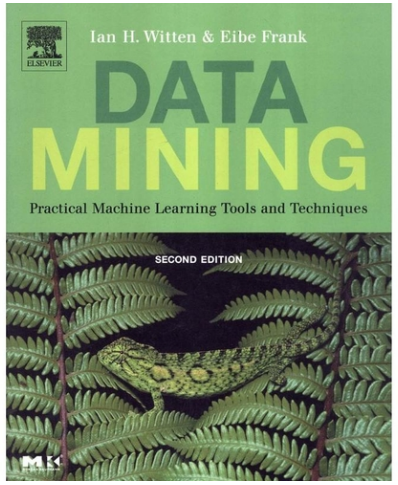
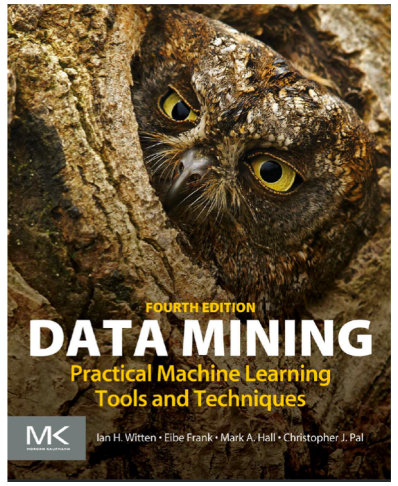
# Machine Learning for Cyber-Security & Artificial Intelligence

## Part 1 – Decision trees

Hermes Senger




Adapted from:

**Chapters 3.2, 4.3, 6.1**, Output: Knowledge representation
Data Mining: Practical Machine Learning Tools and Techniques,
4th Edition. By Ian H. Witten, Eibe Frank, Mark A. Hall, Christopher J. Pal.
Morgan Kauffman, 2017.

# Agenda

- Basic
  - Trees as knowledge representation – Chapter 3.2
- Intermediate
  - Constructing decision trees – Chapter 4.3
- Advanced
  - Implementation – Chapter 6.1

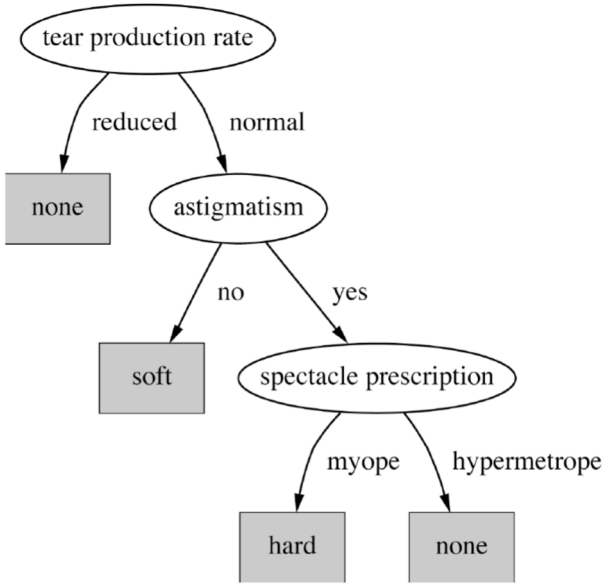
# Agenda

- Basic
  - Trees as knowledge representation – Chapter 3.2
- Intermediate
  - Constructing decision trees – Chapter 4.3
- Advanced
  - Implementation – Chapter 6.1

# Output: representing structural patterns

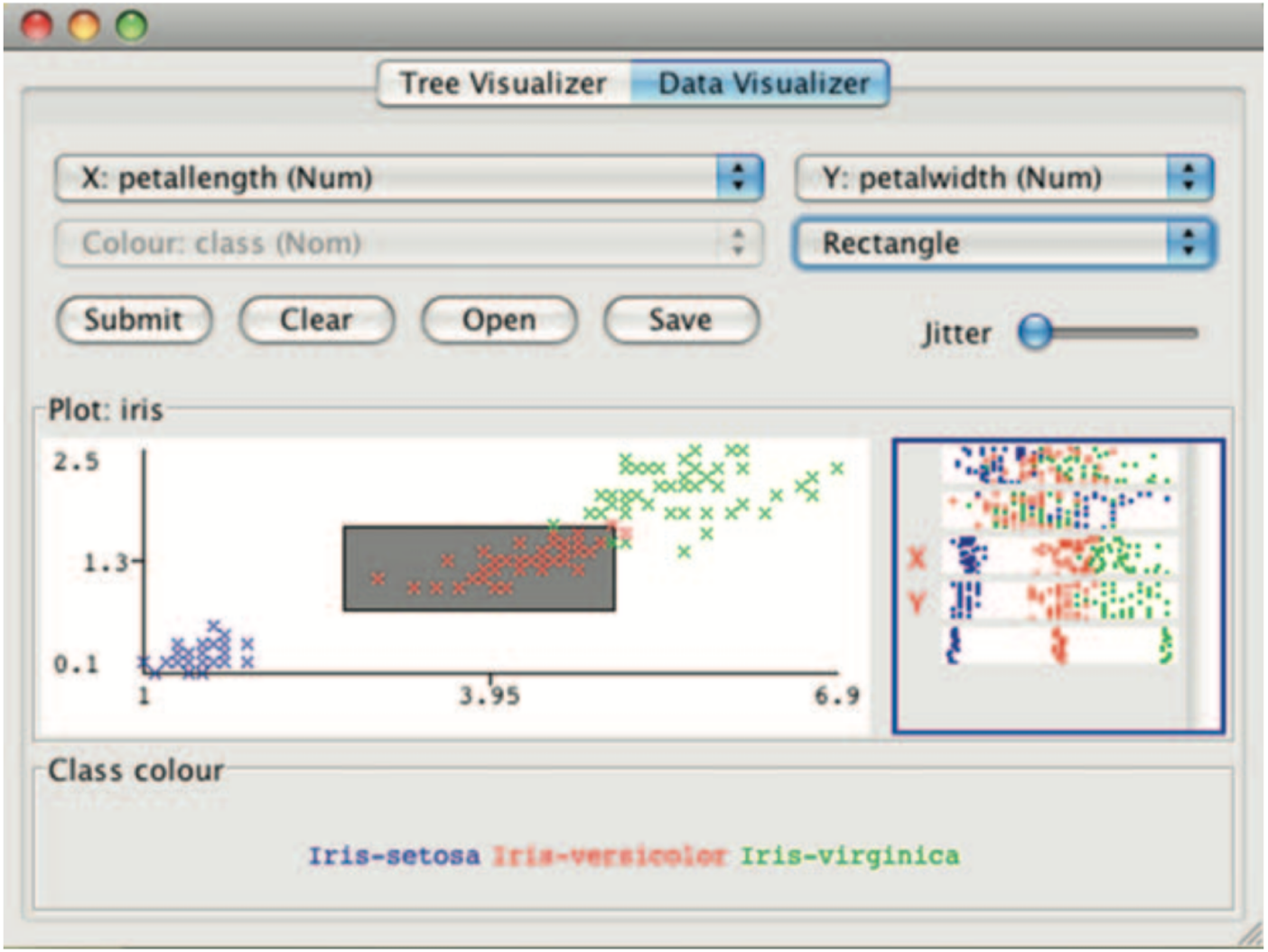
- Many different ways of representing patterns
  - Decision trees, rules, instance-based, …
- Also called “knowledge” representation
- Representation determines inference method
- Understanding the output is the key to understanding the underlying learning methods
- Different types of output for different learning problems (e.g., classification, regression, …)

# Decision trees

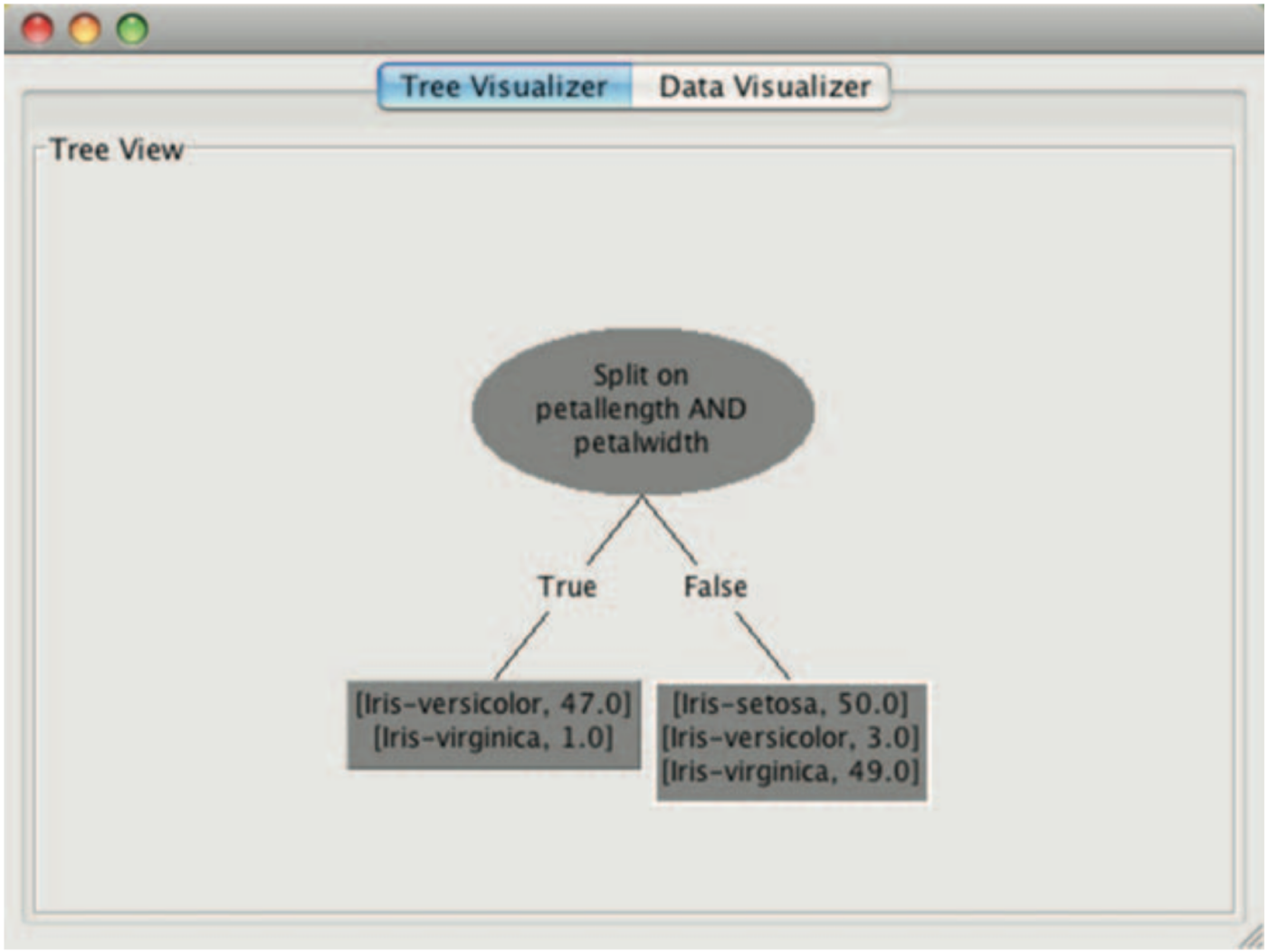



- "Divide-and-conquer" approach produces tree
- Nodes involve testing a particular attribute
- Usually, attribute value is compared to const...
- Other possibilities:
  - Comparing values of two attributes
  - Using a function of one or more attributes
- Leaves assign classification, set of classifications, or probability distribution to instances
- Unknown instance is routed down the tree

# Interactive tree construction I



# Interactive tree construction II



# Procedure

1. Execute Weka
2. Click in the “Tools” menu in the upper bar
3. Search and install userClassifier
4. Enter Explorer -> classify -> tree -> userClassifier
5. Select atribute pairs that allows discriminating one class
6. Select a rectangle/polygon that separates the class
7. Repeat steps 5-6 to cover all instances

# Nominal and numeric attributes in trees

- Nominal:
  number of children usually equal to number values
  $\Rightarrow$ attribute won't get tested more than once
- Other possibility: division into two subsets
- Numeric:
  test whether value is greater or less than constant
  $\Rightarrow$ attribute may get tested several times
  - Other possibility: three-way split (or multi-way split)
    - Integer: *less than, equal to, greater than*
    - Real: *below, within, above*

# Missing values

- Does absence of value have some significance?
- Yes $\Rightarrow$ "missing" is a separate value
- No $\Rightarrow$ "missing" must be treated in a special way
  - Solution A: assign instance to most popular branch
  - Solution B: split instance into pieces
    - Pieces receive weight according to fraction of training instances that go down each branch
    - Classifications from leave nodes are combined using the weights that have percolated to them

# Trees for numeric prediction

- ***Regression***: the process of computing an expression that predicts a numeric quantity
- ***Regression tree***: "decision tree" where each leaf predicts a numeric quantity
  - Predicted value is average value of training instances that reach the leaf
- ***Model tree:*** "regression tree" with linear regression models at the leaf nodes
  - Linear patches approximate continuous function

# Linear regression for the CPU data

PRP =

– 56.1

+ 0.049 MYCT

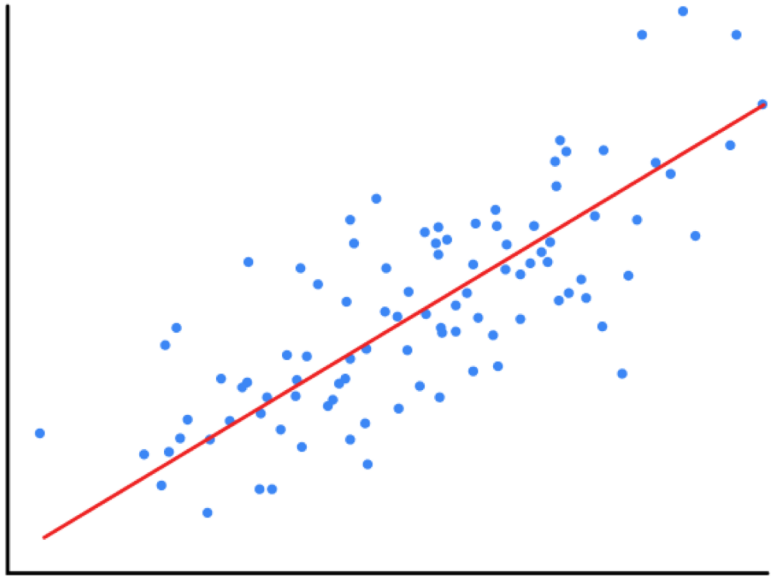
+ 0.015 MMIN

+ 0.006 MMAX

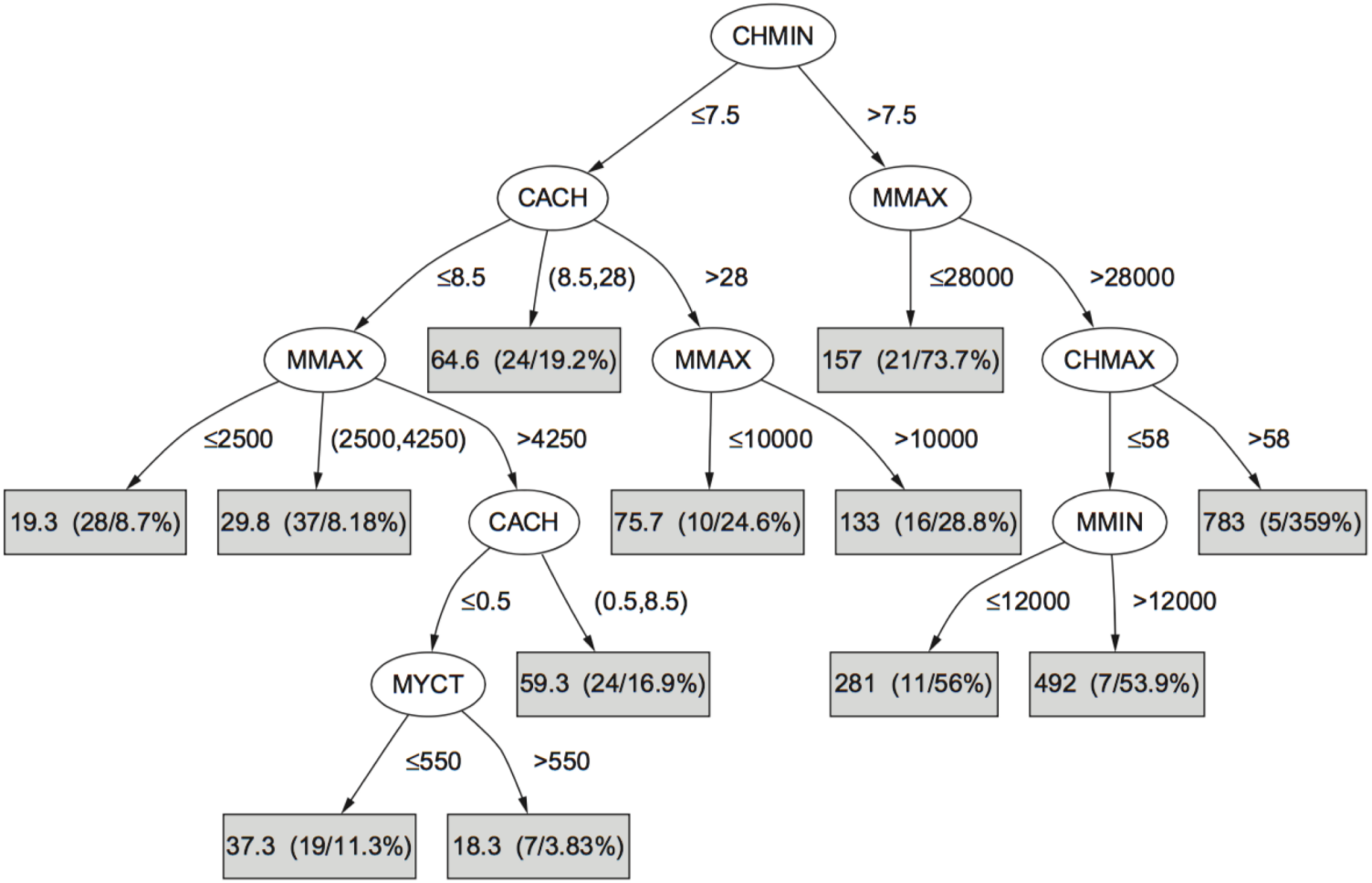
+ 0.630 CACH

– 0.270 CHMIN

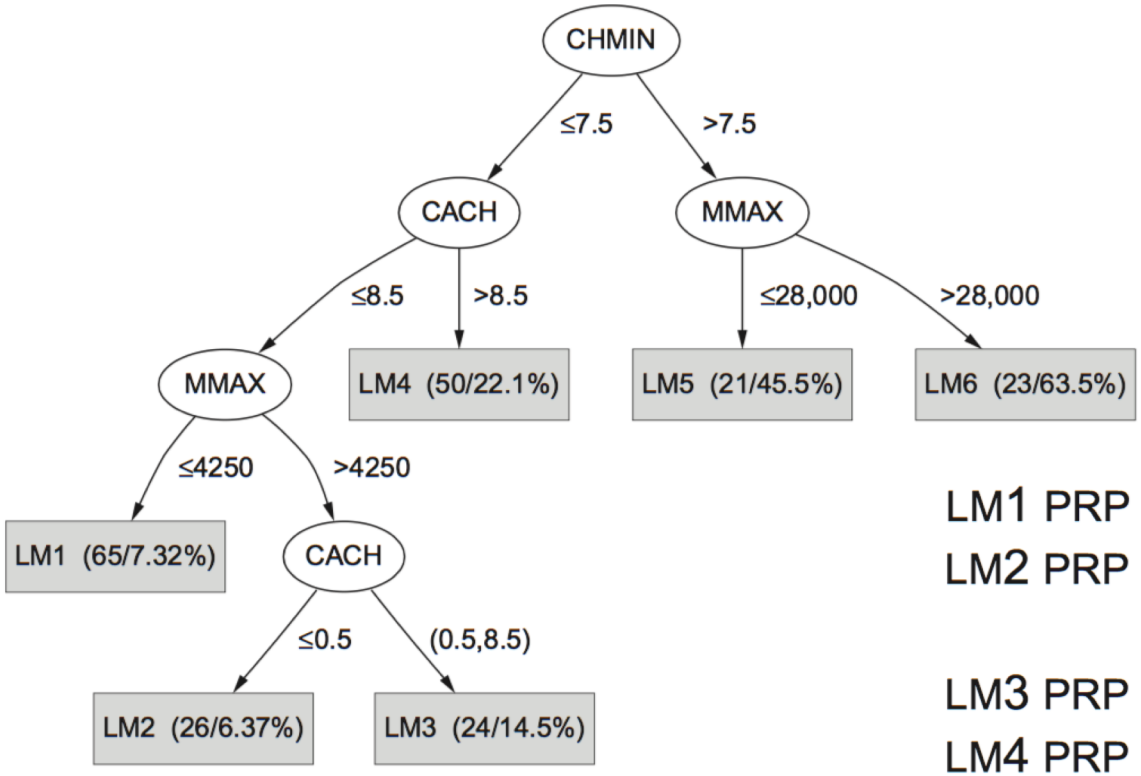
+ 1.46 CHMAX



# Regression tree for the CPU data



# Model tree for the CPU data




LM1 PRP = 8.29 + 0.004 MMAX + 2.77 CHMIN

LM2 PRP = 20.3 + 0.004 MMIN – 3.99 CHMIN
+ 0.946 CHMAX

LM3 PRP = 38.1 + 0.012 MMIN

LM4 PRP = 19.5 + 0.002 MMAX + 0.698 CACH
+ 0.969 CHMAX

LM5 PRP = 285 1.46 MYCT + 1.02 CACH
– 9.39 CHMIN

LM6 PRP = – 65.8 + 0.03 MMIN – 2.94 CHMIN
+ 4.98 CHMAX

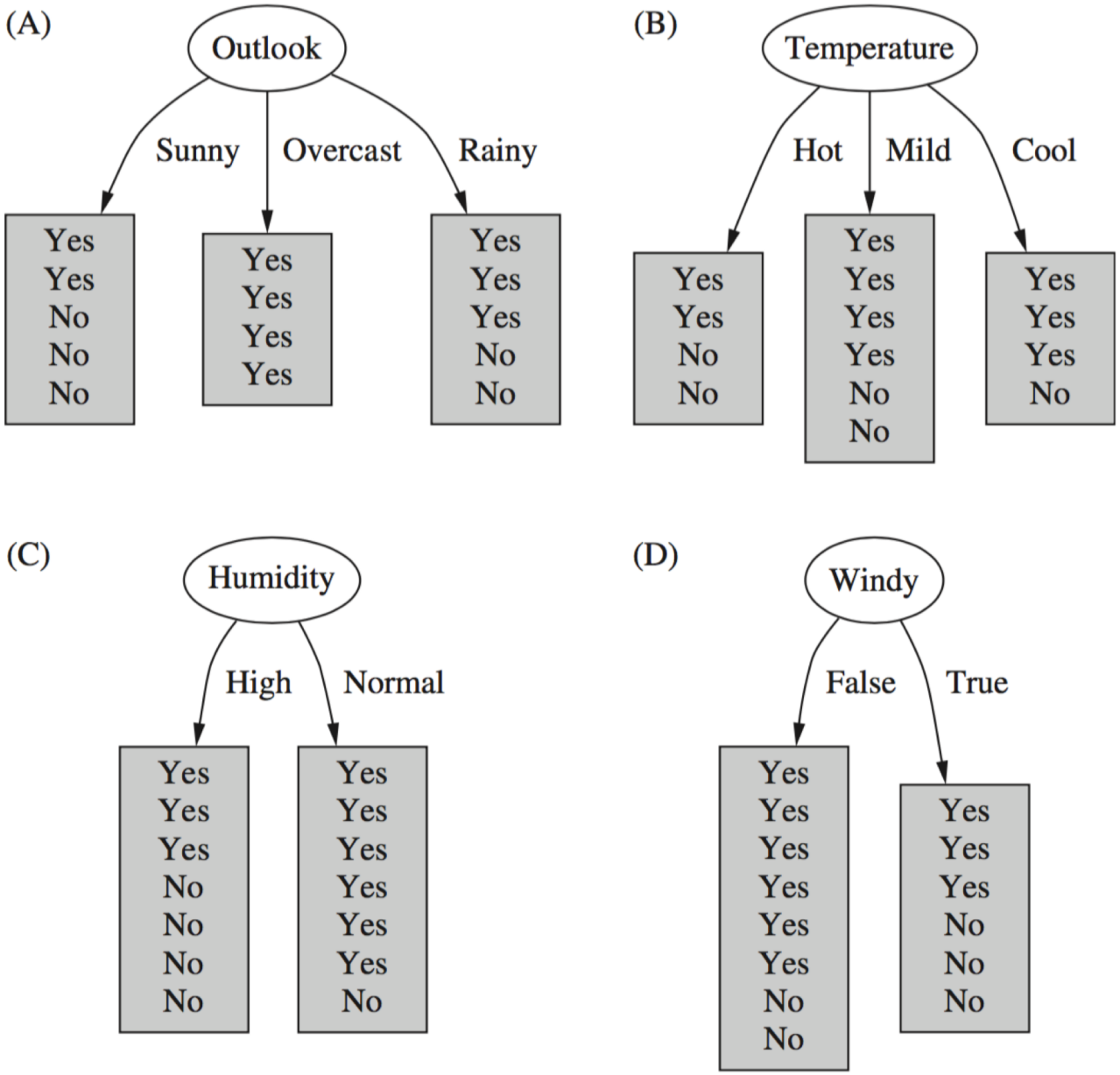
# Agenda

- Basic
  - Trees as knowledge representation – Chapter 3.2
- Intermediate
  - Constructing decision trees – Chapter 4.3
- Advanced
  - Implementation – Chapter 6.1

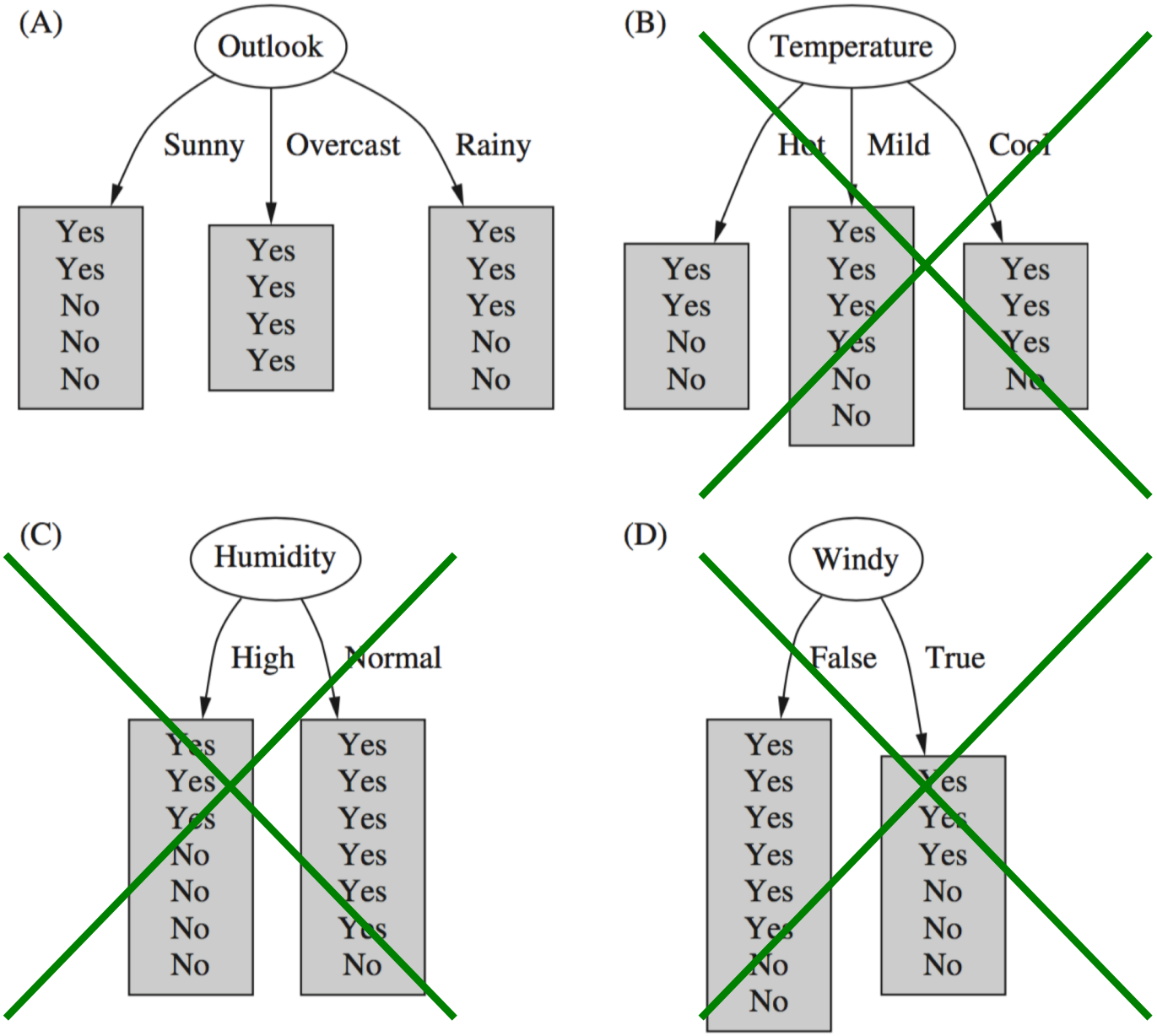
# Constructing decision trees

- Strategy: top down learning using recursive *divide-and-conquer* process
  - First: select attribute for root node
    Create branch for each possible attribute value
  - Then: split instances into subsets
    One for each branch extending from the node
  - Finally: repeat recursively for each branch, using only instances that reach the branch
- Stop if all instances have the same class

# Which attribute to select?



# Which attribute to select?



# Criterion for attribute selection

- Which is the best attribute?
  - Want to get the smallest tree
  - Heuristic: choose the attribute that produces the “purest” nodes
- Popular selection criterion: *information gain*
  - Information gain increases with the average purity of the subsets
- Strategy: amongst attributes available for splitting, **choose attribute that gives greatest information gain**
- Information gain requires measure of ***impurity***
- Impurity measure that it uses is the *entropy* of the class distribution, which is a measure from information theory

# Computing information

- We have a probability distribution: the class distribution in a subset of instances
- The expected information required to determine an outcome (i.e., class value), is the distribution's *entropy*
- Formula for computing the entropy:

$$\text{Entropy}(p_1, p_2, \ldots, p_n) = -p_1 \log p_1 - p_2 \log p_2 \ldots - p_n \log p_n$$

- Using base-2 logarithms, entropy gives the information required in expected *bits*
- Entropy is maximal when all classes are equally likely and minimal when one of the classes has probability 1

# Example: attribute *Outlook*

- *Outlook* = *Sunny* :

$$\text{Info}([2, 3]) = 0.971 \text{ bits}$$

- *Outlook* = *Overcast* :

$$\text{Info}([4, 0]) = 0.0 \text{ bits}$$

- *Outlook* = *Rainy* :

$$\text{Info}([3, 2]) = 0.971 \text{ bits}$$

- Expected information for attribute:

$$\text{Info}([2, 3], [4, 0], [3, 2]) = (5/14) \times 0.971 + (4/14) \times 0 + (5/14) \times 0.971$$

$$= 0.693 \text{ bits}$$

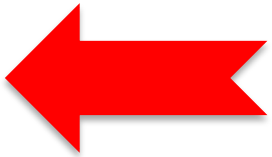
# Computing information gain

- Information gain: information before splitting – information after splitting

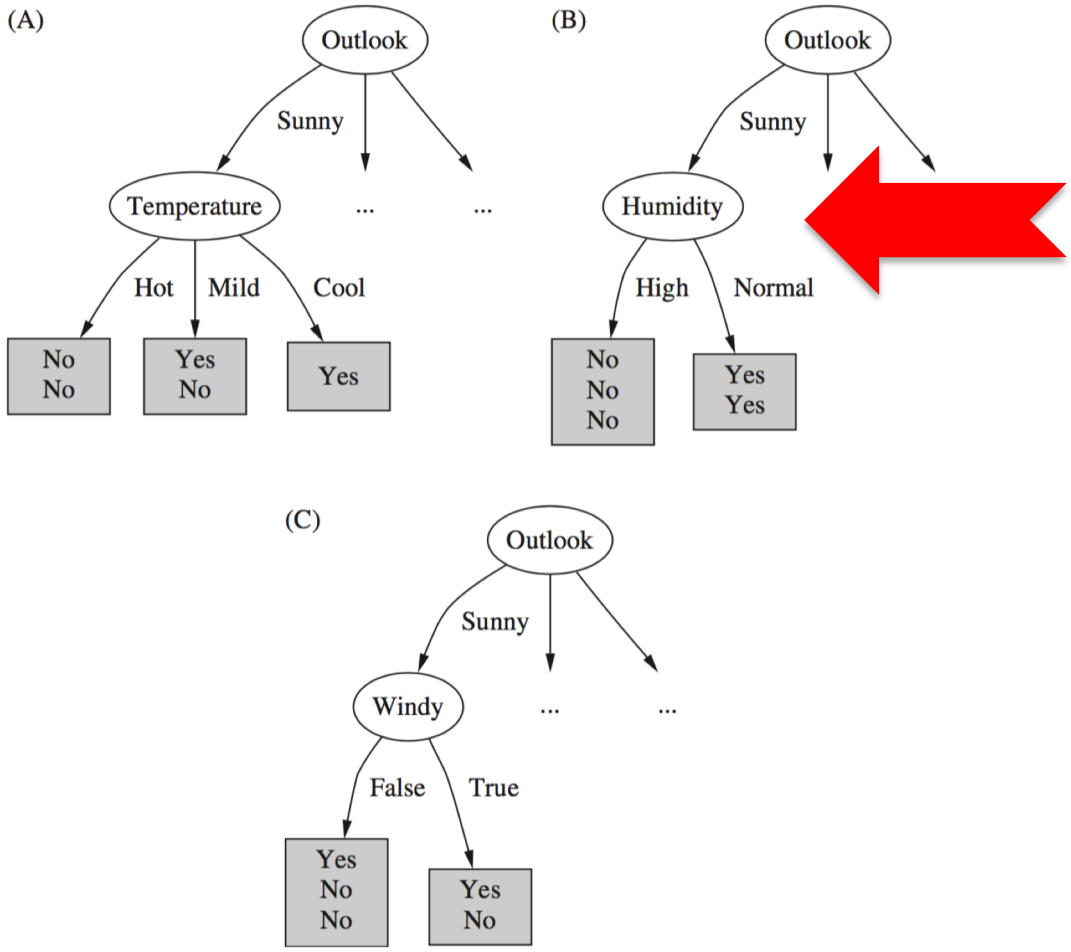
  Gain(*Outlook* ) = Info([9,5]) – info([2,3],[4,0],[3,2])
  = 0.940 – 0.693
  = 0.247 bits

- Information gain for attributes from weather data:

  Gain(*Outlook* ) = 0.247 bits
  Gain(*Temperature* ) = 0.029 bits
  Gain(*Humidity* ) = 0.152 bits
  Gain(*Windy* ) = 0.048 bits



# Continuing to split

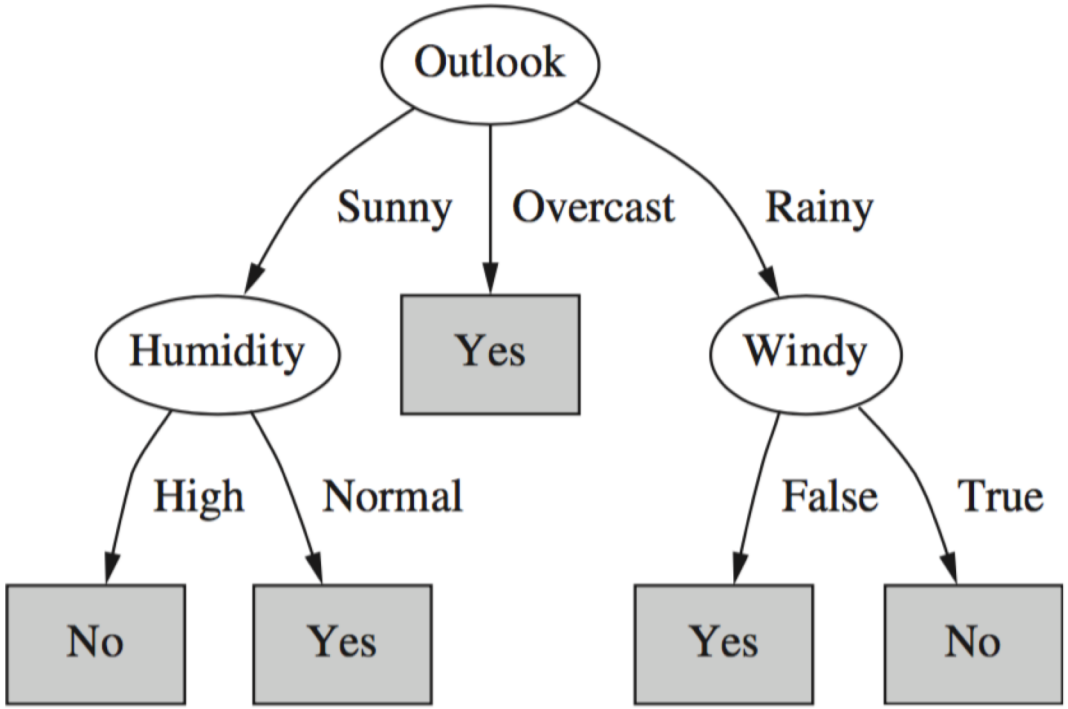



Gain(*Temperature* ) = 0.571 bits

Gain(*Humidity* ) = 0.971 bits

Gain(*Windy* ) = 0.020 bits

# Final decision tree




- Note: not all leaves need to be pure; sometimes identical instances have different classes
  - Splitting stops when data cannot be split any further

# Wishlist for an impurity measure

- Properties we would like to see in an impurity measure:
  - When node is **pure**, measure should be **zero**
  - When impurity is maximal (i.e., all classes equally likely), measure should be maximal
  - Measure should ideally obey ***multistage property*** (i.e., decisions can be made in several stages):

$$\text{Entropy}(p,q,r) = \text{entropy}(p,q+r) + (q+r)\cdot\text{entropy}\left(\frac{q}{q+r},\frac{r}{q+r}\right)$$

- It can be shown that entropy is the only function that satisfies all three properties!
- Note that the multistage property is intellectually pleasing but not strictly necessary in practice

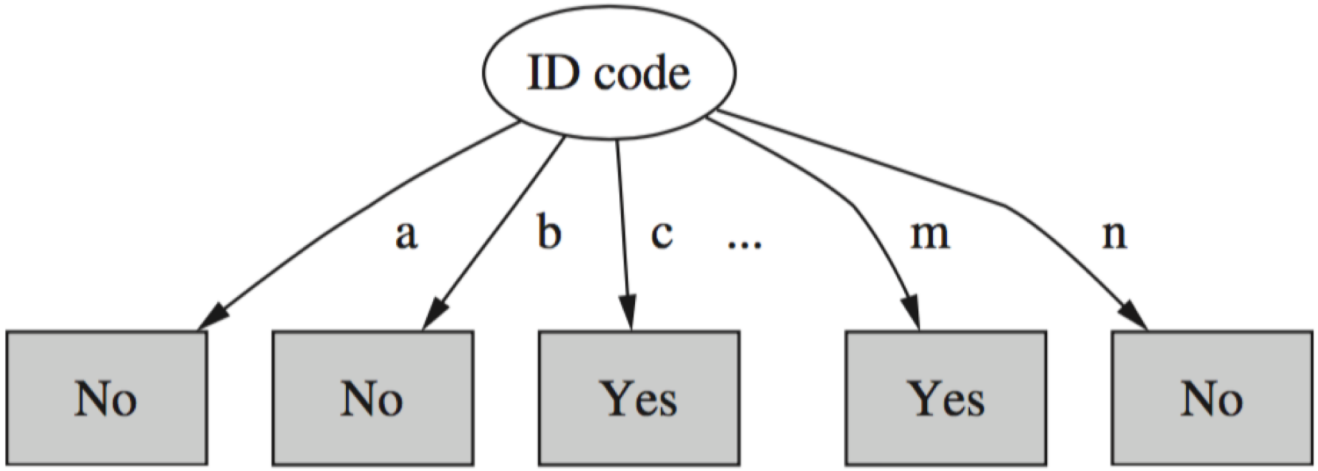
# Highly-branching attributes

- Problematic: attributes with a large number of values (extreme case: ID code)
- Subsets are more likely to be pure if there is a large number of values
  - Information gain is biased towards choosing attributes with a large number of values
  - This may result in ***overfitting*** (selection of an attribute that is non-optimal for prediction)
- An additional problem in decision trees is data *fragmentation*

# Weather data with *ID code*

| ID code | Outlook | Temp. | Humidity | Windy | Play |
|---|---|---|---|---|---|
| A | Sunny | Hot | High | False | No |
| B | Sunny | Hot | High | True | No |
| C | Overcast | Hot | High | False | Yes |
| D | Rainy | Mild | High | False | Yes |
| E | Rainy | Cool | Normal | False | Yes |
| F | Rainy | Cool | Normal | True | No |
| G | Overcast | Cool | Normal | True | Yes |
| H | Sunny | Mild | High | False | No |
| I | Sunny | Cool | Normal | False | Yes |
| J | Rainy | Mild | Normal | False | Yes |
| K | Sunny | Mild | Normal | True | Yes |
| L | Overcast | Mild | High | True | Yes |
| M | Overcast | Hot | Normal | False | Yes |
| N | Rainy | Mild | High | True | No |

# Tree stump for *ID code* attribute




- All (single-instance) subsets have entropy zero!
- This means the information gain is maximal for this ID code attribute (namely 0.940 bits)

# Gain ratio

- ***Gain ratio*** is a modification of the information gain that reduces its bias towards attributes with many values
- Gain ratio takes number and size of branches into account when choosing an attribute
  - It corrects the information gain by taking the *intrinsic information* of a split into account
- Intrinsic information: entropy of the distribution of instances into branches
- Measures how much info do we need to tell which branch a randomly chosen instance belongs to

# Computing the gain ratio

- Example: intrinsic information of ID code

$$\frac{1}{14}(\text{info}([0,1]) + \text{info}([0,1]) + \text{info}([1,0]) + \cdots + \text{info}([1,0]) + \text{info}([0,1]))$$

- Value of attribute should decrease as intrinsic information gets larger
- The *gain ratio* is defined as the information gain of the attribute divided by its intrinsic information
- Example (*outlook* at root node):

| | |
|---|---|
| Gain: 0.940−0.693 | 0.247 |
| Split info: info([5,4,5]) | 1.577 |
| Gain ratio: 0.247/1.577 | 0.156 |

# All gain ratios for the weather data

| Outlook | | Temperature | |
|---|---|---|---|
| Info: | 0.693 | Info: | 0.911 |
| Gain: 0.940-0.693 | 0.247 | Gain: 0.940-0.911 | 0.029 |
| Split info: info([5,4,5]) | 1.577 | Split info: info([4,6,4]) | 1.557 |
| Gain ratio: 0.247/1.577 | 0.157 | Gain ratio: 0.029/1.557 | 0.019 |
| Humidity | | Windy | |
| Info: | 0.788 | Info: | 0.892 |
| Gain: 0.940-0.788 | 0.152 | Gain: 0.940-0.892 | 0.048 |
| Split info: info([7,7]) | 1.000 | Split info: info([8,6]) | 0.985 |
| Gain ratio: 0.152/1 | 0.152 | Gain ratio: 0.048/0.985 | 0.049 |

# More on the gain ratio

- "Outlook" still comes out top
- However: "ID code" has greater gain ratio
  - Standard fix: *ad hoc* test to prevent splitting on that type of identifier attribute
- Problem with gain ratio: it may overcompensate
  - May choose an attribute just because its intrinsic information is very low
  - Standard fix: only consider attributes with greater than average information gain
- Both tricks are implemented in the well-known C4.5 decision tree learner

# Discussion

- Top-down induction of decision trees: ID3, algorithm developed by Ross Quinlan
  - Gain ratio just one modification of this basic algorithm
  - C4.5 tree learner deals with numeric attributes, missing values, noisy data
- Similar approach: CART tree learner
  - Uses Gini index rather than entropy to measure impurity
- There are many other attribute selection criteria! (But little difference in accuracy of result)

# Agenda

- Basic
  - Trees as knowledge representation – Chapter 3.2
- Intermediate
  - Constructing decision trees – Chapter 4.3
- Advanced (not covered during classes)
  - Implementation – Chapter 6.1

# Algorithms for learning trees and rules

- Decision trees
  - From ID3 to C4.5 (pruning, numeric attributes, ...)

# Industrial-strength algorithms

- For an algorithm to be useful in a wide range of real-world applications it must:
  - Permit numeric attributes
  - Allow missing values
  - Be robust in the presence of noise
- Basic scheme needs to be extended to fulfill these requirements

# From ID3 to C4.5

- Extending ID3:
  - to permit numeric attributes: *straightforward*
  - to deal sensibly with missing values: *trickier*
  - stability for noisy data: *requires pruning mechanism*
- End result: C4.5 (Quinlan)
  - Best-known and (probably) most widely-used learning algorithm
  - Commercial successor: C5.0

# Numeric attributes

- Standard method: binary splits
  - E.g. temp < 45
- Unlike nominal attributes,
  every attribute has many possible split points
- Solution is straightforward extension:
  - Evaluate info gain (or other measure)
    for every possible split point of attribute
  - Choose “best” split point
  - Info gain for best split point is info gain for attribute
- Computationally more demanding

# Weather data (again!)

| Outlook | Temperature | Humidity | Windy | Play |
|---|---|---|---|---|
| Sunny | Hot | High | False | No |
| Sunny | Hot | High | True | No |
| Overcast | Hot | High | False | Yes |
| Rainy | Mild | High | False | Yes |
| Rainy | Cool | Normal | False | Yes |
| Rainy | Cool | Normal | True | No |
| … | … | … | … | … |

```
If outlook = sunny and humidity = high then play = no
If outlook = rainy and windy = true then play = no
If outlook = overcast then play = yes
If humidity = normal then play = yes
If none of the above then play = yes
```

# Weather data (again!)

| Outlook | Temperature | Humidity | Windy | Play |
|---|---|---|---|---|
| Sunny | Hot | High | False | No |
| Sunny | | | | |
| Overcast | | | | |
| Rainy | | | | |
| Rainy | | | | |
| ... | | | | |

| Outlook | Temperature | Humidity | Windy | Play |
|---|---|---|---|---|
| Sunny | 85 | 85 | False | No |
| Sunny | 80 | 90 | True | No |
| Overcast | 83 | 86 | False | Yes |
| Rainy | 70 | 96 | False | Yes |
| Rainy | 68 | 80 | False | Yes |
| Rainy | 65 | 70 | True | No |
| ... | ... | ... | ... | ... |

```
If outlook = sunny and humidity = high then play = no
If outlook = rainy and windy = true then play = no
If outlook = overcast then play = yes
If humidity = normal then play = yes
If none of the
```

```
If outlook = sunny and humidity > 83 then play = no
If outlook = rainy and windy = true then play = no
If outlook = overcast then play = yes
If humidity < 85 then play = no
If none of the above then play = yes
```

# Example

- Split on temperature attribute:

| 64 | 65 | 68 | 69 | 70 | 71 | 72 | 72 | 75 | 75 | 80 | 81 | 83 | 85 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Yes | No | Yes | Yes | Yes | No | No | Yes | Yes | Yes | No | Yes | Yes | No |

  - E.g., temperature $< 71.5$: yes/4, no/2
    temperature $\geq 71.5$: yes/5, no/3

  - Info([4,2],[5,3])
    = 6/14 info([4,2]) + 8/14 info([5,3])
    = 0.939 bits

- Place split points halfway between values
- Can evaluate all split points in one pass!

# Can avoid repeated sorting

- Sort instances by the values of the numeric attribute
  - Time complexity for sorting: $O(n \log n)$
- Does this have to be repeated at each node of the tree?
- No! Sort order for children can be derived from sort order for parent
  - Time complexity of derivation: $O(n)$
  - Drawback: need to create and store an array of sorted indices for each numeric attribute

# Binary *vs* multiway splits

- Splitting (multi-way) on a nominal attribute exhausts all information in that attribute
  - Nominal attribute is tested (at most) once on any path in the tree
- Not so for binary splits on numeric attributes!
  - Numeric attribute may be tested several times along a path in the tree
- Disadvantage: tree is hard to read
- Remedy:
  - pre-discretize numeric attributes, *or*
  - use multi-way splits instead of binary ones

# Computing multi-way splits

- Simple and efficient way of generating multi-way splits: greedy algorithm
- But: dynamic programming can find optimum multi-way split in $O(n^2)$ time
  - imp ($k$, $i$, $j$ ) is the impurity of the best split of values $x_i$ … $x_j$ into $k$ sub-intervals
  - imp ($k$, 1, $i$ ) =
    $\min_{0<j<i}$ imp ($k$–1, 1, $j$ ) + imp (1, $j$+1, $i$ )
  - imp ($k$, 1, $N$ ) gives us the best $k$-way split
- In practice, greedy algorithm works as well

# Missing values

- C4.5 applies method of fractional instances:
  - Split instances with missing values into pieces
  - A piece going down a branch receives a weight proportional to the popularity of the branch
  - weights sum to 1
- Info gain works with fractional instances
  - use sums of weights instead of counts
- During classification, split the instance into pieces in the same way
  - Merge probability distribution using weights

# Pruning

- Prevent overfitting the training data: “prune the decision tree
- Two strategies:
  - *Postpruning*
    take a fully-grown decision tree and discard unreliable parts
  - *Prepruning*
    stop growing a branch when information becomes unreliable
- Postpruning is preferred in practice—prepruning can “stop early”

# Prepruning

- Based on statistical significance test
  - Stop growing the tree when there is no *statistically significant* association between any attribute and the class at a particular node
- Most popular test: *chi-squared test*
- Quinlan's classic tree learner ID3 used chi-squared test in addition to information gain
  - Only statistically significant attributes were allowed to be selected by the information gain procedure

# Early stopping

| | a | b | class |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 1 |
| 3 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 |

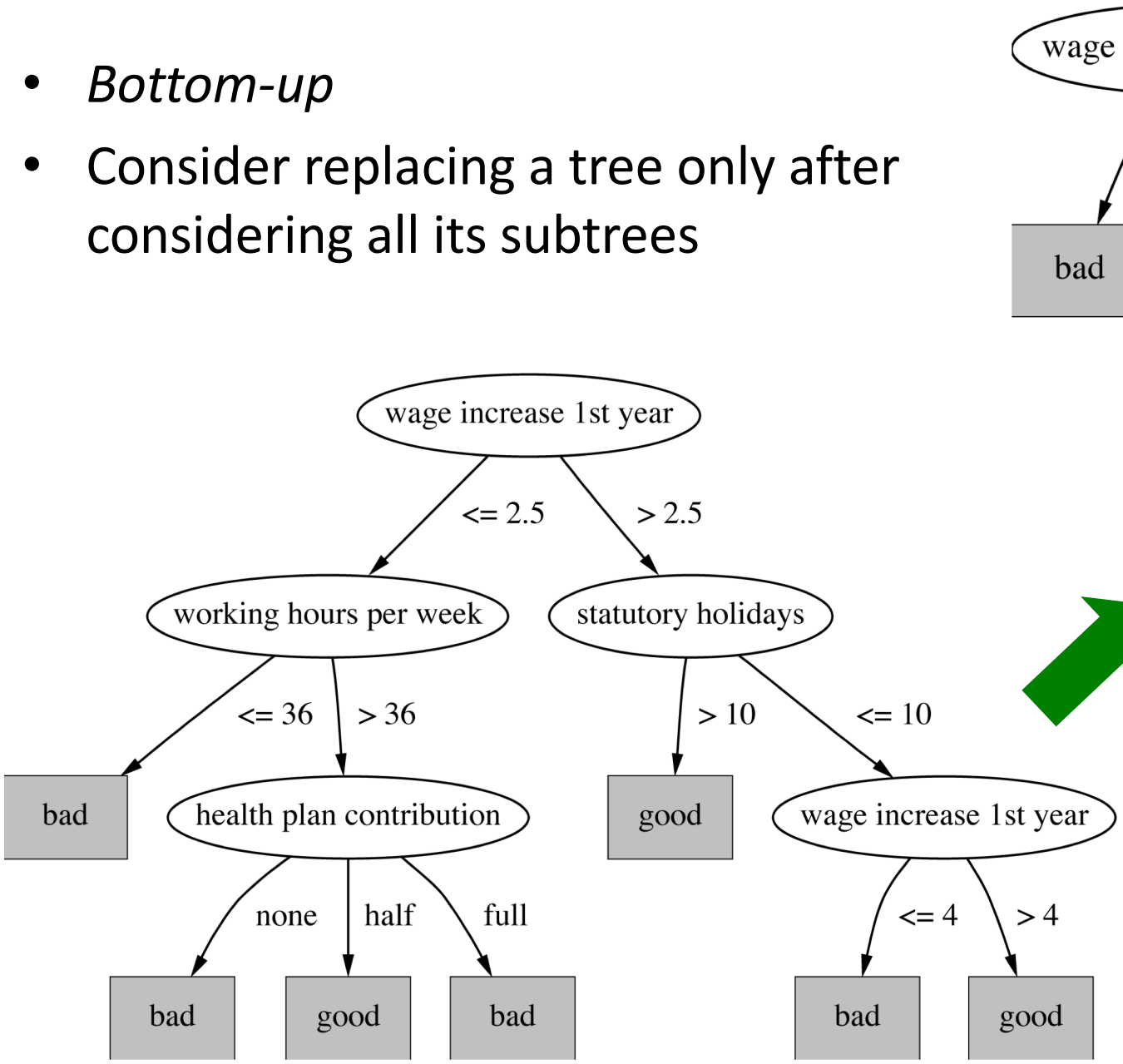
- Pre-pruning may stop the growth process prematurely: *early stopping*
- Classic example: XOR/Parity-problem
  - No *individual* attribute exhibits any significant association with the class
  - Structure is only visible in fully expanded tree
  - Prepruning won't expand the root node
- But: XOR-type problems rare in practice
- And: prepruning faster than postpruning

# Postpruning

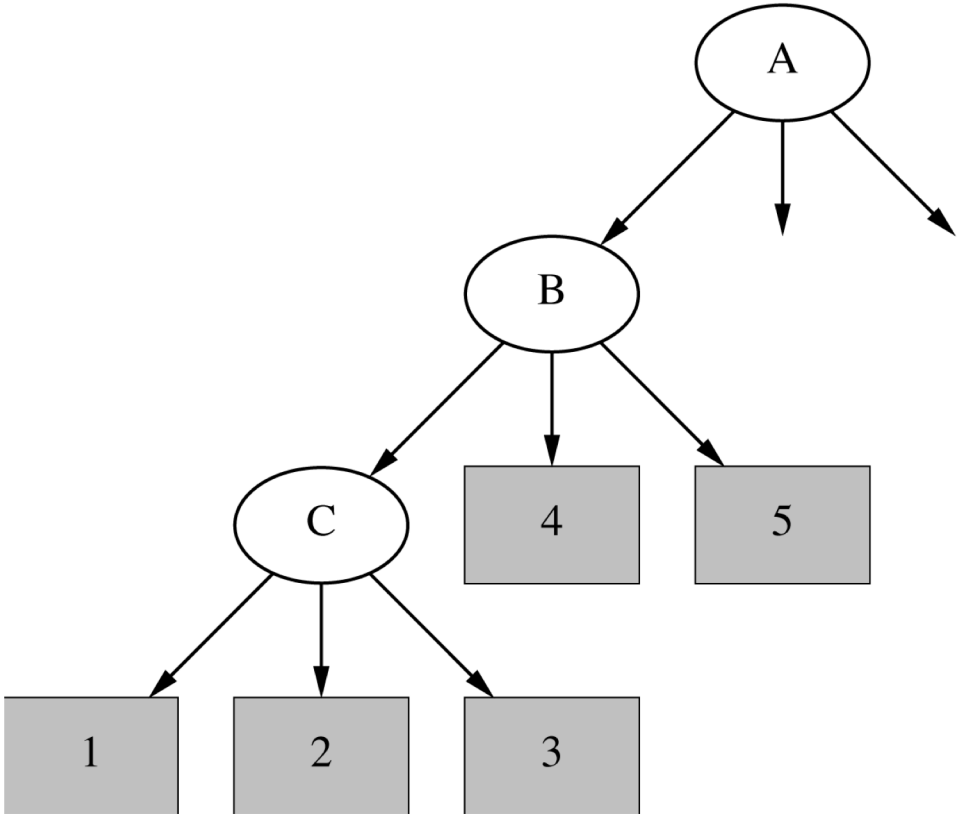
- First, build full tree
- Then, prune it
  - Fully-grown tree shows all attribute interactions
- Problem: some subtrees might be due to chance effects
- Two pruning operations:
  - Subtree replacement
  - Subtree raising
- Possible strategies:
  - error estimation
  - significance testing
  - MDL principle

# Subtree replacement

- *Bottom-up*
- Consider replacing a tree only after considering all its subtrees

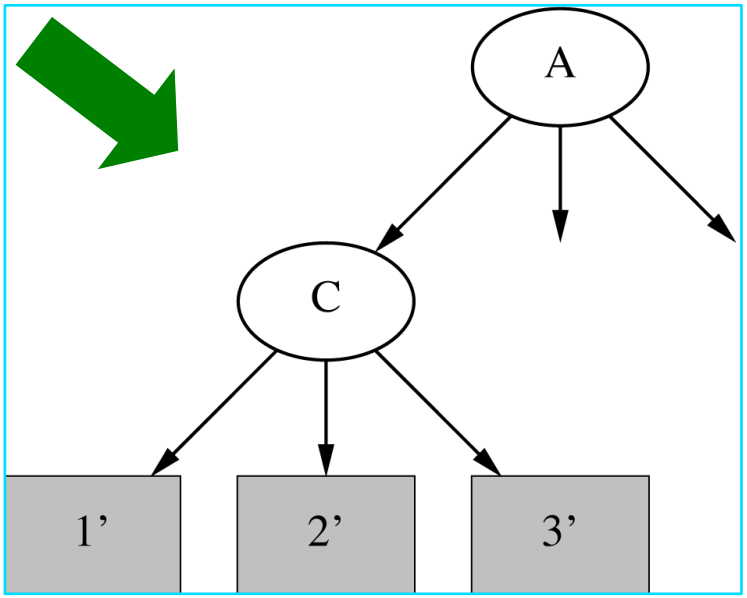


# Subtree raising




- Delete node
- Redistribute instances
- Slower than subtree replacement

  *(Worthwhile?)*



# Estimating error rates

- Prune only if it does not increase the estimated error
- Error on the training data is NOT a useful estimator *(would result in almost no pruning)*
- One possibility: use hold-out set for pruning (yields “reduced-error pruning”)
- C4.5’s method
  - Derive confidence interval from training data
  - Use a heuristic limit, derived from this, for pruning
  - Standard Bernoulli-process-based method
  - Shaky statistical assumptions (based on training data)

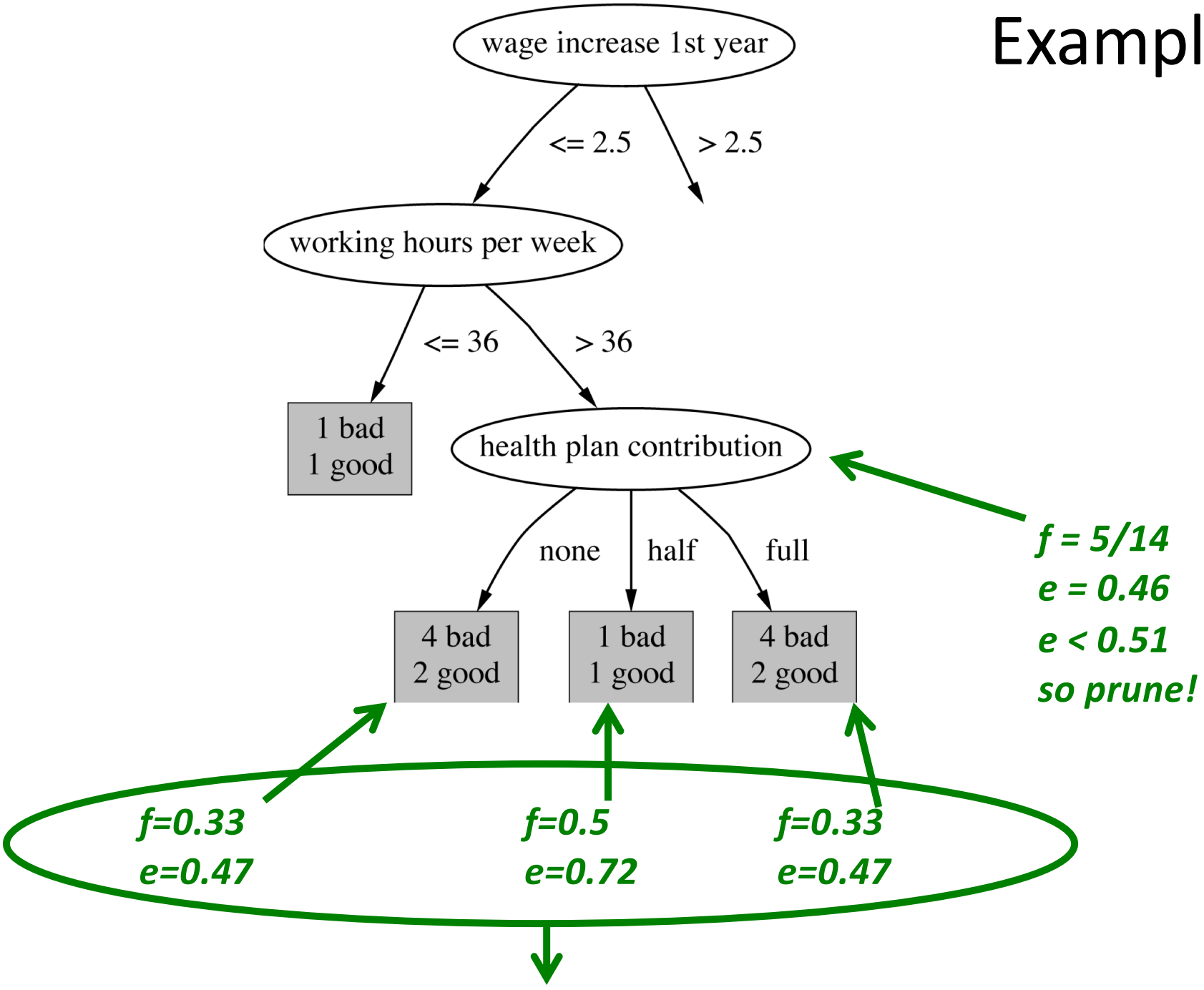
# C4.5's method

- Error estimate for subtree is weighted sum of error estimates for all its leaves
- Error estimate for a node:

$$e = \left( f + \frac{z^2}{2N} + z\sqrt{\frac{f}{N} + \frac{f^2}{N} + \frac{z^2}{4N^2}} \right) \Bigg/ \left( 1 + \frac{z^2}{N} \right)$$

- If $c$ = 25% then $z$ = 0.69 (from normal distribution)
- $f$ is the error on the training data
- $N$ is the number of instances covered by the leaf

# Example


wage increase 1st year
<= 2.5
> 2.5
working hours per week
<= 36
> 36
1 bad
1 good
health plan contribution
none
half
full
4 bad
2 good
1 bad
1 good
4 bad
2 good
f = 5/14
e = 0.46
e < 0.51
so prune!
f=0.33
e=0.47
f=0.5
e=0.72
f=0.33
e=0.47
Combined using ratios 6:2:6 gives 0.51

# Complexity of tree induction

- Assume
  - $m$ attributes
  - $n$ training instances
  - tree depth $O$ (log $n$)
- Building a tree $O(m\, n \log n)$
- Subtree replacement $O(n)$
- Subtree raising $O(n (\log n)^2)$
  - Every instance may have to be redistributed at every node between its leaf and the root
  - Cost for redistribution (on average): $O$ (log $n$)
- Total cost: $O(m\, n \log n) + O(n (\log n)^2)$

# From trees to rules

- Simple way: one rule for each leaf
- C4.5rules: greedily prune conditions from each rule if this reduces its estimated error
  - Can produce duplicate rules
  - Check for this at the end
- Then
  - look at each class in turn
  - consider the rules for that class
  - find a “good” subset (guided by MDL)
- Then rank the subsets to avoid conflicts
- Finally, remove rules (greedily) if this decreases error on the training data

# C4.5: choices and options

- C4.5rules slow for large and noisy datasets
- Successor algorithm C5.0rules uses a different technique
  - Much faster and a bit more accurate
- C4.5 has two parameters
  - Confidence value (default 25%):
    lower values incur heavier pruning
  - Minimum number of instances in the two most popular branches (default 2)
- Time complexity of C4.5 is actually greater than what was stated above:
  - For each numeric split point that has been identified, the *entire* training set is scanned to find the closest actual point

# Cost-complexity pruning

- C4.5's postpruning often does not prune enough
  - Tree size continues to grow when more instances are added even if performance on independent data does not improve
  - But: it is very fast and popular in practice
- Can be worthwhile in some cases to strive for a more compact tree at the expense of more computational effort
  - *Cost-complexity pruning* method from the CART (Classification and Regression Trees) learning system achieves this
  - Applies cross-validation or a hold-out set to choose an appropriate tree size for the final tree

# Cost-complexity pruning details

- Basic idea:
  - First prune subtrees that, relative to their size, lead to the smallest increase in error on the training data
  - Increase in error (α): average error increase per leaf of subtree
  - Bottom-up pruning based on this criterion generates a *sequence* of successively smaller trees
  - Each candidate tree in the sequence corresponds to one particular threshold value $\alpha_i$
- Which tree to chose as the final model?
  - Use either a hold-out set or cross-validation to estimate the error for each $\alpha_i$
  - Rebuild tree on entire training set using chosen value of $\alpha$

# Discussion

*TDIDT: Top-Down Induction of Decision Trees*

- The most extensively studied method of machine learning used in data mining
- Different criteria for attribute/test selection rarely make a large difference
- Different pruning methods mainly change the size of the resulting pruned tree
- C4.5 builds *univariate* decision trees: each node tests a single attribute
- Some TDITDT systems can build *multivariate* trees (e.g., the famous CART tree learner)

# Discussion and Bibliographic Notes

- CART's pruning method (Breiman et al. 1984) can often produce smaller trees than C4.5's method
- C4.5's overfitting problems have been investigated empirically by Oates and Jensen (1997)
- A complete description of C4.5, the early 1990s version, appears as a excellent and readable book (Quinlan 1993)
- An MDL-based heuristic for C4.5 Release 8 that combats overfitting of numeric attributes is described by Quinlan (1998)
- The more recent version of Quinlan's tree learner, C5.0, is also available as open-source code