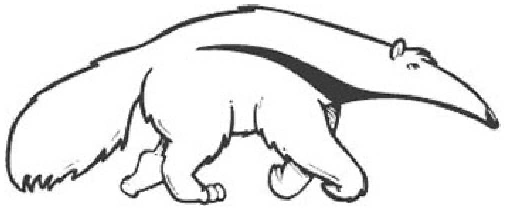
Introduction to Artificial Intelligence

CS171, Spring Quarter, 2020
Introduction to Artificial Intelligence
Prof. Richard Lathrop



Read Beforehand: All assigned reading so far

BREN:ICS INFORMATION AND COMPUTER SCIENCES

UNIVERSITY of CALIFORNIA IRVINE

Final Review

- Bayesian Networks: R&N Chap 14.1-14.5
- Game Search: R&N Chap 5.1-5.4
- Constraint Satisfaction: R&N Chap 6.1-6.4, except 6.3.3
- Machine Learning: R&N Chap 18.1-18.12, 20.2

Review Bayesian Networks Chapter 14.1-5

- **Basic concepts and vocabulary of Bayesian networks.**
  - Nodes represent random variables.
  - Directed arcs represent (informally) direct influences.
  - Conditional probability tables, P(Xi | Parents(Xi)).
- **Given a Bayesian network:**
  - Write down the full joint distribution it represents.
- **Given a full joint distribution in factored form:**
  - Draw the Bayesian network that represents it.
- **Given a variable ordering and background assertions of conditional independence among the variables:**
  - Write down the factored form of the full joint distribution, as simplified by the conditional independence assertions.
- **Use the network to find answers to probability questions about it.**

Bayesian Networks

- Represent dependence/independence via a directed graph
  - Nodes = random variables
  - Edges = direct dependence
- Structure of the graph ⇔ Conditional independence

- Recall the chain rule of repeated conditioning:

$$P(X_1, X_2, X_3 ..., X_N) = P(X_1 | X_2, X_3 ..., X_N) P(X_2 | X_3, ..., X_N) \cdots P(X_N)$$

$$\underbrace{P(X_1, X_2, X_3 ..., X_N)}_{\text{The full joint distribution}} = \underbrace{\prod_{i=1}^{n} P(X_i | parents(X_i))}_{\text{The graph-structured approximation}}$$

- Requires that graph is acyclic (no directed cycles)
- 2 components to a Bayesian network
  - The graph structure (conditional independence assumptions)
  - The numerical probabilities (of each variable given its parents)

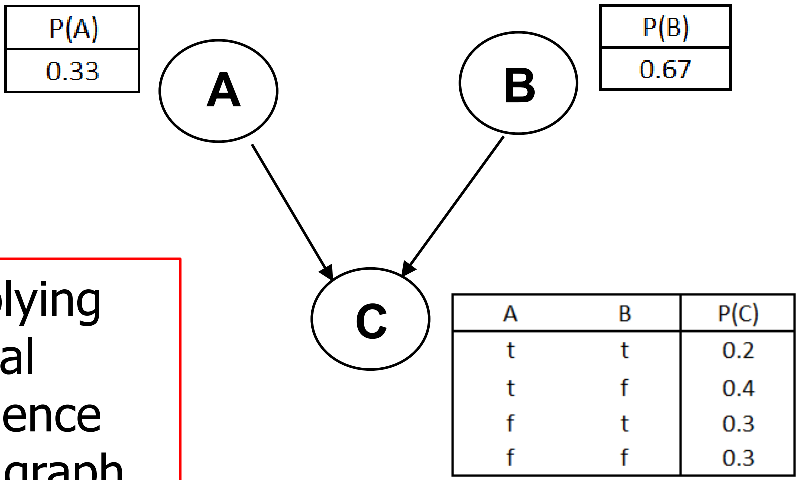
Bayesian Network

- A Bayesian network specifies a joint distribution in a structured form:

Full factorization

$$p(A,B,C) = p(C|A,B)p(A|B)p(B)$$
$$= p(C|A,B)p(A)p(B)$$

After applying conditional independence from the graph



P(A)
0.33

P(B)
0.67

A	B	P(C)
t	t	0.2
t	f	0.4
f	t	0.3
f	f	0.3

- Dependence/independence represented via a directed graph:

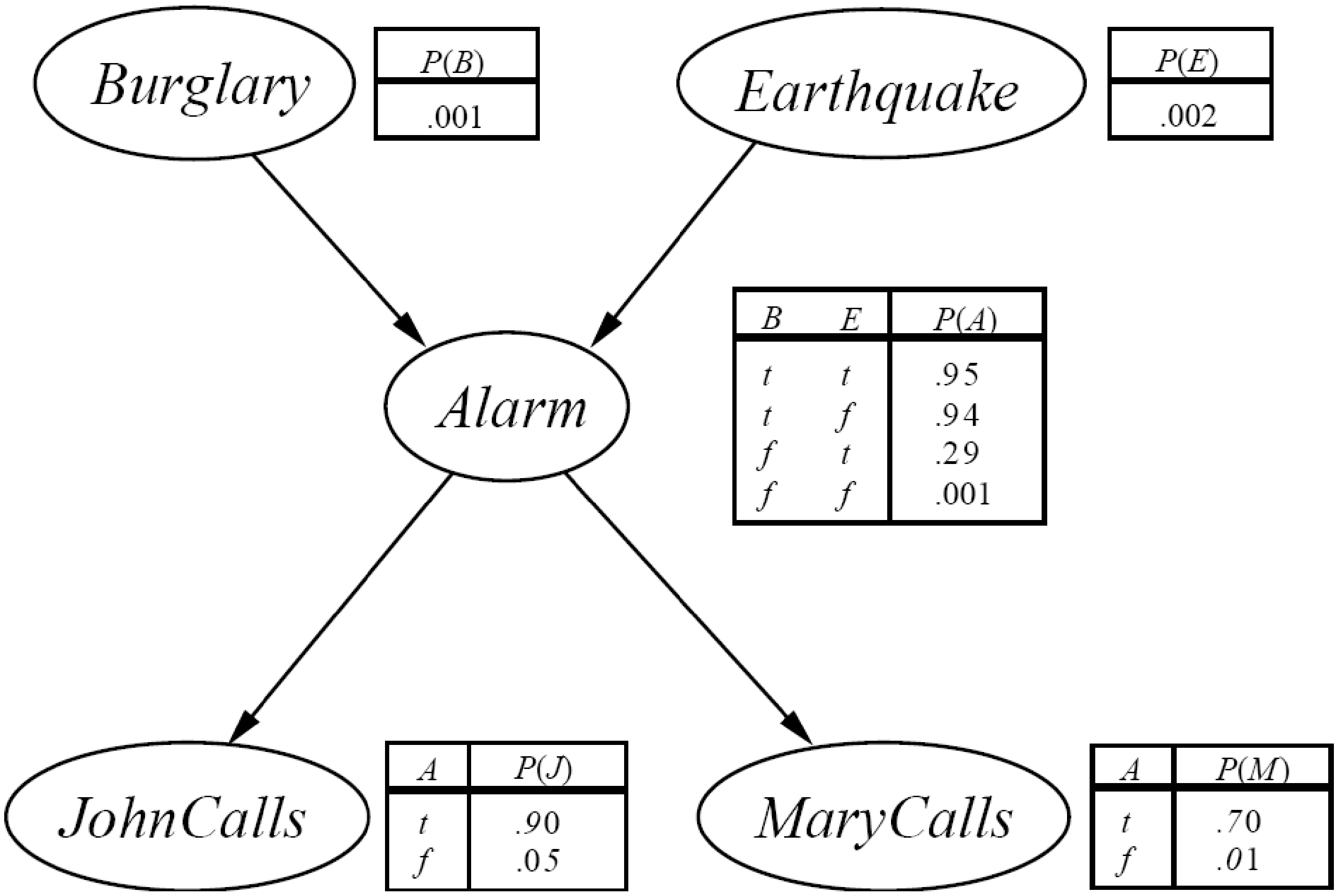
– Node	= random variable
– Directed Edge	= conditional dependence
– Absence of Edge	= conditional independence

- Allows concise view of joint distribution relationships:
  - Graph nodes and edges show conditional relationships between variables.
  - Tables provide probability data.

Burglar Alarm Example

- Consider the following 5 binary variables:
  - B = a burglary occurs at your house
  - E = an earthquake occurs at your house
  - A = the alarm goes off
  - J = John calls to report the alarm
  - M = Mary calls to report the alarm
- Sample Query: What is P(B|M, J) ?
- Using full joint distribution to answer this question requires
  - $2^5 - 1 = 31$ parameters
- Can we use prior domain knowledge to come up with a Bayesian network that requires fewer probabilities?

The Resulting Bayesian Network



P(B)
.001

P(E)
.002

B	*E*	*P(A)*
t	*t*	.95
t	*f*	.94
f	*t*	.29
f	*f*	.001

A	*P(J)*
t	.90
f	.05

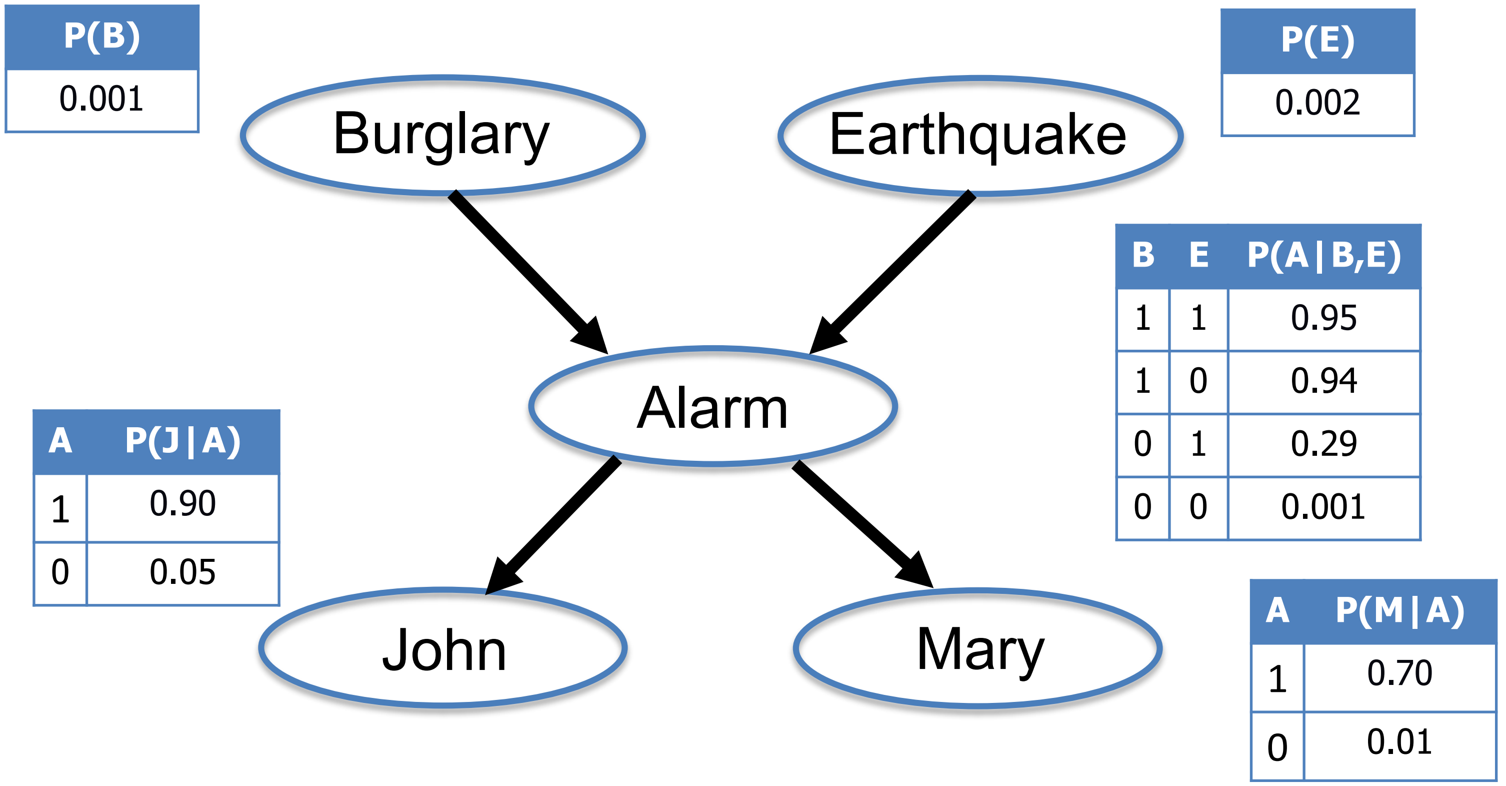
A	*P(M)*
t	.70
f	.01

Example of Answering a Simple Query

- What is P(¬j, m, a, ¬e, b) = P(J = false ∧ M=true ∧ A=true ∧ E=false ∧ B=true)

P(J, M, A, E, B) ≈ P(J | A) P(M | A) P(A| E, B) P(E) P(B) ; by conditional independence

P(¬j, m, a, ¬e, b) ≈ P(¬j | a) P(m | a) P(a| ¬e, b) P(¬e) P(b)
= 0.10 x 0.70 x 0.94 x 0.998 x 0.001 ≈ .0000657



P(B)
0.001

P(E)
0.002

B	E	P(A|B,E)
1	1	0.95
1	0	0.94
0	1	0.29
0	0	0.001

A	P(J|A)
1	0.90
0	0.05

A	P(M|A)
1	0.70
0	0.01

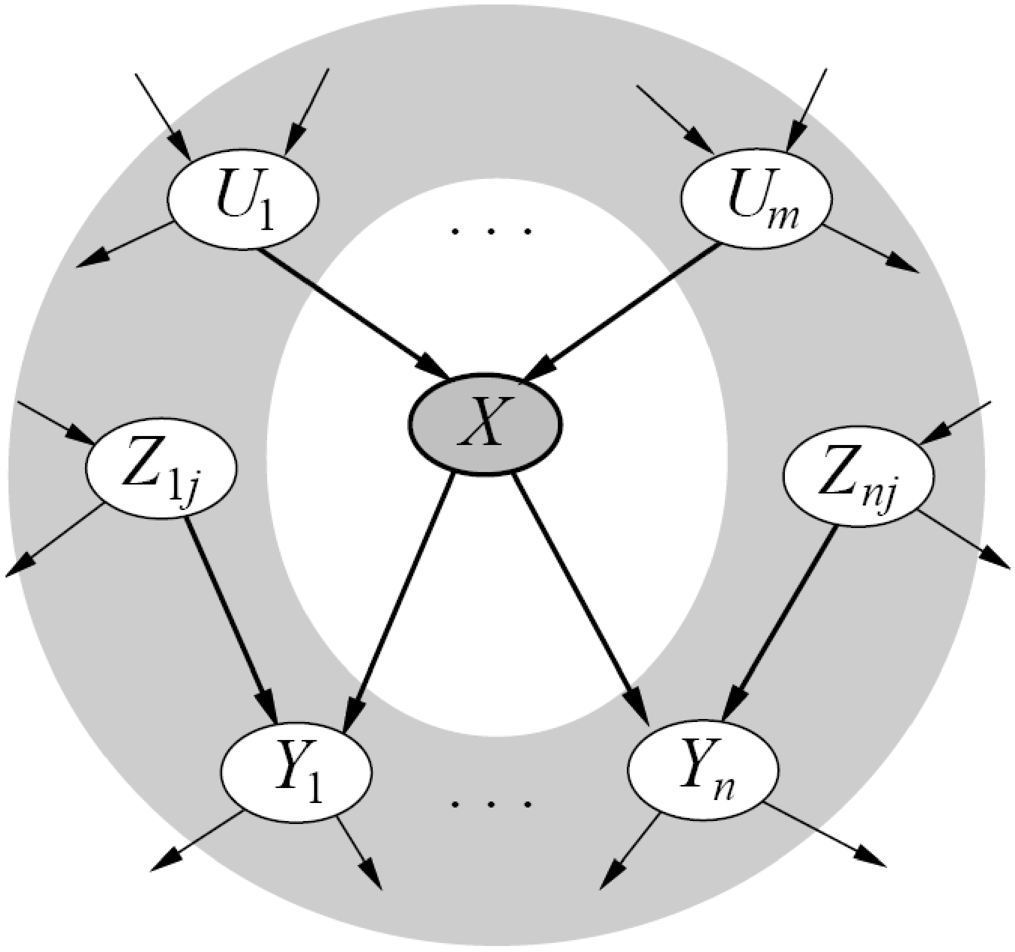
Given a graph, can we "read off" conditional independencies?

The "Markov Blanket" of X (the gray area in the figure)

X is conditionally independent of everything else, GIVEN the values of:

- * X's parents
- * X's children
- * X's children's parents

X is conditionally independent of its non-descendants, GIVEN the values of its parents.



Summary

- Bayesian networks represent a joint distribution using a graph
- The graph encodes a set of conditional independence assumptions
- Answering queries (or inference or reasoning) in a Bayesian network amounts to computation of appropriate conditional probabilities
- Probabilistic inference is intractable in the general case
  - Can be done in linear time for certain classes of Bayesian networks (polytrees: at most one directed path between any two nodes)
  - Usually faster and easier than manipulating the full joint distribution

Review Adversarial (Game) Search Chapter 5.1-5.4

- Minimax Search with Perfect Decisions (5.2)
  - Impractical in most cases, but theoretical basis for analysis
- Minimax Search with Cut-off (5.4)
  - Replace terminal leaf utility by heuristic evaluation function
- Alpha-Beta Pruning (5.3)
  - The fact of the adversary leads to an advantage in search!
- Practical Considerations (5.4)
  - Redundant path elimination, look-up tables, etc.

Games as Search

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
  - Winner gets reward, loser gets penalty.
  - “Zero sum” means the sum of the reward and the penalty is a constant.

- Formal definition as a search problem:
  - **Initial state:** Set-up specified by the rules, e.g., initial board configuration of chess.
  - **Player(s):** Defines which player has the move in a state.
  - **Actions(s):** Returns the set of legal moves in a state.
  - **Result(s,a):** Transition model defines the result of a move.
  - (**2nd ed.: Successor function:** list of (move,state) pairs specifying legal moves.)
  - **Terminal-Test(s):** Is the game finished? True if finished, false otherwise.
  - **Utility function(s,p):** Gives numerical value of terminal state s for player p.
    - E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    - E.g., win (+1), lose (0), and draw (1/2) in chess.

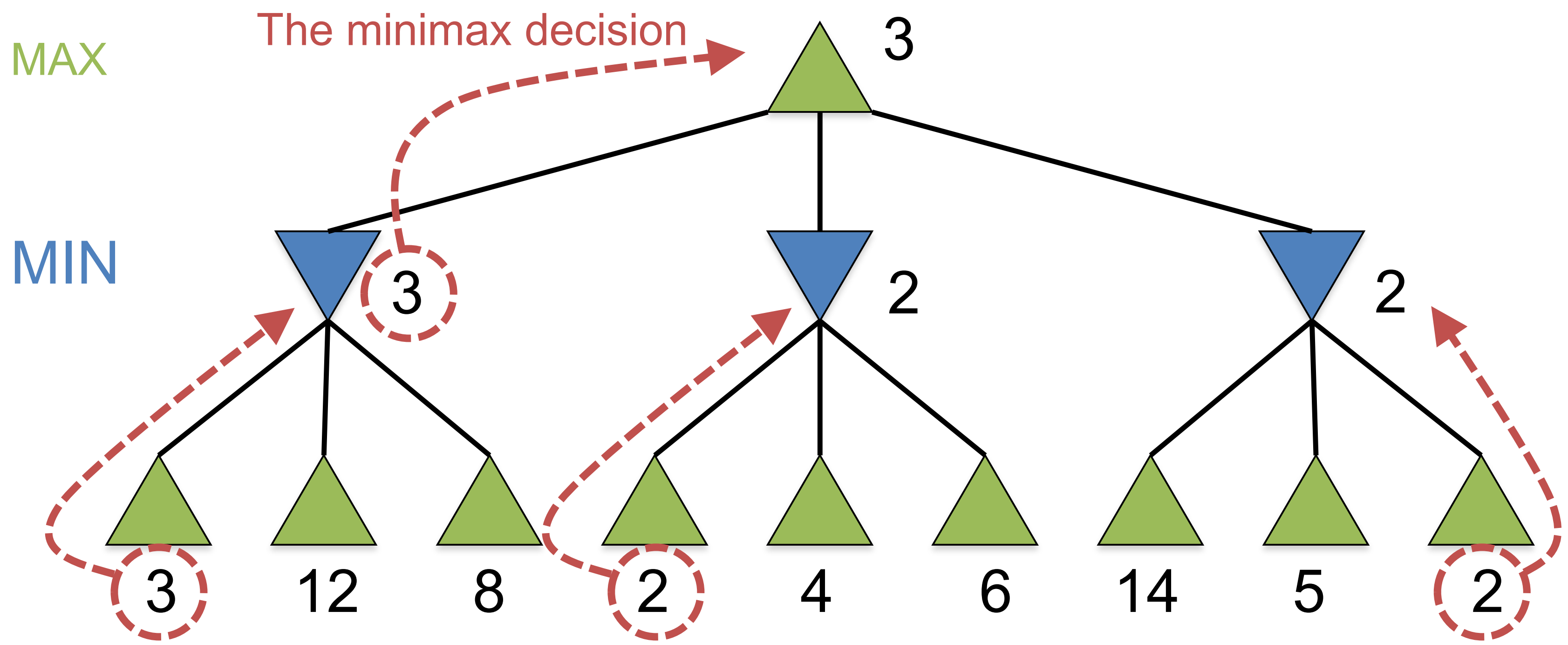
- MAX uses search tree to determine “best” next move.

An optimal procedure: The Min-Max method

Will find the <u>optimal strategy and best next move </u>for Max:

- 1. Generate the whole game tree, down to the leaves.
- 2. Apply utility (payoff) function to each leaf.
- 3. Back-up values from leaves through branch nodes:
  - a Max node computes the Max of its child values
  - a Min node computes the Min of its child values
- 4. At root: choose move leading to the child of highest value.

Two-ply Game Tree



Minimax maximizes the utility of the worst-case outcome for MAX

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
   return arg max_{a ∈ ACTIONS(state)} MIN-VALUE(Result(state,a))
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$
   for a  in ACTIONS(state) do
      $v \leftarrow$ MAX(v,MIN-VALUE(Result(state,a)))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$
   for a  in ACTIONS(state) do
      $v \leftarrow$ MIN(v,MAX-VALUE(Result(state,a)))
   return v
```

Properties of minimax

- **<u>Complete?</u>**
  - Yes (if tree is finite).

- **<u>Optimal?</u>**
  - Yes (against an optimal opponent).
  - Can it be beaten by an opponent playing sub-optimally?
    - No. (Why not?)

- **<u>Time complexity?</u>**
  - O(b^m)

- **<u>Space complexity?</u>**
  - O(bm) (depth-first search, generate all actions at once)
  - O(m) (backtracking search, generate actions one at a time)

Cutting off search

MINIMAXCUTOFF is identical to MINIMAXVALUE except

1. TERMINAL? is replaced by CUTOFF?
2. UTILITY is replaced by EVAL

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

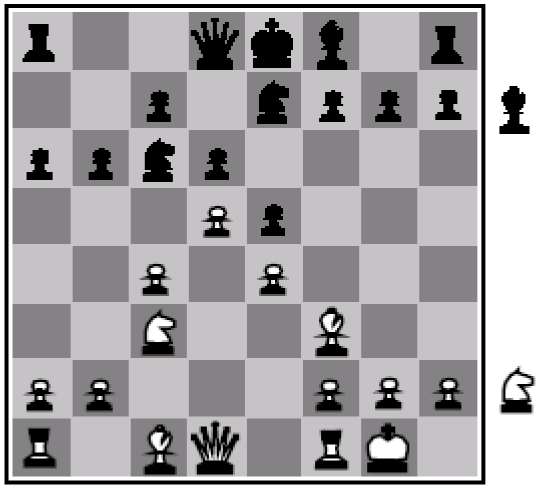
Static (Heuristic) Evaluation Functions

- An Evaluation Function:
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

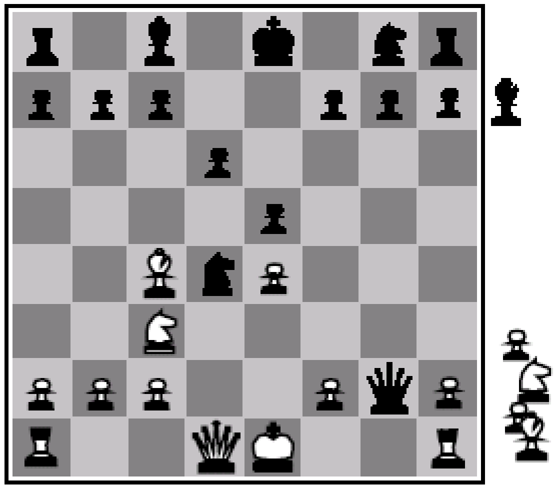
- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

- If the board evaluation is X for a player, it's -X for the opponent
  - "Zero-sum game"

Evaluation functions


Black to move

White slightly better


White to move

Black winning

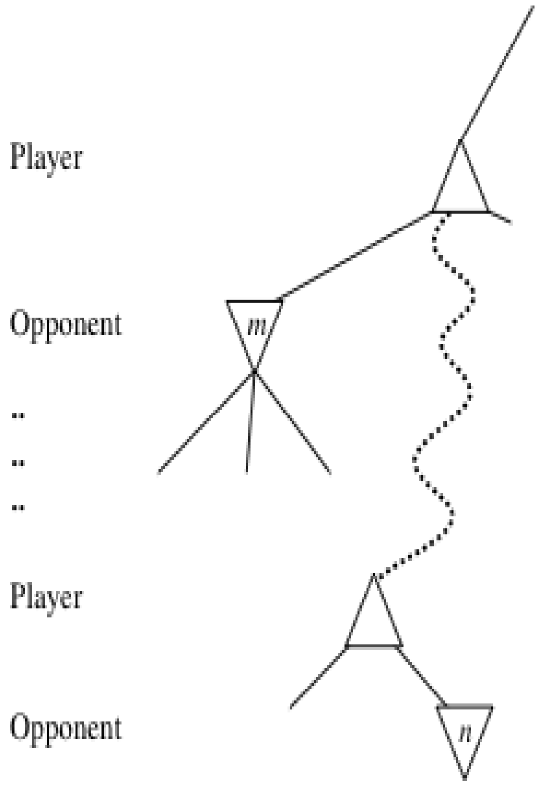
For chess, typically *linear* weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

General alpha-beta pruning

- Consider a node *n* in the tree ---
- If player has a better choice at:
  - Parent node of n
  - Or any choice point further up
- Then *n* will never be reached in play.
- Hence, when that much is known about *n*, it can be pruned.



Alpha-beta Algorithm

- Depth first search
  - only considers nodes along a single path from root at any time

α = highest-value choice found at any choice point of path for MAX

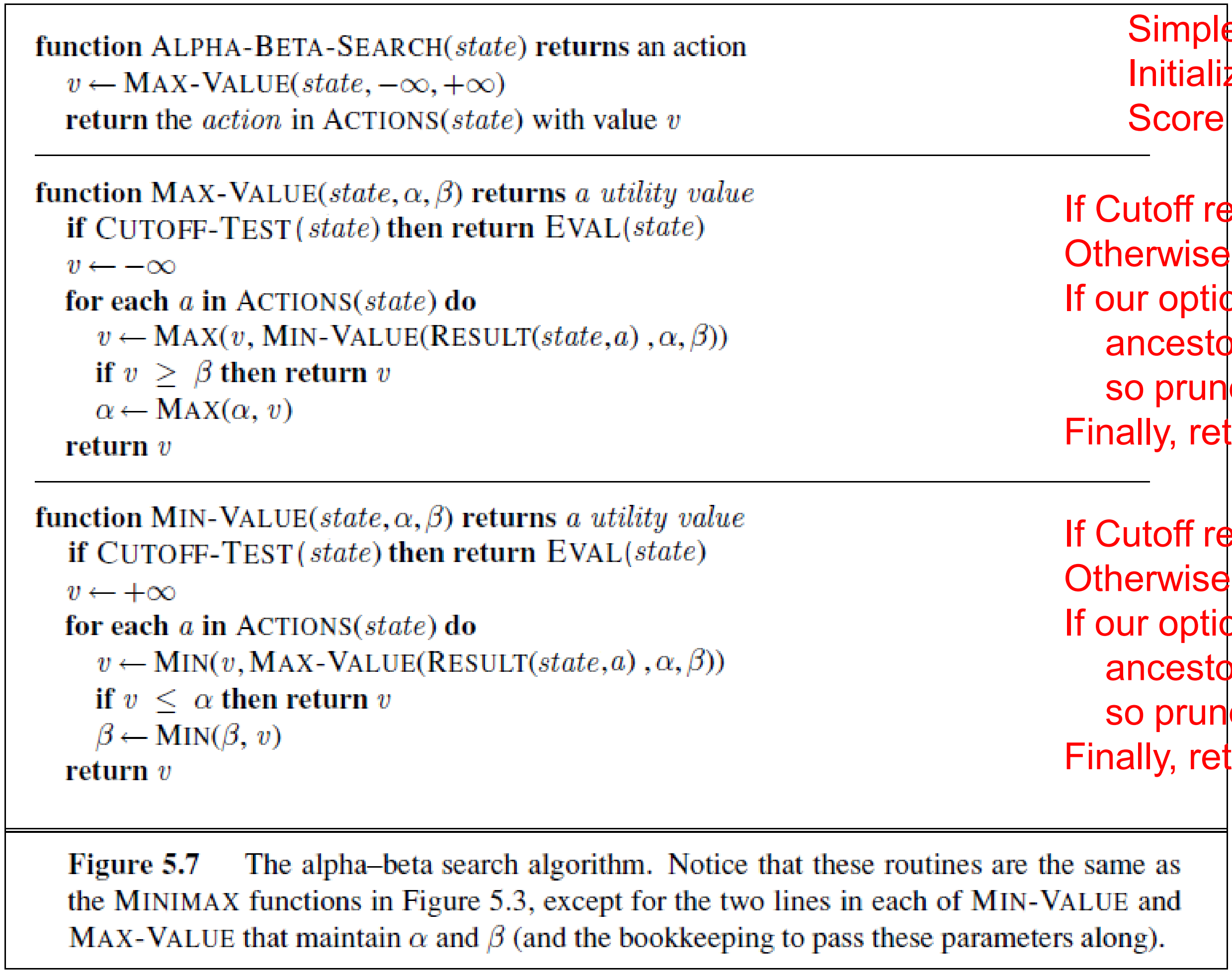
(initially, $\alpha = -\text{infinity}$)

β = lowest-value choice found at any choice point of path for MIN

(initially, $\beta = +\text{infinity}$)

- Pass current values of α and β down to child nodes during search.
- Update values of α and β during search:
  - MAX updates α at MAX nodes
  - MIN updates β at MIN nodes
- Prune remaining branches at a node when $\alpha \geq \beta$

Recursive α-β pruning: R&N Fig. 5.7



```
function ALPHA-BETA-SEARCH(state) returns an action
   $v \leftarrow$ MAX-VALUE(state, $-\infty, +\infty$)
   return the action in ACTIONS(state) with value $v$
```

```
function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
   if CUTOFF-TEST(state) then return EVAL(state)
   $v \leftarrow -\infty$
   for each a in ACTIONS(state) do
      $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT(state,a), $\alpha$, $\beta$))
      if $v \geq \beta$ then return $v$
      $\alpha \leftarrow$ MAX($\alpha$, $v$)
   return $v$
```

```
function MIN-VALUE(state, $\alpha$, $\beta$) returns a utility value
   if CUTOFF-TEST(state) then return EVAL(state)
   $v \leftarrow +\infty$
   for each a in ACTIONS(state) do
      $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT(state,a), $\alpha$, $\beta$))
      if $v \leq \alpha$ then return $v$
      $\beta \leftarrow$ MIN($\beta$, $v$)
   return $v$
```

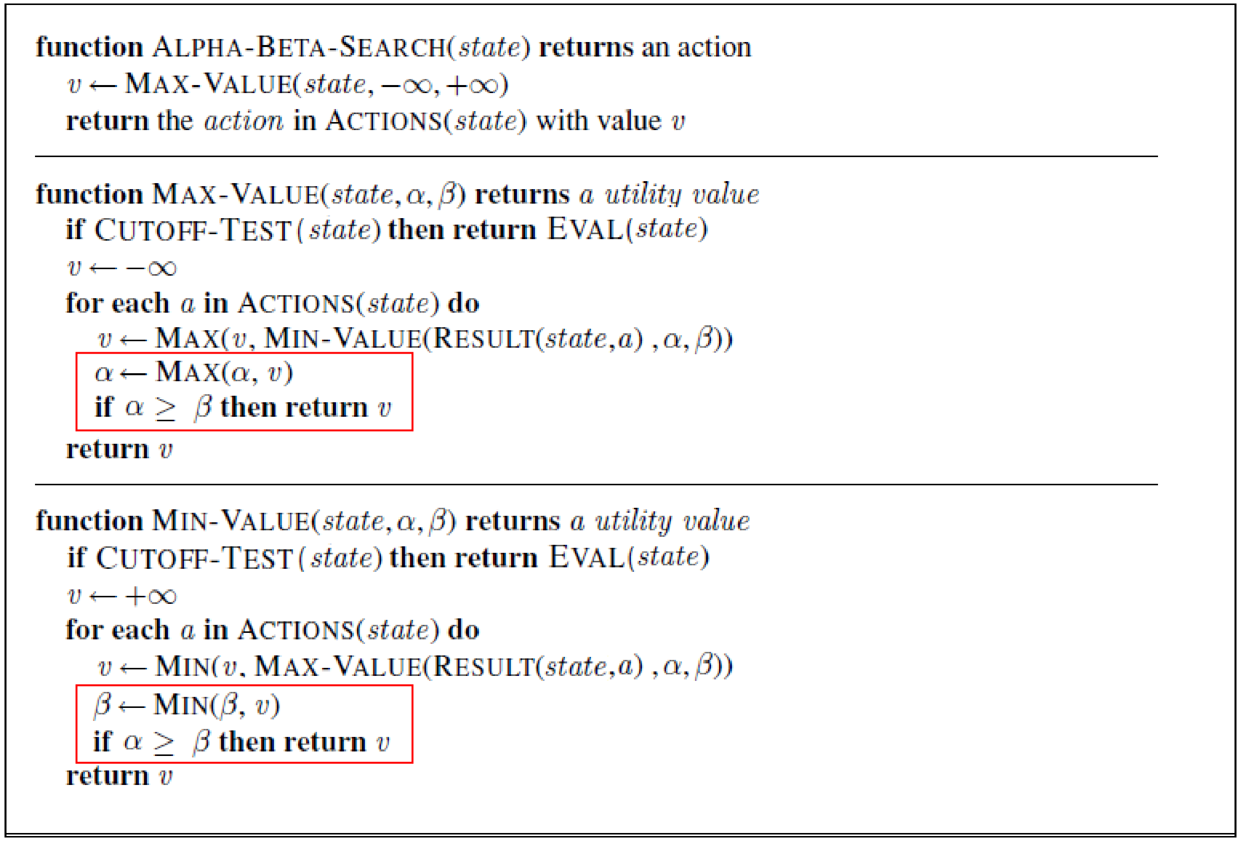
Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Simple stub to call recursion functions
Initialize alpha, beta; get best value
Score each action; return best action

If Cutoff reached, return Eval heuristic
Otherwise, find our best child:
If our options become too good, our min
   ancestor will never let us come this way,
   so prune now & return best value so far
Finally, return the best value we found

If Cutoff reached, return Eval heuristic
Otherwise, find our worst child:
If our options become too bad, our max
   ancestor will never let us come this way,
   so prune now & return worst value so far
Finally, return the worst value we found

Recursive α-β pruning variant: Prune when α ≥ β



```
function ALPHA-BETA-SEARCH(state) returns an action
   $v \leftarrow$ MAX-VALUE(state, $-\infty, +\infty$)
   return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\alpha, \beta$) returns a utility value
   if CUTOFF-TEST(state) then return EVAL(state)
   $v \leftarrow -\infty$
   for each a in ACTIONS(state) do
      $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT(state,a), $\alpha, \beta$))
      $\alpha \leftarrow$ MAX($\alpha$, $v$)
      if $\alpha \geq \beta$ then return $v$
   return $v$

function MIN-VALUE(state, $\alpha, \beta$) returns a utility value
   if CUTOFF-TEST(state) then return EVAL(state)
   $v \leftarrow +\infty$
   for each a in ACTIONS(state) do
      $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT(state,a), $\alpha, \beta$))
      $\beta \leftarrow$ MIN($\beta$, $v$)
      if $\alpha \geq \beta$ then return $v$
   return $v$
```

This variant has a conceptually simpler pruning rule (α ≥ β), but when pruning occurs it makes one extra call to MAX(). Both variants yield the same pruning behavior, and **both are considered correct on tests.**

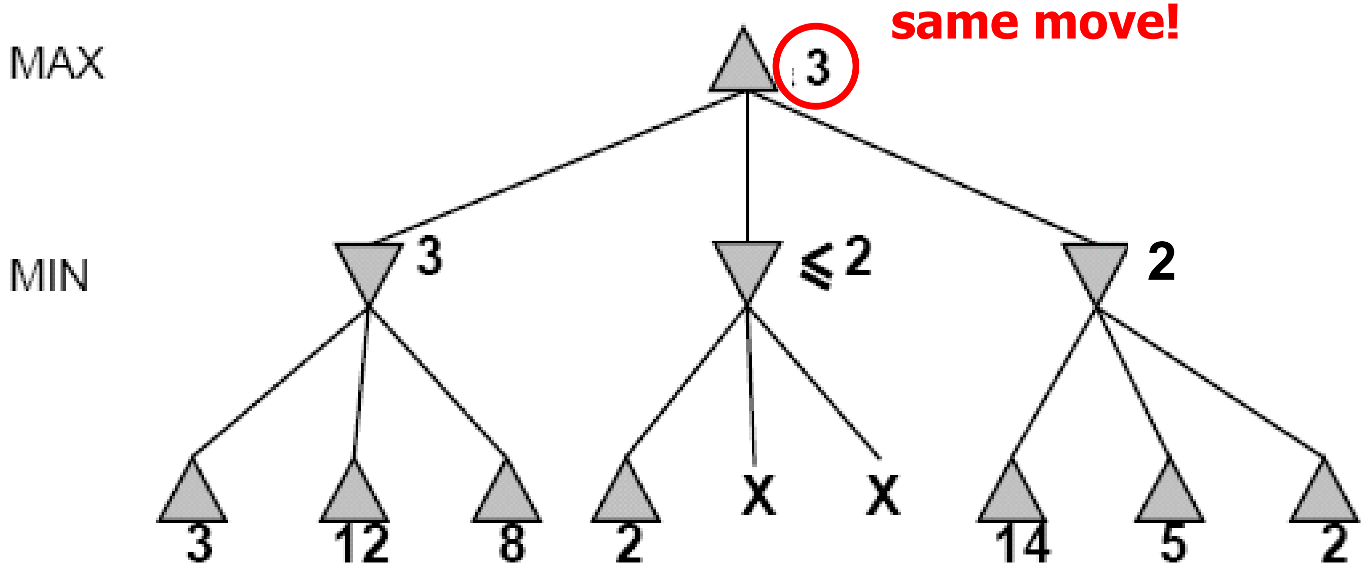
When to Prune?

- **Prune whenever $\alpha \geq \beta$.**
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    - **Max nodes update alpha** based on children's returned values.
  - Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.
    - **Min nodes update beta** based on children's returned values.

α/β Pruning vs. Returned Node Value

- Some students are confused about the use of α/β pruning vs. the returned value of a node
- <u>α/β are used **ONLY FOR PRUNING**</u>
  - α/β have no effect on anything other than pruning
  - IF (α >= β) THEN prune & return current node value
- <u>Returned node value = “best” child seen so far</u>
  - Maximum child value seen so far for MAX nodes
  - Minimum child value seen so far for MIN nodes
  - If you prune, return to parent <u>“best” child so far</u>
- <u>Returned node value is received by parent</u>

Alpha-Beta Example



Max calculates the same node value, and makes the same move!

Review Detailed Example of Alpha-Beta Pruning in lecture slides.

Review Constraint Satisfaction
R&N 6.1-6.4 (except 6.3.3)

- What is a CSP?

- Backtracking search for CSPs
  - Choose a variable, then choose an order for values
  - Minimum Remaining Values (MRV), Degree Heuristic (DH), Least Constraining Value (LCV)

- Constraint propagation
  - Forward Checking (FC), Arc Consistency (AC-3)

- Local search for CSPs
  - Min-conflicts heuristic

Constraint Satisfaction Problems

- What is a CSP?
  - Finite set of variables, $X_1, X_2, \ldots, X_n$
  - Nonempty domain of possible values for each: $D_1, \ldots, D_n$
  - Finite set of constraints, $C_1, \ldots, C_m$
    - Each constraint C_i limits the values that variables can take, e.g., $X_1 \neq X_2$
  - Each constraint C_i is a pair: C_i = (*scope*, *relation*)
    - Scope = tuple of variables that participate in the constraint
    - Relation = list of allowed combinations of variables

      May be an explicit list of allowed combinations

      May be an abstract relation allowing membership testing & listing

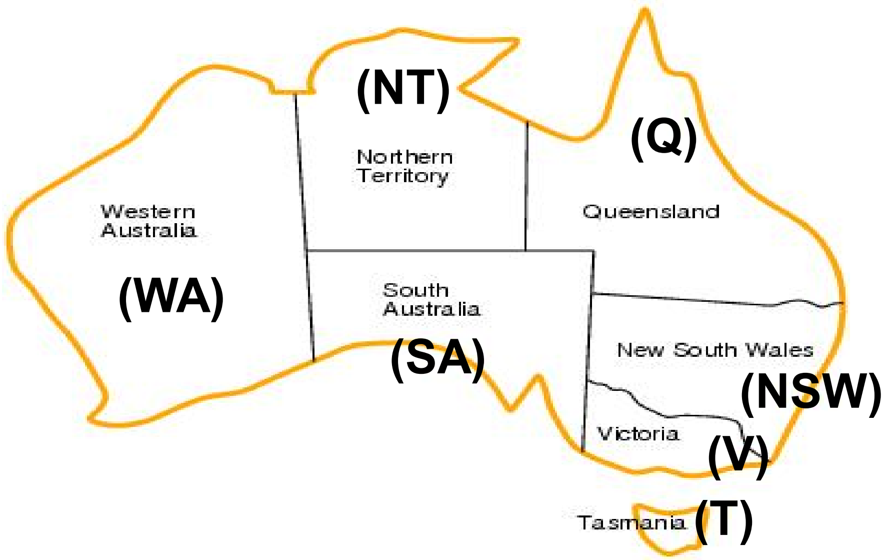
- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain-specific expertise required)

CSPs --- what is a solution?

- A ***state*** is an ***assignment*** of values to some variables.
  - ***Complete*** assignment
    - = every variable has a value.
  - ***Partial*** assignment
    - = some variables have no values.
  - ***Consistent*** assignment
    - = assignment does not violate any constraints

- A ***solution*** is a ***complete*** and ***consistent*** assignment.

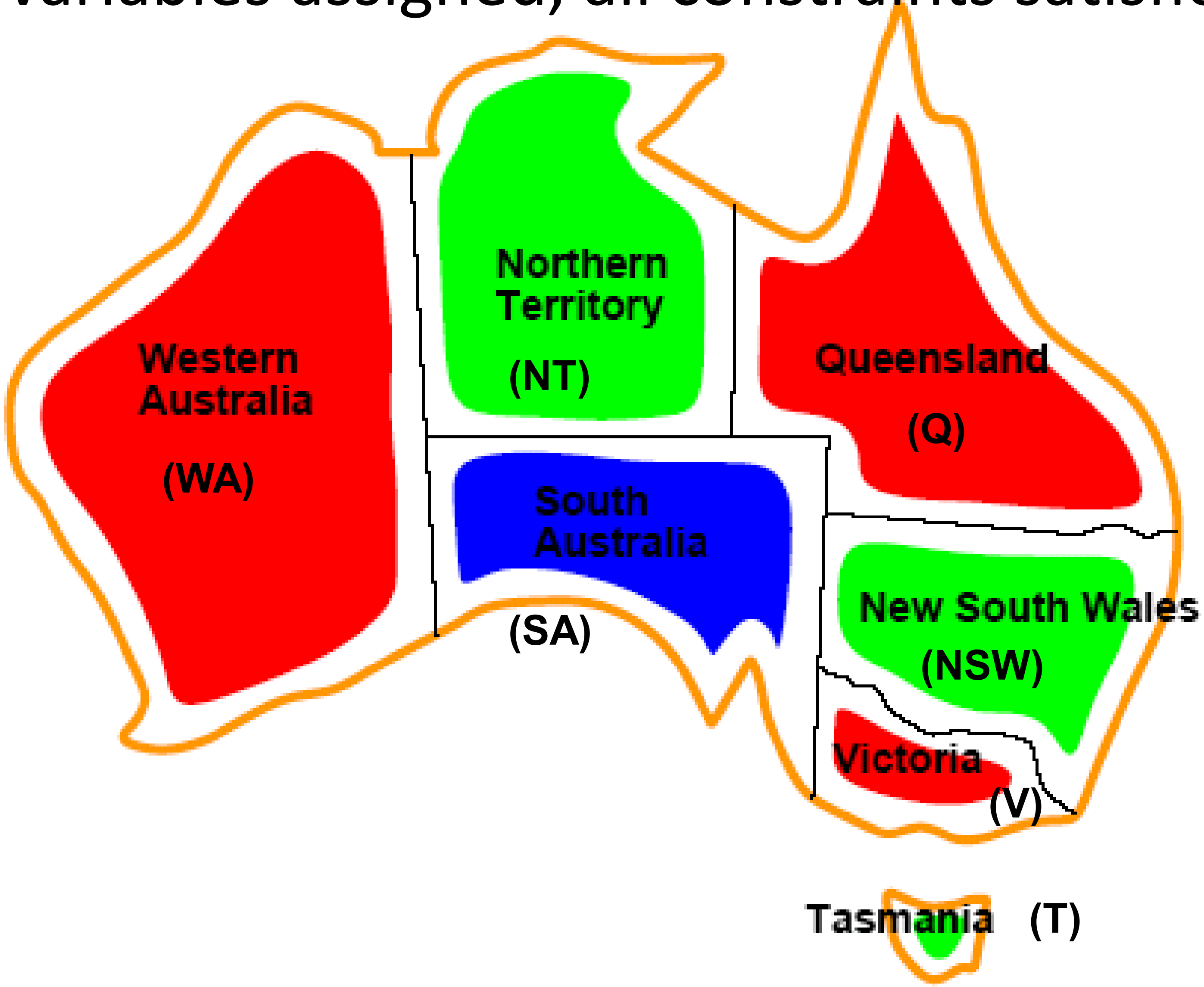
CSP example: map coloring



- **Variables:** *WA, NT, Q, NSW, V, SA, T*
- **Domains:** $D_i = \{red, green, blue\}$
- **Constraints:** Adjacent regions must have different colors, e.g., $WA \neq NT$.

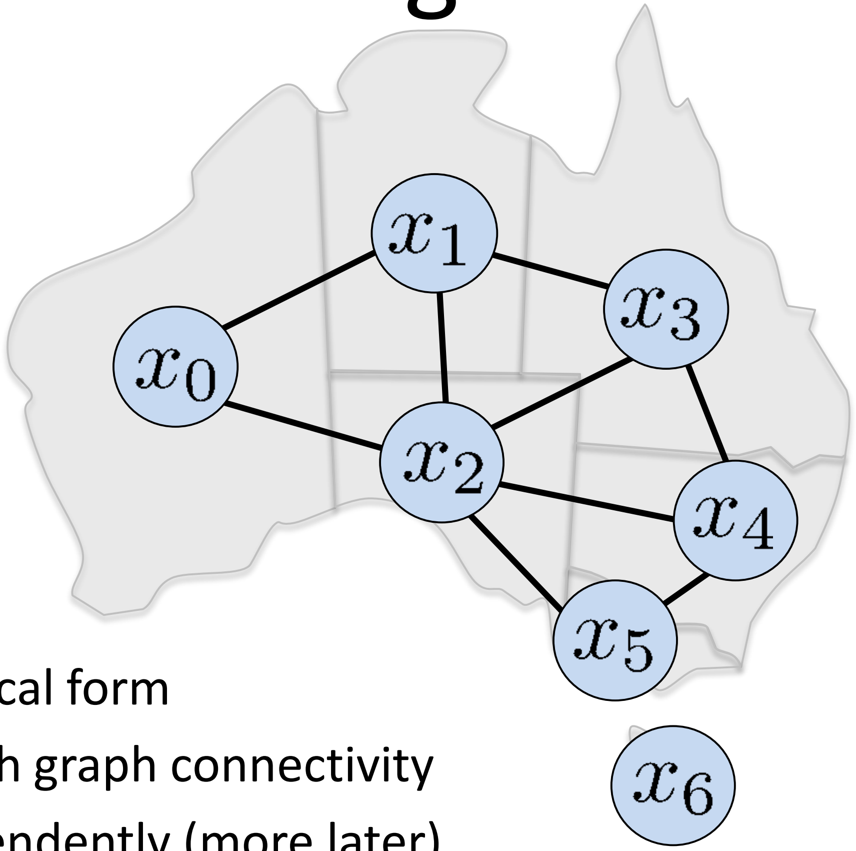
Example: Map coloring solution

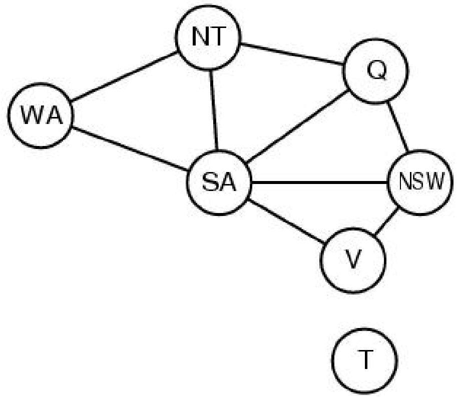
All variables assigned, all constraints satisfied.



Example: Map Coloring

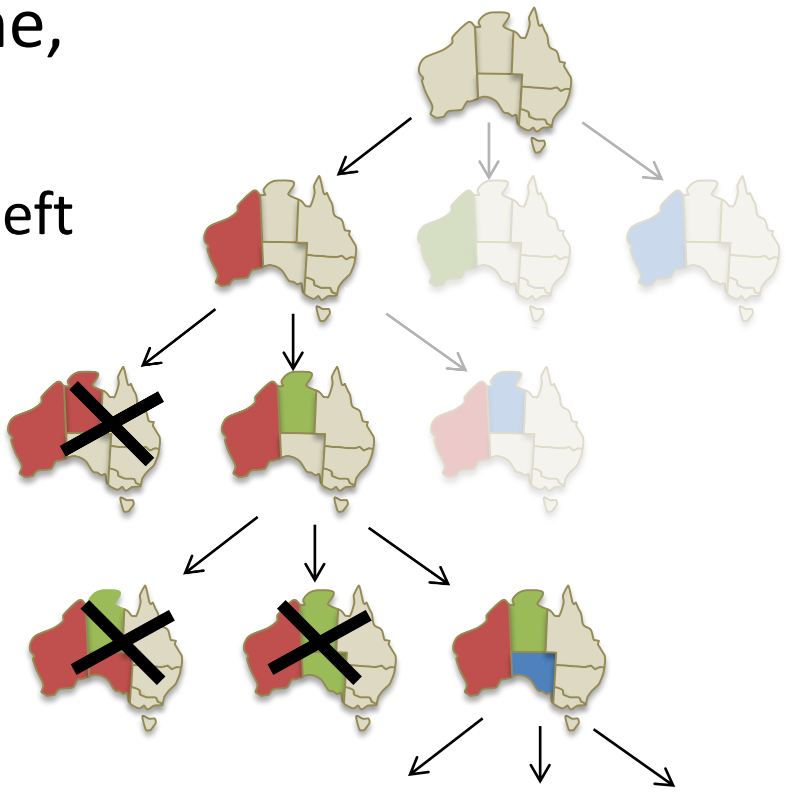
- Constraint graph
  - Vertices: variables
  - Edges: constraints
    (connect involved variables)
- Graphical model
  - Abstracts the problem to a canonical form
  - Can reason about problem through graph connectivity
  - Ex: Tasmania can be solved independently (more later)
- Binary CSP
  - Constraints involve at most two variables
  - Sometimes called "pairwise"





Backtracking search

- Similar to depth-first search
  - At each level, pick a single variable to expand
  - Iterate over the domain values of that variable
- Generate children one at a time,
  - One child per value
  - Backtrack when no legal values left
- Uninformed algorithm
  - Poor general performance

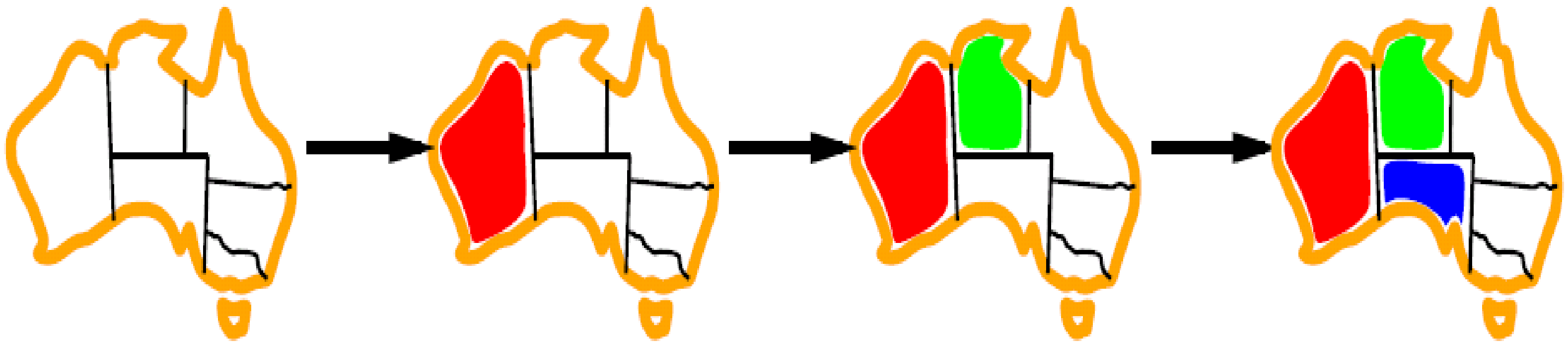


Backtracking search (Figure 6.5)

```
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({} , csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assignment,csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RRECURSIVE-BACTRACKING(assignment, csp)
            if result ≠ failure  then return result
            remove {var=value} from assignment
    return failure
```

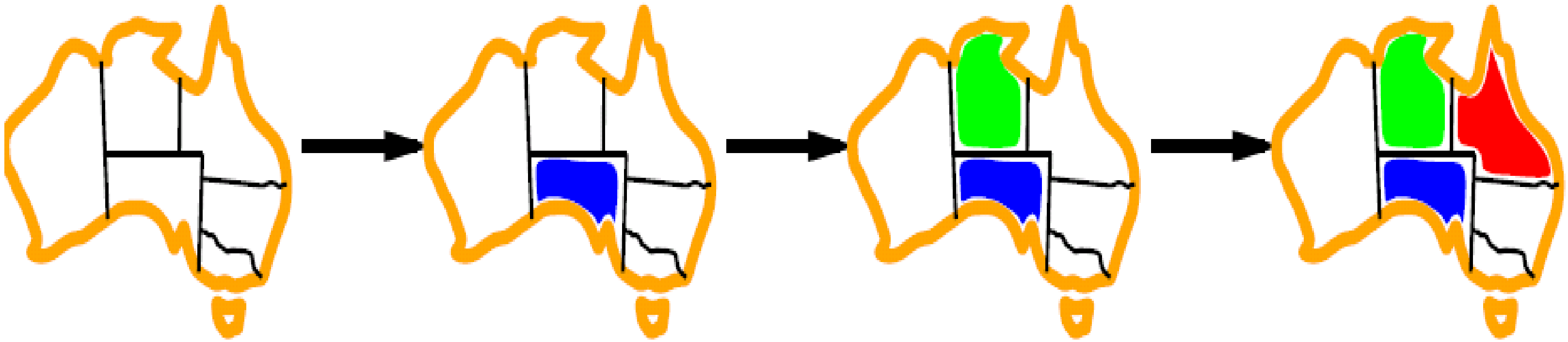
Minimum remaining values (MRV)



var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

- A.k.a. most constrained variable heuristic
- *Heuristic Rule*: choose variable with the fewest legal moves
  - e.g., will immediately detect failure if X has no legal values

Degree heuristic for the initial variable



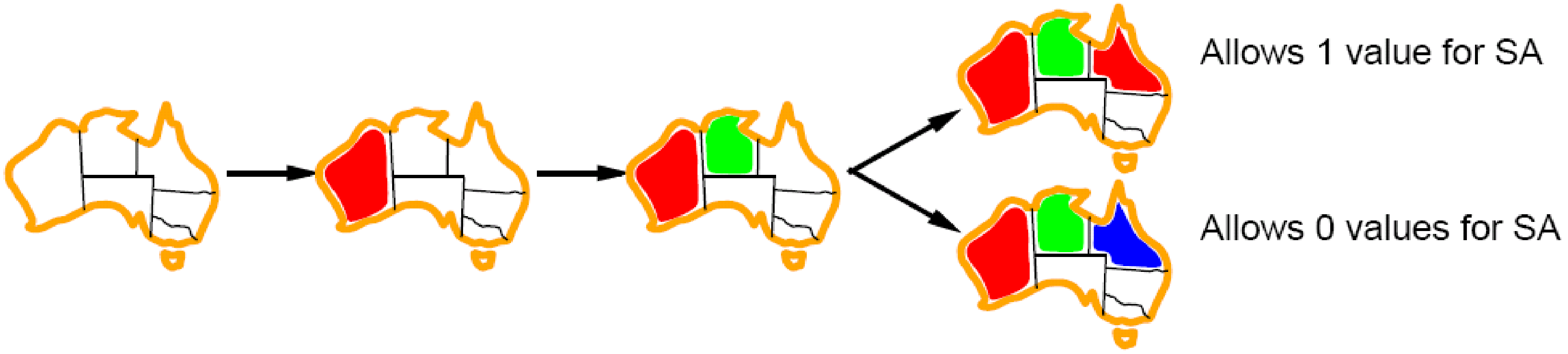
- *Heuristic Rule*: select variable that is involved in the largest number of constraints on other unassigned variables.
- Degree heuristic can be useful as a tie breaker.
- *In what order should a variable's values be tried?*

Backtracking search (Figure 6.5)

```
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({} , csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assignment,csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RRECURSIVE-BACTRACKING(assignment, csp)
            if result ≠ failure  then return result
            remove {var=value} from assignment
    return failure
```

Least constraining value for value-ordering



- Least constraining value heuristic

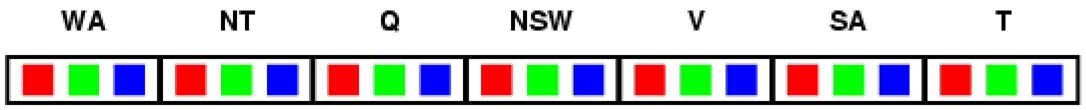
- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments

Look-ahead: Constraint propagation

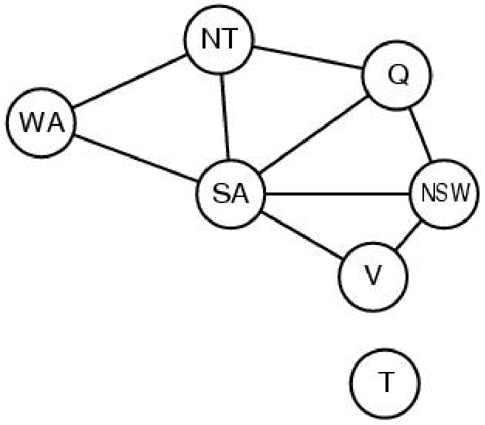
- Intuition:
  - Some domains have values that are <u>inconsistent</u> with the values in some other domains
  - Propagate constraints to remove inconsistent values
  - Thereby reduce future branching factors
- Forward checking
  - Check each unassigned neighbor in constraint graph
- Arc consistency (AC-3 in R&N)
  - Full arc-consistency everywhere until quiescence
  - Can run as a preprocessor
    - Remove obvious inconsistencies
  - Can run after each step of backtracking search
    - Maintaining Arc Consistency (MAC)

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - ONLY check neighbors of most recently assigned variable

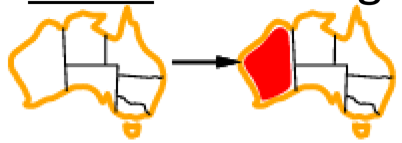


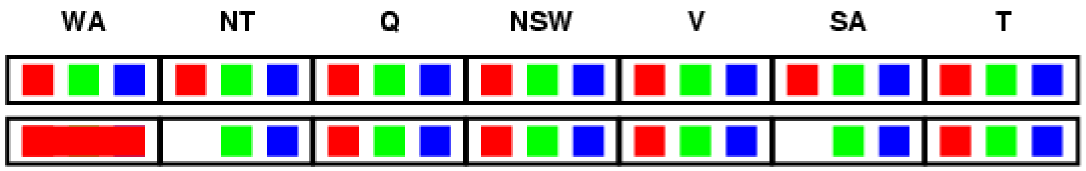
WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦



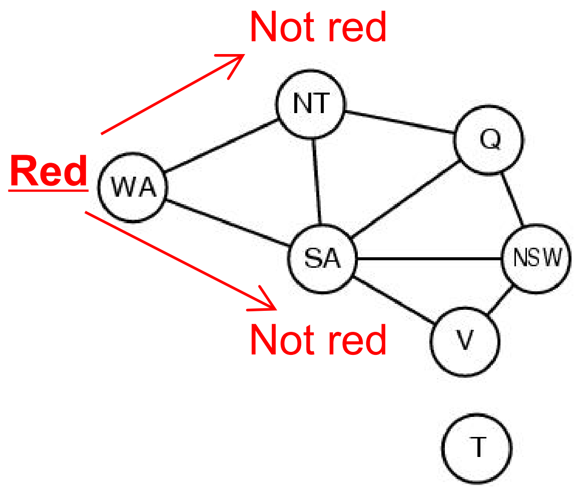
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - ONLY check neighbors of most recently assigned variable





WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦

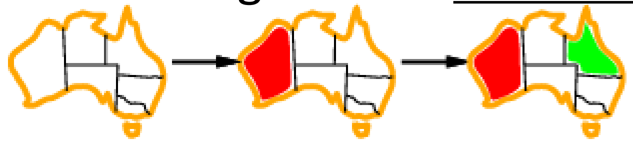


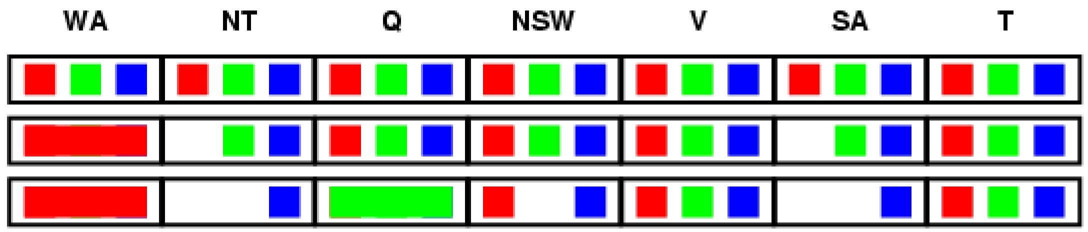
Assign {WA = red}
Effect on other variables (neighbors of WA):

- NT can no longer be red
- SA can no longer be red

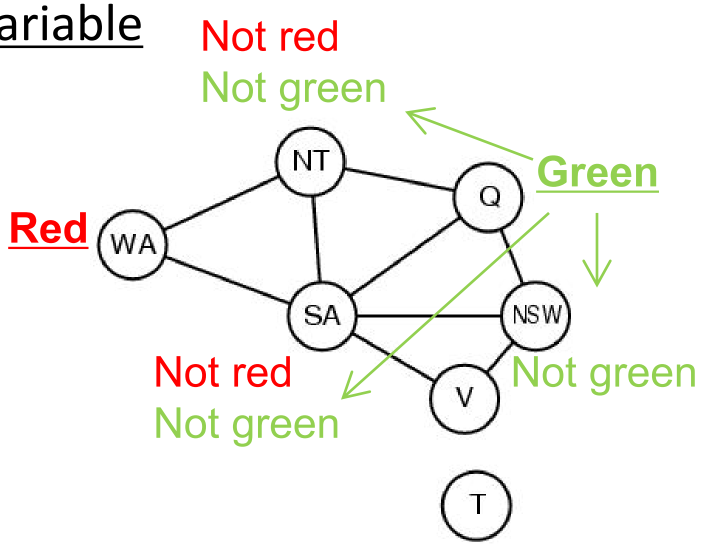
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - Check neighbors of <u>most recently assigned variable</u>





WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦



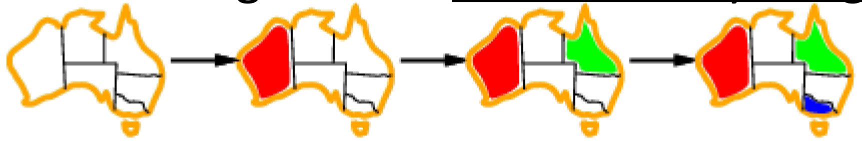
Assign {Q = green}
Effect on other variables (neighbors of Q):

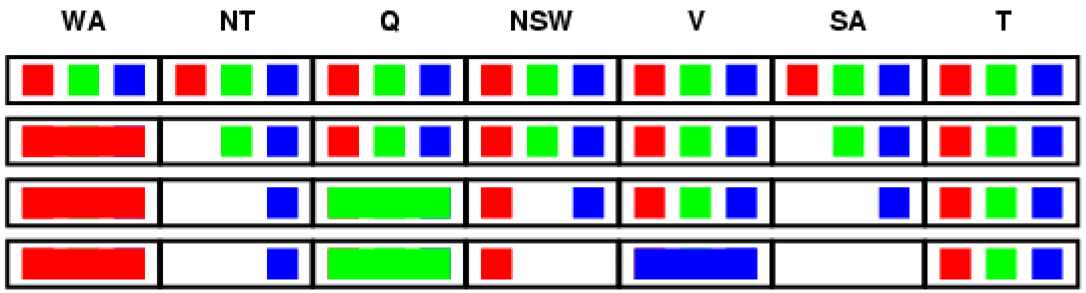
- NT can no longer be green
- SA can no longer be green
- NSW can no longer be green

<u>(We already have failure, but FC is too simple to detect it now)</u>

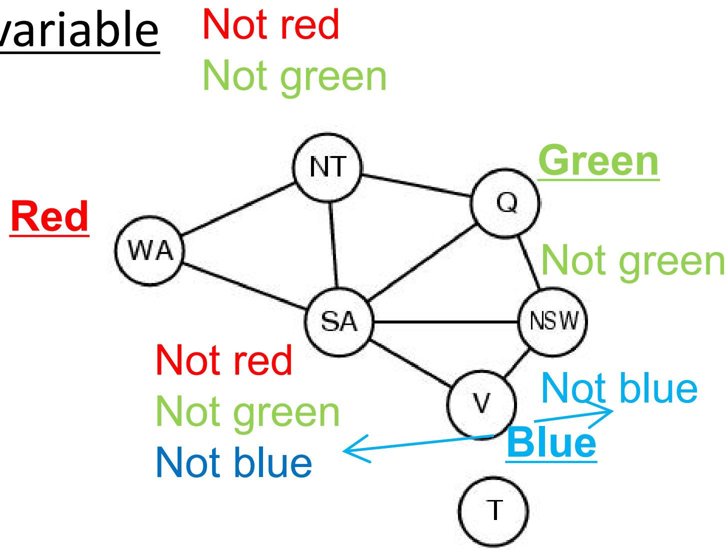
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - Check neighbors of <u>most recently assigned variable</u>





WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦



Assign {V = blue}
Effect on other variables (neighbors of V):

- NSW can no longer be blue
- SA can no longer be blue **(no values possible!)**

Forward checking has detected that this partial assignment is inconsistent with any complete assignment

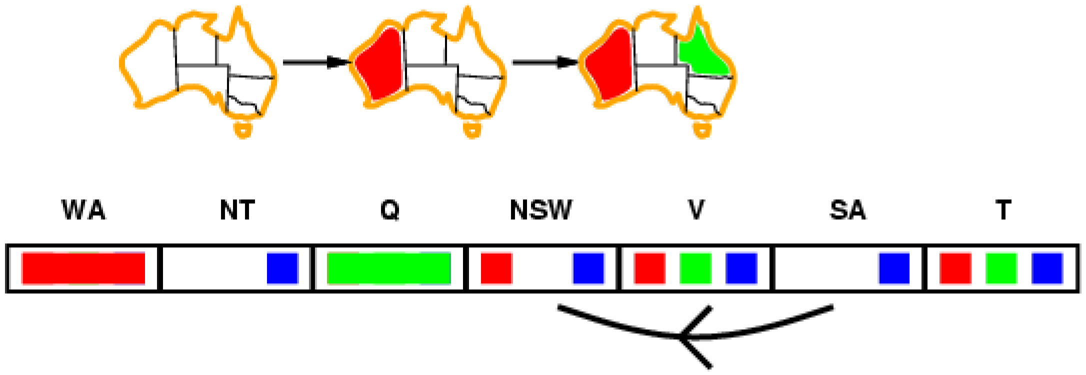
Arc consistency (AC-3) algorithm

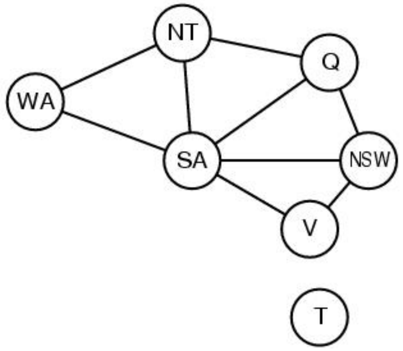
- An Arc $X \rightarrow Y$ is consistent iff for *every* value x of X there is *some* value y of Y that is consistent with x
- Put all arcs $X \rightarrow Y$ on a queue
  - Each undirected constraint graph arc is two directed arcs
  - Undirected $X — Y$ becomes directed $X \rightarrow Y$ and $Y \rightarrow X$
  - $X \rightarrow Y$ and $Y \rightarrow X$ both go on queue, separately
- Pop one arc $X \rightarrow Y$ and remove any inconsistent values from X
- If any change in X, put all arcs $Z \rightarrow X$ back on queue, where Z is any neighbor of X that is not equal to Y
- Continue until queue is empty

Arc consistency (AC-3)

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff (iff = if and only if)

  for every value *x* of *X* there is some allowed value *y for Y* *(note: directed!)*





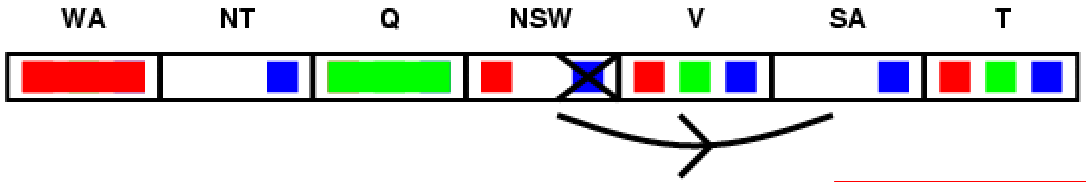
- Consider state after WA=red, Q=green
  - $SA \rightarrow NSW$ is consistent because

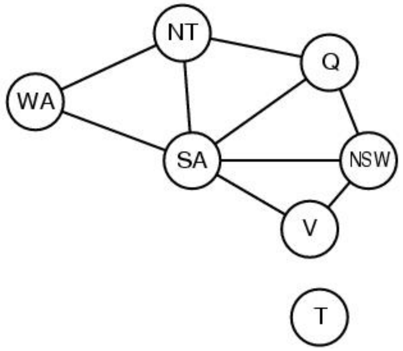
    SA = blue and NSW = red satisfies all constraints on SA and NSW

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value *x* of *X* there is some allowed value *y for Y* *(note: directed!)*





If *X* loses a value, neighbors of *X* need to be rechecked

- Consider state after WA=red, Q=green
  - NSW $\rightarrow$ SA consistent if

    NSW = red and SA = blue

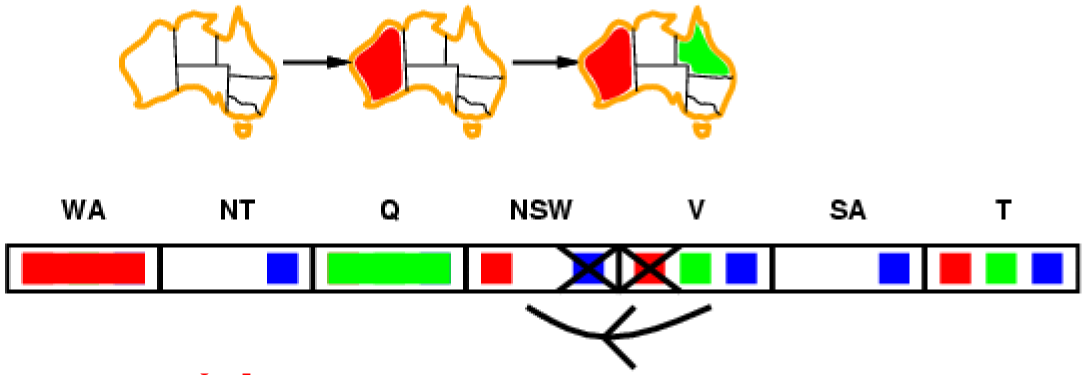
    NSW = blue and SA = ??? => NSW = blue can be pruned

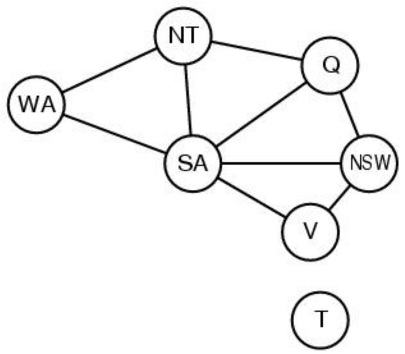
    No current domain value for SA is consistent

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value *x* of *X* there is some allowed value *y for Y* *(note: directed!)*



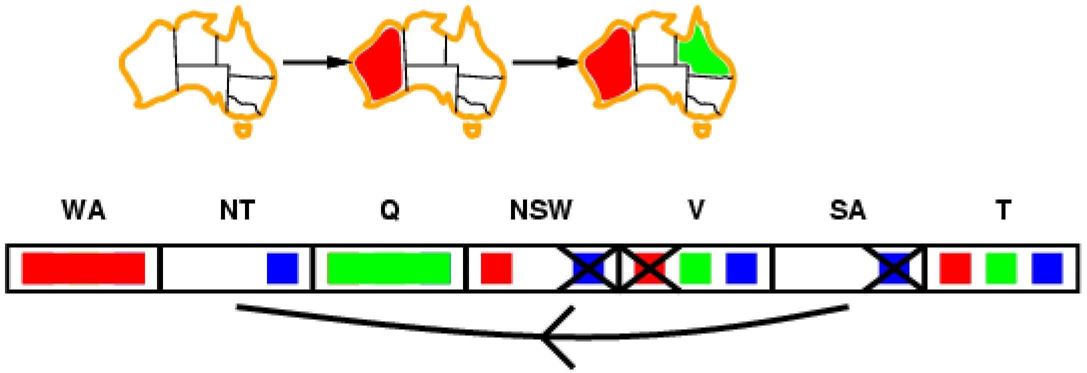


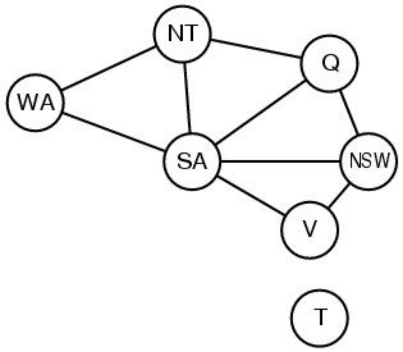
- **Enforce arc consistency:**
  - arc can be made consistent by removing blue from NSW
- **Continue to propagate constraints:**
  - Check V $\rightarrow$ NSW : not consistent for V = red; remove red from V

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value *x* of *X* there is some allowed value *y for Y* *(note: directed!)*





- Continue to propagate constraints
- SA $\rightarrow$ NT not consistent:
  - **And cannot be made consistent! Failure!**
- Arc consistency detects failure earlier than FC
  - But requires more computation: is it worth the effort?

Local search: min-conflicts heuristic

- Use complete-state representation
  - Initial state = all variables assigned values
  - Successor states = change 1 (or more) values

- For CSPs
  - allow states with unsatisfied constraints (unlike backtracking)
  - operators **reassign** variable values
  - hill-climbing with n-queens is an example

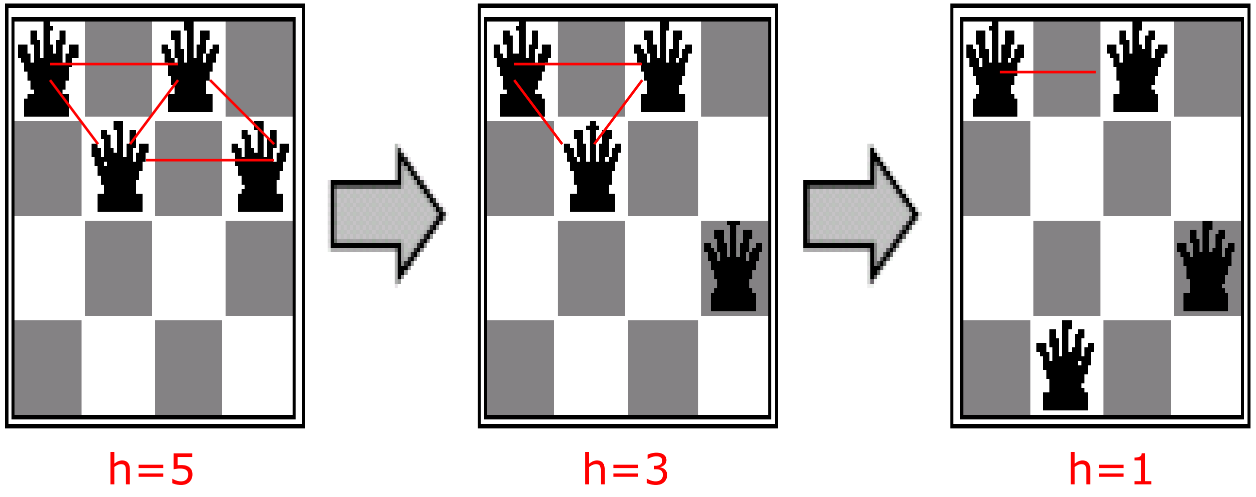
- **Variable selection:** randomly select any conflicted variable
- **Value selection:** *min-conflicts heuristic*
  - Select new value that results in a minimum number of conflicts with the other variables

Local search: min-conflicts heuristic

```
function MIN-CONFLICTS(csp, max_steps) return solution or failure
    inputs: csp, a constraint satisfaction problem
            max_steps, the number of steps allowed before giving up

    current ← a (random) initial complete assignment for csp
    for i = 1 to max_steps do
            if current is a solution for csp then return current
            var ← a randomly chosen, conflicted variable from
                  VARIABLES[csp]
            value ← the value v for var that minimize
    CONFLICTS(var,v,current,csp)
            set var = value in current
    return failure
```

Min-conflicts example 1



Use of min-conflicts heuristic in hill-climbing.

Summary

- CSPs
  - special kind of problem: states defined by values of a fixed set of variables, goal test defined by constraints on variable values
- Backtracking = depth-first search, one variable assigned per node
- Heuristics: variable order & value selection heuristics help a lot
- Constraint propagation
  - does additional work to constrain values and detect inconsistencies
  - Works effectively when combined with heuristics
- Iterative min-conflicts is often effective in practice.
- Graph structure of CSPs determines problem complexity
  - e.g., tree structured CSPs can be solved in linear time.

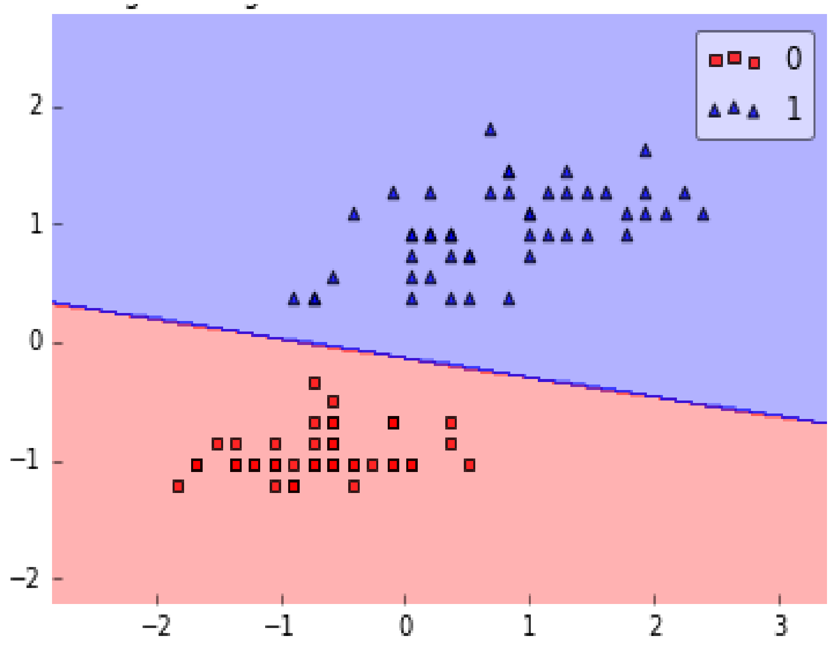
Review Intro Machine Learning Chapter 18.1-18.4

- Understand Attributes, Target Variable, Error (loss) function, Classification & Regression, Hypothesis (Predictor) function
- What is Supervised Learning?
- Decision Tree Algorithm
- Entropy & Information Gain
- Tradeoff between train and test with model complexity
- Cross validation

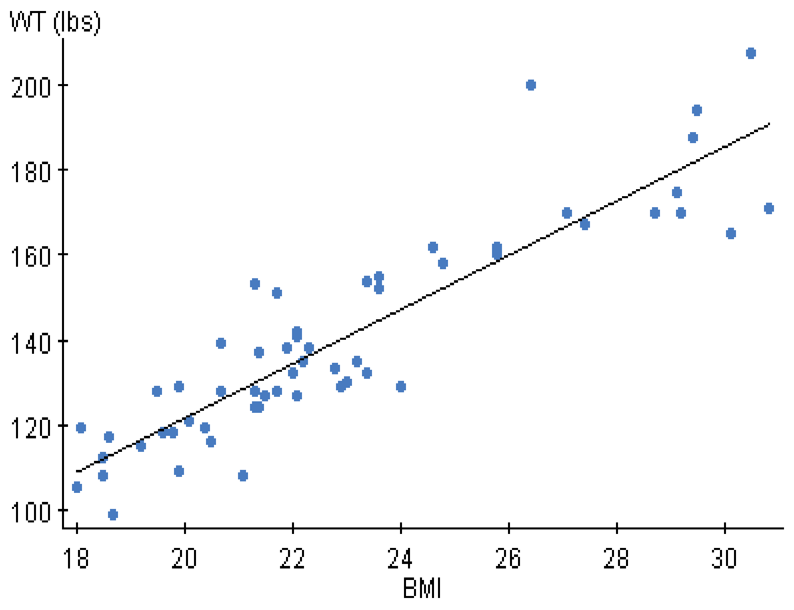
Supervised Learning

- Use supervised learning – training data is given with correct output
- We write program to reproduce this output with new test data
- Eg : face detection
- Classification : face detection, spam email
- Regression : Netflix guesses how much you will rate the movie

Classification Graph



Regression Graph



Terminology

- Attributes
  - Also known as features, variables, independent variables, covariates
- Target Variable
  - Also known as goal predicate, dependent variable, …
- Classification
  - Also known as discrimination, supervised classification, …
- Error function
  - Also known as objective function, loss function, …

Inductive or Supervised learning

- Let x = input vector of attributes (feature vectors)

- Let f(x) = target label
  - The implicit mapping from x to f(x) is unknown to us
  - We only have training data pairs, D = {**x**, **f(x)**} available

- We want to learn a mapping from x to f(x)
  - Our hypothesis function is h(x, θ)
  - h(x, θ) ≈ f(x) for all training data points x
  - θ are the parameters of our predictor function h

- Examples:
  - h(x, θ) = sign($\theta_1 x_1 + \theta_2 x_2 + \theta_3$) (perceptron)
  - h(x, θ) = $\theta_0 + \theta_1 x_1 + \theta_2 x_2$ (regression)
  - $h_k(x) = (x_1 \wedge x_2) \vee (x_3 \wedge \neg x_4)$

Empirical Error Functions

- $E(h) = \Sigma_x$ distance$[h(x, \theta) , f(x)]$

  Sum is over all training pairs in the training data D

  Examples:

  distance = squared error if h and f are real-valued (regression)

  distance = delta-function if h and f are categorical (classification)

  In learning, we get to choose

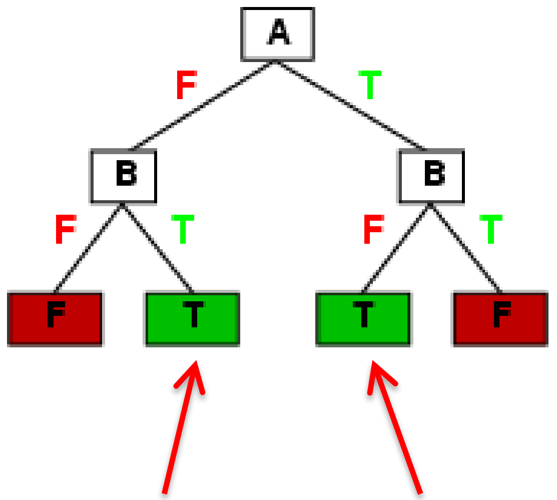
  1. what class of functions h(..) we want to learn
     – potentially a huge space! ("hypothesis space")

  2. what error function/distance we want to use
     - should be chosen to reflect real "loss" in problem
     - but often chosen for mathematical/algorithmic convenience

Decision Tree Representations

- Decision trees are fully expressive
  - Can represent any Boolean function (in DNF)
  - Every path in the tree could represent 1 row in the truth table
  - Might yield an exponentially large tree
    - Truth table is of size 2^d, where d is the number of attributes

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



A xor B = $(\neg A \wedge B) \vee (A \wedge \neg B)$ in DNF

Decision Tree Representations

- Decision trees are DNF representations
  - often used in practice → often result in compact approximate representations for complex functions
  - E.g., consider a truth table where most of the variables are irrelevant to the function

- Simple DNF formulae can be easily represented
  - E.g., $f = (A \land B) \lor (\neg A \land D)$
  - DNF = disjunction of conjunctions

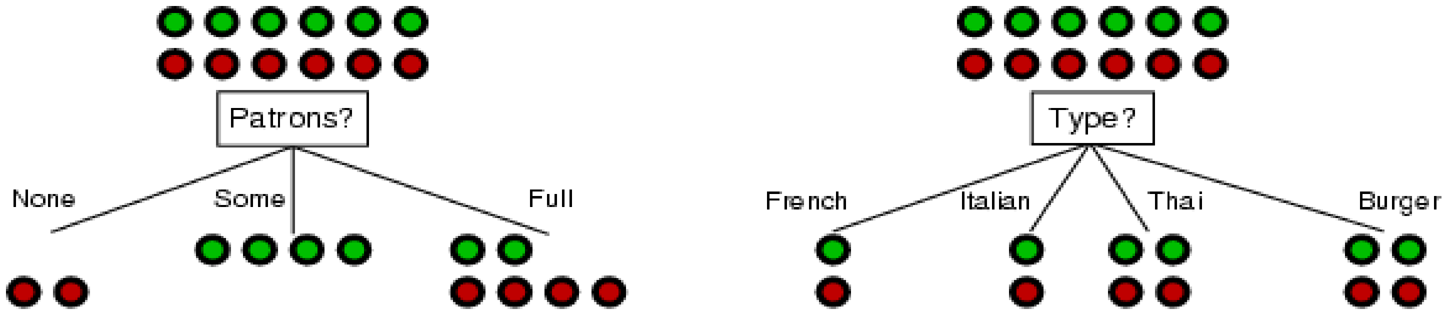
- Trees can be very inefficient for certain types of functions
  - Parity function: 1 only if an even number of 1's in the input vector
    - Trees are very inefficient at representing such functions
  - Majority function: 1 if more than ½ the inputs are 1's
    - Also inefficient

Pseudocode for Decision tree learning

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MODE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value $v_i$ of best do
            $examples_i$ ← {elements of examples with best = $v_i$}
            subtree ← DTL($examples_i$, attributes − best, MODE(examples))
            add a branch to tree with label $v_i$ and subtree subtree
        return tree
```

Choosing an attribute

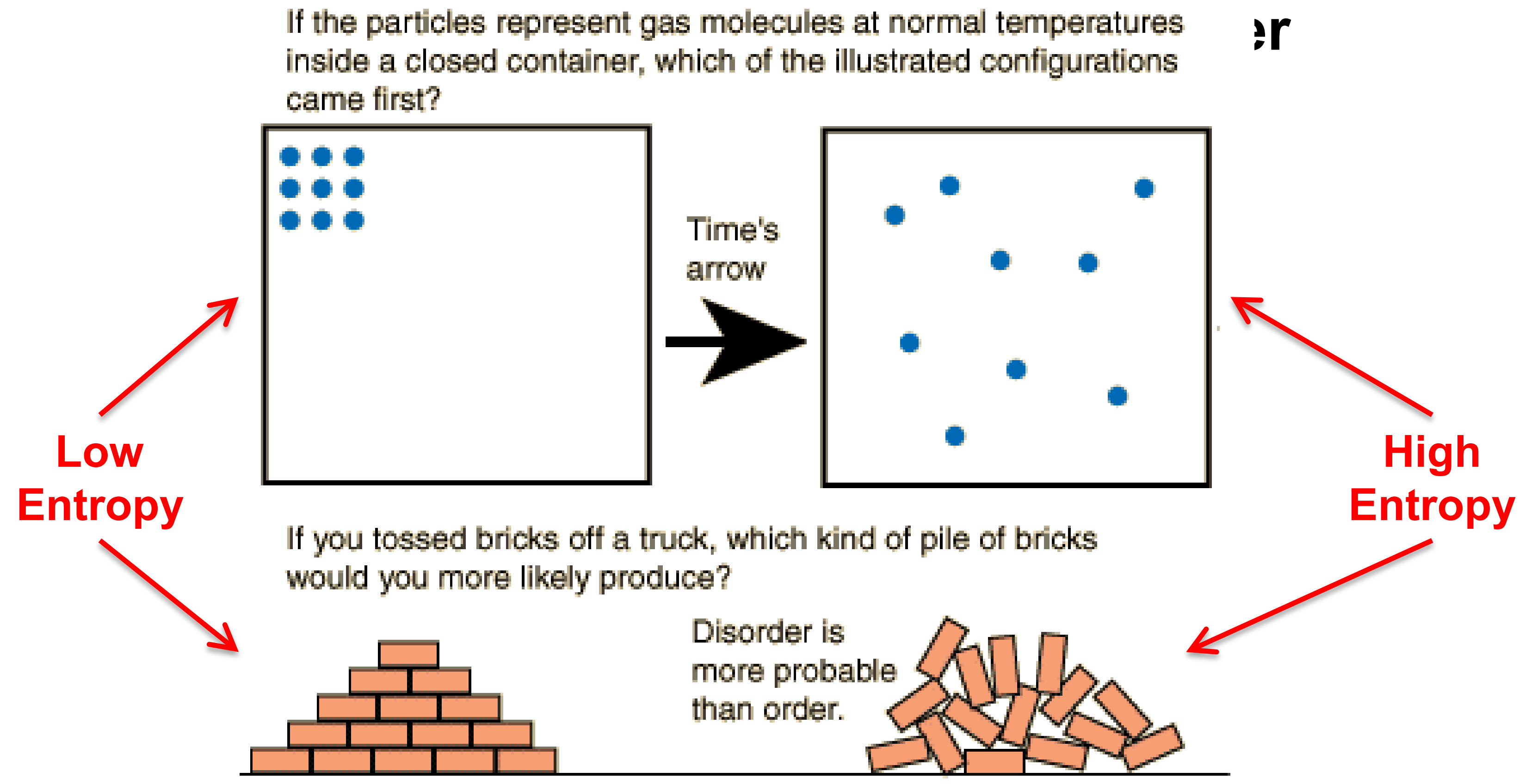
- Idea: a good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"



- *Patrons?* is a better choice
  - How can we quantify this?
  - One approach would be to use the classification error E directly (greedily)
    - Empirically it is found that this works poorly
  - **Much better is to use information gain (next slides)**
  - Other metrics are also used, e.g., Gini impurity, variance reduction
    - Often very similar results to information gain in practice

Entropy and Information

- “Entropy” is a measure of randomness



er
If the particles represent gas molecules at normal temperatures inside a closed container, which of the illustrated configurations came first?
Time's arrow
Low Entropy
High Entropy
If you tossed bricks off a truck, which kind of pile of bricks would you more likely produce?
Disorder is more probable than order.

https://www.youtube.com/watch?v=ZsY4WcQOrfk

Entropy, H(p), with only 2 outcomes

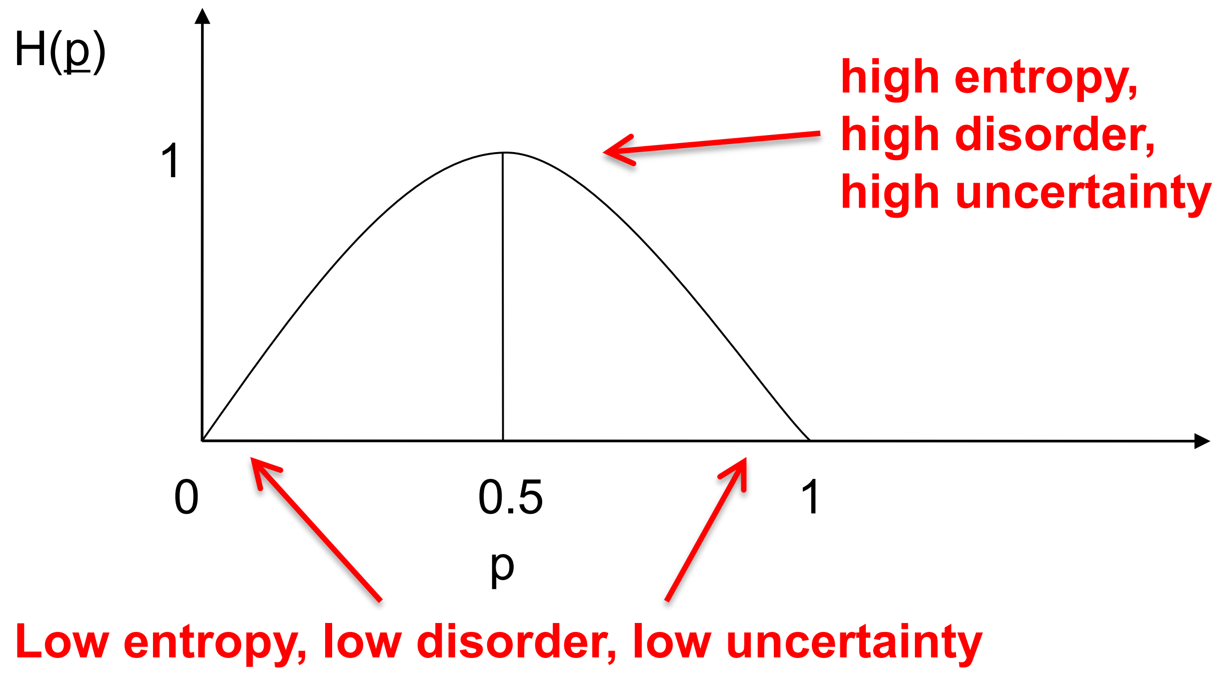
Consider 2 class problem:

p = probability of class #1,

$1 - p$ = probability of class #2

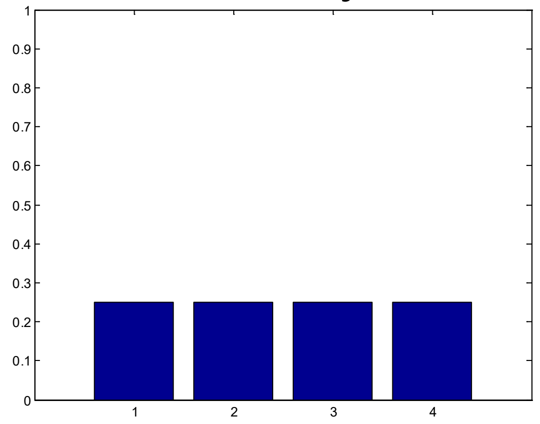
In binary case:

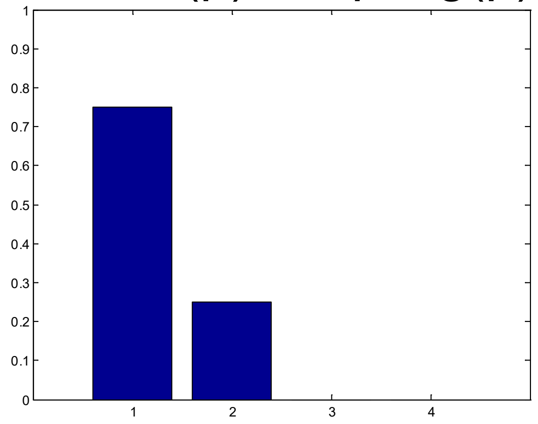
$$H(p) = -p \log p - (1-p) \log (1-p)$$

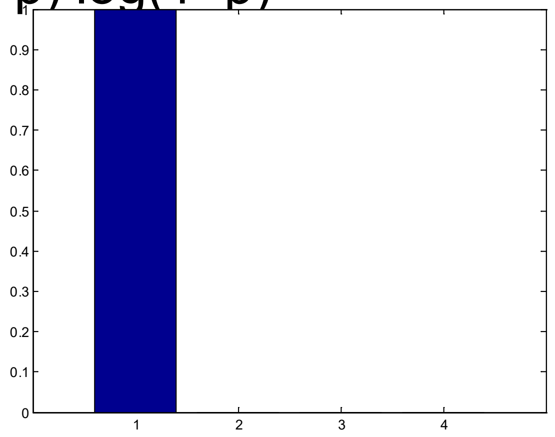


Entropy and Information

- **Entropy $H(X) = E[\log 1/P(X)] = \sum_{x \in X} P(x) \log 1/P(x)$**
  $= -\sum_{x \in X} P(x) \log P(x)$
  - Log base two, units of entropy are "bits"
  - If only two outcomes: $H(p) = -p \log(p) - (1-p) \log(1-p)$







$H(x) = .25 \log 4 + .25 \log 4 + .25 \log 4 + .25 \log 4 = \log 4 = 2$ bits

Max entropy for 4 outcomes

$H(x) = .75 \log 4/3 + .25 \log 4 = 0.8133$ bits

$H(x) = 1 \log 1 = 0$ bits

Min entropy

Information Gain

- H(P) = current entropy of class distribution P at a particular node,
  before further partitioning the data

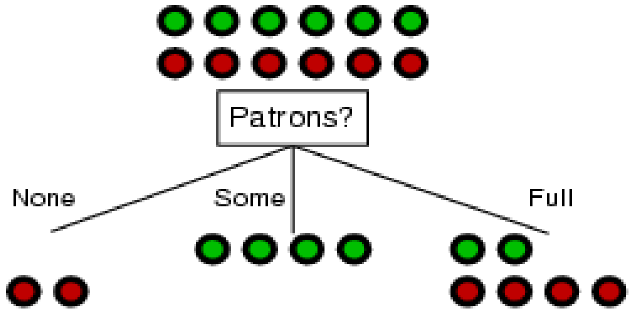
- H(P | A) = conditional entropy given attribute A
  = weighted average entropy of conditional class distribution,
  after partitioning the data according to the values in A

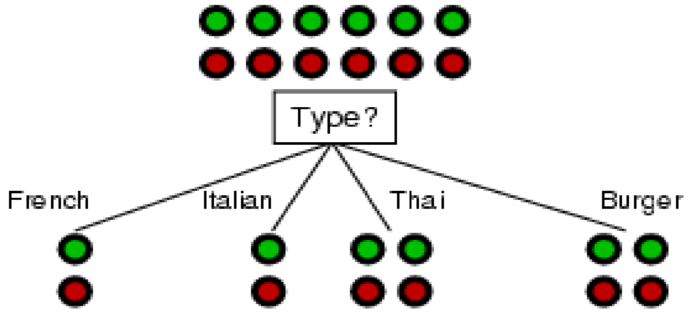
- Gain(A) = H(P) – H(P | A)
  - Sometimes written IG(A) = InformationGain(A)

- Simple rule in decision tree learning
  - **At each internal node, split on the node with the largest information gain [or equivalently, with smallest H(P|A)]**

- Note that by definition, conditional entropy can't be greater than the entropy, so Information Gain must be non-negative

Choosing an attribute





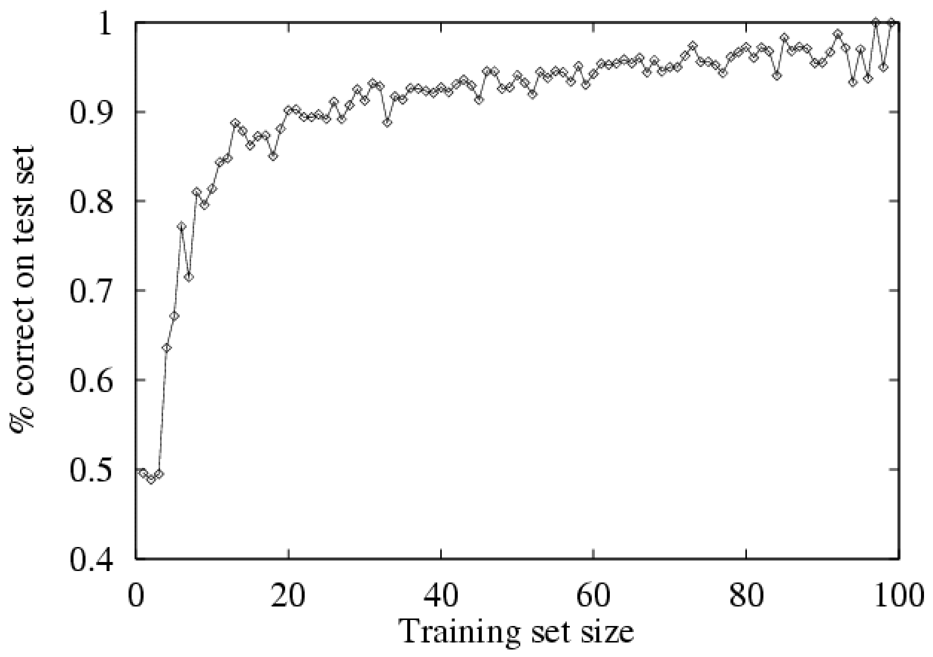
IG(Patrons) = 0.541 bits

IG(Type) = 0 bits

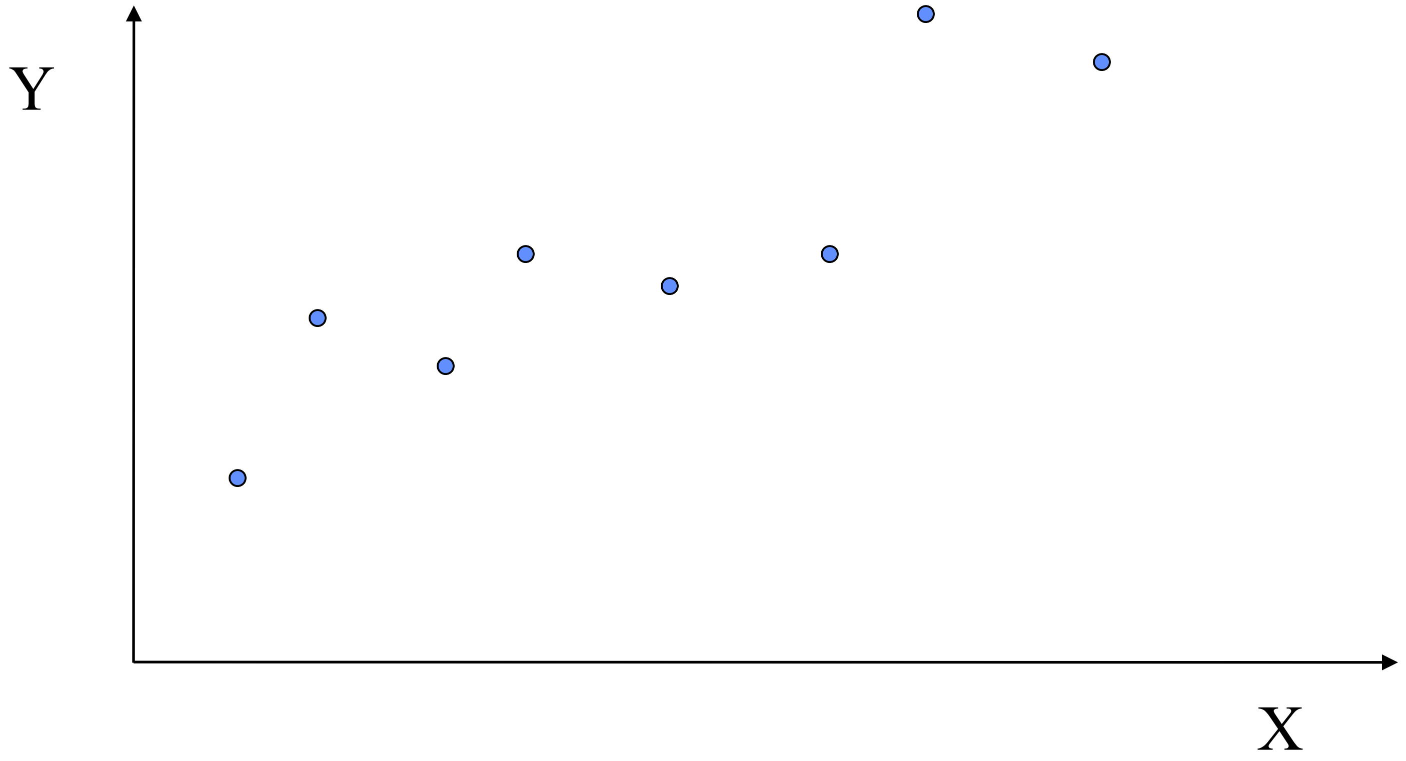
Example of Test Performance

Restaurant problem

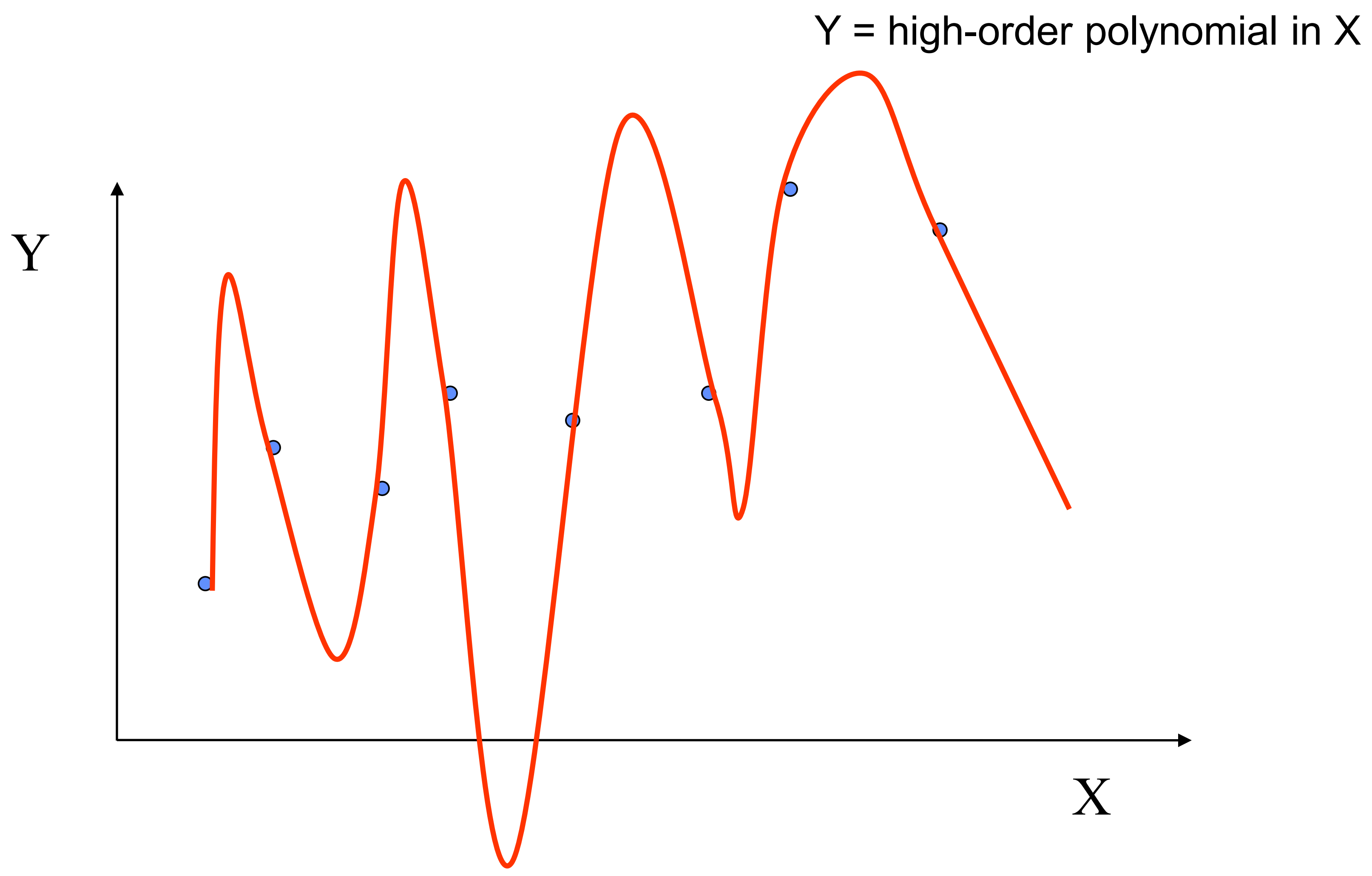
- simulate 100 data sets of different sizes
- train on this data, and assess performance on an independent test set
- learning curve = plotting accuracy as a function of training set size
- typical "diminishing returns" effect (some nice theory to explain this)



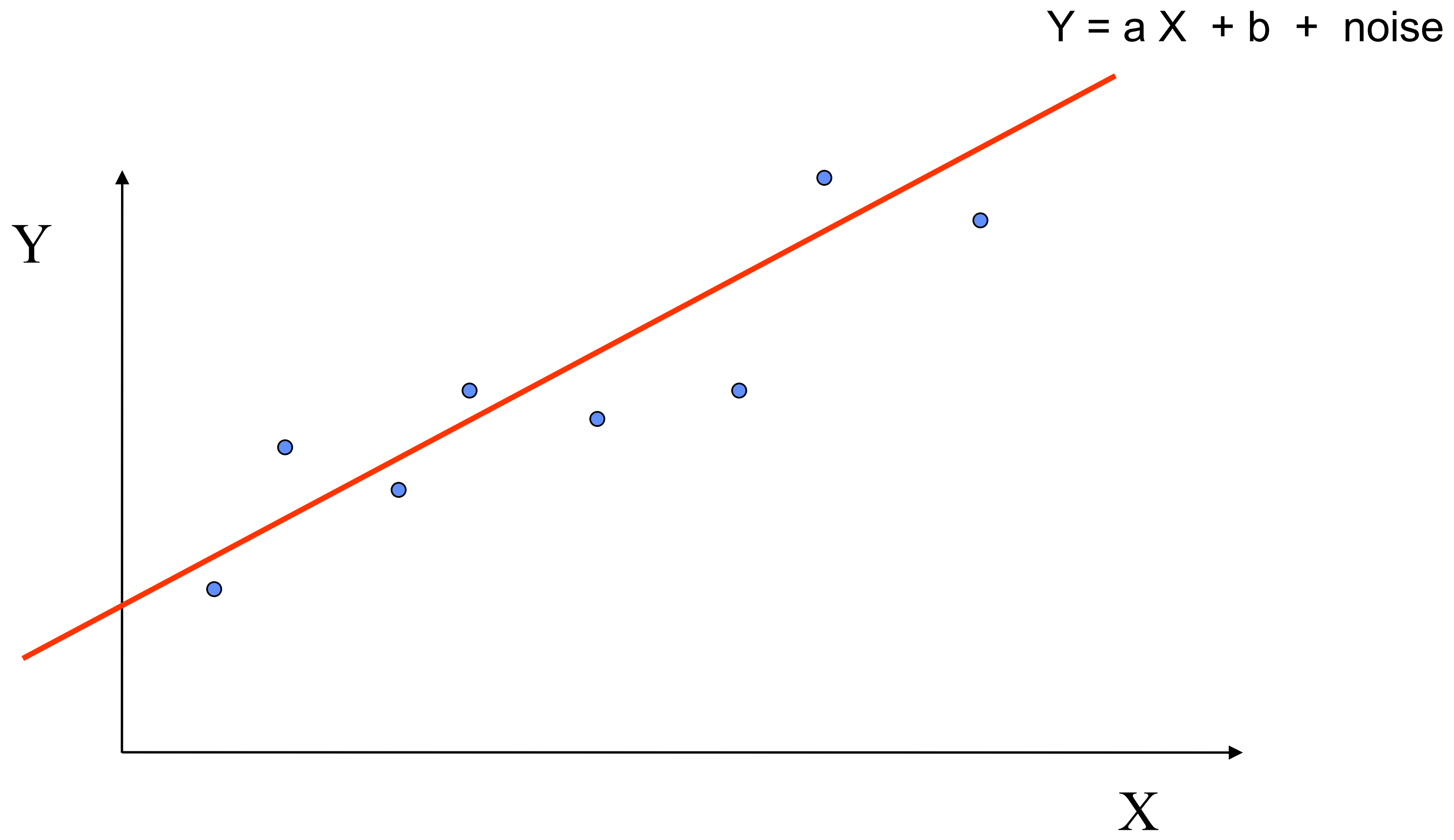
Overfitting and Underfitting



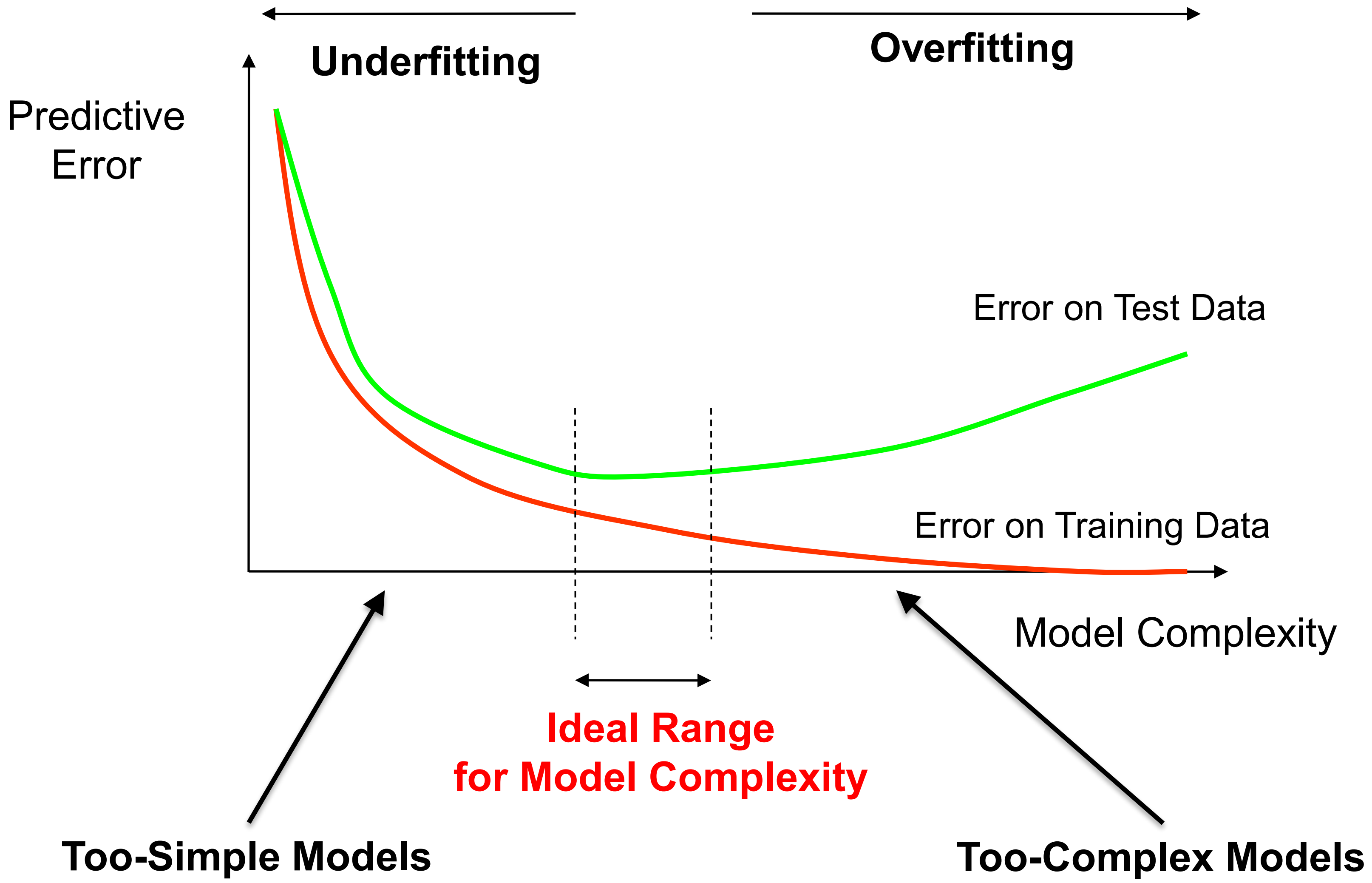
A Complex Model



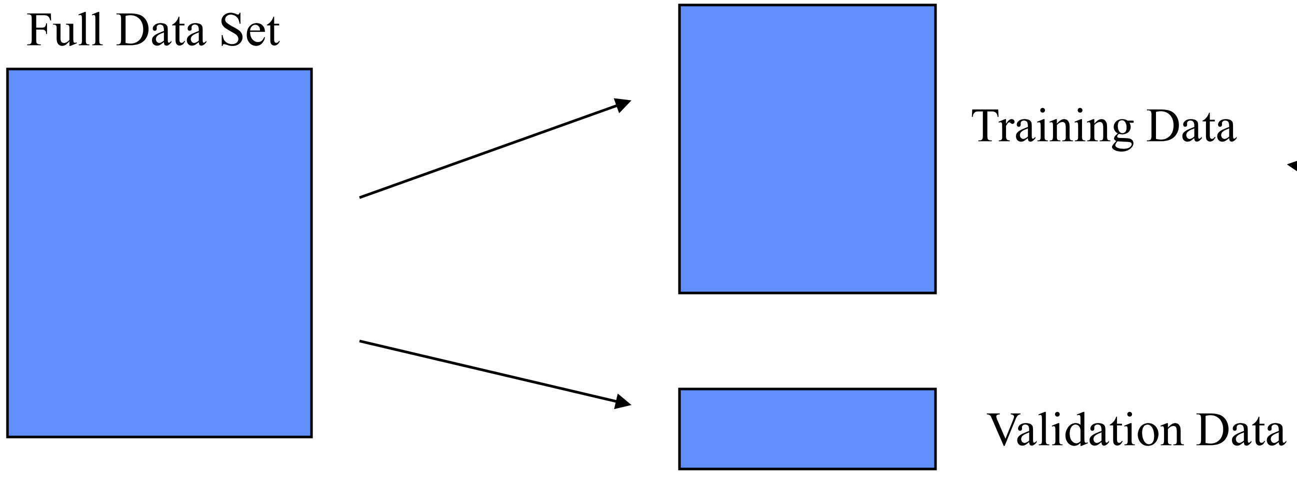
A Much Simpler Model



How Overfitting affects Prediction



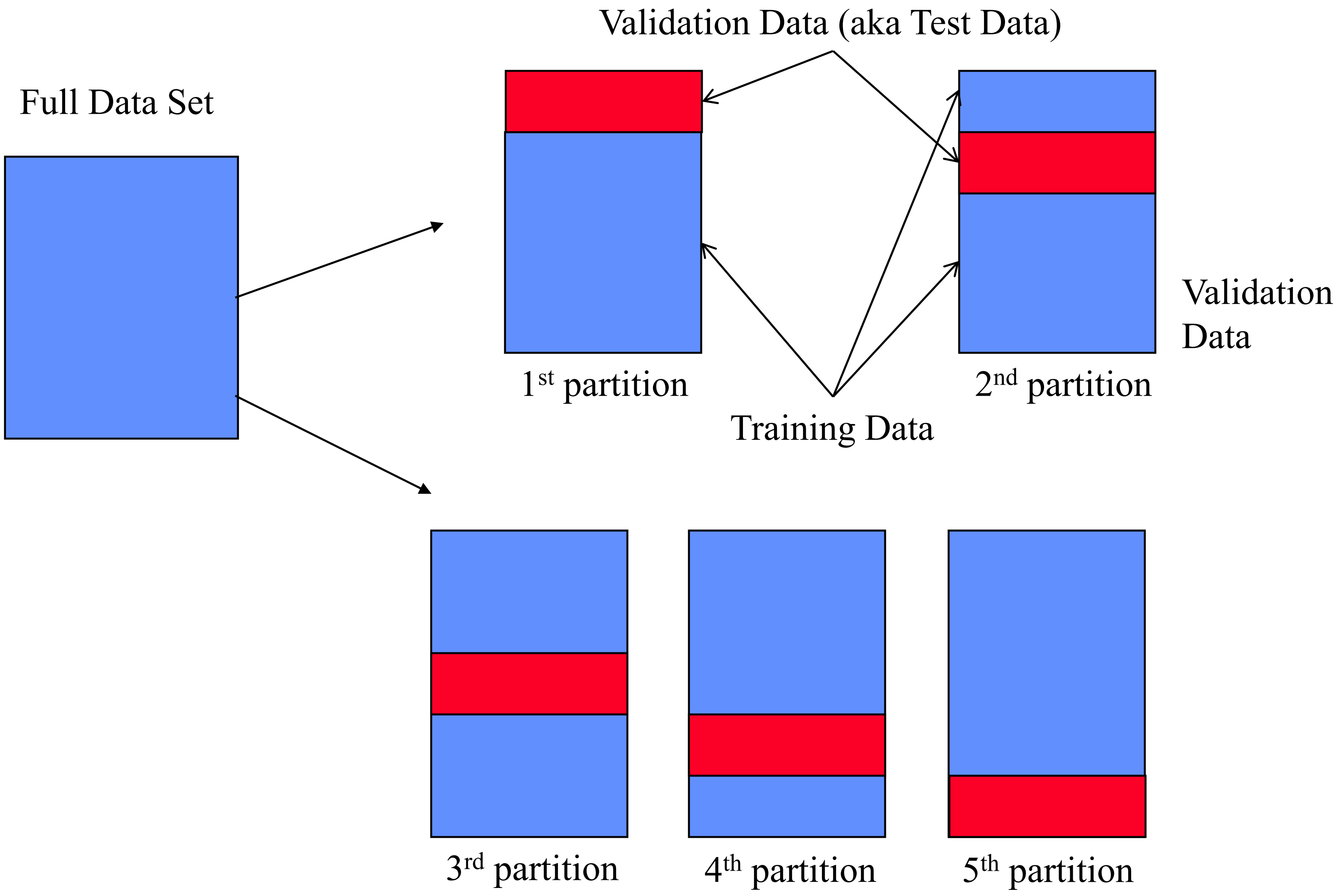
Training and Validation Data



Idea: train each model on the "training data"

and then test each model's accuracy on the validation data

Disjoint Validation Data Sets



The k-fold Cross-Validation Method

- Why just choose one particular 90/10 "split" of the data?
  - In principle we could do this multiple times

- "k-fold Cross-Validation" (e.g., k=10)
  - randomly partition our full data set into k disjoint subsets (each roughly of size n/k, n = total number of training data points)
    - for i = 1:10 (here k = 10)
      - train on 90% of data,
      - Acc(i) = accuracy on other 10%
    - end
    - Cross-Validation-Accuracy = $1/k \sum_i$ Acc(i)
  - choose the method with the highest cross-validation accuracy
  - common values for k are 5 and 10
  - Can also do "leave-one-out" where k = n

You will be expected to know

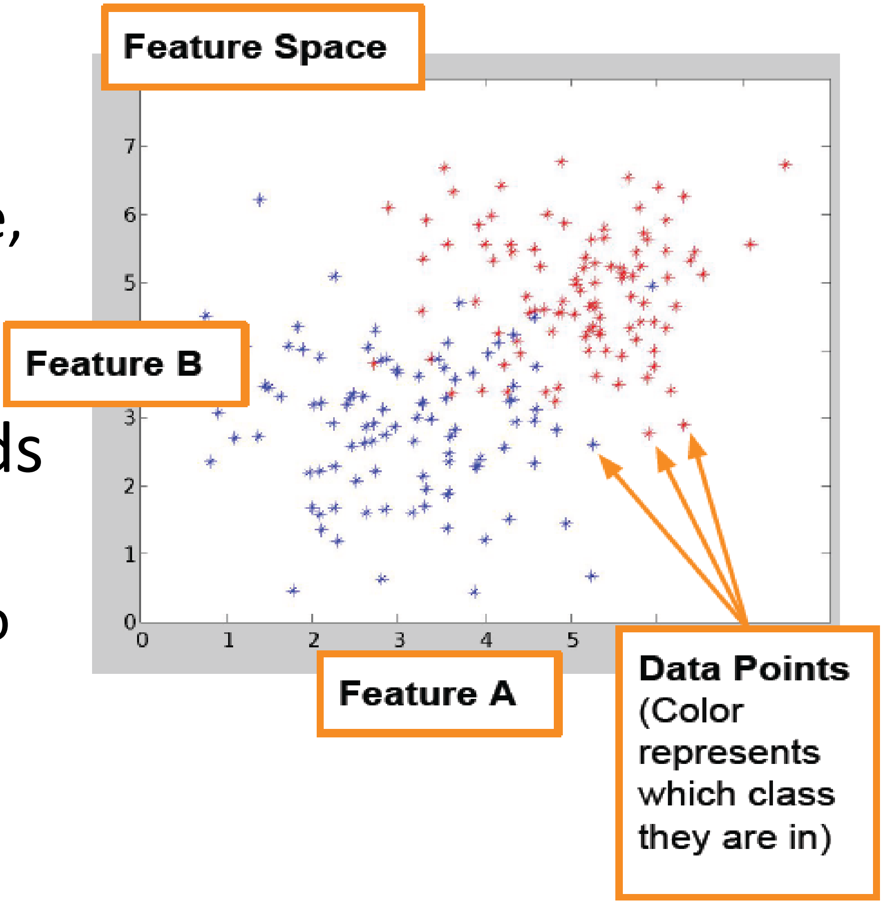
- Understand Attributes, Error function, Classification, Regression, Hypothesis (Predictor function)
- What is Supervised Learning?
- Decision Tree Algorithm
- Entropy
- Information Gain
- Tradeoff between train and test with model complexity
- Cross validation

Review Machine Learning Classifiers Chapters 18.5-18.12; 20.2.2

- Decision Regions and Decision Boundaries
- Classifiers:
  - Decision trees
  - K-nearest neighbors
  - Perceptrons
  - Support vector Machines (SVMs), Neural Networks
  - Naïve Bayes

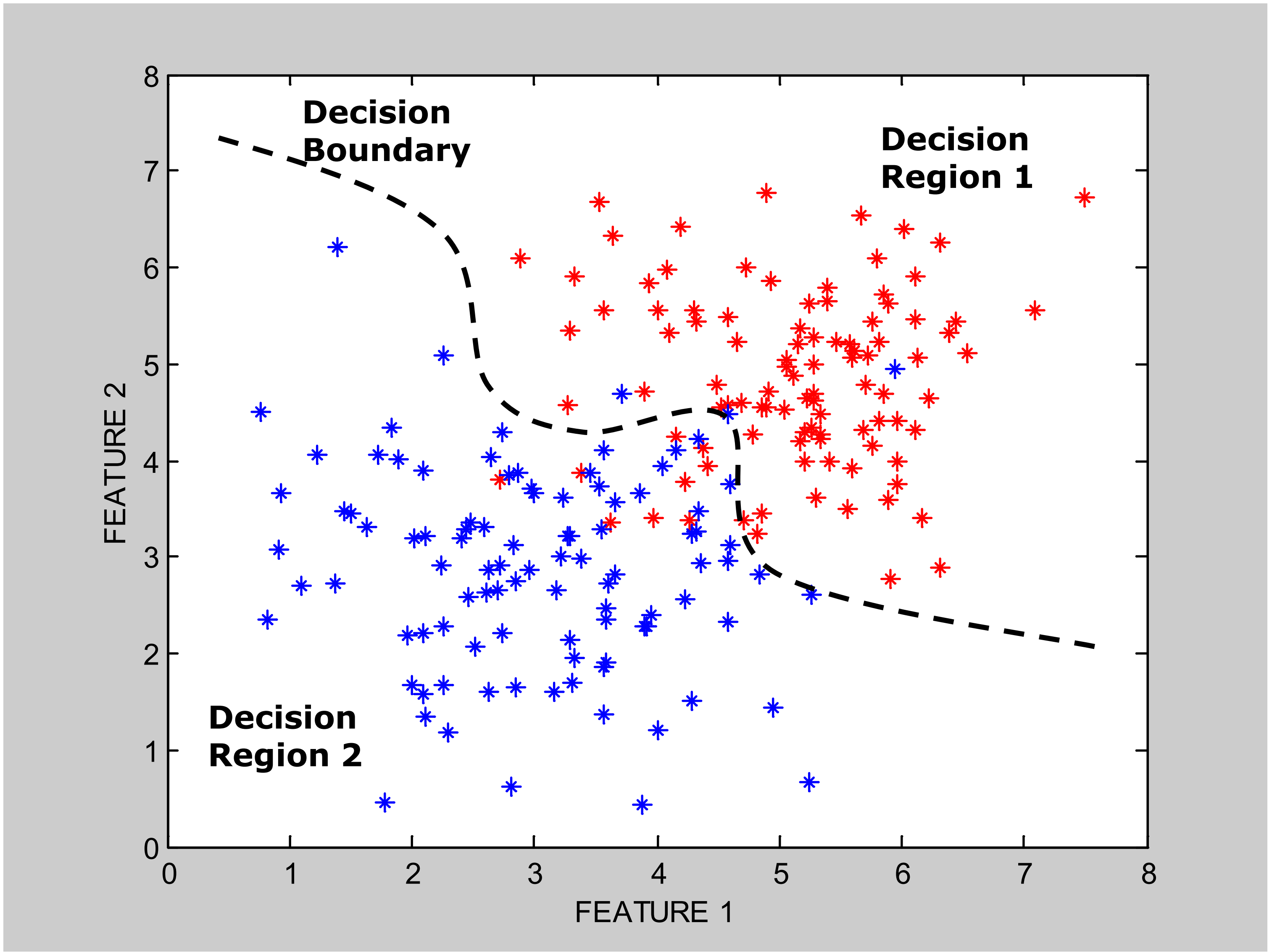
A Different View on Data Representation

- Data pairs can be plotted in "feature space"
- Each axis represents one feature.
  - This is a d dimensional space, where d is the number of features.
- Each data case corresponds to one point in the space.
  - In this figure we use color to represent their class label.



Decision Boundaries

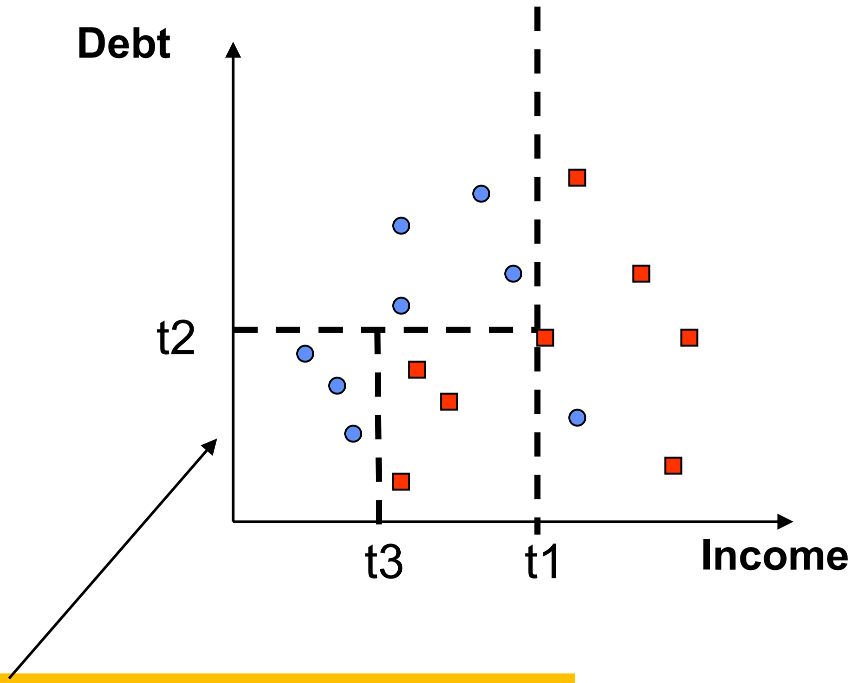
Can we find a boundary that separates the two classes?

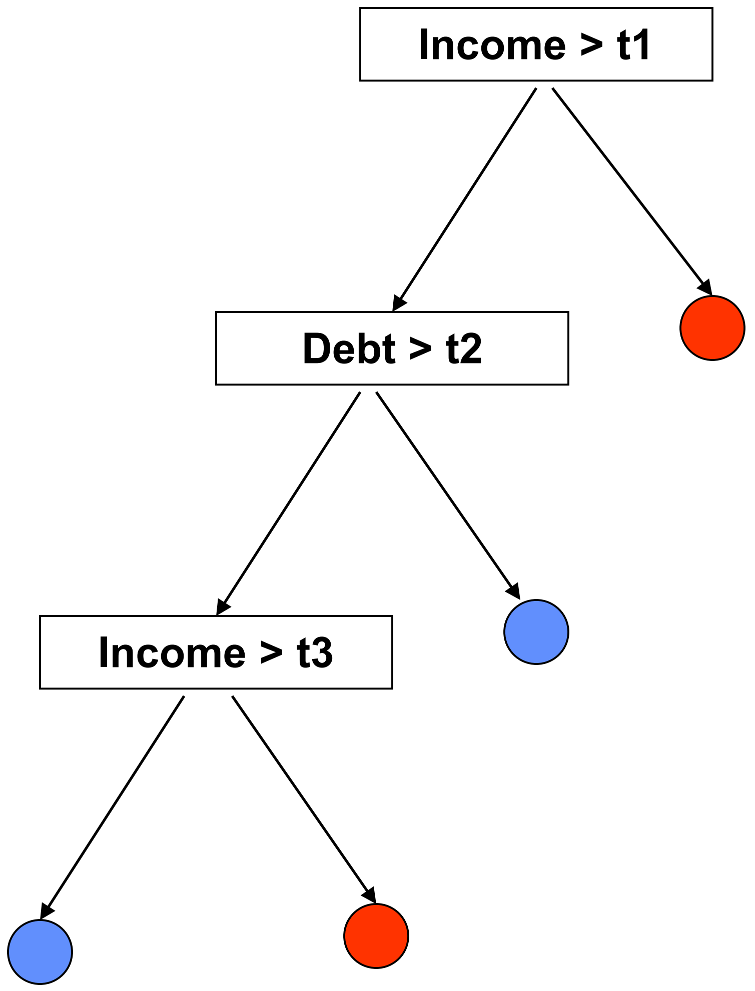


Classification in Euclidean Space

- A classifier is a partition of the feature space into disjoint decision regions
  - Each region has a label attached
  - Regions with the same label need not be contiguous
  - For a new test point, find what decision region it is in, and predict the corresponding label
- Decision boundaries = boundaries between decision regions
  - The “dual representation” of decision regions
- Learning a classifier $\Leftrightarrow$ searching for the decision boundaries that optimize our objective function

Decision Tree Example



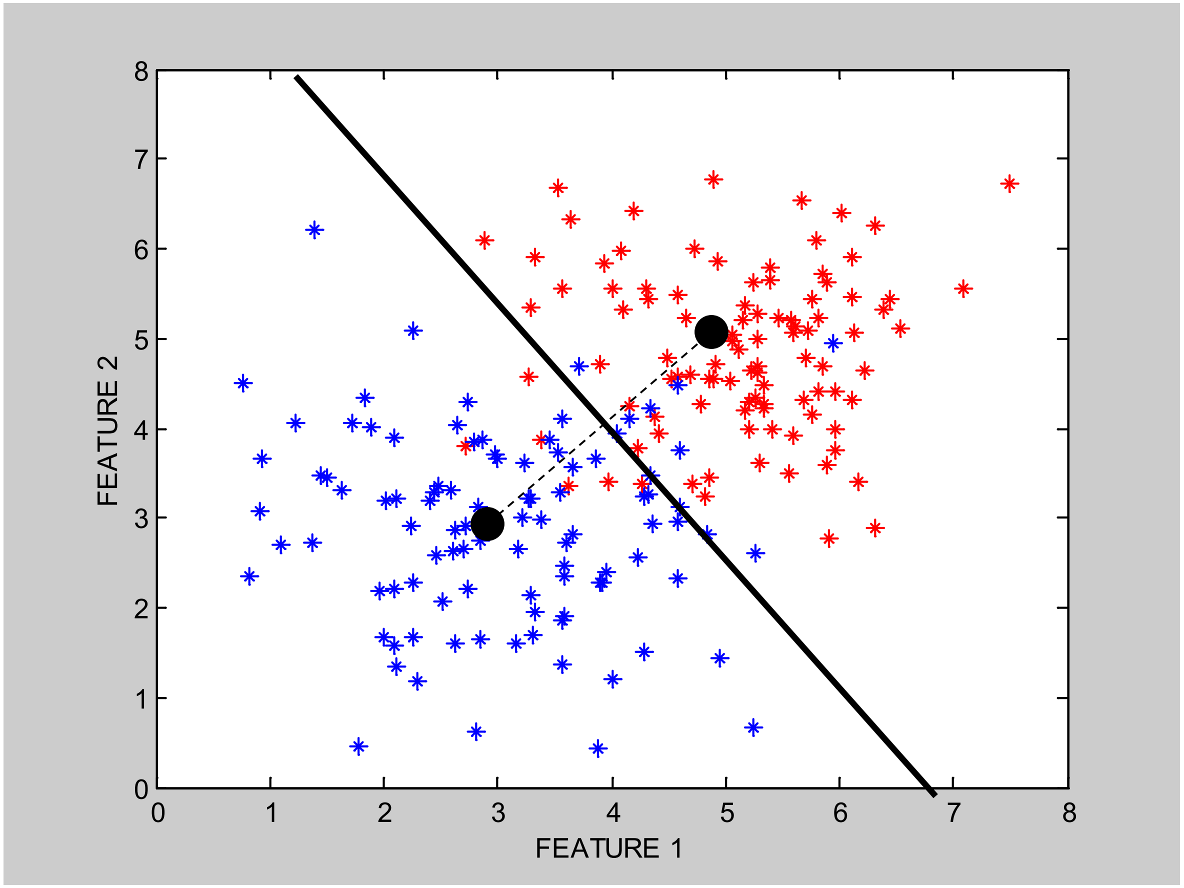


Note: tree boundaries are linear and axis-parallel

A Simple Classifier: Minimum Distance Classifier

- Training
  - Separate training vectors by class
  - Compute the mean for each class, $\underline{\mu}_k$, k = 1,… m
- Prediction
  - Compute the closest mean to a test vector $\underline{x}'$ (using Euclidean distance)
  - Predict the corresponding class
- In the 2-class case, the decision boundary is defined by the locus of the hyperplane that is halfway between the 2 means and is orthogonal to the line connecting them
- This is a very simple-minded classifier – easy to think of cases where it will not work very well

Minimum Distance Classifier



Another Example: Nearest Neighbor Classifier

- The nearest-neighbor classifier
  - Given a test point x′, compute the distance between x′ and each input data point
  - Find the closest neighbor in the training data
  - Assign x′ the class label of this neighbor
  - (sort of generalizes minimum distance classifier to exemplars)

- The nearest neighbor classifier results in piecewise linear decision boundaries

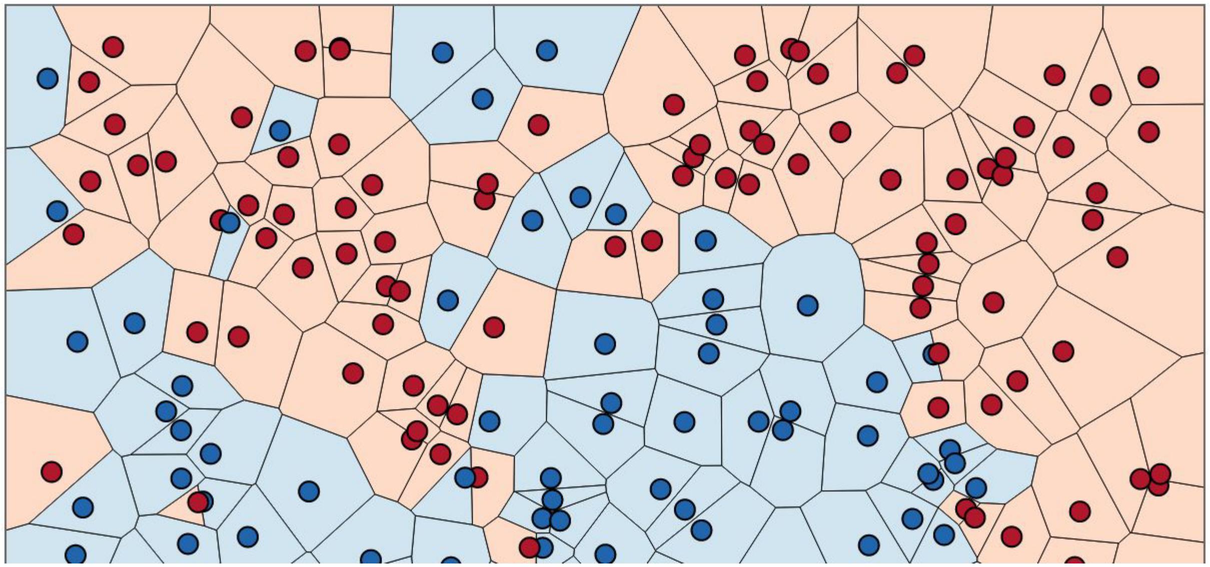
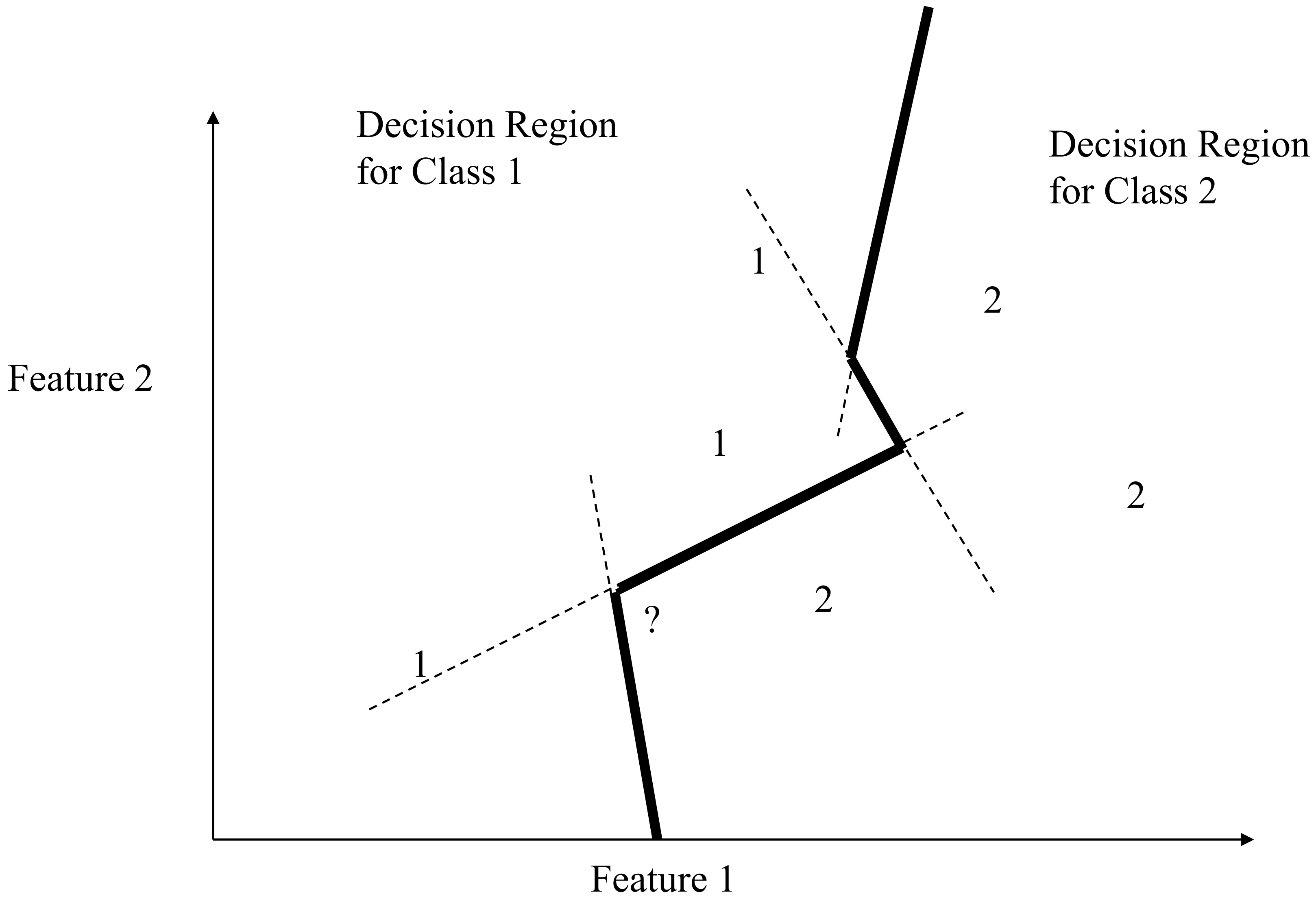


Image Courtesy: http://scott.fortmann-roe.com/docs/BiasVariance.html

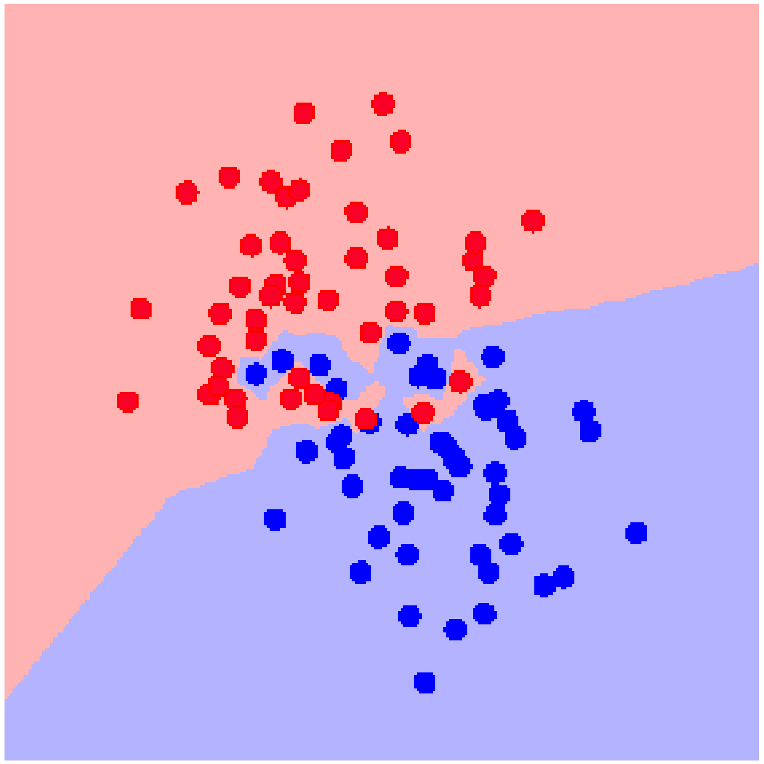
Overall Boundary = Piecewise Linear



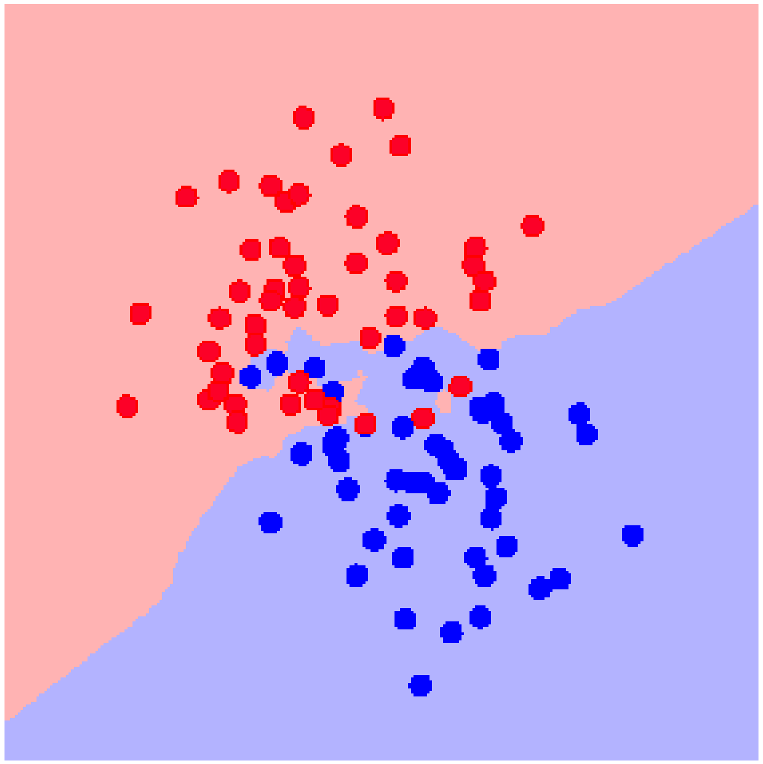
kNN Decision Boundary

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

K = 1



K = 3



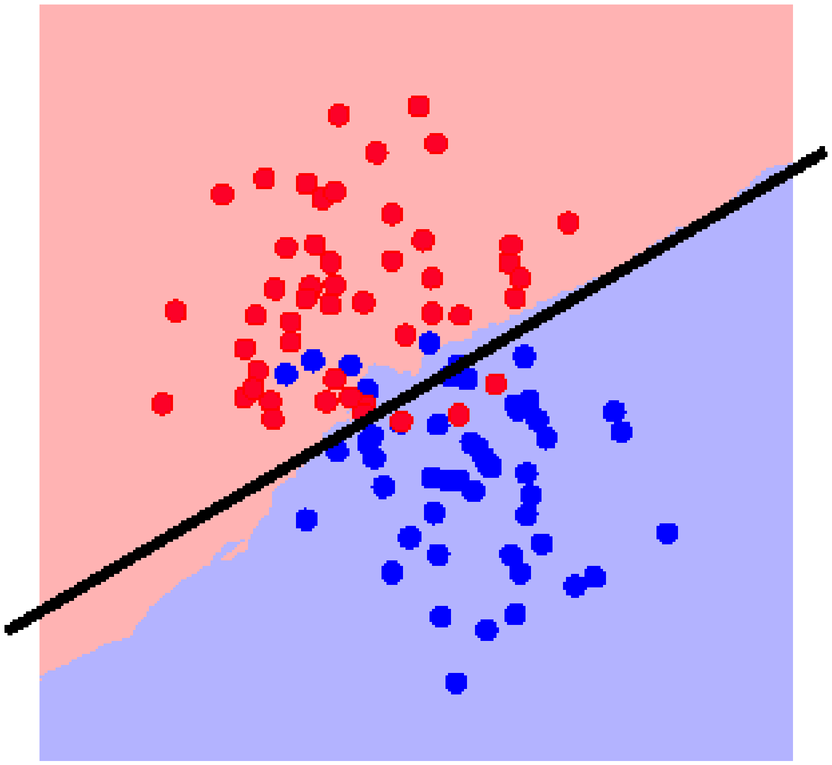
kNN Decision Boundary

- piecewise linear decision boundary
- Increasing k "simplifies" decision boundary
  - Majority voting means less emphasis on individual points

- True ("best") decision boundary
  - In this case is linear
  - Compared to kNN: not bad!

Larger K $\Longrightarrow$ Smoother boundary

K = 25



Linear Classifiers

- Linear classifiers classification decision based on the value of a linear combination of the characteristics.
  - Linear decision boundary (single boundary for 2-class case)
- We can always represent a linear decision boundary by a linear equation:

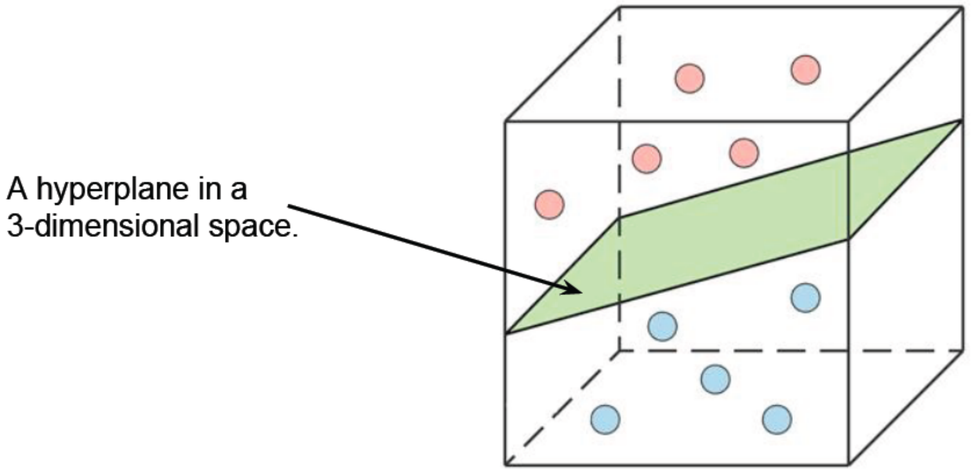
$$w_1 x_1 + w_2 x_2 + \ldots + w_d x_d = \sum_j w_j x_j = w^T x = 0$$

- The w_i are weights; the x_i are feature values

Linear Classifiers

$$w_1x_1 + w_2x_2 + \ldots + w_dx_d = \sum_j w_jx_j = w^Tx = 0$$

- This equation defines a hyperplane in d dimensions
  - A hyperplane is a subspace whose dimension is one less than that of its ambient space.
  - If a space is 3-dimensional, its hyperplanes are the 2-dimensional planes;
  - if a space is 2-dimensional, its hyperplanes are the 1-dimensional lines.

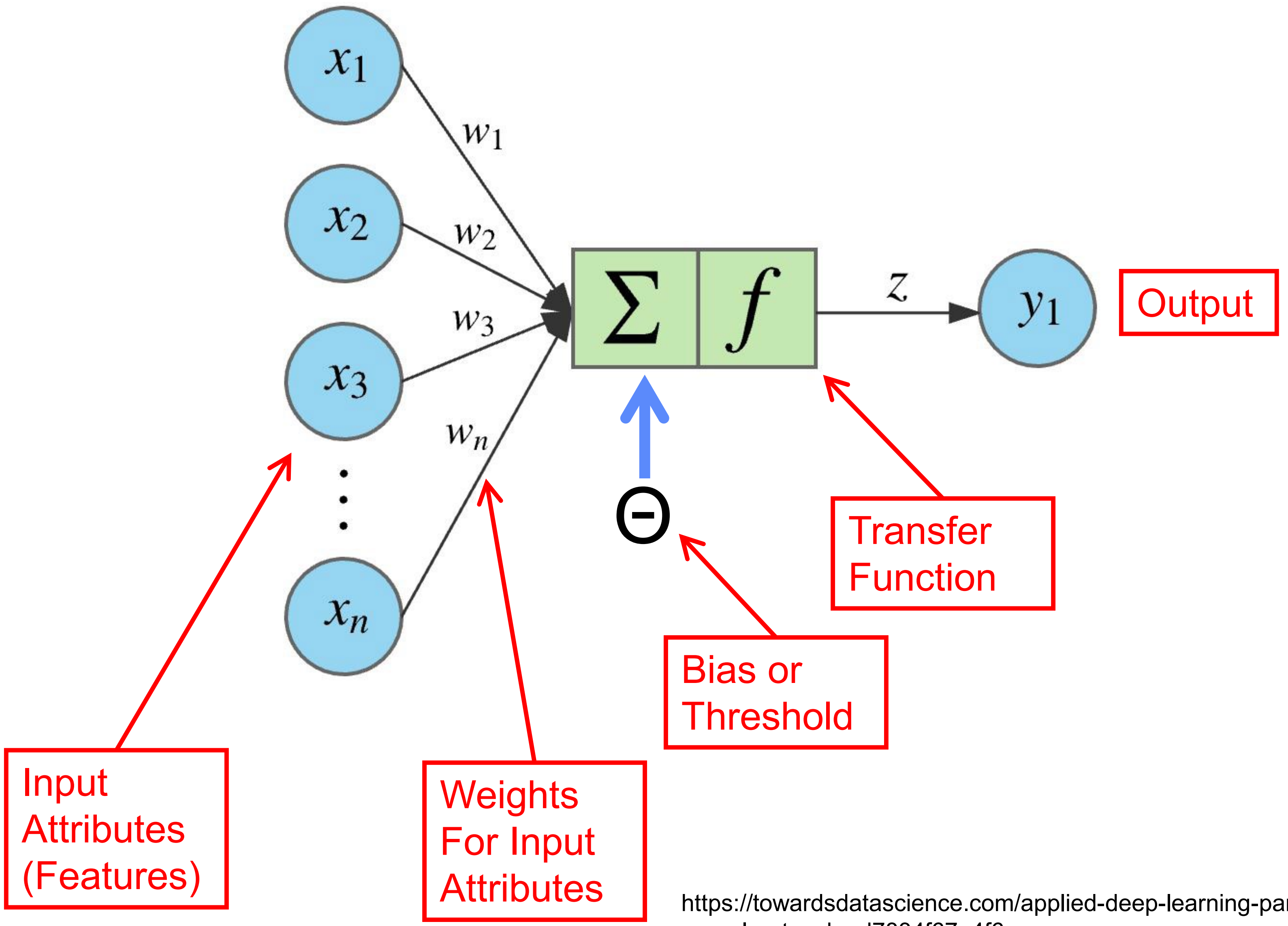


https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

Linear Classifiers

- For prediction we simply see if $\sum_j w_j x_j > 0$ for new data x.
  - If so, predict x to be positive
  - If not, predict x to be negative
- Learning consists of searching in the d-dimensional weight space for the set of weights (the linear boundary) that minimizes an error measure
- A threshold can be introduced by a “dummy” feature
  - The feature value is always 1.0
  - Its weight corresponds to (the negative of) the threshold
- Note that a minimum distance classifier is a special case of a linear classifier

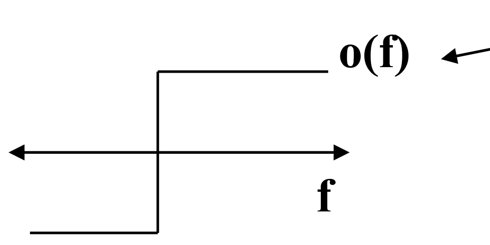
The Perceptron Classifier
(pages 729-731 in text)



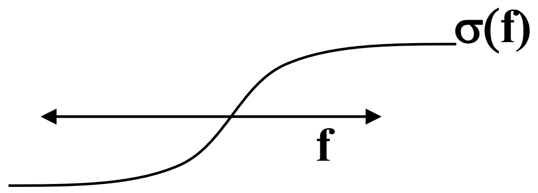
https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

Two different types of perceptron output

x-axis below is f(x) = f = weighted sum of inputs
y-axis is the perceptron output



Thresholded output,
takes values +1 or -1



Sigmoid output, takes
real values between -1 and +1

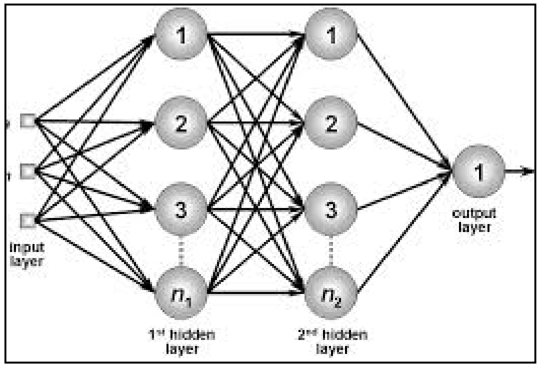
The sigmoid is in effect an approximation
to the threshold function above, but
has a gradient that we can use for learning

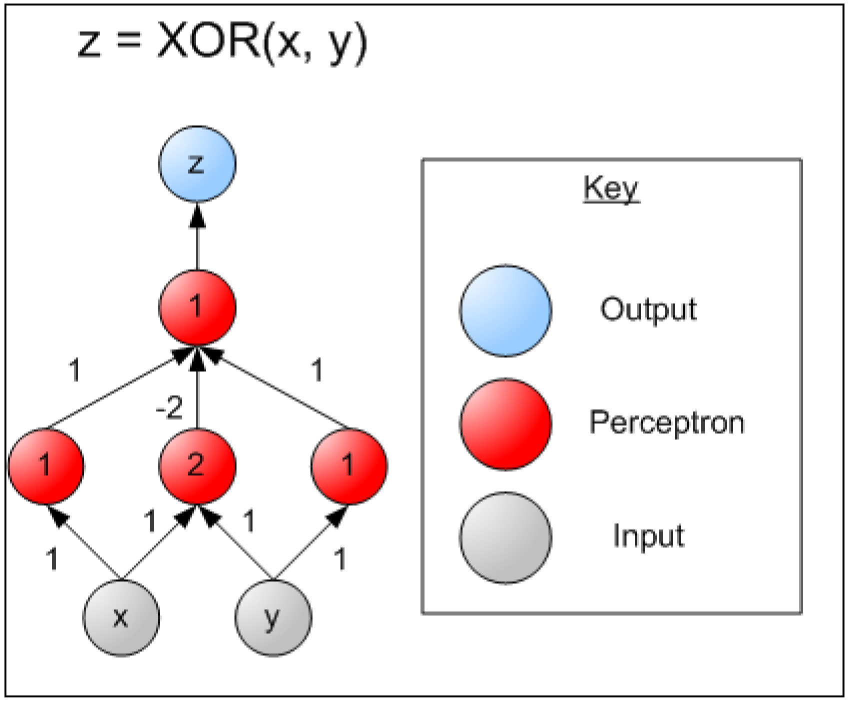
Sigmoid function is defined as

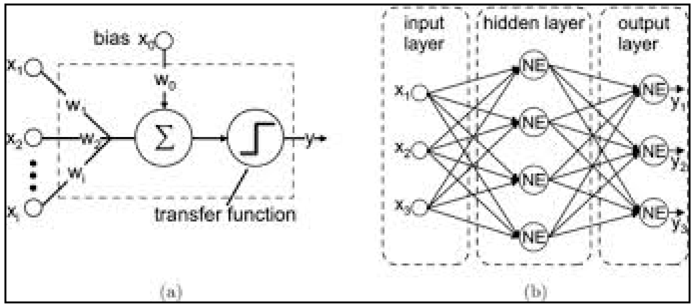
$$\sigma[f] = [\, 2 / (1 + \exp[-f]) \,] - 1$$

Multi-Layer Perceptrons (Artificial Neural Networks)

(sections 18.7.3-18.7.4 in textbook)







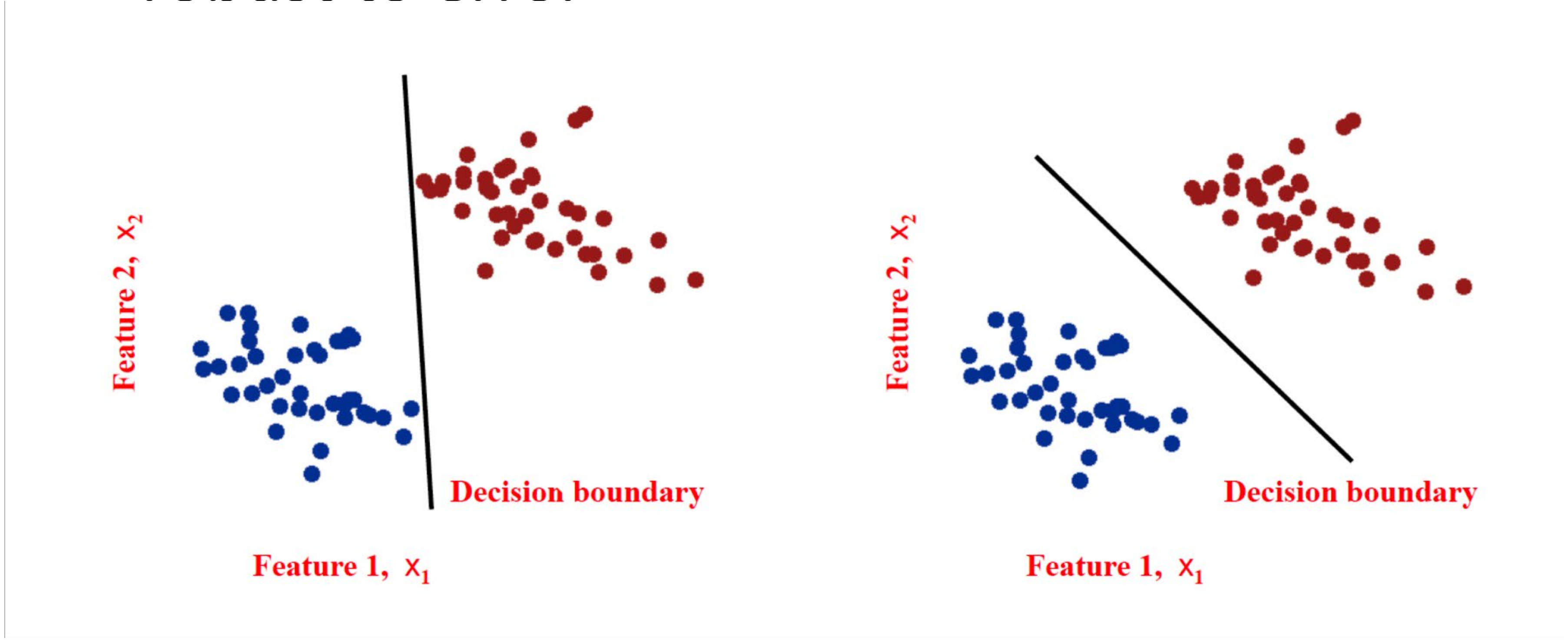
Multi-Layer Perceptrons (Artificial Neural Networks)

(sections 18.7.3-18.7.4 in textbook)

- What if we took K perceptrons and trained them in parallel and then took a weighted sum of their sigmoidal outputs?
  - This is a multi-layer neural network with a single "hidden" layer (the outputs of the first set of perceptrons) What if we hooked them up into a general Directed Acyclic Graph?
  - Can create simple "neural circuits" (but no feedback; not fully general)
  - Often called neural networks with hidden units
- How would we train such a model?
  - Backpropagation algorithm = clever way to do gradient descent
  - Bad news: many local minima and many parameters
    - training is hard and slow
  - Good news: can learn general non-linear decision boundaries
  - Generated much excitement in AI in the late 1980's and 1990's
  - New current excitement with very large "deep learning" networks

Which decision boundary is "better"?

- Both have zero training error (perfect training accuracy).
- But one seems intuitively better, more robust to error



Support Vector Machines (SVM): “Modern perceptrons” (section 18.9, R&N)

- A modern linear separator classifier
  - Essentially, a perceptron with a few extra wrinkles

- Constructs a **“maximum margin separator”**
  - A linear decision boundary with the largest possible distance from the decision boundary to the example points it separates
  - “Margin” = Distance from decision boundary to closest example
  - The “maximum margin” helps SVMs to generalize well

- Can embed the data in a non-linear higher dimension space
  - Constructs a linear separating hyperplane in that space
    - **This can be a non-linear boundary in the original space**
  - Algorithmic advantages and simplicity of linear classifiers
  - Representational advantages of non-linear decision boundaries

- **Currently most popular “off-the shelf” supervised classifier.**

Constructs a “maximum margin separator”

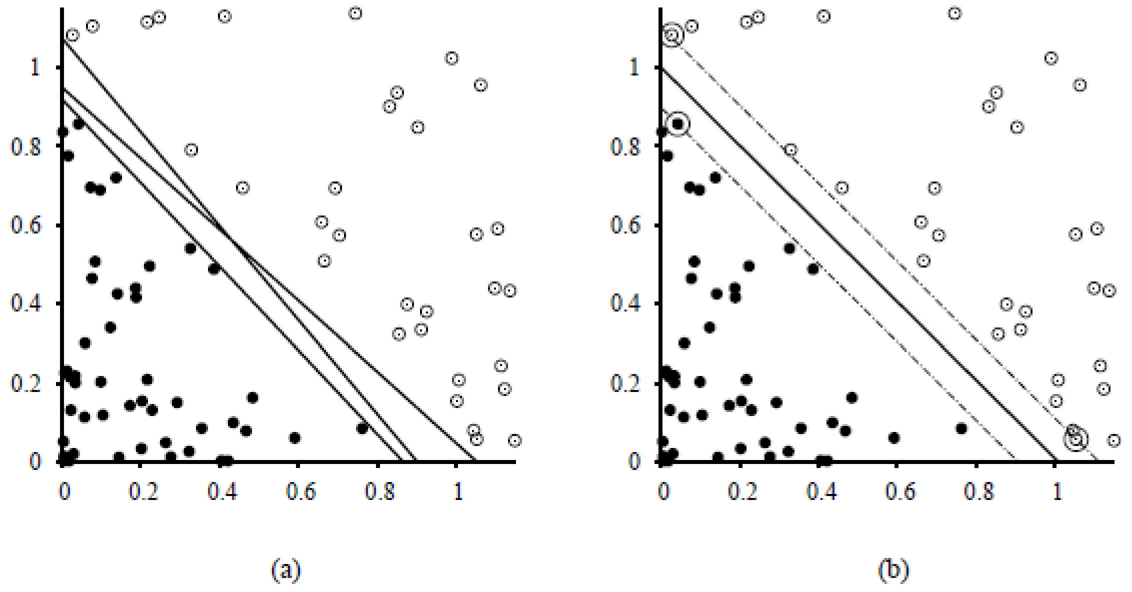


Figure 18.30 **FILES:** . Support vector machine classification: (a) Two classes of points (black and white circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the **margin** (area between dashed lines). The **support vectors** (points with large circles) are the examples closest to the separator.

Can embed the data in a non-linear higher dimension space

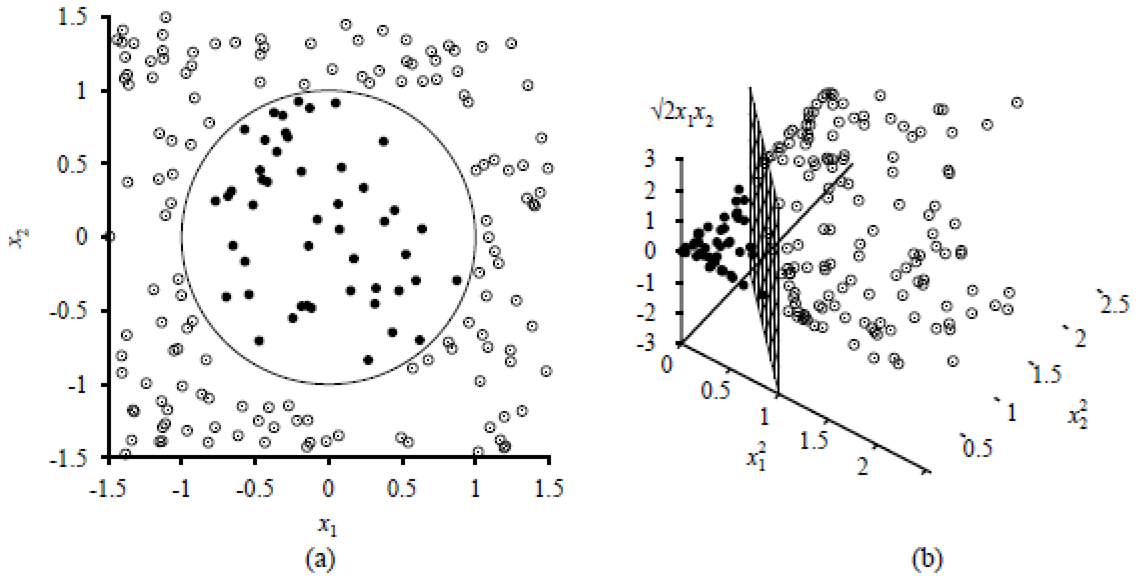
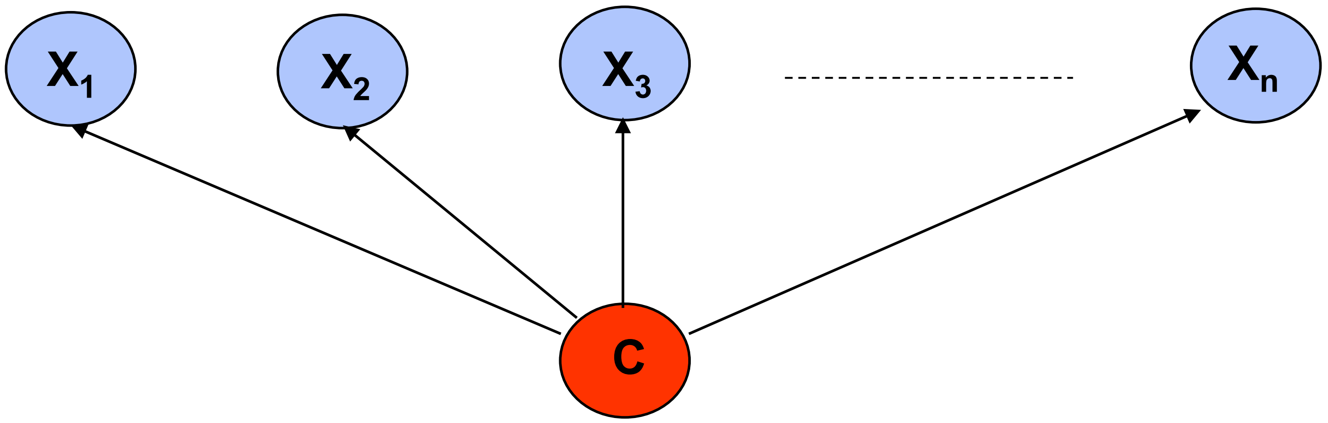


Figure 18.31 **FILES: .** (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 18.29(b) gives a closeup of the separator in (b).

Naïve Bayes Model

(section 20.2.2 R&N 3rd ed.)



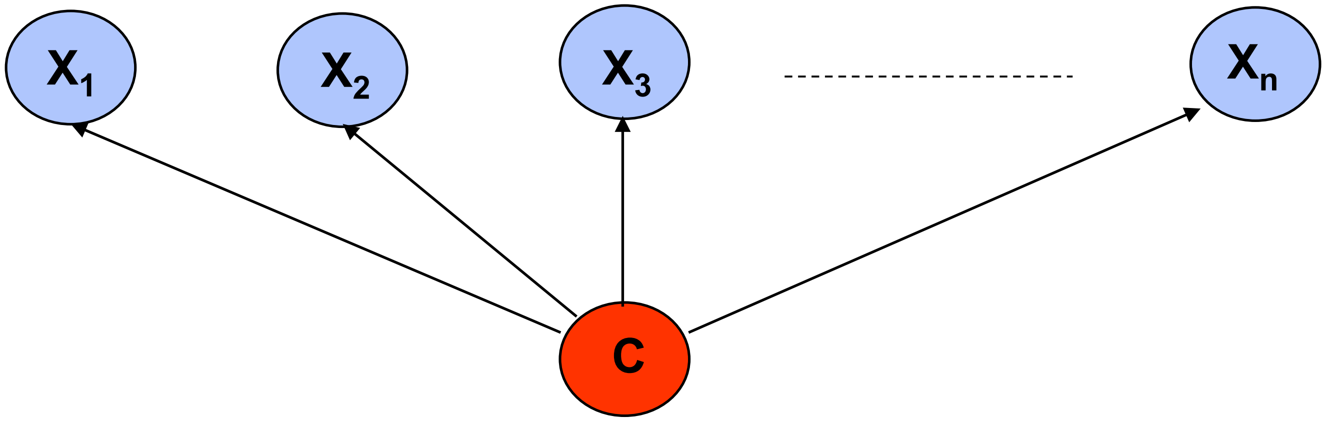
Basic Idea: We want to estimate $P(C \mid X_1,\ldots X_n)$, but it's hard to think about computing the probability of a class from input attributes of an example.

Solution: Use Bayes' Rule to turn $P(C \mid X_1,\ldots X_n)$ into a proportionally equivalent expression that involves only $P(C)$ and $P(X_1,\ldots X_n \mid C)$.
Then assume that feature values are conditionally independent given class, which allows us to turn $P(X_1,\ldots X_n \mid C)$ into $\Pi_i \; P(X_i \mid C)$.

$$P(C \mid X_1,\ldots X_n) = P(C)\, P(X_1,\ldots X_n \mid C) / P(X_1,\ldots X_n) \propto P(C)\, \Pi_i \; P(X_i \mid C)$$

We estimate $P(C)$ easily from the frequency with which each class appears within our training data, and we estimate $P(X_i \mid C)$ easily from the frequency with which each X_i appears in each class C within our training data.

Naïve Bayes Model (section 20.2.2 R&N 3rd ed.)



By Bayes Rule: $P(C \mid X_1,\ldots X_n)$ is proportional to $P(C)\ \Pi_i\ P(X_i \mid C)$
[note: denominator $P(X_1,\ldots X_n)$ is constant for all classes, may be ignored.]

Features Xi are conditionally independent given the class variable C

- choose the class value c_i with the highest $P(c_i \mid x_1,\ldots, x_n)$
- simple to implement, often works very well
- e.g., spam email classification: X's = counts of words in emails

Conditional probabilities $P(X_i \mid C)$ can easily be estimated from labeled date

- Problem: Need to avoid zeroes, e.g., from limited training data
- Solutions: Pseudo-counts, beta[a,b] distribution, etc.

Naïve Bayes Model (2)

$$P(C \mid X_1,\ldots X_n) \approx \alpha \; \Pi \; P(X_i \mid C) \; P(C)$$

Probabilities P(C) and P(Xi | C) can easily be estimated from labeled data

P(C = cj) ≈ #(Examples with class label cj) / #(Examples)

P(Xi = xik | C = cj)
≈ #(Examples with Xi value xik and class label cj)
/ #(Examples with class label cj)

Usually easiest to work with logs

$$\log [\, P(C \mid X_1,\ldots X_n) \,]$$
$$= \log \alpha + \Sigma \; [\, \log P(X_i \mid C) + \log P(C) \,]$$

DANGER: Suppose ZERO examples with Xi value xik and class label cj ?
An unseen example with Xi value xik will NEVER predict class label cj !

Practical solutions: Pseudocounts, e.g., add 1 to every #() , etc.
Theoretical solutions: Bayesian inference, beta distribution, etc.

Final Review

- Bayesian Networks: R&N Chap 14.1-14.5
- Game Search: R&N Chap 5.1-5.4
- Constraint Satisfaction: R&N Chap 6.1-6.4, except 6.3.3
- Machine Learning: R&N Chap 18.1-18.12, 20.2

GOOD LUCK!