CS9840
Learning and Computer Vision
Prof. Olga Veksler

Lecture 6
Linear Machines
Information Theory (a little BIT)

Today

- Optimization with Gradient descent
- Linear Classifier
  - Two classes
  - Multiple classes
  - Perceptron Criterion Function
    - Batch perceptron rule
    - Single sample perceptron rule
  - Minimum Squared Error (MSE) rule
    - Pseudoinverse
- Generalized Linear Classifier
- Gradient Descent Based learning
- Mutual Information

Optimization

- How to minimize a function of a single variable

$$\mathbf{J}(\mathbf{x}) = (\mathbf{x}-5)^2$$

- From calculus, take derivative, set it to 0

$$\frac{\mathbf{d}}{\mathbf{dx}}\mathbf{J}(\mathbf{x}) = \mathbf{0}$$

- Solve the resulting equation
  - maybe easy or hard to solve

- Example above is easy:

$$\frac{\mathbf{d}}{\mathbf{dx}}\mathbf{J}(\mathbf{x}) = \mathbf{2}(\mathbf{x}-\mathbf{5}) = \mathbf{0} \Rightarrow \mathbf{x} = \mathbf{5}$$

Optimization

- How to minimize a function of many variables

$$\mathbf{J}(\mathbf{x}) = \mathbf{J}(\mathbf{x}_1, \ldots, \mathbf{x}_d)$$

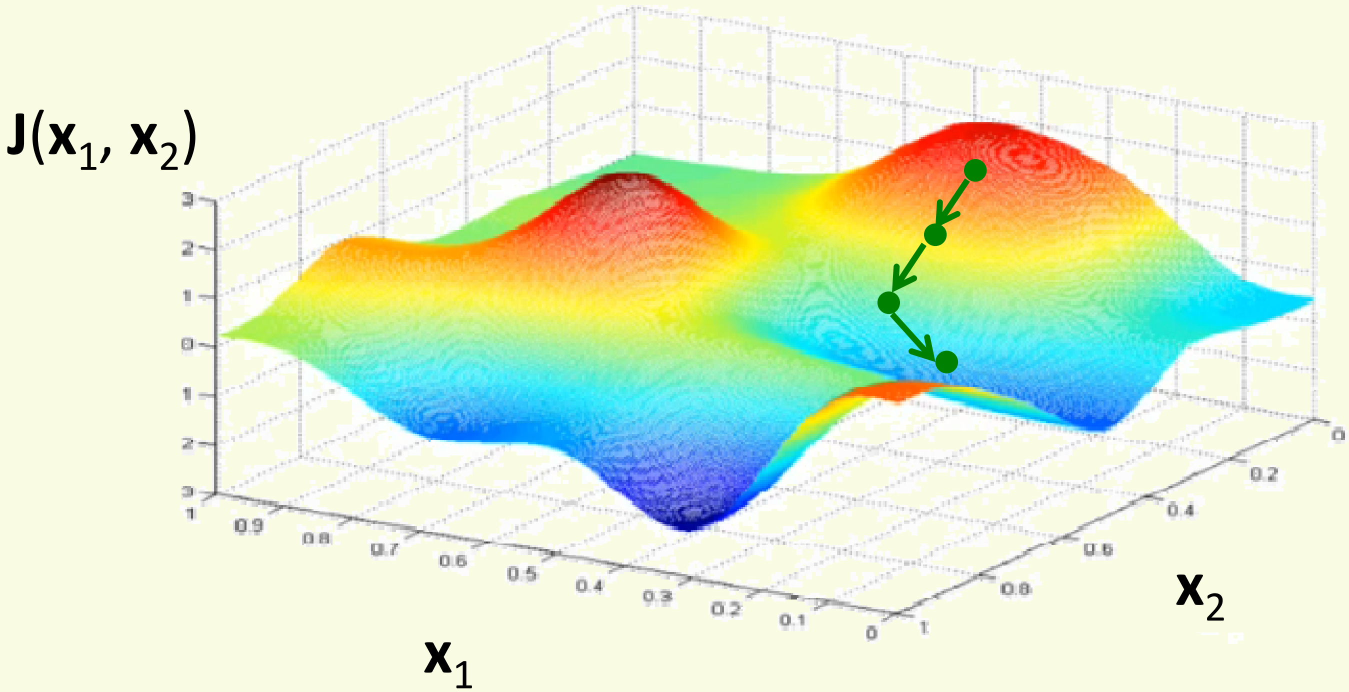
- From calculus, take partial derivatives, set them to 0

gradient

$$\begin{bmatrix} \frac{\partial}{\partial \mathbf{x}_1} \mathbf{J}(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial \mathbf{x}_d} \mathbf{J}(\mathbf{x}) \end{bmatrix} = \nabla \mathbf{J}(\mathbf{x}) = \mathbf{0}$$

- Solve the resulting system of **d** equations
- It may not be possible to solve the system of equations above analytically

Optimization: Gradient Direction

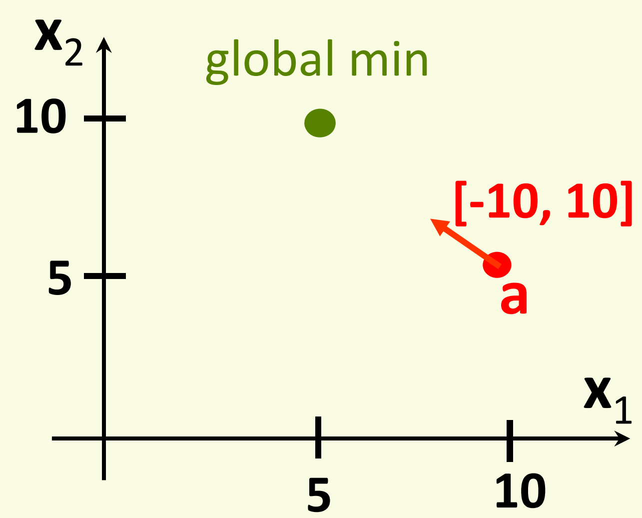


- Gradient ∇**J**(**x**) points in the direction of steepest increase of function **J**(**x**)
- **-** ∇**J**(**x**) points in the direction of steepest decrease

Picture from Andrew Ng

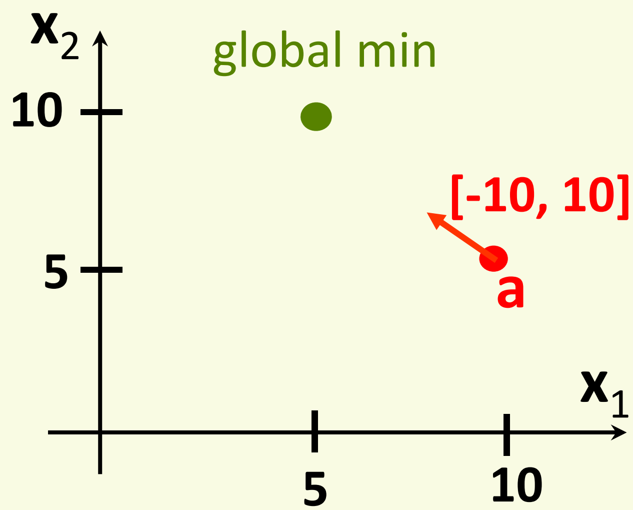
Gradient Direction in 2D

- $J(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1 - 5)^2 + (\mathbf{x}_2 - 10)^2$
- $\frac{\partial}{\partial \mathbf{x}_1} J(\mathbf{x}) = 2(\mathbf{x}_1 - 5)$
- $\frac{\partial}{\partial \mathbf{x}_2} J(\mathbf{x}) = 2(\mathbf{x}_2 - 10)$
- Let **a** = [10, 5]
- $-\frac{\partial}{\partial \mathbf{x}_1} J(\mathbf{a}) = -10$
- $-\frac{\partial}{\partial \mathbf{x}_2} J(\mathbf{a}) = 10$



Gradient Descent: Step Size

- $\mathbf{J}(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1 - 5)^2 + (\mathbf{x}_2 - 10)^2$
- Which step size to take?
- Controlled by parameter α
  - called **learning rate**
- From previous example:
  - **a =** [10 5]
  - **-∇J(a)** = [-10 10]
- Let α **=** 0.2
- **a** - α **∇J(a)** = [10 5]+0.2 [-10 10]=[8 7]
- **J**(10, 5) = 50
- **J**(8,7) = 18



Gradient Descent Algorithm

k = 1

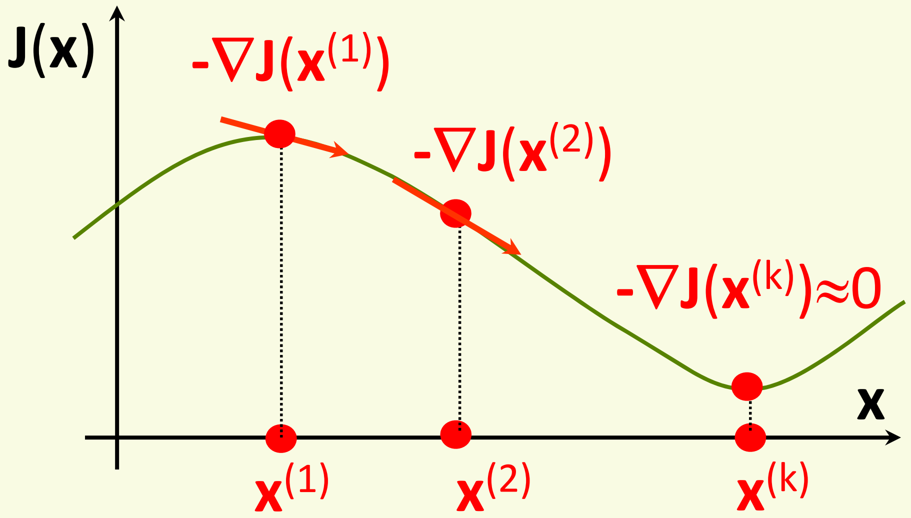
$\mathbf{x}^{(1)}$ = any initial guess

choose α, ε

while $\alpha||\nabla \mathbf{J}(\mathbf{x}^{(k)})|| > \varepsilon$

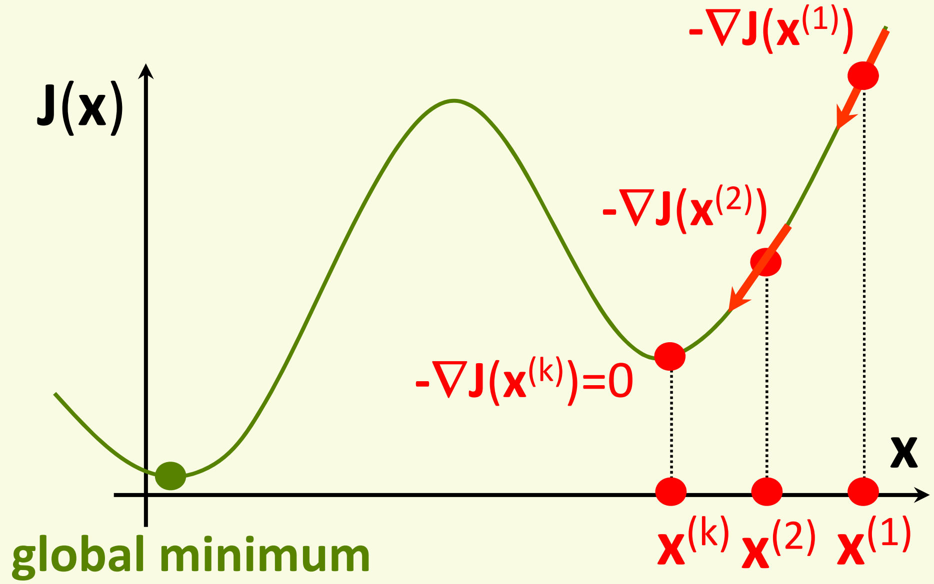
$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla \mathbf{J}(\mathbf{x}^{(k)})$

k = **k** + 1



Gradient Descent: Local Minimum

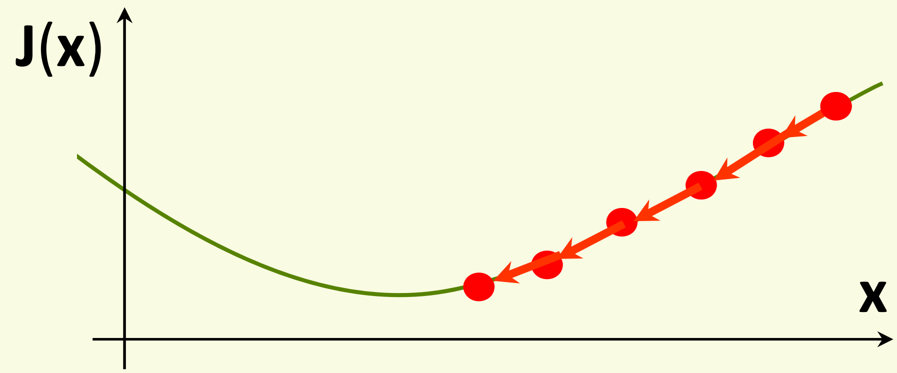
- Not guaranteed to find global minimum
  - gets stuck in local minimum



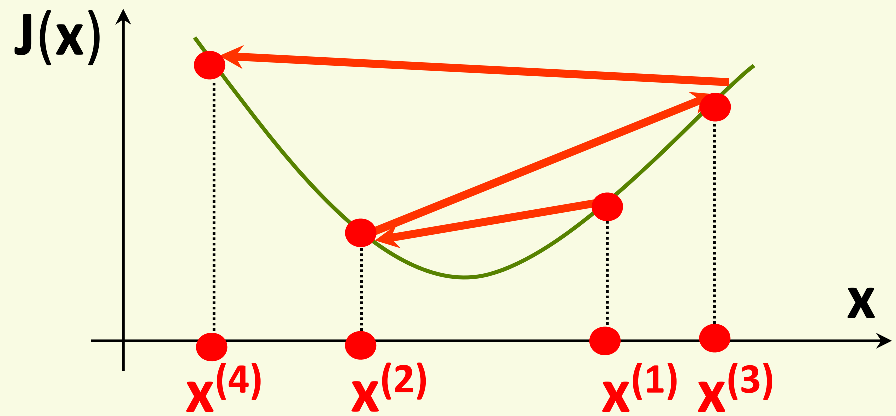
- Still gradient descent is very popular because it is simple and applicable to any differentiable function

How to Set Learning Rate α?

- If α too small, too many iterations to converge



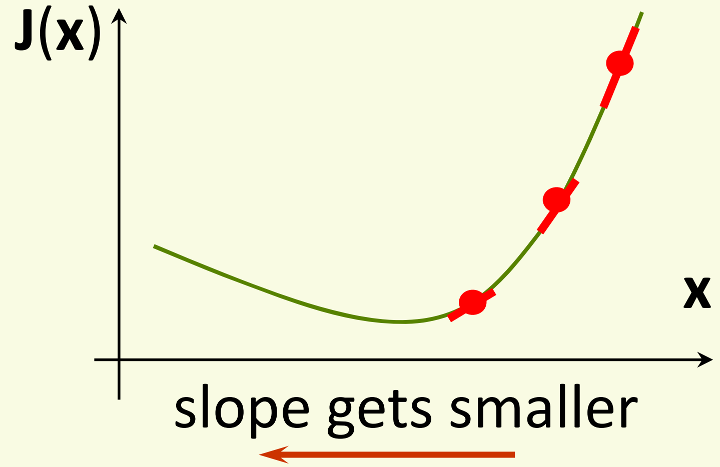
- If α too large, may overshoot the local minimum and possibly never even converge



- It helps to compute **J**(**x**) as a function of iteration number, to make sure we are properly minimizing it

How to Set Learning Rate α?

- As we approach local minimum, often gradient gets smaller
- Step size may get smaller automatically, even if α is fixed
- So it may be unnecessary to decrease α over time in order not to overshoot a local minimum



Variable Learning Rate

- If desired, can change learning rate α at each iteration

$\mathbf{k} = 1$

$\mathbf{x}^{(1)}$ = any initial guess

choose α, ε

while $\alpha \|\nabla \mathbf{J}(\mathbf{x}^{(k)})\| > \varepsilon$

$\quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla \mathbf{J}(\mathbf{x}^{(k)})$

$\quad \mathbf{k} = \mathbf{k} + 1$

→

$\mathbf{k} = 1$

$\mathbf{x}^{(1)}$ = any initial guess

choose ε

while $\alpha \|\nabla \mathbf{J}(\mathbf{x}^{(k)})\| > \varepsilon$

$\quad$ choose $\alpha^{(k)}$

$\quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha^{(k)} \nabla \mathbf{J}(\mathbf{x}^{(k)})$

$\quad \mathbf{k} = \mathbf{k} + 1$

Variable Learning Rate

- Usually don't keep track of all intermediate solutions

$\mathbf{k} = 1$

$\mathbf{x}^{(1)}$ = any initial guess

choose α, ε

while $\alpha \|\nabla \mathbf{J}(\mathbf{x}^{(k)})\| > \varepsilon$

$\quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla \mathbf{J}(\mathbf{x}^{(k)})$

$\quad \mathbf{k} = \mathbf{k} + 1$

→

$\mathbf{x}$ = any initial guess

choose α, ε

while $\alpha \|\nabla \mathbf{J}(\mathbf{x})\| > \varepsilon$

$\quad \mathbf{x} = \mathbf{x} - \alpha \nabla \mathbf{J}(\mathbf{x})$

Advanced Optimization Methods

- There are more advanced gradient-based optimization methods
- Such as conjugate gradient
  - automatically pick a good learning rate α
  - usually converge faster
  - however more complex to understand and implement
  - in Matlab, use **fminunc** for various advanced optimization methods

Last Time: Supervised Learning

- Training samples (or examples)

$$\mathbf{x}^1, \mathbf{x}^2, \dots \mathbf{x}^n$$

- Each example is typically multi-dimensional
  - $\mathbf{x}^i = [\mathbf{x}^i_1, \mathbf{x}^i_2, \dots, \mathbf{x}^i_d]$
  - $\mathbf{x}^i$ is often called a *feature vector*
- Know desired output for each example

$$\mathbf{y}^1, \mathbf{y}^2, \dots \mathbf{y}^n$$

  - regression: continuous $\mathbf{y}$
  - classification: finite $\mathbf{y}$

Last Time: Supervised Learning

- Wish to design a *machine* **f**(**x**,**w**) s.t.

$$\mathbf{f}(\mathbf{x},\mathbf{w}) = \mathbf{y}$$

  - How do we choose **f**?
    - last lecture studied kNN classifier
    - this lecture in on liner classifier
    - many other choices
  - W is typically multidimensional vector of weights (also called *parameters*)

$$\mathbf{w} = [\mathbf{w}_1,\mathbf{w}_2,\ldots\mathbf{w}_k]$$

  - By modifying **w**, the machine "learns"

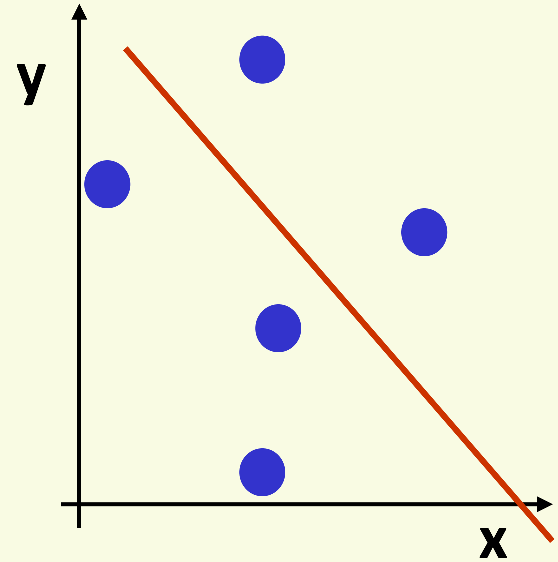
Training and Testing Phases

- Divide all labeled samples $\mathbf{x}^1$, $\mathbf{x}^2$,…, $\mathbf{x}^n$ into *training* and *test* sets
- Training phase
  - Uses training samples
  - goal is to “teach” the machine
  - find weights $\mathbf{w}$ s.t. $\mathbf{f}(\mathbf{x}^i,\mathbf{w}) = \mathbf{y}^i$ “as much as possible”
    - “as much as possible” needs to be defined
- Testing phase
  - Uses only test samples
  - for evaluating how well our machine works on unseen examples

Loss Function

- How to quantify "$\mathbf{f}(\mathbf{x}^i,\mathbf{w}) = \mathbf{y}^i$ as much as possible"?
- $\mathbf{f}(\mathbf{x},\mathbf{w})$ has to be "close" to the true output $\mathbf{y}$
- Define Loss (or Error, or Criterion) function $\mathbf{L}$
- Typically first define per-sample loss $\mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w})$
  - for classification, $\mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w}) = \mathbf{I}[\mathbf{f}(\mathbf{x}^i,\mathbf{w}) \neq \mathbf{y}^i]$
    - where $\mathbf{I}[\text{true}] = 1$, $\mathbf{I}[\text{false}] = 0$
  - for regression, $\mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w}) = \| \mathbf{f}(\mathbf{x}^i,\mathbf{w}) - \mathbf{y}^i \|^2$,
    - how far is the estimated output from the correct one?
- Then loss function $\mathbf{L} = \sum_i \mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w})$
  - classification: counts number of missclassified examples
  - regression: sums distances to the correct output

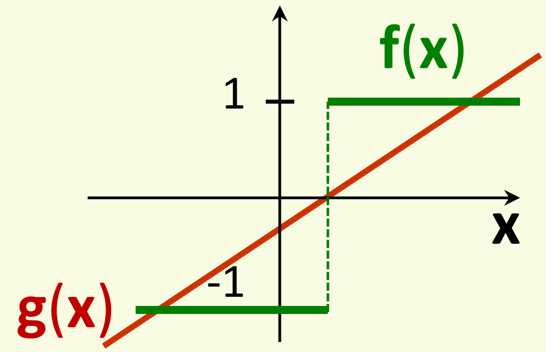
Linear Machine: Regression



- $\mathbf{f}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\sum_{i=1,2,\ldots d} \mathbf{w}_i\mathbf{x}_i$
- In vector notation
  - $\mathbf{x}= [\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_d]$
  - $\mathbf{f}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\mathbf{w}^t\mathbf{x}$
- This is standard linear regression
  - line fitting
  - assume $\mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w}) = \|\mathbf{f}(\mathbf{x}^i,\mathbf{w}) - \mathbf{y}^i\|^2$
- optimal **w** can be found by solving a system of linear equations

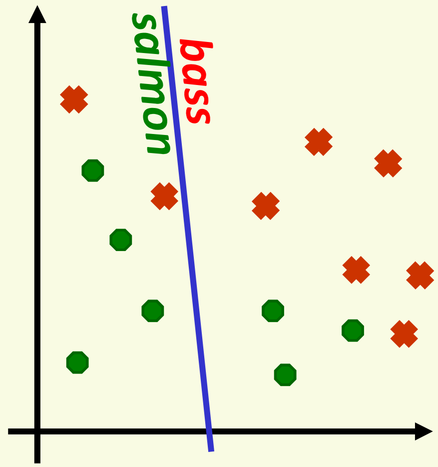
$$\mathbf{w}^* = [\Sigma\mathbf{x}^i (\mathbf{x}^i)^T]^{-1} \Sigma\mathbf{y}^i\mathbf{x}^i$$

Linear Machine: Classification

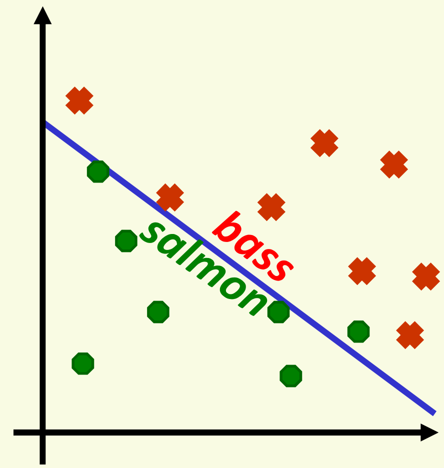


- First consider the two-class case
- We choose the following encoding:
  - **y** = 1 for the first class
  - **y** = -1 for the second class
- Linear classifier
  - $-\infty \leq \mathbf{w}_0+\mathbf{x}_1\mathbf{w}_1 + \ldots + \mathbf{x}_d\mathbf{w}_d \leq \infty$
  - we need **f**(**x**,**w**) to be either +1 or -1
  - let $\mathbf{g}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\mathbf{x}_1\mathbf{w}_1 + \ldots + \mathbf{x}_d\mathbf{w}_d = \mathbf{w}_0+\mathbf{w}^t\mathbf{x}$
  - let **f**(**x**,**w**) = sign(**g**(**x**,**w**))
    - 1 if **g**(**x**,**w**) is positive
    - -1 if **g**(**x**,**w**) is negative
    - other choices for **g**(**x**,**w**) are also used
  - **g**(**x**,**w**) is called the **discriminant function**

Linear Classifier: Decision Boundary



bad boundary



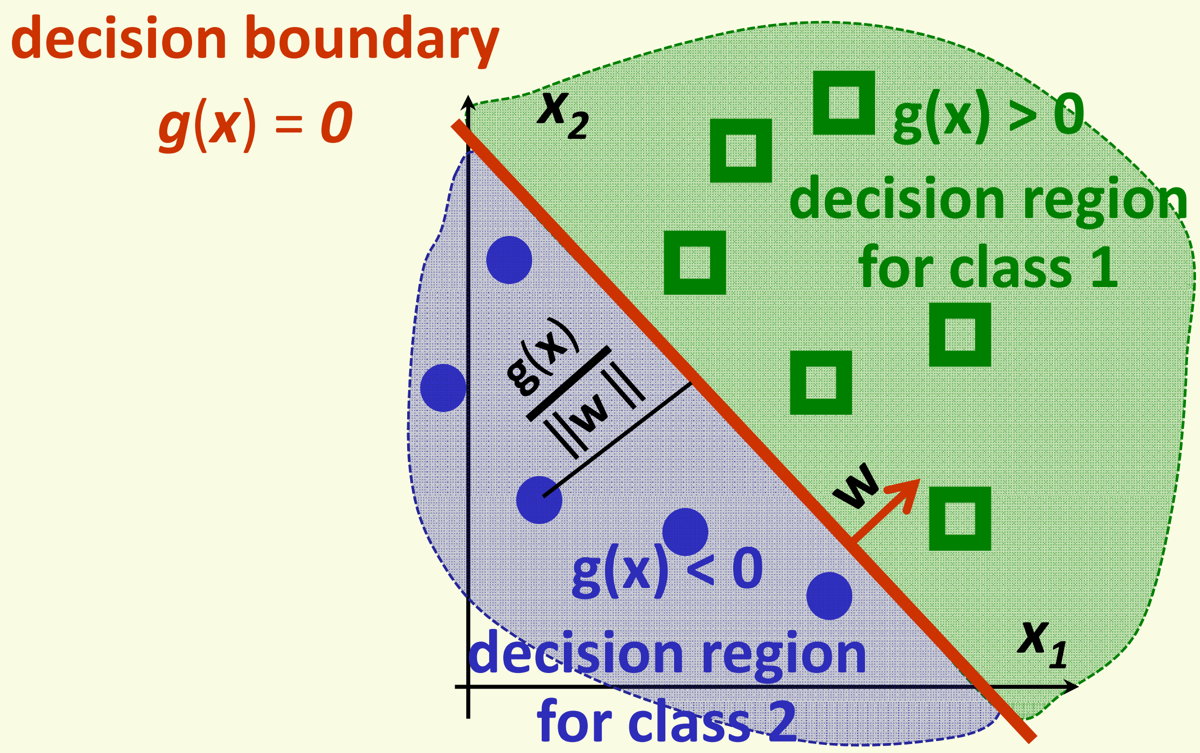
better boundary

- $\mathbf{f}(\mathbf{x},\mathbf{w}) = \text{sign}(\mathbf{g}(\mathbf{x},\mathbf{w})) = \text{sign}(\mathbf{w}_0+\mathbf{x}_1\mathbf{w}_1 + \ldots + \mathbf{x}_d\mathbf{w}_d)$
- Decision boundary is linear
- Find the best linear boundary to separate two classes
- Search for best $\mathbf{w} = [\mathbf{w}_0,\mathbf{w}_1,\ldots,\mathbf{w}_d]$ to minimize training error

More on Linear Discriminant Function (LDF)

- LDF: $\mathbf{g(x,w) = w_0 + x_1 w_1 + \dots + x_d w_d}$
- Written using vector notation $\mathbf{g(x) = w^t x + w_0}$

weight vector bias or threshold



More on Linear Discriminant Function (LDF)

- Decision boundary: $\mathbf{g}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\mathbf{x}_1\mathbf{w}_1 + \dots + \mathbf{x}_d\mathbf{w}_d = 0$
- This is a hyperplane, by definition
  - a point in 1D
  - a line in 2D
  - a plane in 3D
  - a hyperplane in higher dimensions

Multiple Classes

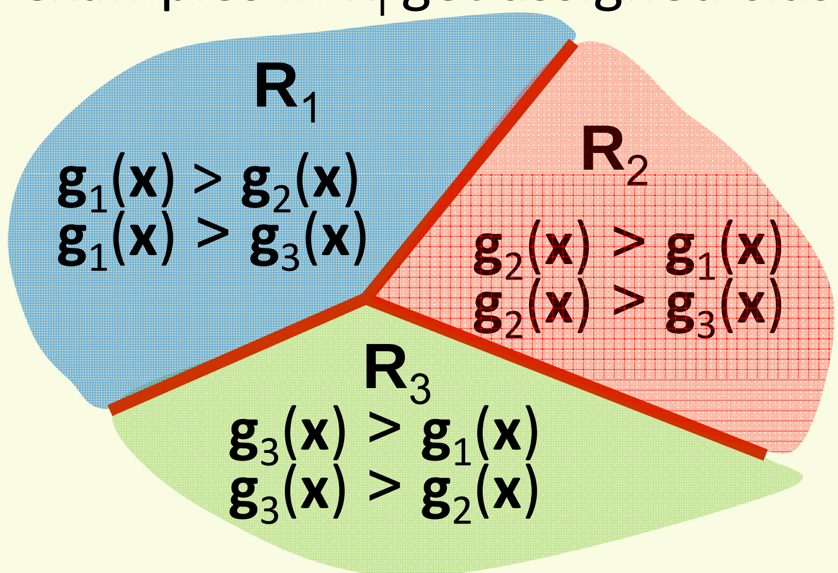
- We have **m** classes
- Define **m** linear discriminant functions

$$\mathbf{g}_i(\mathbf{x}) = \mathbf{w}_i^t\mathbf{x} + \mathbf{w}_{i0} \quad \text{for } \mathbf{i} = 1, 2, \ldots \mathbf{m}$$

- Assign **x** to class **i** if

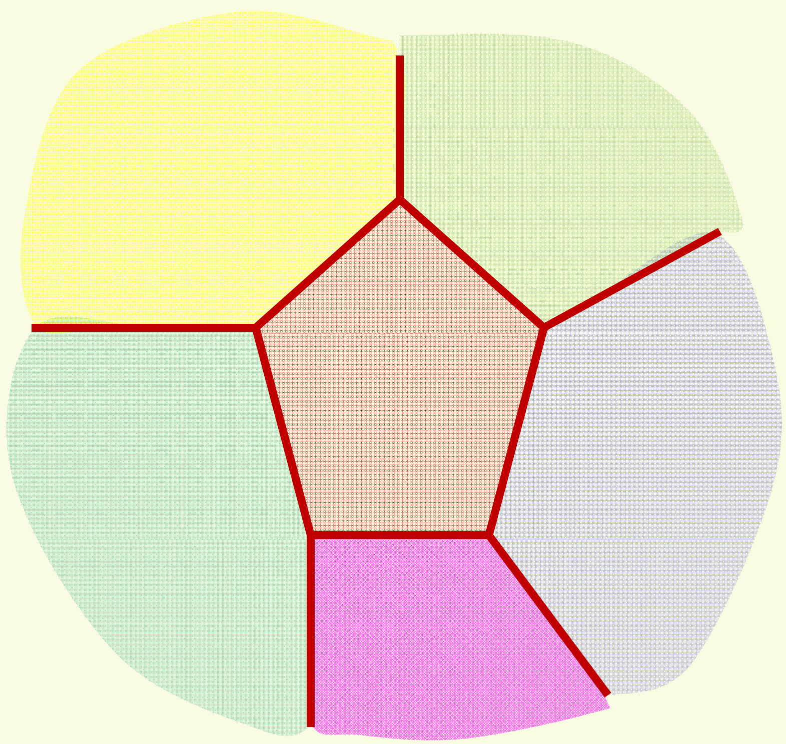
$$\mathbf{g}_i(\mathbf{x}) > \mathbf{g}_j(\mathbf{x}) \text{ for all } \mathbf{j} \neq \mathbf{i}$$

- Let $\mathbf{R}_i$ be the decision region for class **i**
  - That is all examples in $\mathbf{R}_i$ get assigned class **i**



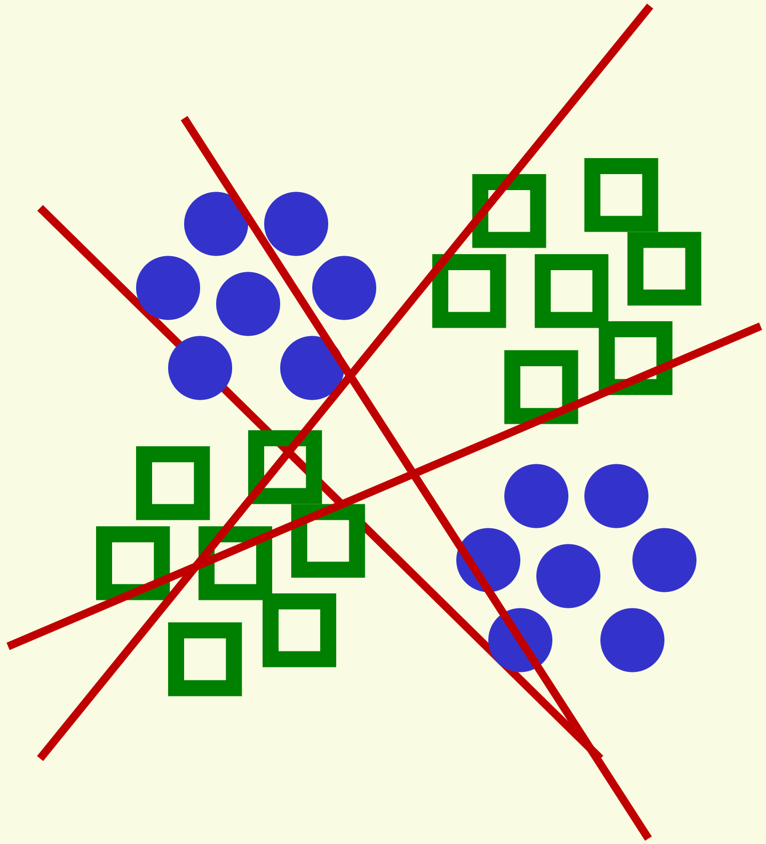
Multiple Classes

- Can be shown that decision regions are convex
- In particular, they must be spatially contiguous



Failure Cases for Linear Classifier

- Thus applicability of linear classifiers is limited to mostly unimodal distributions, such as Gaussian
- Not unimodal data
- Need non-contiguous decision regions
- Linear classifier will fail



Linear Classifiers

- Linear classifiers give simple decision boundary
  - try simpler models first
- Linear classifiers are optimal for certain type of data
  - Gaussian distributions with equal covariance
- May not be optimal for other data distributions, but they are very simple to use

Fitting Parameters w

- Linear discriminant function $\mathbf{g(x)} = \mathbf{w}^t\mathbf{x} + \mathbf{w_0}$
- Can rewrite it $\mathbf{g(x)} = [\mathbf{w}_0 \quad \mathbf{w}^t] \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} = \mathbf{a}^t\mathbf{z} = \mathbf{g(z)}$

  new weight vector ***a***

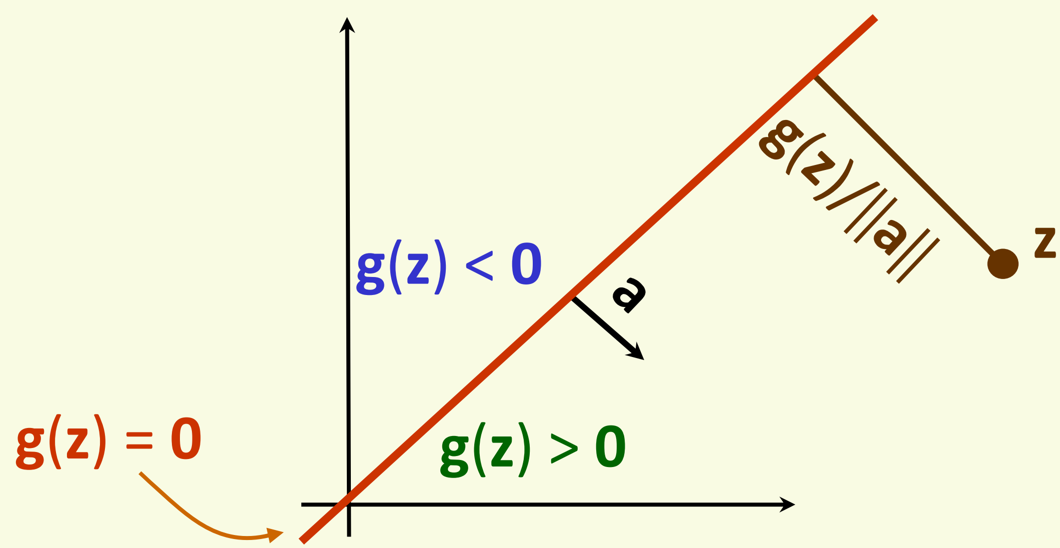
  new feature vector ***z***

- **z** is called augmented feature vector
- new problem equivalent to the old $\mathbf{g(z)} = \mathbf{a}^t\mathbf{z}$

$$\mathbf{a} = \begin{bmatrix} \mathbf{w}_0 \\ \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_d \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} 1 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_d \end{bmatrix}$$

Augmented Feature Vector

- Feature augmenting is done to simplify notation
- From now on we assume that we have augmented feature vectors
  - given samples $\mathbf{x}^1, \ldots, \mathbf{x}^n$ convert them to augmented samples $\mathbf{z}^1, \ldots, \mathbf{z}^n$ by adding a new dimension of value 1
- $\mathbf{g(z) = a^t z}$



Training Error

- For the rest of the lecture, assume we have 2 classes
- Samples $\mathbf{z}^1, \ldots, \mathbf{z}^n$ some in class 1, some in class 2
- Use these samples to determine weights $\mathbf{a}$ in the discriminant function $\mathbf{g}(\mathbf{z}) = \mathbf{a}^t\mathbf{z}$
- Want to minimize number of misclassified samples
- Recall that $\begin{cases} \mathbf{g}(\mathbf{z}^i) > 0 & \Rightarrow \text{class 1} \\ \mathbf{g}(\mathbf{z}^i) < 0 & \Rightarrow \text{class 2} \end{cases}$
- Thus training error is 0 if $\begin{cases} \mathbf{g}(\mathbf{z}^i) > 0 & \forall \mathbf{z}^i \text{ class 1} \\ \mathbf{g}(\mathbf{z}^i) < 0 & \forall \mathbf{z}^i \text{ class 2} \end{cases}$

Simplifying Notation Further

- Thus training error is 0 if $\begin{cases} \mathbf{a}^t\mathbf{z}^i > 0 & \forall \mathbf{z}^i \text{ class 1} \\ \mathbf{a}^t\mathbf{z}^i < 0 & \forall \mathbf{z}^i \text{ class 2} \end{cases}$

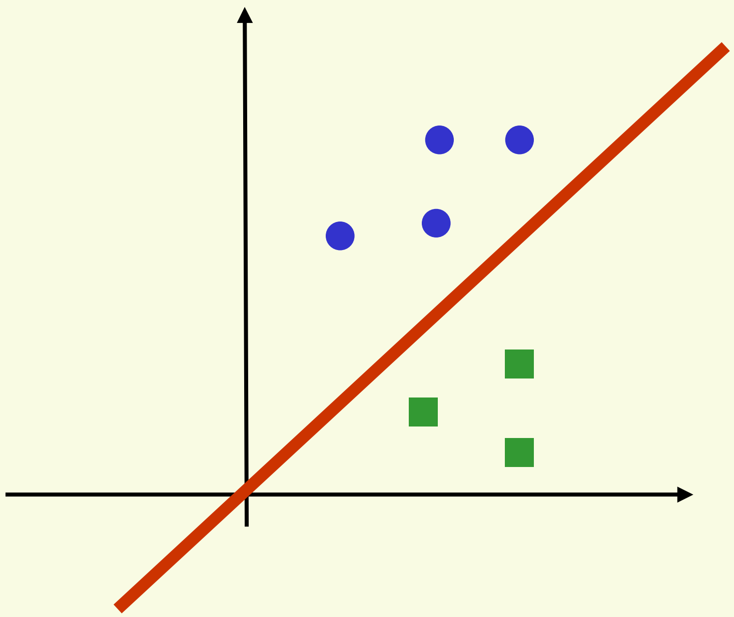
- Equivalently, training error is 0 if $\begin{cases} \mathbf{a}^t\mathbf{z}^i > 0 & \forall \mathbf{z}^i \text{ class 1} \\ \mathbf{a}^t(-\mathbf{z}^i) > 0 & \forall \mathbf{z}^i \text{ class 2} \end{cases}$

- Problem "normalization":
    1. replace all examples $\mathbf{z}^i$ from class 2 by $-\mathbf{z}^i$
    2. seek weights **a** s.t. $\mathbf{a}^t\mathbf{z}^i > 0$ for $\forall \mathbf{z}^i$

- If exists, such **a** is called a ***separating*** or ***solution*** vector
- Original samples $\mathbf{x}^1, \dots \mathbf{x}^n$ can also be linearly separated

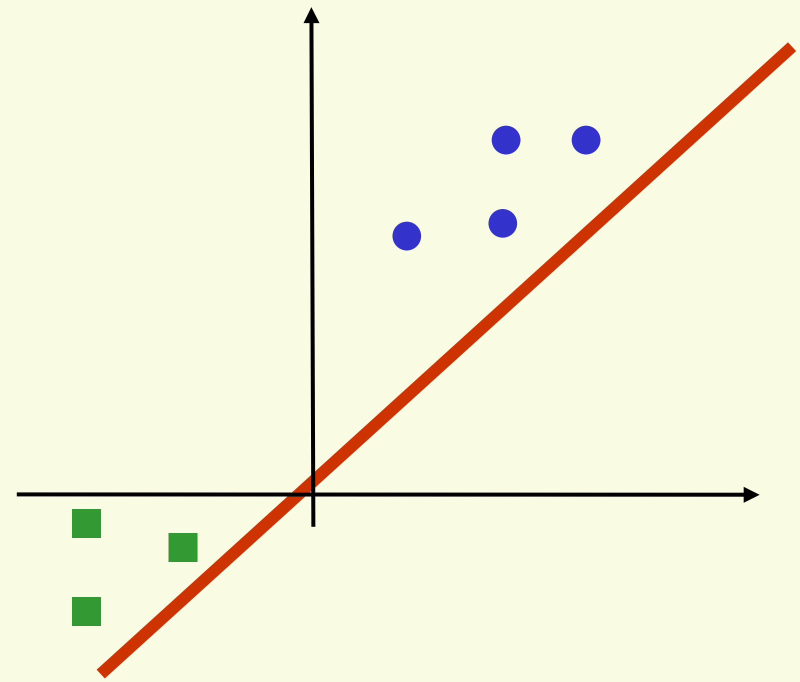
Effect of Normalization

before normalization



seek a hyperplane that separates samples from different categories

after normalization

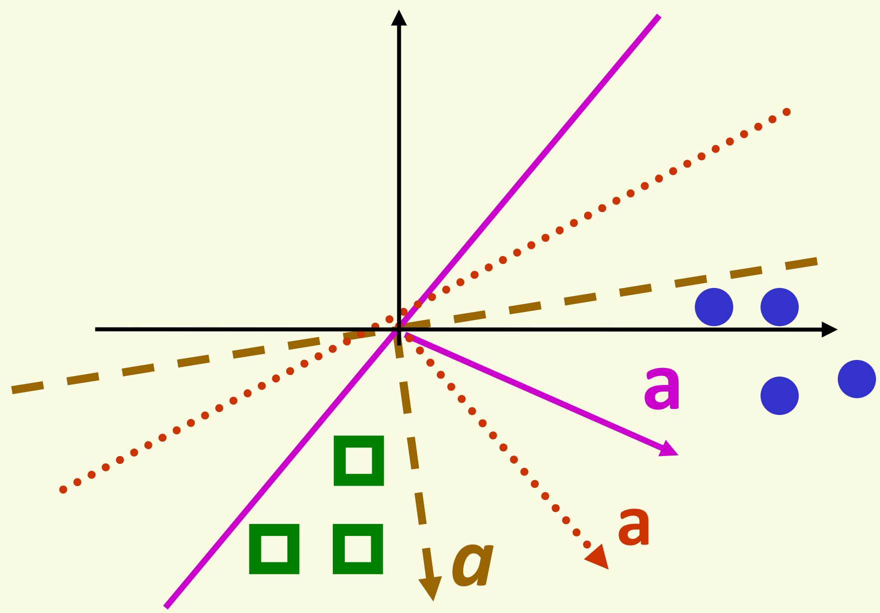


seek hyperplane that puts normalized samples on the same (positive) side

Solution Region

- Find weight vector **a** s.t. for all samples $\mathbf{z}^1,\ldots,\mathbf{z}^n$

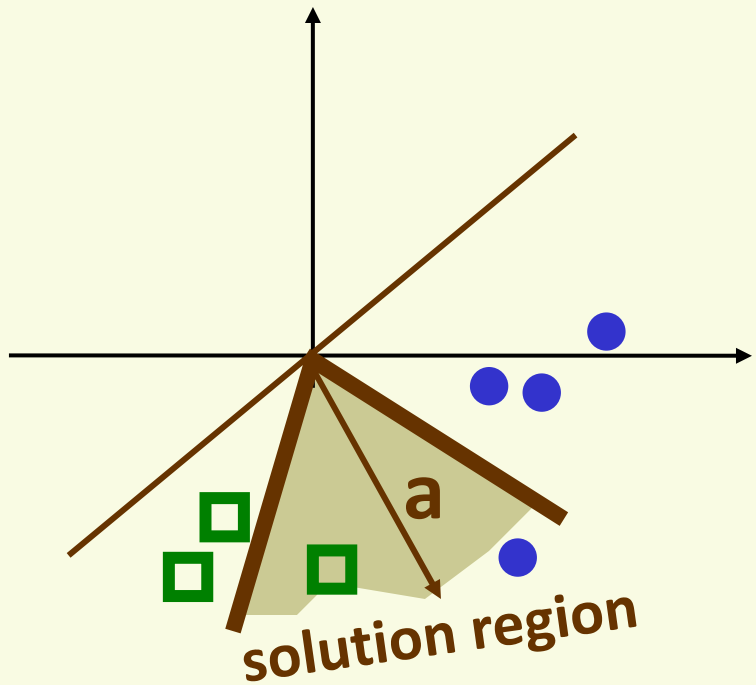
$$\mathbf{a^t z^i} = \sum_{k=0}^{d} \mathbf{a_k z_d^i} > \mathbf{0}$$



- If there is one such **a**, then there are infinitely many **a**

Solution Region

- Solution region: the set of all possible solutions for **a**



Minimum Squared Error Optimization (MSE)

- Linear Regression is a very well understood problem
- Problem is not regression, but let's convert to regression!

$\mathbf{a}^t\mathbf{z}^i > 0$ for all samples $\mathbf{z}^i$
solve system of linear inequalities

↓

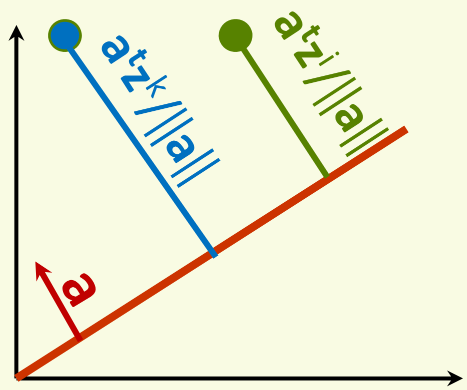
$\mathbf{a}^t\mathbf{z}^i = \boldsymbol{b}_i$ for all samples $\mathbf{z}^i$
solve system of linear equations

- MSE procedure
  - choose positive constants $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$
  - try to find weight vector $\mathbf{a}$ s.t. $\mathbf{a}^t\mathbf{z}^i = \mathbf{b}_i$ for all samples $\mathbf{z}^i$
  - if succeed, then $\mathbf{a}$ is a solution because $\mathbf{b}_i$'s are positive
  - consider all the samples (not just the misclassified ones)

MSE: Margins

- By setting $\mathbf{a}^t\mathbf{z}^i = \mathbf{b}_i$, we expect $\mathbf{z}^i$ to be at a relative distance $\mathbf{b}_i$ from the separating hyperplane
- Thus $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ are expected relative distances of examples from the separating hyperplane
- Should make $\mathbf{b}_i$ small if sample $\mathbf{i}$ is expected to be near separating hyperplane, and make $\mathbf{b}_i$ larger otherwise
- In the absence of any such information, there are good reasons to set

$$\mathbf{b}_1 = \mathbf{b}_2 = \ldots = \mathbf{b}_n = 1$$



MSE: Matrix Notation

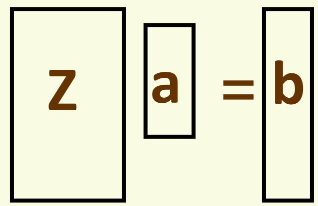
- Solve system of **n** equations $\begin{cases} \mathbf{a}^t \mathbf{z}^1 = \mathbf{b}_1 \\ \vdots \\ \mathbf{a}^t \mathbf{z}^n = \mathbf{b}_n \end{cases}$
- Using matrix notation:

$$\underbrace{\begin{bmatrix} \mathbf{z}_0^1 & \mathbf{z}_1^1 & \cdots & \mathbf{z}_\mathbf{d}^1 \\ \mathbf{z}_0^2 & \mathbf{z}_1^2 & \cdots & \mathbf{z}_\mathbf{d}^2 \\ \vdots & & & \vdots \\ \vdots & & & \vdots \\ \mathbf{z}_0^\mathbf{n} & \mathbf{z}_1^\mathbf{n} & \cdots & \mathbf{z}_\mathbf{d}^\mathbf{n} \end{bmatrix}}_{\mathbf{Z}} \underbrace{\begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_\mathbf{d} \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \vdots \\ \mathbf{b}_\mathbf{n} \end{bmatrix}}_{\mathbf{b}}$$

- Solve a linear system **Za** = **b**

MSE:Approximate Solution

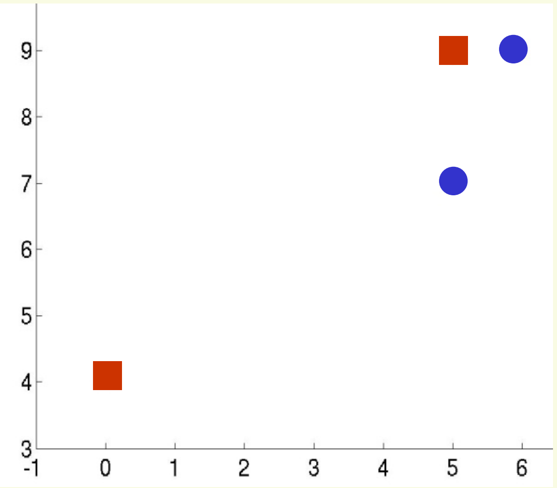
- Typically **Z** is overdetermined
  - more rows (examples) than columns (features)



- No exact solution for **Za** = **b** in this case
- Find an approximate solution **a**, that is $\mathbf{Za} \approx \mathbf{b}$
  - approximate solution **a** does not necessarily give a separating hyperplane in the separable case
  - but hyperplane corresponding to an approximate **a** may still be a good solution
- Least Squares Solution: $\mathbf{a} = (\mathbf{Z}^t\mathbf{Z})^{-1}\,\mathbf{Z}^t\mathbf{b}$

MSE: Example

- Class 1: (6 9), (5 7)
- Class 2: (5 9), (0 4)
- Add extra feature and "normalize"

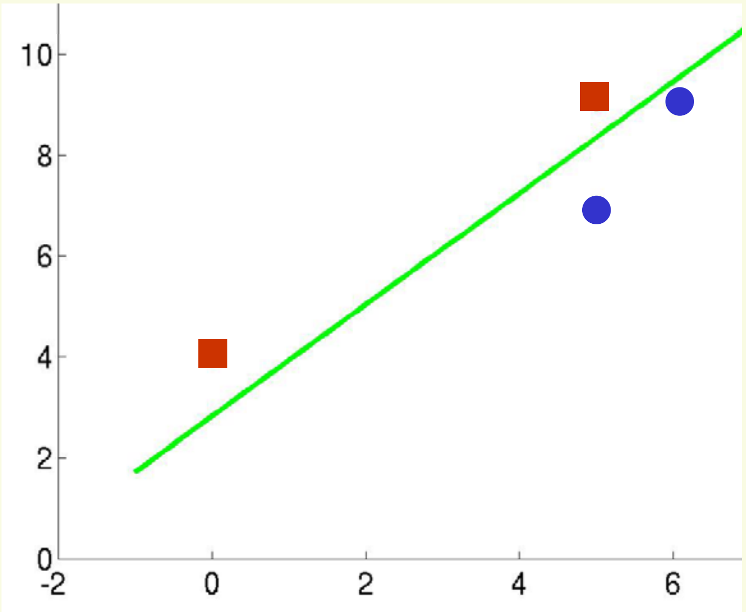


$$\mathbf{z}^1 = \begin{bmatrix} 1 \\ 6 \\ 9 \end{bmatrix} \quad \mathbf{z}^2 = \begin{bmatrix} 1 \\ 5 \\ 7 \end{bmatrix} \quad \mathbf{z}^3 = \begin{bmatrix} -1 \\ -5 \\ -9 \end{bmatrix} \quad \mathbf{z}^4 = \begin{bmatrix} -1 \\ 0 \\ -4 \end{bmatrix}$$

- $$\mathbf{Z} = \begin{bmatrix} 1 & 6 & 9 \\ 1 & 5 & 7 \\ -1 & -5 & -9 \\ -1 & 0 & -4 \end{bmatrix}$$

MSE: Example

- Choose $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$



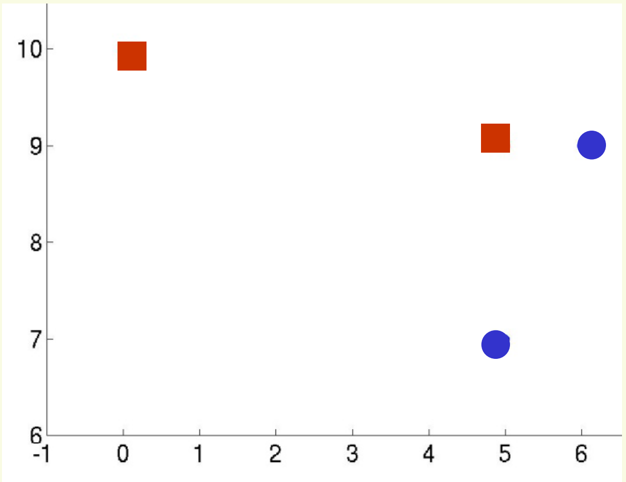
- Use **a=Z\b** to solve in Matlab

$$\mathbf{a} = \begin{bmatrix} 2.7 \\ 1.0 \\ -0.9 \end{bmatrix}$$

- Note **a** is an approximation since $\mathbf{Za} = \begin{bmatrix} 0.4 \\ 1.3 \\ 0.6 \\ 1.1 \end{bmatrix} \neq \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$
- Gives a separating hyperplane since $\mathbf{Za} > 0$

MSE: Example

- Class 1: (6 9), (5 7)
- Class 2: (5 9), (0 10)
- One example is far compared to others from separating hyperplane

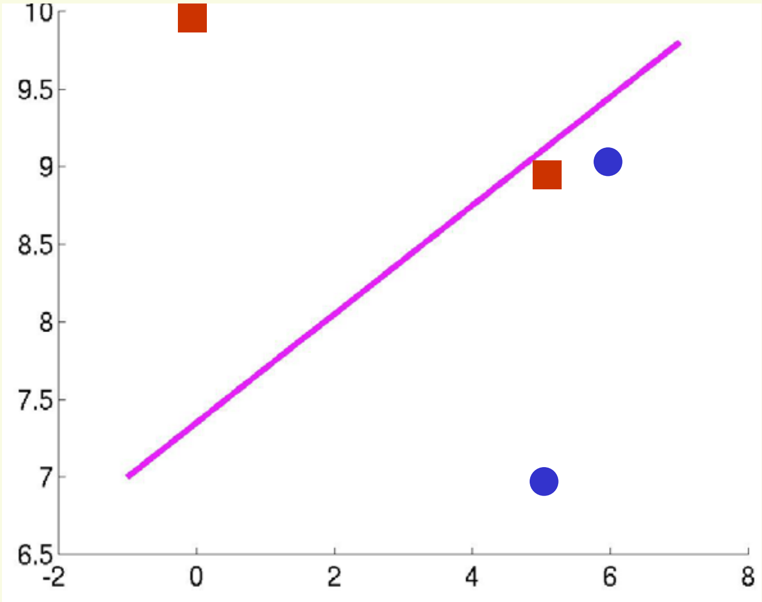


$$\mathbf{z}^1 = \begin{bmatrix} 1 \\ 6 \\ 9 \end{bmatrix} \quad \mathbf{z}^2 = \begin{bmatrix} 1 \\ 5 \\ 7 \end{bmatrix} \quad \mathbf{z}^3 = \begin{bmatrix} -1 \\ -5 \\ -9 \end{bmatrix} \quad \mathbf{z}^4 = \begin{bmatrix} -1 \\ 0 \\ -10 \end{bmatrix}$$

- $\mathbf{Z} = \begin{bmatrix} 1 & 6 & 9 \\ 1 & 5 & 7 \\ -1 & -5 & -9 \\ -1 & 0 & -10 \end{bmatrix}$

MSE: Example

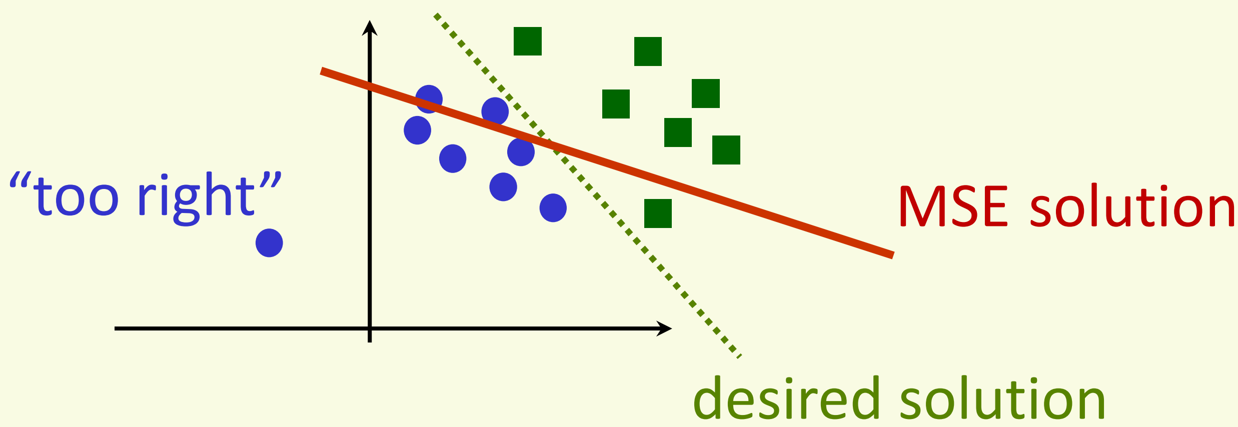
- Choose $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$



- Solve $\mathbf{a} = \mathbf{Z} \backslash \mathbf{b} = \begin{bmatrix} 3.2 \\ 0.2 \\ -0.4 \end{bmatrix}$
- $\mathbf{Za} = \begin{bmatrix} 0.2 \\ 0.9 \\ -0.04 \\ 1.16 \end{bmatrix} \neq \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$
- Does not give a separating hyperplane since $\mathbf{a}^t \mathbf{z}^3 < \boldsymbol{0}$

MSE: Problems

- MSE wants all examples to be at the same distance from the separating hyperplane
- Examples that are “too right”, i.e. too far from the boundary cause problems



- No problems with convergence though, both in separable and non-separable cases
- Can fix it in linearly separable case, i.e find better **b**

Another Approach: Design a Loss Function

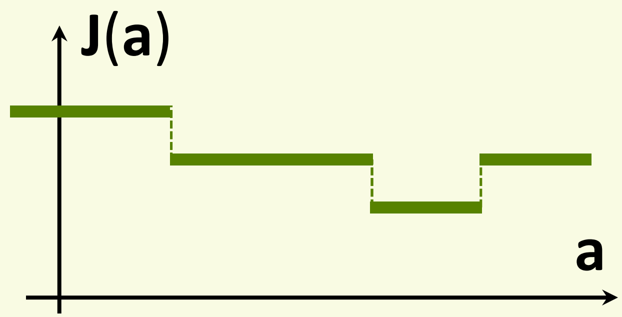
- Find weight vector **a** s.t. $\forall \mathbf{z}^1, \ldots, \mathbf{z}^n$, $\mathbf{a}^t \mathbf{z}^i > 0$
- Design a loss function **J**(**a**), which is minimum when **a** is a solution vector
- Let **Z**(**a**) be the set of examples misclassified by **a**

$$\mathbf{Z}(\mathbf{a}) = \{ \mathbf{z}^i \mid \mathbf{a}^t \mathbf{z}^i < 0 \}$$

- Natural choice: number of misclassified examples

$$\mathbf{J}(\mathbf{a}) = |\mathbf{Z}(\mathbf{a})|$$

- Unfortunately, can't be minimized with gradient descent
  - piecewise constant, gradient zero or does not exist

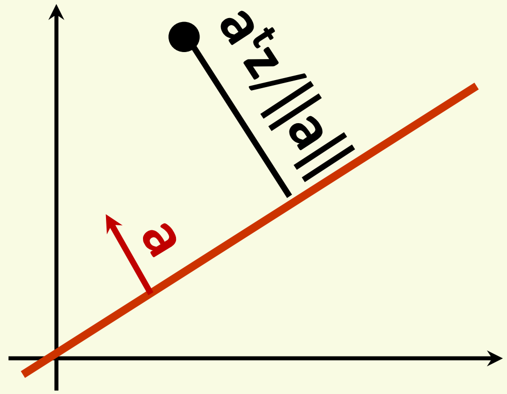


Perceptron Loss Function

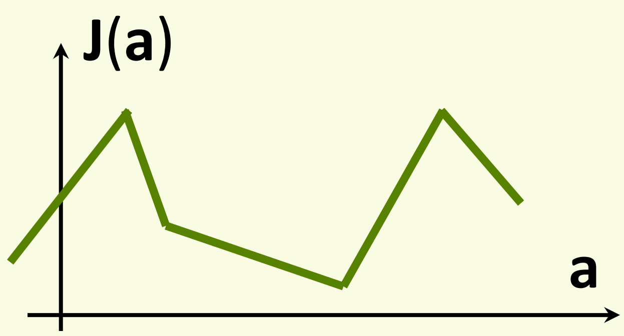
- Better choice: Perceptron loss function

$$J_p(\mathbf{a}) = \sum_{\mathbf{z} \in Z(\mathbf{a})} \left(-\mathbf{a}^t\mathbf{z}\right)$$

- If $\mathbf{z}$ is misclassified, $\mathbf{a}^t\mathbf{z} < 0$
- Thus $\mathbf{J}(\mathbf{a}) \geq 0$
- $\mathbf{J}_p(\mathbf{a})$ is proportional to the sum of distances of misclassified examples to decision boundary



- $\mathbf{J}_p(\mathbf{a})$ is piecewise linear and suitable for gradient descent



Optimizing with Gradient Descent

$$J_p(\mathbf{a}) = \sum_{\mathbf{z} \in Z(\mathbf{a})} \left(-\mathbf{a}^t \mathbf{z}\right)$$

- Gradient of $\mathbf{J}_p(\mathbf{a})$ is $\nabla \mathbf{J}_p(\mathbf{a}) = \sum_{\mathbf{z} \in Z(\mathbf{a})} (-\mathbf{z})$
  - cannot solve $\nabla \mathbf{J}_p(\mathbf{a}) = 0$ analytically because of $\mathbf{Z}(\mathbf{a})$
- Recall update rule for gradient descent

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k+1)} - \alpha \nabla \mathbf{J}(\mathbf{x}^{(k)})$$

- Gradient decent update rule for $\mathbf{J}_p(\mathbf{a})$ is:

$$\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} + \boldsymbol{\alpha} \sum_{\mathbf{z} \in Z(\mathbf{a})} \mathbf{z}$$

  - called **batch rule** because it is based on all examples
  - true gradient descent

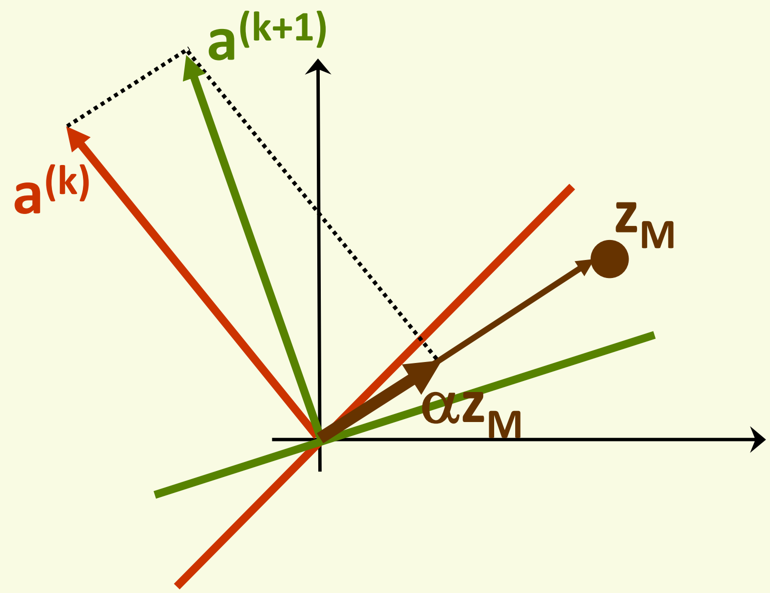
Perceptron Single Sample Rule

- Gradient decent single sample rule for $\mathbf{J}_p(\mathbf{a})$ is

$$\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} + \alpha \cdot \mathbf{z}_M$$

  - $\mathbf{z}_M$ is one sample misclassified by $\mathbf{a}^{(k)}$
  - must have a consistent way to visit samples

- Geometric Interpretation:



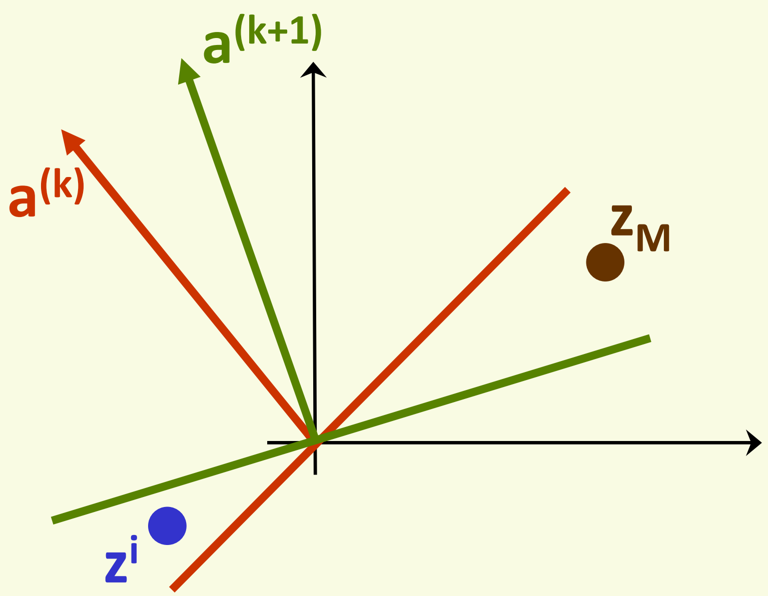
- $\mathbf{z}_M$ misclassified by $\mathbf{a}^{(k)}$

$$\left(\mathbf{a}^{(k)}\right)^t \mathbf{z}_M \le 0$$

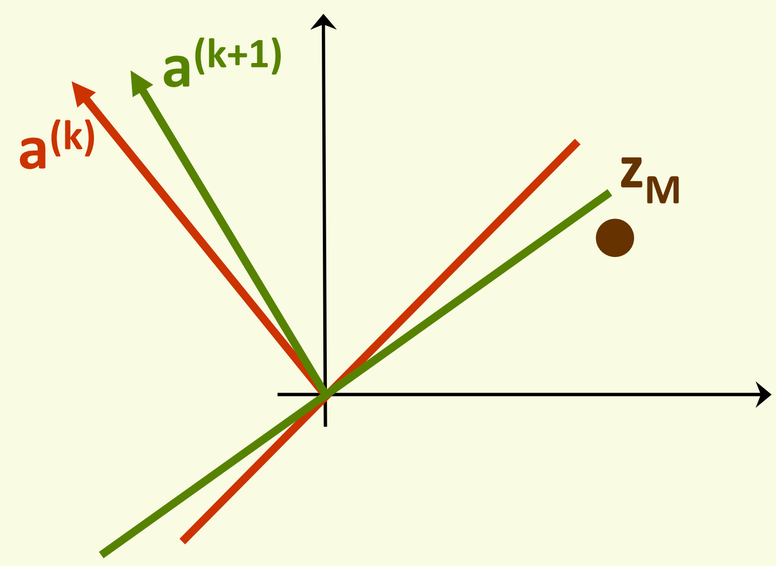
- $\mathbf{z}_M$ is on the wrong side of decision boundary
- adding $\alpha \cdot \mathbf{z}_M$ to a moves decision boundary in the right direction

Perceptron Single Sample Rule

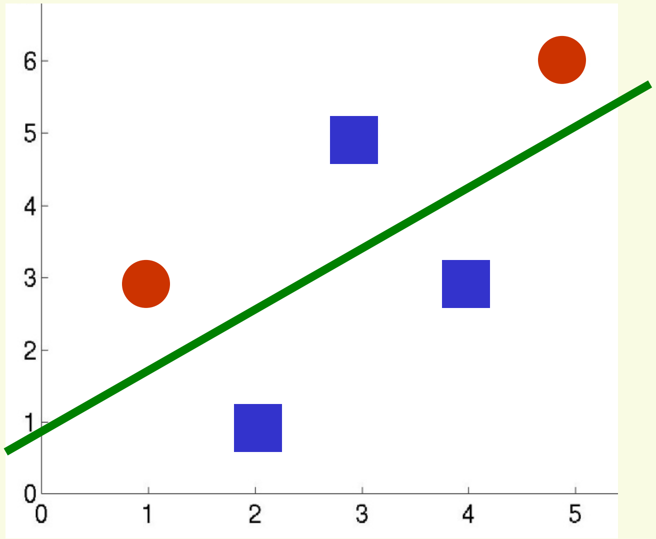
if α is too large, previously correctly classified sample $\mathbf{z}^i$ is now misclassified



if α is too small, $\mathbf{z}_M$ is still misclassified



Non-Linearly Separable Case

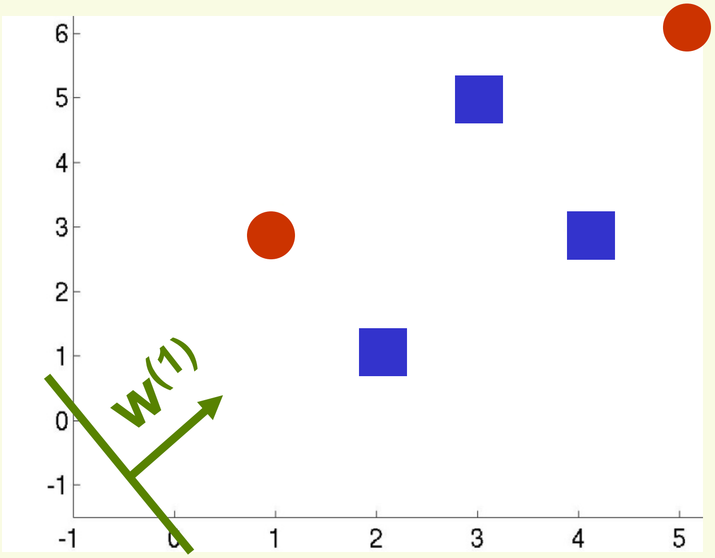


- Suppose we have examples:
  - class 1: [2,1], [4,3], [3,5]
  - class 2: [1,3] , [5,6]
  - not linearly separable
- Still would like to get approximate separation
- Good line choice is shown in green
- Let us run gradient descent
  - Add extra feature and "normalize"

$$\mathbf{z}^1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad \mathbf{z}^2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \quad \mathbf{z}^3 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad \mathbf{z}^4 = \begin{bmatrix} -1 \\ -1 \\ -3 \end{bmatrix} \quad \mathbf{z}^5 = \begin{bmatrix} -1 \\ -5 \\ -6 \end{bmatrix}$$

Non-Linearly Separable Case

- single sample perceptron rule
- Initial weights $\mathbf{a}^{(1)} = [1 \;\; 1 \;\; 1]$
- This is line $\mathbf{x}_1 + \mathbf{x}_2 + 1 = 0$
- Use fixed learning rate $\alpha = 1$
- Rule is: $\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} + \mathbf{z}_M$



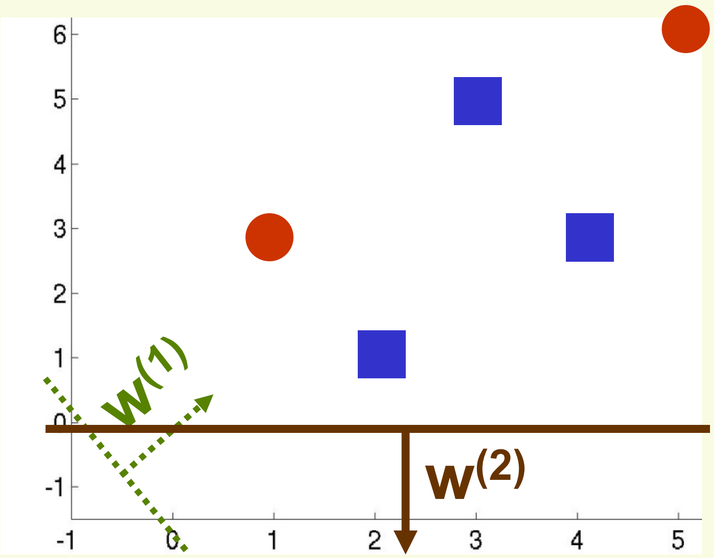
$$\mathbf{z}^1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad \mathbf{z}^2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \quad \mathbf{z}^3 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad \mathbf{z}^4 = \begin{bmatrix} -1 \\ -1 \\ -3 \end{bmatrix} \quad \mathbf{z}^5 = \begin{bmatrix} -1 \\ -5 \\ -6 \end{bmatrix}$$

- $\mathbf{a}^t\mathbf{z}^1 = [1\ 1\ 1] \cdot [1\ 2\ 1]^t > 0$
- $\mathbf{a}^t\mathbf{z}^2 = [1\ 1\ 1] \cdot [1\ 4\ 3]^t > 0$
- $\mathbf{a}^t\mathbf{z}^3 = [1\ 1\ 1] \cdot [1\ 3\ 5]^t > 0$

Non-Linearly Separable Case

- $\mathbf{a}^{(1)} = [1\ \ 1\ \ 1]$
- rule is: $\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} + \mathbf{z}_M$

$$\mathbf{z}^1 = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad \mathbf{z}^2 = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \quad \mathbf{z}^3 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad \mathbf{z}^4 = \begin{bmatrix} -1 \\ -1 \\ -3 \end{bmatrix} \quad \mathbf{z}^5 = \begin{bmatrix} -1 \\ -5 \\ -6 \end{bmatrix}$$



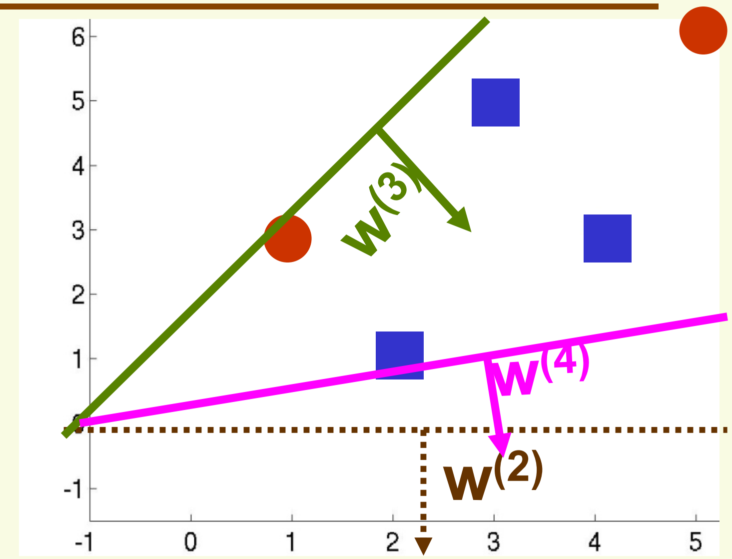
- $\mathbf{a}^t\mathbf{z}^4 = [1\ 1\ 1] \cdot [-1\ -1\ -3]^t = -5 < 0$
- *Update:* $\mathbf{a}^{(2)} = \mathbf{a}^{(1)} + \mathbf{z}_M = [1\ \ 1\ \ 1] + [-1\ \ -1\ \ -3] = [0\ \ 0\ \ -2]$

- $\mathbf{a}^t\mathbf{z}^5 = [0\ 0\ -2] \cdot [-1\ -5\ -6]^t = 12 > 0$
- $\mathbf{a}^t\mathbf{z}^1 = [0\ 0\ -2] \cdot [1\ 2\ 1]^t < 0$
- *Update:* $\mathbf{a}^{(3)} = \mathbf{a}^{(2)} + \mathbf{z}_M = [0\ \ 0\ \ -2] + [1\ \ 2\ \ 1] = [1\ \ 2\ \ -1]$

Non-Linearly Separable Case

- $\mathbf{a}^{(3)} = [1 \;\; 2 \;\; -1]$
- rule is: $\mathbf{a}^{(k+1)} = \mathbf{a}^{(k)} + \mathbf{z}_M$

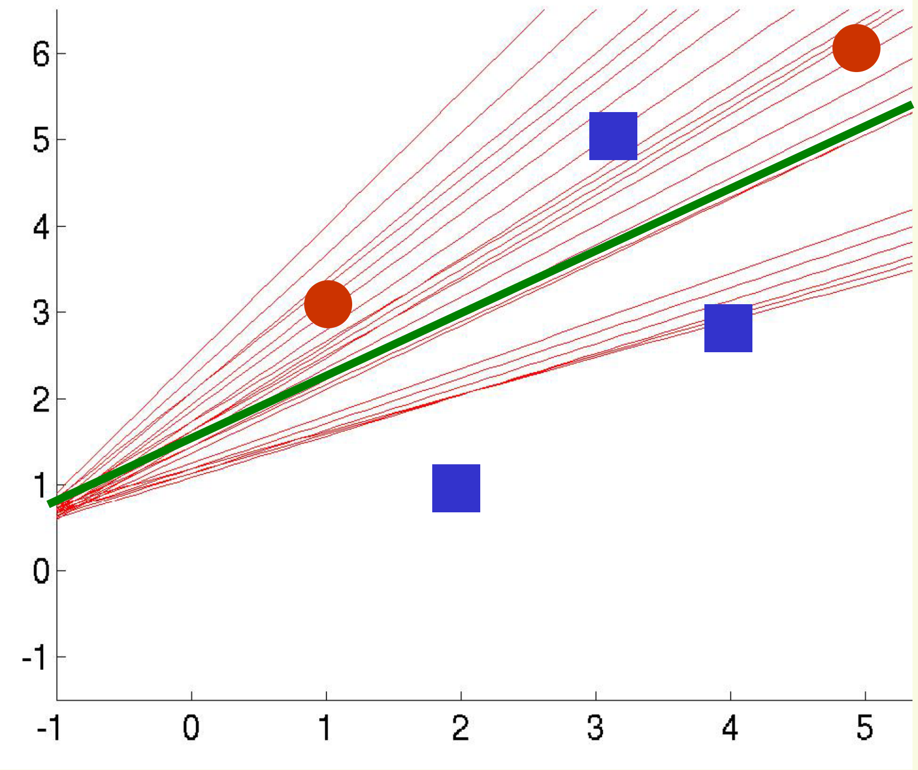
$$\mathbf{z}^1 = \begin{bmatrix}1\\2\\1\end{bmatrix} \quad \mathbf{z}^2 = \begin{bmatrix}1\\4\\3\end{bmatrix} \quad \mathbf{z}^3 = \begin{bmatrix}1\\3\\5\end{bmatrix} \quad \mathbf{z}^4 = \begin{bmatrix}-1\\-1\\-3\end{bmatrix} \quad \mathbf{z}^5 = \begin{bmatrix}-1\\-5\\-6\end{bmatrix}$$



- $\mathbf{a}^t\mathbf{z}^2 = [1\;4\;3] \cdot [1 \;\; 2\; -1]^t = 6 > 0$
- $\mathbf{a}^t\mathbf{z}^3 = [1\;3\;5] \cdot [1 \;\; 2\; -1]^t = 2 > 0$
- $\mathbf{a}^t\mathbf{z}^4 = [-1\;-1\;-3] \cdot [1 \;\; 2\; -1]^t = 0$
- *Update:* $\mathbf{a}^{(4)} = \mathbf{a}^{(3)} + \mathbf{z}_M = [1 \;\; 2 \;\; -1] + [-1 \;\; -1 \;\; -3] = [0 \;\; 1 \;\; -4]$

Non-Linearly Separable Case

- We can continue this forever
  - there is no solution vector **a** satisfying for all $\mathbf{a}^t\mathbf{z}_i > 0$ for all **i**
- Need to stop at a good point
- Solutions at iterations 900 through 915
- Some are good some are not
- How do we stop at a good solution?



Convergence of Perceptron Rules

1. Classes are linearly separable:
    - with fixed learning rate, both single sample and batch rules converge to a correct solution **a**
    - can be any **a** in the solution space
2. Classes are not linearly separable:
    - with fixed learning rate, both single sample and batch do not converge
    - can ensure convergence with appropriate variable learning rate
        - $\alpha \rightarrow 0$ as $\mathbf{k} \rightarrow \infty$
        - example, inverse linear: $\alpha = \mathbf{c/k}$, where **c** is any constant
            - also converges in the linearly separable case
        - no guarantee that we stop at a good point, but there are good reasons to choose inverse linear learning rate

- Practical Issue: both single sample and batch algorithms converge faster if features are roughly on the same scale
    - see kNN lecture on feature normalization

Batch vs. Single Sample Rules

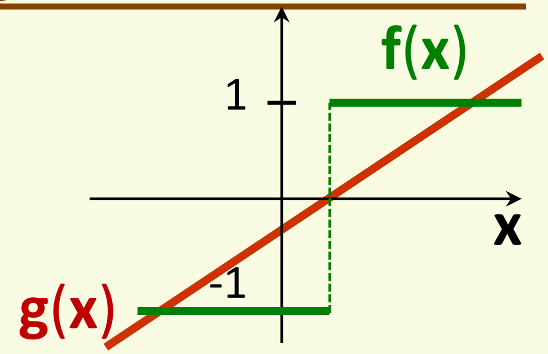
Batch

- True gradient descent, full gradient computed
- Smoother gradient because all samples are used
- Takes longer to converge

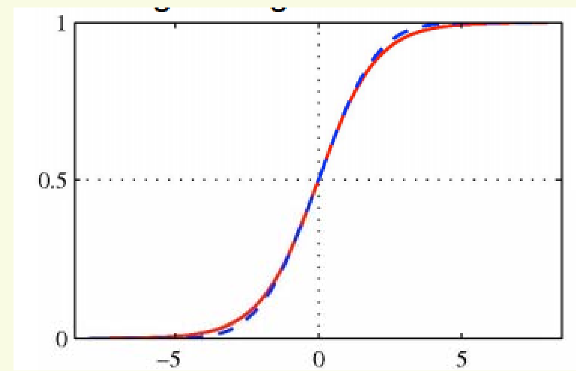
Single Sample

- Only partial gradient is computed
- Noisier gradient, therefore may concentrates more than necessary on any isolated training examples (those could be noise)
- Converges faster
- Easier to analyze

Linear Machine: Logistic Regression



- Despite the name, used for classification, not regression
- Instead of putting **g**(**x**) through a sign function, can put it through a smooth function
  - smooth function is better for gradient descent
- Logistic sigmoid function

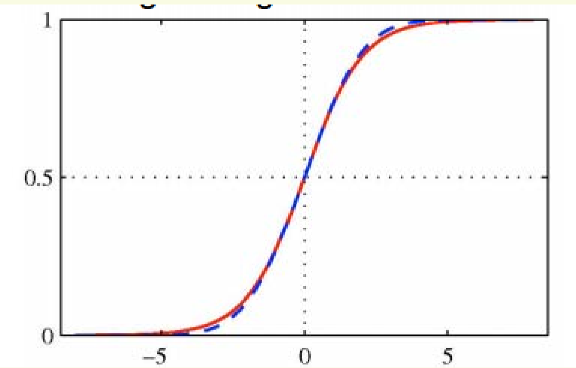


- $\mathbf{g}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\mathbf{x}_1\mathbf{w}_1 + \dots + \mathbf{x}_d\mathbf{w}_d$
- let $\mathbf{f}(\mathbf{x},\mathbf{w}) = \sigma(\mathbf{g}(\mathbf{x},\mathbf{w}))$

$$\sigma(a) = \frac{1}{1+exp(-a)}$$

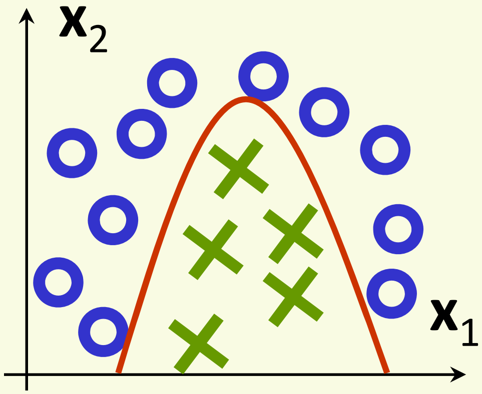
Linear Machine: Logistic Regression

- **f**(**x**,**w**) = σ(**g**(**x**,**w**))
  - bigger 0.5 if **g**(**x**,**w**) is positive
    - decide class 1
  - less 0.5 if **g**(**x**,**w**) is negative
    - decide class 2
- Has an interesting probabilistic interpretation
- P(class 1|**x**) = σ(**g**(**x**,**w**))
- Under a certain loss function, can be optimized exactly with gradient decent



$$\sigma(a) = \frac{1}{1 + exp(-a)}$$

Generalized Linear Classifier



- Can use other discriminant functions, like quadratics

  $\mathbf{g}(\mathbf{x}) = \mathbf{w}_0+\mathbf{w}_1\mathbf{x}_1+\mathbf{w}_2\mathbf{x}_2+ \mathbf{w}_{12}\mathbf{x}_1\mathbf{x}_2 +\mathbf{w}_{11}\mathbf{x}_1^{\ 2} +\mathbf{w}_{22}\mathbf{x}_2^{\ 2}$

- Methodology is almost the same as in the linear case:
  - $\mathbf{f}(\mathbf{x}) = \text{sign}(\mathbf{w}_0+\mathbf{w}_1\mathbf{x}_1+\mathbf{w}_2\mathbf{x}_2+\mathbf{w}_{12}\mathbf{x}_1\mathbf{x}_2 +\mathbf{w}_{11}\mathbf{x}_1^{\ 2} + \mathbf{w}_{22}\mathbf{x}_2^{\ 2})$
  - $\mathbf{z} = [\,1 \quad \mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_1\mathbf{x}_2 \quad \mathbf{x}_1^{\ 2} \quad \mathbf{x}_2^{\ 2}]$
  - $\mathbf{a} = [\,\mathbf{w}_0 \quad \mathbf{w}_1 \quad \mathbf{w}_2 \quad \mathbf{w}_{12} \quad \mathbf{w}_{11} \quad \mathbf{w}_{22}]$
  - "normalization": multiply negative class samples by -1
  - all the other procedures remain the same, i.e. gradient descent to minimize Perceptron loss function, or MSE procedure, etc.

Generalized Linear Classifier

- In general, to the liner function:

$$\mathbf{g}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\Sigma_{i=1\ldots d}\ \mathbf{w}_i\mathbf{x}_i$$

- can add quadratic terms:

$$\mathbf{g}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\Sigma_{i=1\ldots d}\ \mathbf{w}_i\mathbf{x}_i+\Sigma_{i=1\ldots d}\ \Sigma_{j=1,..d}\ \mathbf{w}_{ij}\mathbf{x}_i\mathbf{x}_j$$

- This is still a linear function in its parameters **w**
- $\mathbf{g}(\mathbf{y},\mathbf{v}) = \mathbf{v}_0+\mathbf{v}^t\mathbf{y}$

$$\mathbf{v}_0 = \mathbf{w}_0$$

$$\mathbf{y} = [\mathbf{x}_1 \quad \mathbf{x}_2 \ldots \quad \mathbf{x}_d \quad \mathbf{x}_1\mathbf{x}_1 \quad \mathbf{x}_1\mathbf{x}_2 \quad \ldots \quad \mathbf{x}_d\mathbf{x}_d]$$

$$\mathbf{v} = [\mathbf{w}_1 \quad \mathbf{w}_2 \ldots \mathbf{w}_d \quad \mathbf{w}_{11} \quad \mathbf{w}_{12} \quad \ldots \quad \mathbf{w}_{dd}]$$

- Can use all the same training methods as before

Generalized Linear Classifier

- Generalized linear classifier

$$\mathbf{g}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\sum_{i=1\ldots m} \mathbf{w}_i\mathbf{h}_i(\mathbf{x})$$

- $\mathbf{h}(\mathbf{x})$ are called basis function, can be arbitrary functions
  - in strictly linear case, $\mathbf{h}_i(\mathbf{x})= \mathbf{x}_i$

- Linear function in its parameters $\mathbf{w}$

$$\mathbf{g}(\mathbf{x},\mathbf{w}) = \mathbf{w}_0+\mathbf{w}^t\mathbf{h}$$

$$\mathbf{h} = [\mathbf{h}_1(\mathbf{x}) \;\; \mathbf{h}_2(\mathbf{x}) \;\; \ldots \;\; \mathbf{h}_m(\mathbf{x})]$$

$$[\mathbf{w}_1 \;\; \ldots \;\; \mathbf{w}_m]$$

- Can use all the same training methods as before

Generalized Linear Classifier

- Usually face severe overfitting
  - too many degrees of freedom
  - Boundary can "curve" to fit to the noise in the data
- Helps to regularize by keeping **w** small
  - small **w** means the boundary is not as curvy
- Usually add $\boldsymbol{\lambda}||\mathbf{w}||^2$ to the loss function
- Recall quadratic loss function

$$\mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w}) = ||\ \mathbf{f}(\mathbf{x}^i,\mathbf{w}) - \mathbf{y}^i\ ||^2$$

- Regularized version

$$\mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w}) = ||\ \mathbf{f}(\mathbf{x}^i,\mathbf{w}) - \mathbf{y}^i\ ||^2 + \boldsymbol{\lambda}||\mathbf{w}||^2$$

- How to set $\boldsymbol{\lambda}$?
- With cross-validation

Learning by Gradient Descent

- Can have classifiers even more general
- More general than generalized linear ☺
- Suppose we suspect that the machine has to have functional form $\mathbf{f}(\mathbf{x},\mathbf{w})$, not necessarily linear
- Pick differentiable per-sample loss function $\mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w})$
- Need to find $\mathbf{w}$ that minimizes $\mathbf{L} = \Sigma_i \mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w})$
- Use gradient-based minimization:
  - Batch rule: $\mathbf{w} = \mathbf{w} - \alpha \nabla \mathbf{L}(\mathbf{w})$
  - Or single sample rule: $W = W - \alpha \nabla \mathbf{L}(\mathbf{x}^i,\mathbf{y}^i,\mathbf{w})$

Information theory

- Information Theory regards information as only those symbols that are uncertain to the receiver
  - **only infrmatn esentil to understnd mst b tranmitd**
- Shannon made clear that uncertainty is the very commodity of communication
- The amount of information, or uncertainty, output by an information source is a measure of its entropy
- In turn, a source's entropy determines the amount of bits per symbol required to encode the source's information
- Messages are encoded with strings of 0 and 1 (bits)

Information theory

- Suppose we toss a **fair** die with 8 sides
  - need 3 bits to transmit the results of each toss
  - 1000 throws will need 3000 bits to transmit
- Suppose the die is biased
  - side A occurs with probability 1/2, chances of throwing B are 1/4, C are 1/8, D are 1/16, E are 1/32, F 1/64, G and H are 1/128
  - Encode A= 0, B = 10, C = 110, D = 1110,…, so on until G = 1111110, H = 1111111
  - We need, on average, 1/2+2/4+3/8+4/16+5/32+6/64+7/128+7/128 = 1.984 bits to encode results of a toss
  - 1000 throws require 1984 bits to transmit
  - Less bits to send = less "information"
  - Biased die tosses contain less "information" than unbiased die tosses (know in advance biased sequence will have a lot of A's)
  - What's the number of bits in the best encoding?
- Extreme case: if a die always shows side A, a sequence of 1,000 tosses has no information, 0 bits to encode

Information theory

- if a die is fair (any side is equally likely, or uniform distribution), for any toss we need log(8) = 3 bits
- Suppose any of n events is equally likely (uniform distribution)
  - P(x) = 1/n, therefore -log P = -log(1/n) = log n
- In the "good" encoding strategy for our biased die example, every side x has -log p(x) bits in its code
- Expected number of bits is

$$-\sum_{x} p(x) \log p(x)$$

Shannon's Entropy

$$H[p(x)] = -\sum_x p(x) \log p(x) = \sum_x p(x) \log \frac{1}{p(x)}$$

- How much randomness (or uncertainty) is there in the value of signal x if it has distribution p(x)
  - For uniform distribution (every event is equally likely), H[x] is maximum
  - If p(x) = 1 for some event x, then H[x] = 0
  - Systems with one very common event have less entropy than systems with many equally probable events
- Gives the expected length of optimal encoding (in binary bits) of a message following distribution p(x)
  - doesn't actually give this optimal encoding

Conditional Entropy of X given Y

$$H[x \mid y] = \sum_{x,y} p(x,y) \log \frac{1}{p(x \mid y)} = -\sum_{x,y} p(x,y) \log p(x \mid y)$$

- Measures average uncertainty about x when y is known
- Property:
  - $H[x] \geq H[x|y]$, which means after seeing new data (y), the uncertainty about x is not increased, on average

Mutual Information of X and Y

$$I[x,y] = H(x) - H(x \mid y)$$

- Measures the average reduction in uncertainty about x after y is known
- or, equivalently, it **measures the amount of information that y conveys about x**
- Properties
  - I(x,y) = I(y,x)
  - I(x,y) $\geq$ 0
  - If x and y are independent, then I(x,y) = 0
  - I(x,x) = H(x)

MI for Feature Selection

$$I[x,c] = H(c) - H(c \mid x)$$

- Let x be a proposed feature and c be the class
- If I[x,c] is high, we can expect feature x be good at predicting class c