(\underline{w} * \underline{x}(j))$: predict output for data point j

    $\underline{w} \leftarrow \underline{w} + \alpha\,(y(j) - \hat{y}(j))\,\underline{x}(j)$: "gradient-like" step

  - Converges if data are linearly separable

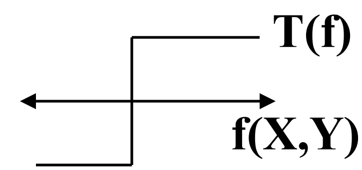


y(j) predicted **correctly**: no update

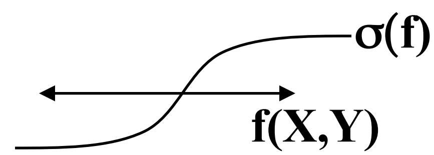


Surrogate loss functions

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



  - Usually some smooth function of distance
    - Example: "sigmoid", looks like an "S"

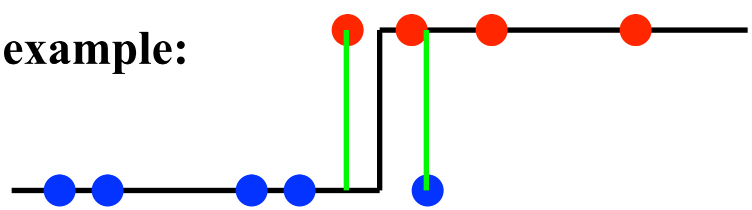


  - Now, measure e.g. MSE

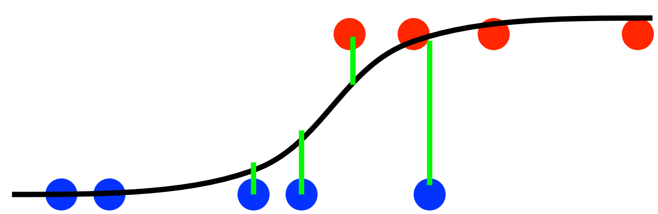
$$J_\sigma(\underline{w}) = (1/N) \sum_i \left(\sigma(f(x_i)) - t(i) \right)^2$$

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

1D example:



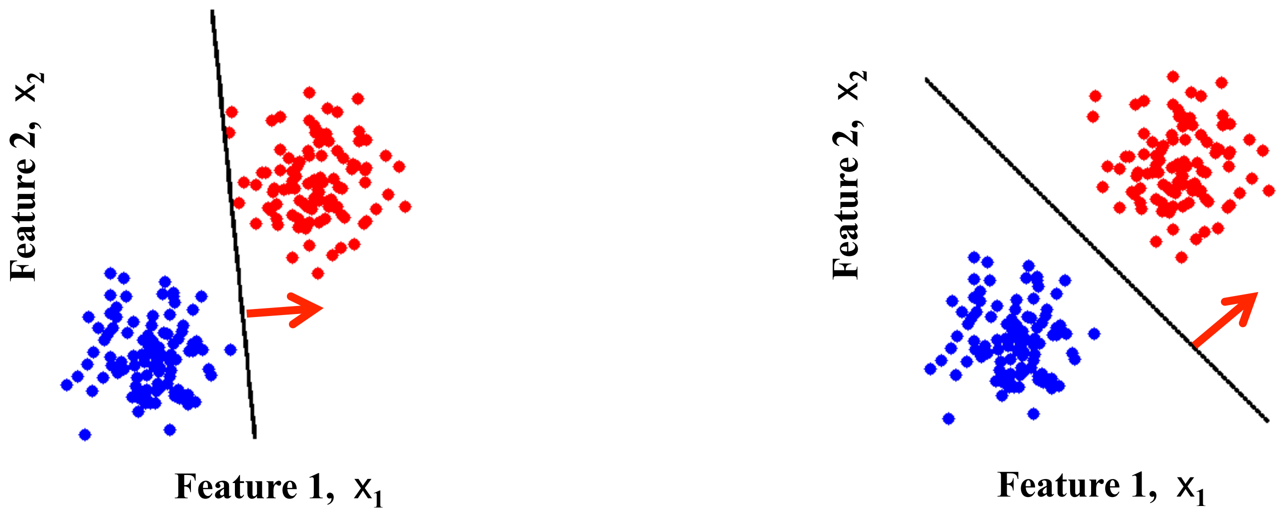
Classification error = MSE = 2/9



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…

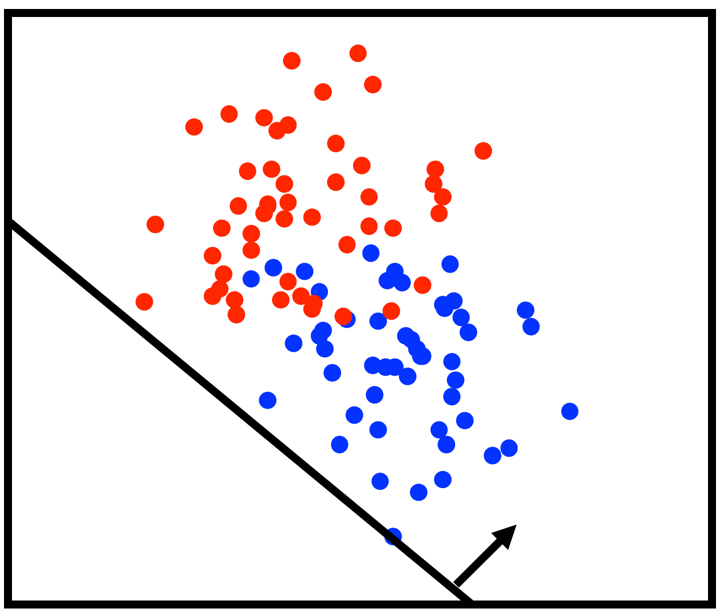


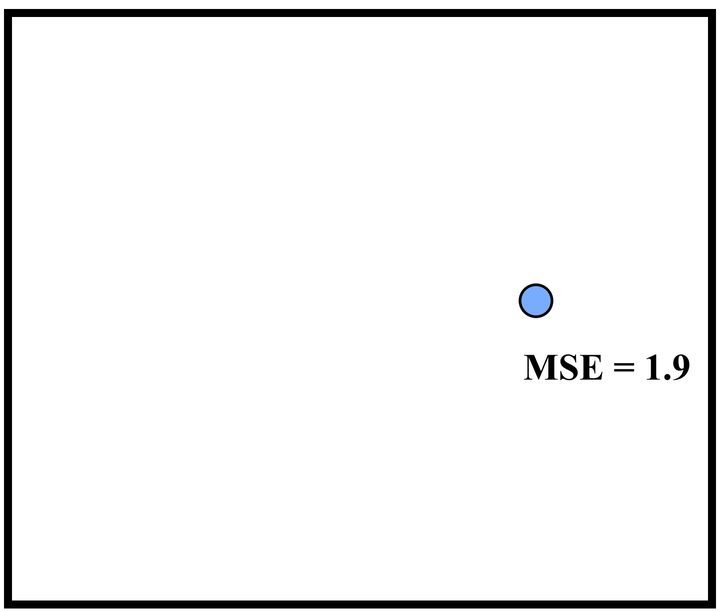
- Side benefit of "smoothed" error function
  - Encourages data to be far from the decision boundary
  - See more examples of this principle later...

Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of $f(X1,X2) = a*X1 + b*X2 + c$

- Example: 2D feature space ⇔ parameter space

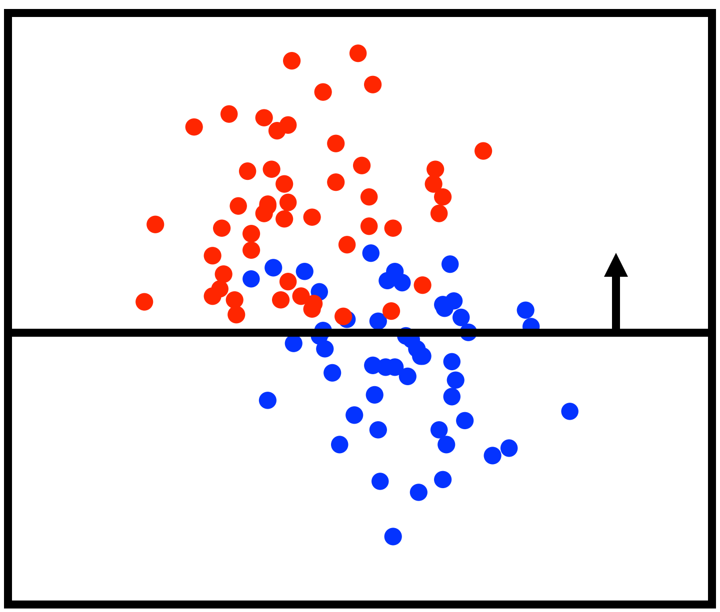


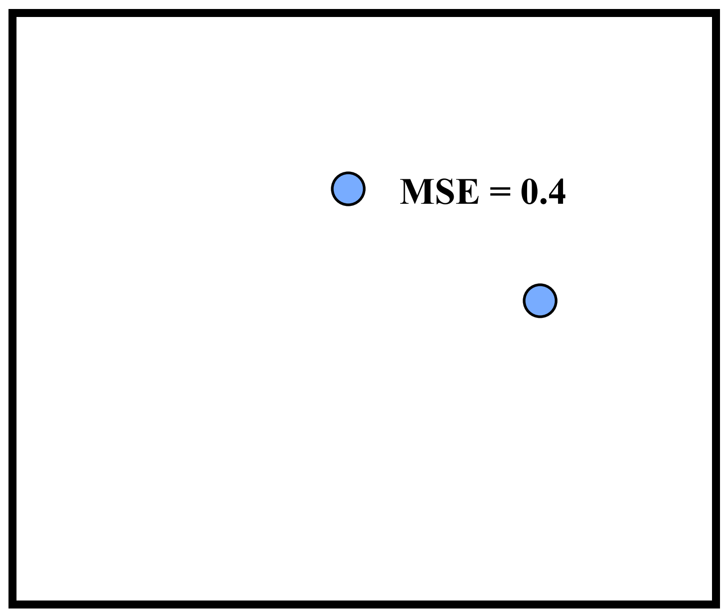


Training the Classifier

- Once we have a smooth measure of quality, we can find the “best” settings for the parameters of $f(X1,X2) = a*X1 + b*X2 + c$

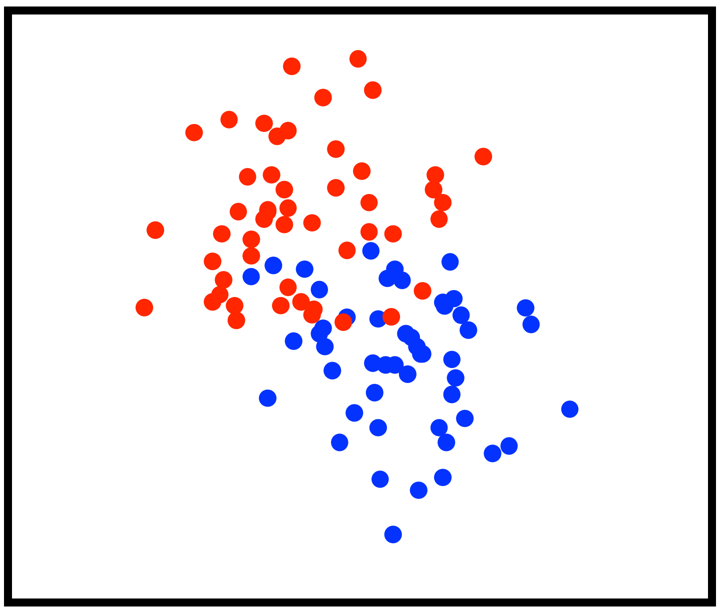
- Example: 2D feature space ⇔ parameter space

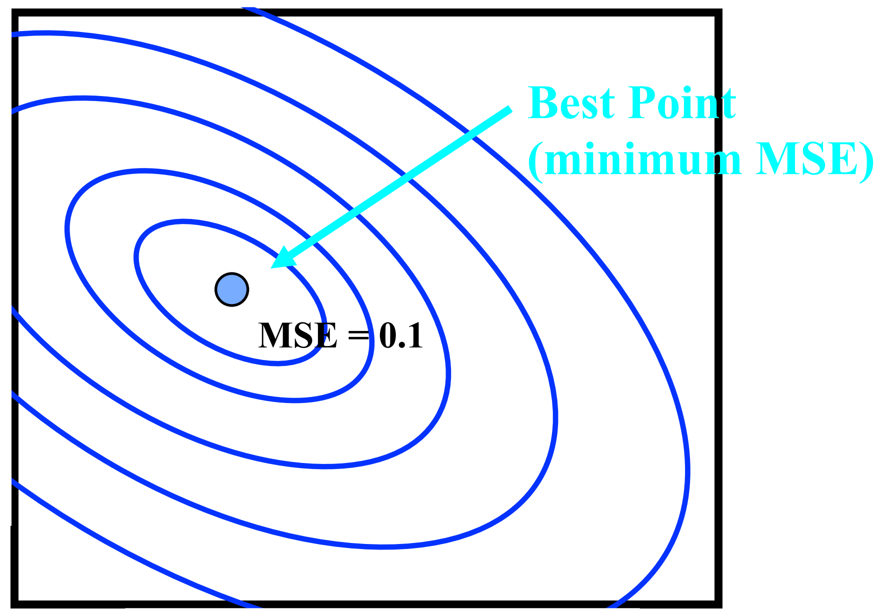




Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of f(X1,X2) = a*X1 + b*X2 + c
- Finding the minimum MSE in parameter space…



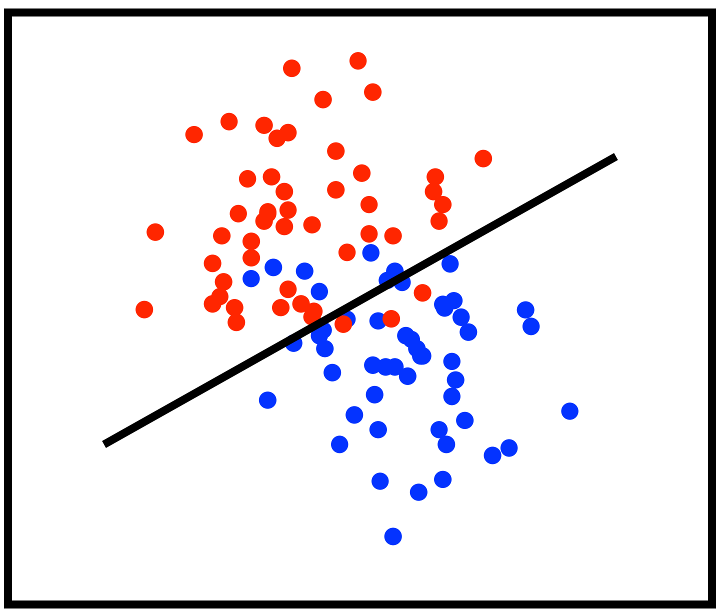


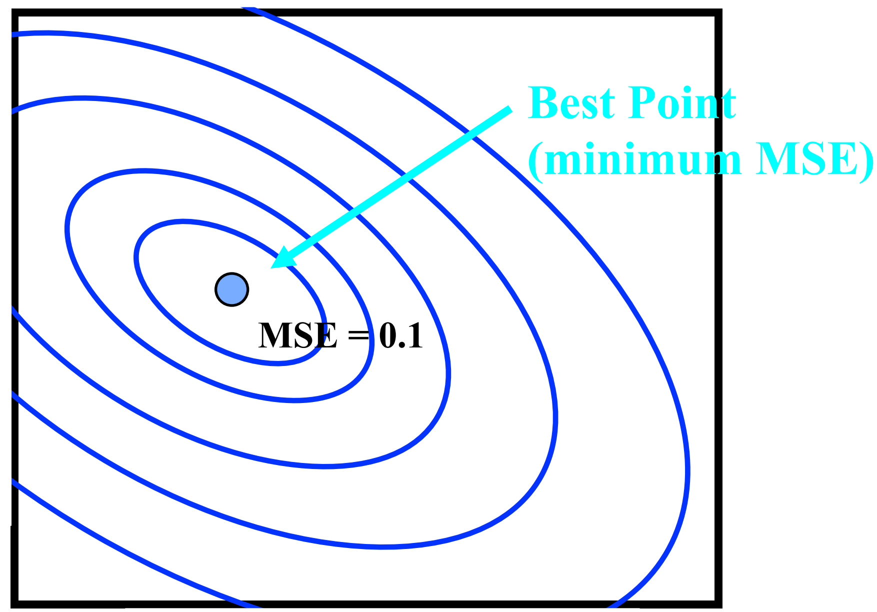
- [a b c] = ?

[arctan(A/B), c] = [-pi/4, 1]

Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of
  f(X1,X2) = a*X1 + b*X2 + c
- Finding the minimum MSE in parameter space…





- [a b c] = ?

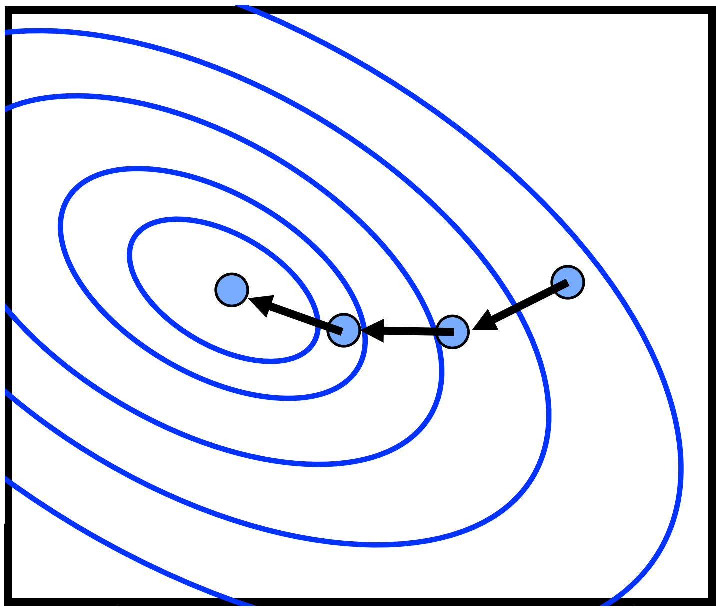
[arctan(A/B), c] = [-pi/4, 1]

Finding the Best MSE

- As in linear regression, this is now just optimization

- Methods:
  - Gradient descent
    - Improve MSE by small changes in parameters ("small" = learning rate)
  - Or, substitute your favorite optimization algorithm…
    - Coordinate descent
    - Stochastic search
    - Genetic algorithms

Gradient Descent



Gradient Equations

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

$$C(\underline{w} = [a, b, c]) = \frac{1}{N} \sum_i (\ \sigma(a x_1^{(i)} + b x_2^{(i)} + c) - y^{(i)} \ )^2$$

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

$$\frac{\partial C}{\partial a} = \frac{1}{N} \sum_i 2 \boxed{(\sigma(w \cdot x) - y^{(i)})} \ \boxed{\partial\sigma(w \cdot x) \ \ x_1(i)}$$

Error between class and prediction

Sensitivity of prediction to changes in parameter "a"

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

Saturating Functions

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

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

- Derivative is

$$\partial\sigma(x) = \sigma(x)\,(1-\sigma(x))$$

- Matlab Implementation:

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

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

Aside on logistic regression

- "Logistic regression" often refers to a different loss function than MSE
- Logistic loss function:

$$C(\underline{w}) = \frac{1}{N} \sum_i y \log \sigma(wx^T) + (1-y) \log(1-\sigma(wx^T))$$

- Interpretable as a (log) conditional probability
  - σ(w x) $\approx$ Pr[y=1]
  - Might talk about this more later

- Nicely behaved: convex, unique optimum
- BUT, we'll use MSE here...

Gradient Decent Algorithm (BATCH)

- Algorithm outline
  - Initialize the weights (e.g., randomly)
  - Loop “until convergence”
    - for each example calculate the output
    - calculate the difference between the output and the target
    - update each of the d+1 weights using the gradient update rule

$$w_j \leftarrow w_j - \eta \, (\partial E / \partial w_j)$$

  - Convergence condition:
    - when change in MSE is sufficiently small, stop iterating
  - Halt and return weights

Incremental Training Algorithm

- "Incremental Gradent Descent" – online version
- Often faster than batch gradient algorithm
- Algorithm outline
  - initialize the weights (e.g., randomly)

  - loop through all N examples (this is 1 iteration)
    - for each example calculate the output
    - calculate the difference between the output and the target
    - update each of the d+1 weights using the **single example** gradient update rule
      - Like the full gradient, but only involves one training example

  - after all N examples are gone through
    - check if the overall error (MSE) has decreased significantly since the previous iteration
    - if not, then perform another iteration through all N examples
    - if so, then halt and return weights

Gradient Descent Learning Rule

- Online (single-example) weight update rule:
  - $$\boxed{w_j \;\leftarrow\; w_j + \eta \, (t(i) - \sigma(f(i))) \; \partial\sigma(f(i)) \; x_j(i)}$$
  - $t(i)$ is the target class of the i^{th} training example
  - $f(i)$ is the weighted sum (respectively) for the ith example
  - w_j is the jth input weight
  - $x_j(i)$ is the jth input feature value, for the i^{th} example
  - η is called the learning rate: a small positive number, $0 < \eta < 1$

- An example of how this works:
  - Say w_j and $x_j(i)$ are both positive:
    - say $t(i) > f(i)$ => we increase the value of the weight
    - say $t(i) < f(i)$ => we decrease the value of the weight
    - η controls how quickly we increase or decrease the weight

Pseudocode for Logistic Regression

Initialize each weight (e.g., randomly)

```
iteration=0;
While (convergence_criterion  not achieved)
    for i=1:N
    calculate the output of the network for example i
           for j = 1: d+1
                  update weight j using the update rule
           end
    end
    calculate convergence_criterion
    ++ iteration
    (optional) plot current location of decision boundary
end
```

Summary

- Linear classifier ⇔ perceptron

- Visualizing the decision boundary

- Measuring quality of a decision boundary
  - MSE criterion

- Learning the weights of a linear classifer from data
  - Reduces to an optimization problem
  - For MSE (and some others) we can do gradient descent
  - Batch gradient descent vs. Incremental gradient descent
  - Gradient equations & update rules