Data Mining

Practical Machine Learning Tools and Techniques

Slides for Chapter 4 of *Data Mining* by I. H. Witten and E. Frank

Algorithms: The basic methods

- Inferring rudimentary rules
- Statistical modeling
- Constructing decision trees
- Constructing rules
- Association rule learning
- Linear models
- Instance-based learning
- Clustering

WEKA The University of Waikato

Simplicity first

- Simple algorithms often work very well!
- There are many kinds of simple structure, eg:
  - One attribute does all the work
  - All attributes contribute equally & independently
  - A weighted linear combination might do
  - Instance-based: use a few prototypes
  - Use simple logical rules
- Success of method depends on the domain

Inferring rudimentary rules

- 1R: learns a 1-level decision tree
  - I.e., rules that all test one particular attribute
- Basic version
  - One branch for each value
  - Each branch assigns most frequent class
  - Error rate: proportion of instances that don't belong to the majority class of their corresponding branch
  - Choose attribute with lowest error rate

(*assumes nominal attributes*)

Pseudo-code for 1R

```
For each attribute,
   For each value of the attribute, make a rule as follows:
      count how often each class appears
      find the most frequent class
      make the rule assign that class to this attribute-value
   Calculate the error rate of the rules
Choose the rules with the smallest error rate
```

- Note: "missing" is treated as a separate attribute value

Evaluating the weather attributes

Outlook	Temp	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
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

Attribute	Rules	Errors	Total errors
Outlook	Sunny → No	2/5	4/14
	Overcast → Yes	0/4	
	Rainy → Yes	2/5	
Temp	Hot → No*	2/4	5/14
	Mild → Yes	2/6	
	Cool → Yes	1/4	
Humidity	High → No	3/7	4/14
	Normal → Yes	1/7	
Windy	False → Yes	2/8	5/14
	True → No*	3/6	

* indicates a tie

Dealing with numeric attributes

- Discretize numeric attributes
- Divide each attribute's range into intervals
  - Sort instances according to attribute's values
  - Place breakpoints where class changes (majority class)
  - This minimizes the total error
- Example: *temperature* from weather data

```
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
```

Outlook	Temperature	Humidity	Windy	Play
Sunny	85	85	False	No
Sunny	80	90	True	No
Overcast	83	86	False	Yes
Rainy	75	80	False	Yes
…	…	…	…	…

The problem of overfitting

- This procedure is very sensitive to noise
  - One instance with an incorrect class label will probably produce a separate interval
- Also: *time stamp* attribute will have zero errors
- Simple solution:
  enforce minimum number of instances in majority class per interval
- Example (with min = 3):

```
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

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
```

With overfitting avoidance

- Resulting rule set:

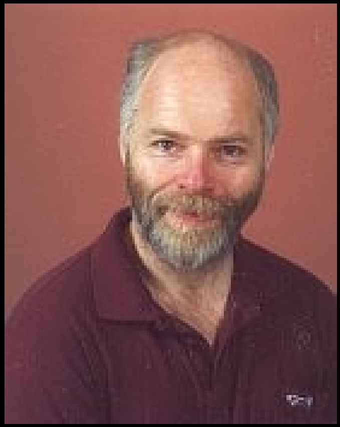
Attribute	Rules	Errors	Total errors
Outlook	Sunny → No	2/5	4/14
	Overcast → Yes	0/4	
	Rainy → Yes	2/5	
Temperature	≤ 77.5 → Yes	3/10	5/14
	> 77.5 → No*	2/4	
Humidity	≤ 82.5 → Yes	1/7	3/14
	> 82.5 and ≤ 95.5 → No	2/6	
	> 95.5 → Yes	0/1	
Windy	False → Yes	2/8	5/14
	True → No*	3/6	

Discussion of 1R

- 1R was described in a paper by Holte (1993)
  - Contains an experimental evaluation on 16 datasets (using *cross-validation* so that results were representative of performance on future data)
  - Minimum number of instances was set to 6 after some experimentation
  - 1R's simple rules performed not much worse than much more complex decision trees
- Simplicity first pays off!

Very Simple Classification Rules Perform Well on Most Commonly Used Datasets

Robert C. Holte, Computer Science Department, University of Ottawa



Discussion of 1R: Hyperpipes

- Another simple technique: build one rule for each class
  - Each rule is a conjunction of tests, one for each attribute
  - For numeric attributes: test checks whether instance's value is inside an interval
    - Interval given by minimum and maximum observed in training data
  - For nominal attributes: test checks whether value is one of a subset of attribute values
    - Subset given by all possible values observed in training data
  - Class with most matching tests is predicted

Statistical modeling

- "Opposite" of 1R: use all the attributes
- Two assumptions: Attributes are
  - *equally important*
  - *statistically independent* (given the class value)
    - I.e., knowing the value of one attribute says nothing about the value of another (if the class is known)
- Independence assumption is never correct!
- But … this scheme works well in practice

Probabilities for weather data

Outlook			Temperature			Humidity			Windy			Play	
	Yes	*No*		*Yes*	*No*		*Yes*	*No*		*Yes*	*No*	*Yes*	*No*
Sunny	2	3	Hot	2	2	High	3	4	False	6	2	9	5
Overcast	4	0	Mild	4	2	Normal	6	1	True	3	3		
Rainy	3	2	Cool	3	1								
Sunny	2/9	3/5	Hot	2/9	2/5	High	3/9	4/5	False	6/9	2/5	9/14	5/14
Overcast	4/9	0/5	Mild	4/9	2/5	Normal	6/9	1/5	True	3/9	3/5		
Rainy	3/9	2/5	Cool	3/9	1/5								

Outlook	Temp	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
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

Probabilities for weather data

Outlook			Temperature			Humidity			Windy			Play	
	Yes	*No*		*Yes*	*No*		*Yes*	*No*		*Yes*	*No*	*Yes*	*No*
Sunny	2	3	Hot	2	2	High	3	4	False	6	2	9	5
Overcast	4	0	Mild	4	2	Normal	6	1	True	3	3		
Rainy	3	2	Cool	3	1								
Sunny	2/9	3/5	Hot	2/9	2/5	High	3/9	4/5	False	6/9	2/5	9/14	5/14
Overcast	4/9	0/5	Mild	4/9	2/5	Normal	6/9	1/5	True	3/9	3/5		
Rainy	3/9	2/5	Cool	3/9	1/5								

- A new day:

Outlook	Temp.	Humidity	Windy	Play
Sunny	Cool	High	True	?

Likelihood of the two classes

For “yes” = $2/9 \times 3/9 \times 3/9 \times 3/9 \times 9/14 = 0.0053$

For “no” = $3/5 \times 1/5 \times 4/5 \times 3/5 \times 5/14 = 0.0206$

Conversion into a probability by normalization:

P(“yes”) = 0.0053 / (0.0053 + 0.0206) = 0.205

P(“no”) = 0.0206 / (0.0053 + 0.0206) = 0.795

Bayes's rule

- Probability of event *H* given evidence *E*:

$$Pr[H|E] = \frac{Pr[E|H]\,Pr[H]}{Pr[E]}$$

- *A priori* probability of *H* : $Pr[H]$
  - Probability of event *before* evidence is seen
- *A posteriori* probability of *H* : $Pr[H|E]$
  - Probability of event *after* evidence is seen

Thomas Bayes

Born: 1702 in London, England
Died: 1761 in Tunbridge Wells, Kent, England

Naïve Bayes for classification

- Classification learning: what's the probability of the class given an instance?
  - Evidence E = instance
  - Event H = class value for instance
- Naïve assumption: evidence splits into parts (i.e. attributes) that are *independent*

$$Pr[H|E] = \frac{Pr[E_1|H]\,Pr[E_2|H] \ldots Pr[E_n|H]\,Pr[H]}{Pr[E]}$$

Weather data example

Outlook	Temp.	Humidity	Windy	Play
Sunny	Cool	High	True	?

← ***Evidence E***

$$Pr[yes|E] = Pr[Outlook = Sunny|yes] \times Pr[Temperature = Cool|yes] \times Pr[Humidity = High|yes] \times Pr[Windy = True|yes] \times \frac{Pr[yes]}{Pr[E]}$$

Probability of class "yes"

$$= \frac{\frac{2}{9} \times \frac{3}{9} \times \frac{3}{9} \times \frac{3}{9} \times \frac{9}{14}}{Pr[E]}$$

The "zero-frequency problem"

- What if an attribute value doesn't occur with every class value?
(e.g. "Humidity = high" for class "yes")
  - Probability will be zero! $Pr[Humidity=High|yes]=0$
  - *A posteriori* probability will also be zero! $Pr[yes|E]=0$
(No matter how likely the other values are!)
- Remedy: add 1 to the count for every attribute value-class combination (*Laplace estimator)*
- Result: probabilities will never be zero!
(also: stabilizes probability estimates)

Modified probability estimates

- In some cases adding a constant different from 1 might be more appropriate
- Example: attribute *outlook* for class *yes*

$$\frac{2+\mu/3}{9+\mu} \qquad \frac{4+\mu/3}{9+\mu} \qquad \frac{3+\mu/3}{9+\mu}$$

Sunny ***Overcast*** ***Rainy***

- Weights don't need to be equal (but they must sum to 1)

$$\frac{2+\mu p_1}{9+\mu} \qquad \frac{4+\mu p_2}{9+\mu} \qquad \frac{3+\mu p_3}{9+\mu}$$

Missing values

- Training: instance is not included in frequency count for attribute value-class combination
- Classification: attribute will be omitted from calculation
- Example:

Outlook	Temp.	Humidity	Windy	Play
?	Cool	High	True	?

Likelihood of “yes” = $3/9 \times 3/9 \times 3/9 \times 9/14 = 0.0238$

Likelihood of “no” = $1/5 \times 4/5 \times 3/5 \times 5/14 = 0.0343$

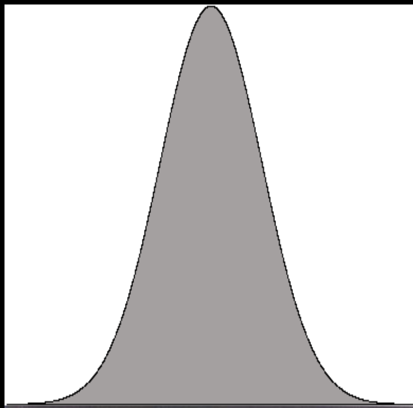
P(“yes”) = 0.0238 / (0.0238 + 0.0343) = 41%

P(“no”) = 0.0343 / (0.0238 + 0.0343) = 59%

Numeric attributes

- Usual assumption: attributes have a *normal* or *Gaussian* probability distribution (given the class)
- The *probability density function* for the normal distribution is defined by two parameters:
  - *Sample mean μ* $\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$
  - *Standard deviation σ* $\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu)^2}$
  - Then the density function *f(x) is*

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Statistics for weather data

Outlook			Temperature		Humidity		Windy			Play	
	Yes	*No*	*Yes*	*No*	*Yes*	*No*		*Yes*	*No*	*Yes*	*No*
Sunny	2	3	64, 68,	65,71,	65, 70,	70, 85,	False	6	2	9	5
Overcast	4	0	69, 70,	72,80,	70, 75,	90, 91,	True	3	3		
Rainy	3	2	72, …	85, …	80, …	95, …					
Sunny	2/9	3/5	μ =73	μ =75	μ =79	μ =86	False	6/9	2/5	9/14	5/14
Overcast	4/9	0/5	σ =6.2	σ =7.9	σ =10.2	σ =9.7	True	3/9	3/5		
Rainy	3/9	2/5									

- Example density value:

$$f(temperature=66|yes)=\frac{1}{\sqrt{2\pi}6.2}e^{-\frac{(66-73)^2}{2\cdot 6.2^2}}=0.0340$$

Classifying a new day

- A new day:

Outlook	Temp.	Humidity	Windy	Play
Sunny	66	90	true	?

Likelihood of "yes" = 2/9 $\times$ 0.0340 $\times$ 0.0221 $\times$ 3/9 $\times$ 9/14 = 0.000036

Likelihood of "no" = 3/5 $\times$ 0.0221 $\times$ 0.0381 $\times$ 3/5 $\times$ 5/14 = 0.000108

P("yes") = 0.000036 / (0.000036 + 0. 000108) = 25%

P("no") = 0.000108 / (0.000036 + 0. 000108) = 75%

- Missing values during training are not included in calculation of mean and standard deviation

Probability densities

- Relationship between probability and density:

$$Pr[c-\frac{\epsilon}{2}<x<c+\frac{\epsilon}{2}]\approx\epsilon\times f(c)$$

- But: this doesn't change calculation of *a posteriori* probabilities because ε cancels out
- Exact relationship:

$$Pr[a\leqslant x\leqslant b]=\int_a^b f(t)\,dt$$

Multinomial naïve Bayes I

- Version of naïve Bayes used for document classification using *bag of words* model
- $n_1, n_2, \ldots, n_k$: number of times word *i* occurs in document
- $P_1, P_2, \ldots, P_k$: probability of obtaining word *i* when sampling from documents in class *H*
- Probability of observing document *E* given class *H* (based on *multinomial distribution*):

$$Pr[E|H] \approx N! \times \prod_{i=1}^{k} \frac{P_i^{n_i}}{n_i!}$$

- Ignores probability of generating a document of the right length (prob. assumed constant for each class)

Multinomial naïve Bayes II

- Suppose dictionary has two words, *yellow* and *blue*
- Suppose Pr[*yellow* | *H*] = 75% and Pr[*blue* | *H*] = 25%
- Suppose *E* is the document "*blue yellow blue*"
- Probability of observing document:

$$Pr[\{\text{blue yellow blue}\}|H] \approx 3! \times \frac{0.75^1}{1!} \times \frac{0.25^2}{2!} = \frac{9}{64} \approx 0.14$$

Suppose there is another class *H'* that has
Pr[*yellow* | *H'*] = 10% and Pr[*yellow* | *H'*] = 90%:

$$Pr[\{\text{blue yellow blue}\}|H'] \approx 3! \times \frac{0.1^1}{1!} \times \frac{0.9^2}{2!} = 0.24$$

- Need to take prior probability of class into account to make final classification
- Factorials don't actually need to be computed
- Underflows can be prevented by using logarithms

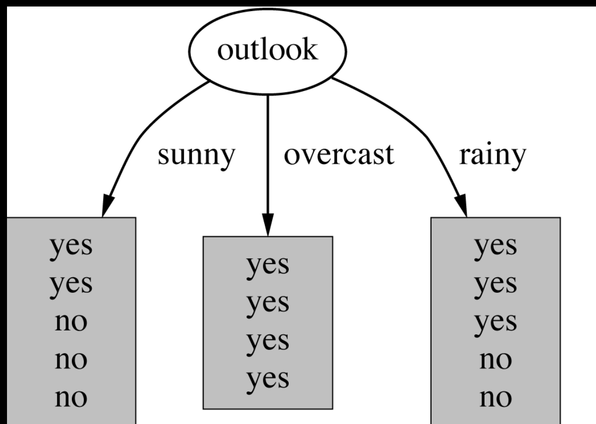
Naïve Bayes: discussion

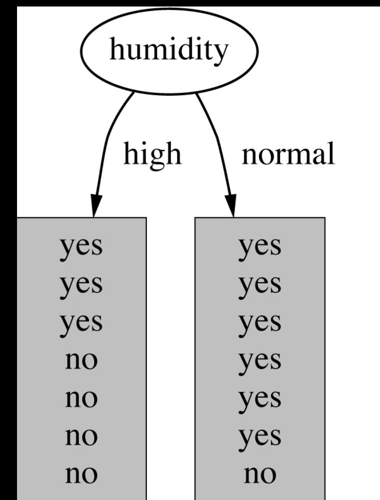
- Naïve Bayes works surprisingly well (even if independence assumption is clearly violated)
- Why? Because classification doesn't require accurate probability estimates *as long as maximum probability is assigned to correct class*
- However: adding too many redundant attributes will cause problems (e.g. identical attributes)
- Note also: many numeric attributes are not normally distributed (→*kernel density estimators*)

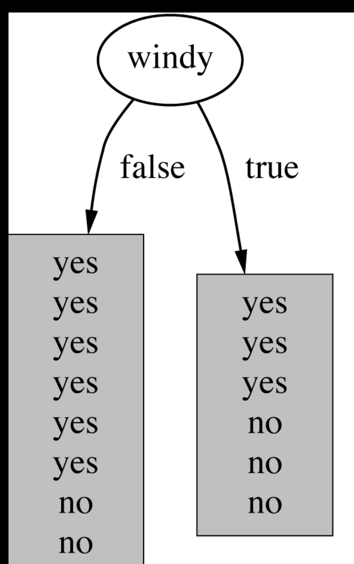
Constructing decision trees

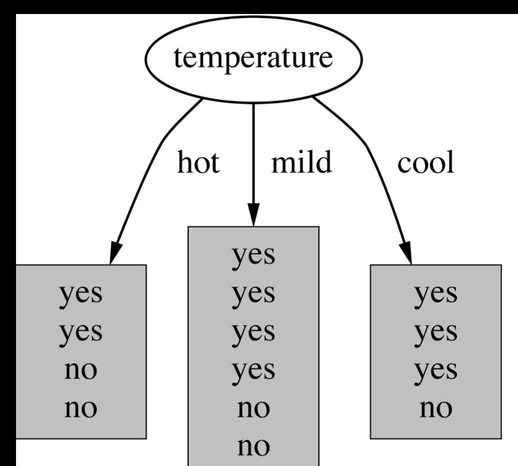
- Strategy: top down
  Recursive *divide-and-conquer* fashion
  - First: select attribute for root node
    Create branch for each possible attribute value
  - Then: split instances into subsets
    One for each branch extending from the node
  - Finally: repeat recursively for each branch, using only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

Which attribute to select?

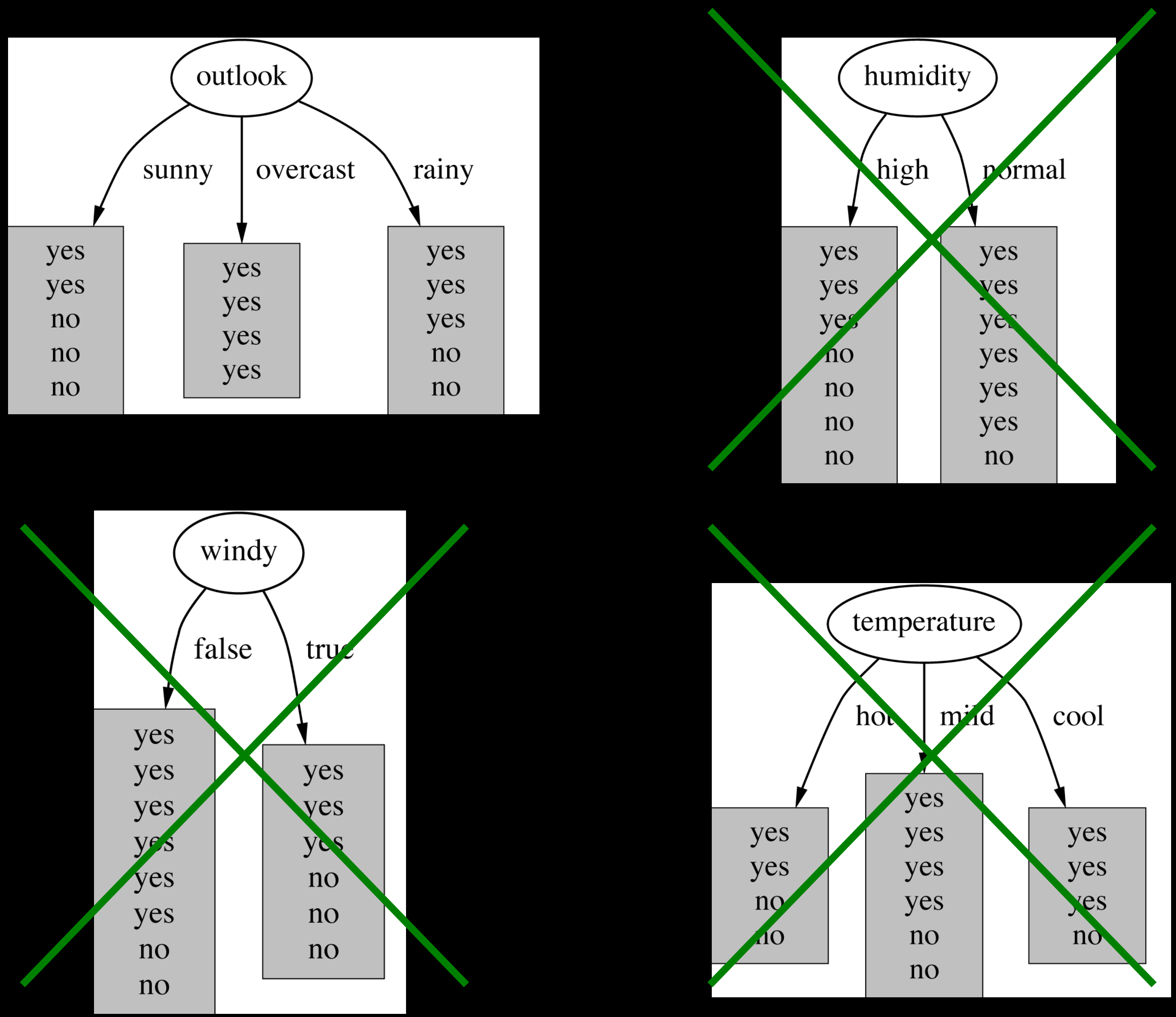








Which attribute to select?



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 *impurity criterion*: *information gain*
  - Information gain increases with the average purity of the subsets
- Strategy: choose attribute that gives greatest information gain

Computing information

- Measure information in *bits*
  - Given a probability distribution, the info required to predict an event is the distribution's *entropy*
  - Entropy gives the information required in bits (can involve fractions of bits!)
- Formula for computing the entropy:

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

Example: attribute *Outlook*

- $Outlook = Sunny$:

$$\text{info}([2,3]) = \text{entropy}(2/5, 3/5) = -2/5 \log(2/5) - 3/5 \log(3/5) = 0.971 \text{ bits}$$

- $Outlook = Overcast$:

$$\text{info}([4,0]) = \text{entropy}(1,0) = -1 \log(1) - 0 \log(0) = 0 \text{ bits}$$

Note: this is normally undefined.

- $Outlook = Rainy$:

$$\text{info}([2,3]) = \text{entropy}(3/5, 2/5) = -3/5 \log(3/5) - 2/5 \log(2/5) = 0.971 \text{ bits}$$

- Expected information for attribute:

$$\text{info}([3,2],[4,0],[3,2]) = (5/14) \times 0.971 + (4/14) \times 0 + (5/14) \times 0.971 = 0.693 \text{ bits}$$

Computing information gain

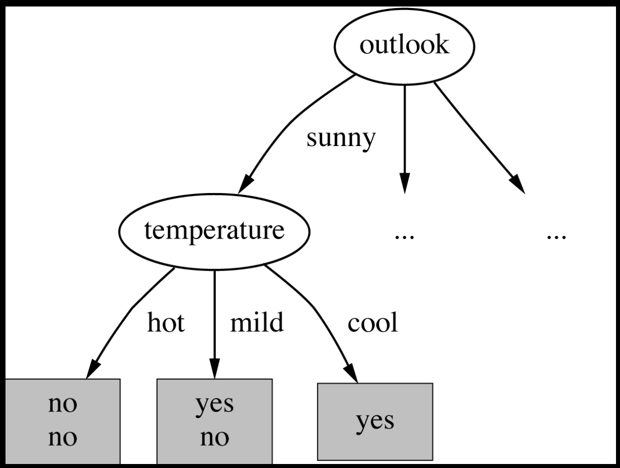
- Information gain: information before splitting – information after splitting

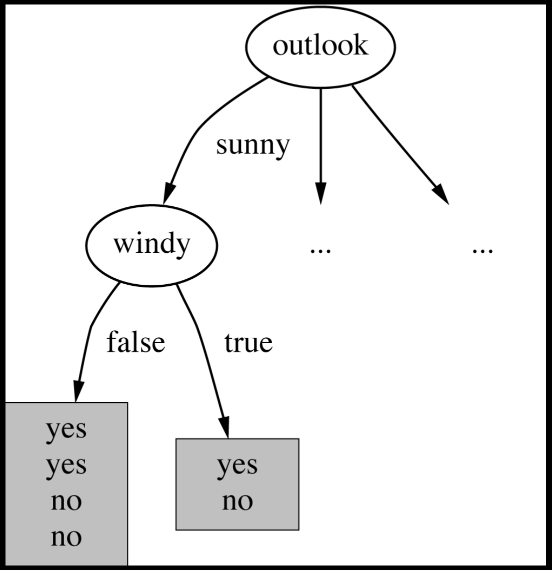
$$\text{gain}(\textit{Outlook}) = \text{info}([9,5]) - \text{info}([2,3],[4,0],[3,2])$$
$$= 0.940 - 0.693$$
$$= 0.247 \text{ bits}$$

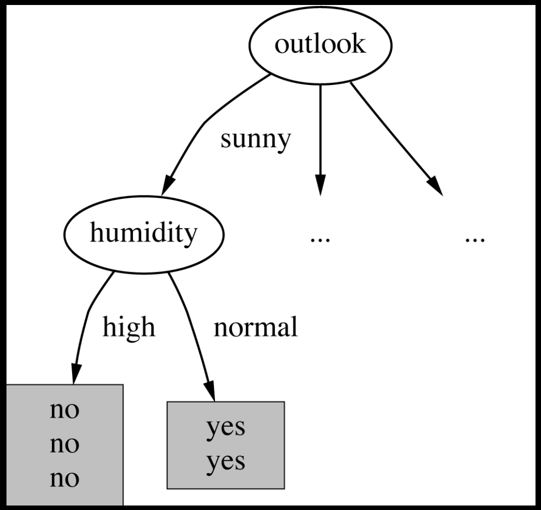
- Information gain for attributes from weather data:

$$\text{gain}(\textit{Outlook}) = 0.247 \text{ bits}$$
$$\text{gain}(\textit{Temperature}) = 0.029 \text{ bits}$$
$$\text{gain}(\textit{Humidity}) = 0.152 \text{ bits}$$
$$\text{gain}(\textit{Windy}) = 0.048 \text{ bits}$$

Continuing to split

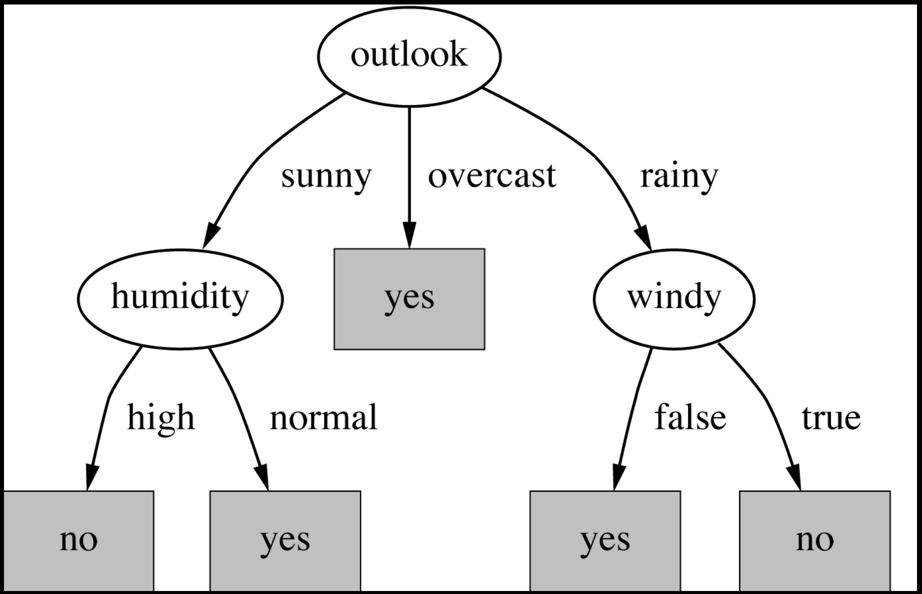






$$\text{gain}(\textit{Temperature}) = 0.571 \text{ bits}$$
$$\text{gain}(\textit{Humidity}) = 0.971 \text{ bits}$$
$$\text{gain}(\textit{Windy}) = 0.020 \text{ bits}$$

Final decision tree



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

Wishlist for a purity measure

- Properties we require from a purity 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 obey *multistage property* (i.e. decisions can be made in several stages):

$$\text{measure}([2,3,4]) = \text{measure}([2,7]) + (7/9) \times \textit{measure}([3,4])$$

- Entropy is the only function that satisfies all three properties!

Properties of the entropy

- The multistage property:

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

- Simplification of computation:

$$\text{info}([2,3,4]) = -2/9 \times \log(2/9) - 3/9 \times \log(3/9) - 4/9 \times \log(4/9)$$

$$= [-2 \times \log 2 - 3 \times \log 3 - 4 \times \log 4 + 9 \times \log 9]/9$$

- Note: instead of maximizing info gain we could just minimize information

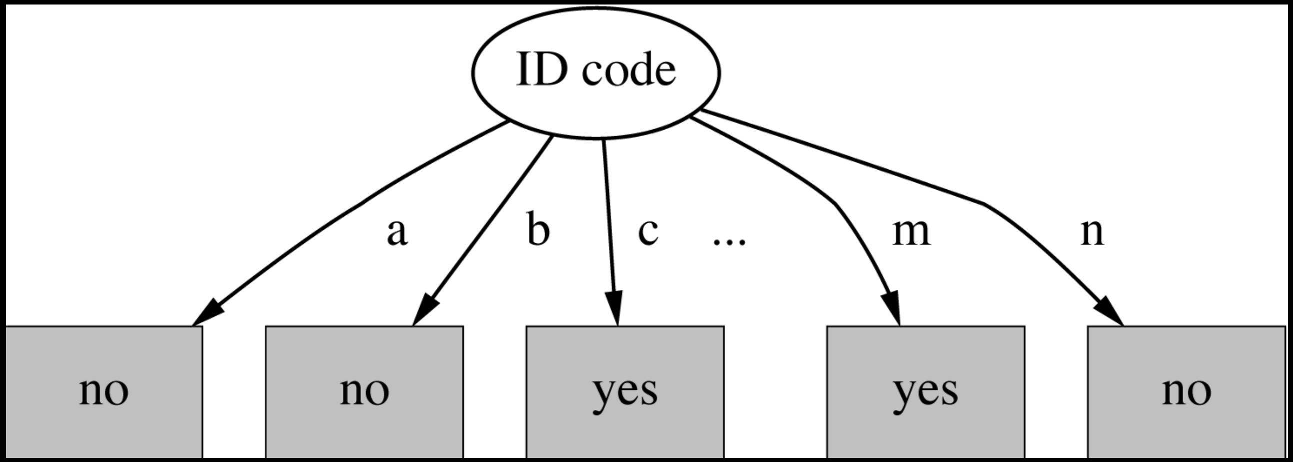
Highly-branching attributes

- 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)
- Another problem: *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



- Entropy of split:

$$\text{info}(\text{ID code}) = \mathit{info}([0,1]) + \mathit{info}([0,1]) + \ldots + \mathit{info}([0,1]) = 0\ \mathit{bits}$$

$\Rightarrow$ Information gain is maximal for ID code (namely 0.940 bits)

Gain ratio

- *Gain ratio*: a modification of the information gain that reduces its bias
- 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 distribution of instances into branches (i.e. how much info do we need to tell which branch an instance belongs to)

Computing the gain ratio

- Example: intrinsic information for ID code

$$\text{info}([1,1,\ldots,1])=14\times(-1/14\times\log(1/14))=3.807\,\text{bits}$$

- Value of attribute decreases as intrinsic information gets larger
- Definition of gain ratio:

$$\text{gain\_ratio}(\textit{attribute})=\frac{\text{gain}(\textit{attribute})}{\text{intrinsic\_info}(\textit{attribute})}$$

- Example:

$$\text{gain\_ratio}(\text{ID code})=\frac{0.940\,\text{bits}}{3.807\,\text{bits}}=0.246$$

Gain ratios for 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 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

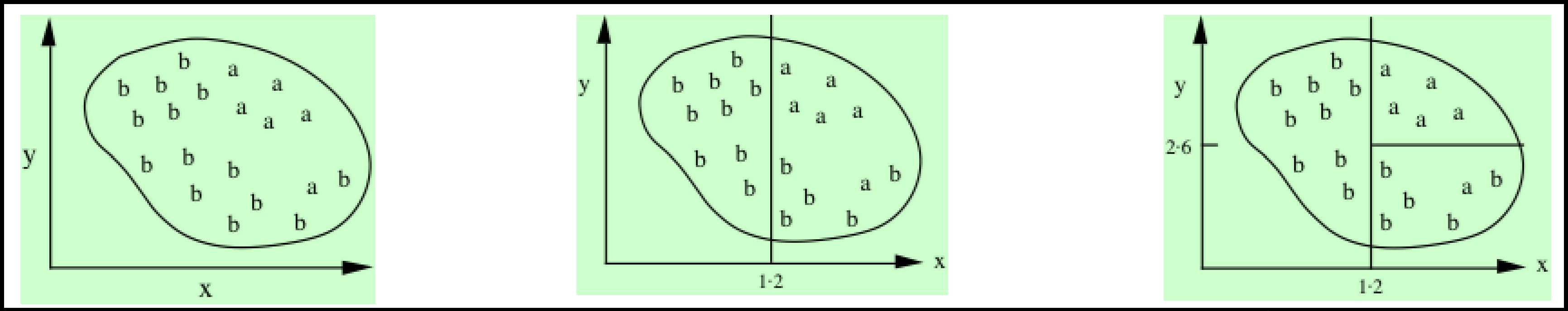
Discussion

- Top-down induction of decision trees: ID3, algorithm developed by Ross Quinlan
  - Gain ratio just one modification of this basic algorithm
  - $\Rightarrow$ C4.5: deals with numeric attributes, missing values, noisy data
- Similar approach: CART
- There are many other attribute selection criteria!
  (But little difference in accuracy of result)

Covering algorithms

- Convert decision tree into a rule set
  - Straightforward, but rule set overly complex
  - More effective conversions are not trivial
- Instead, can generate rule set directly
  - for each class in turn find rule set that covers all instances in it (excluding instances not in the class)
- Called a *covering* approach:
  - at each stage a rule is identified that "covers" some of the instances

Example: generating a rule




```
If true
   then class = a
```

```
If x > 1.2
   then class = a
```

```
If x > 1.2 and y > 2.6
   then class = a
```

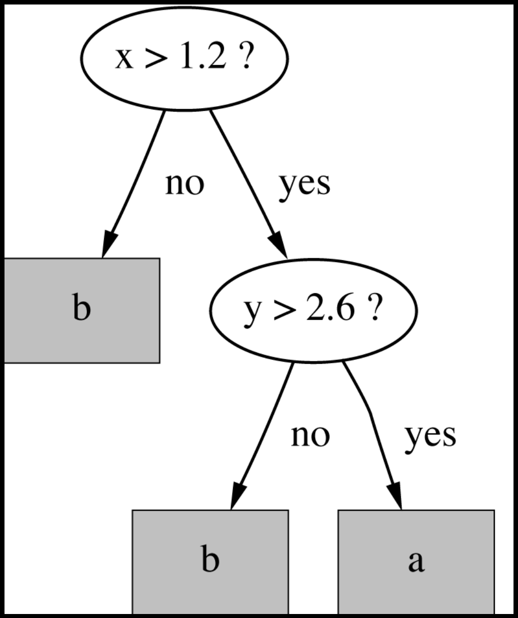
- Possible rule set for class "b":

```
If x ≤ 1.2 then class = b
If x > 1.2 and y ≤ 2.6 then class = b
```

- Could add more rules, get "perfect" rule set

Rules vs. trees

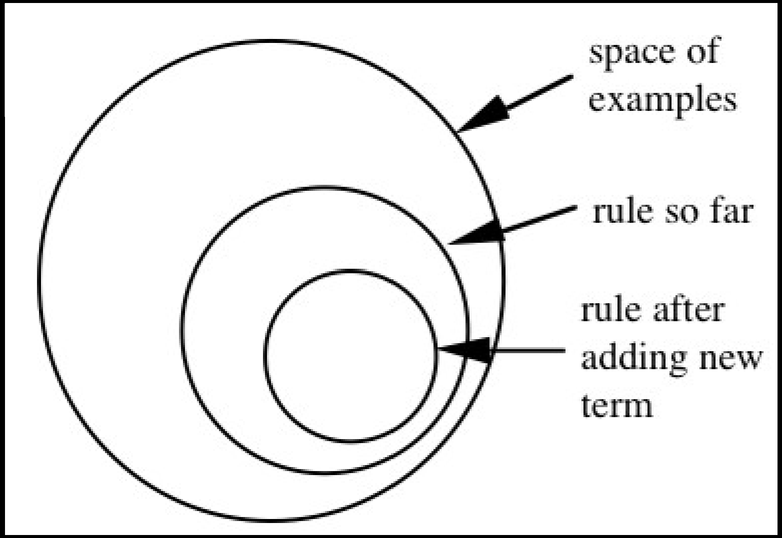
Corresponding decision tree:
(produces exactly the same predictions)



- But: rule sets *can* be more perspicuous when decision trees suffer from replicated subtrees
- Also: in multiclass situations, covering algorithm concentrates on one class at a time whereas decision tree learner takes all classes into account

Simple covering algorithm

- Generates a rule by adding tests that maximize rule's accuracy
- Similar to situation in decision trees: problem of selecting an attribute to split on
  - But: decision tree inducer maximizes overall purity
- Each new test reduces rule's coverage:



Selecting a test

- Goal: maximize accuracy
  - t total number of instances covered by rule
  - p positive examples of the class covered by rule
  - $t - p$ number of errors made by rule
  - ⇒ Select test that maximizes the ratio p/t
- We are finished when $p/t = 1$ or the set of instances can't be split any further

Example: contact lens data

- Rule we seek:
- Possible tests:

```
If ?
    then recommendation = hard
```

Age = Young	2/8
Age = Pre-presbyopic	1/8
Age = Presbyopic	1/8
Spectacle prescription = Myope	3/12
Spectacle prescription = Hypermetrope	1/12
Astigmatism = no	0/12
Astigmatism = yes	4/12
Tear production rate = Reduced	0/12
Tear production rate = Normal	4/12

Modified rule and resulting data

- Rule with best test added:

```
If astigmatism = yes
    then recommendation = hard
```

- Instances covered by modified rule:

Age	Spectacle prescription	Astigmatism	Tear production rate	Recommended lenses
Young	Myope	Yes	Reduced	None
Young	Myope	Yes	Normal	Hard
Young	Hypermetrope	Yes	Reduced	None
Young	Hypermetrope	Yes	Normal	hard
Pre-presbyopic	Myope	Yes	Reduced	None
Pre-presbyopic	Myope	Yes	Normal	Hard
Pre-presbyopic	Hypermetrope	Yes	Reduced	None
Pre-presbyopic	Hypermetrope	Yes	Normal	None
Presbyopic	Myope	Yes	Reduced	None
Presbyopic	Myope	Yes	Normal	Hard
Presbyopic	Hypermetrope	Yes	Reduced	None
Presbyopic	Hypermetrope	Yes	Normal	None

Further refinement

- Current state:

```
If astigmatism = yes
    and ?
  then recommendation = hard
```

- Possible tests:

Age = Young	2/4
Age = Pre-presbyopic	1/4
Age = Presbyopic	1/4
Spectacle prescription = Myope	3/6
Spectacle prescription = Hypermetrope	1/6
Tear production rate = Reduced	0/6
Tear production rate = Normal	4/6

Modified rule and resulting data

- Rule with best test added:

```
If astigmatism = yes
    and tear production rate = normal
  then recommendation = hard
```

- Instances covered by modified rule:

Age	Spectacle prescription	Astigmatism	Tear production rate	Recommended lenses
Young	Myope	Yes	Normal	Hard
Young	Hypermetrope	Yes	Normal	hard
Pre-presbyopic	Myope	Yes	Normal	Hard
Pre-presbyopic	Hypermetrope	Yes	Normal	None
Presbyopic	Myope	Yes	Normal	Hard
Presbyopic	Hypermetrope	Yes	Normal	None

Further refinement

- Current state:

```
If astigmatism = yes
     and tear production rate = normal
     and ?
   then recommendation = hard
```

- Possible tests:

Age = Young	2/2
Age = Pre-presbyopic	1/2
Age = Presbyopic	1/2
Spectacle prescription = Myope	3/3
Spectacle prescription = Hypermetrope	1/3

- Tie between the first and the fourth test
  - We choose the one with greater coverage

The result

- Final rule:

```
If astigmatism = yes
   and tear production rate = normal
   and spectacle prescription = myope
   then recommendation = hard
```

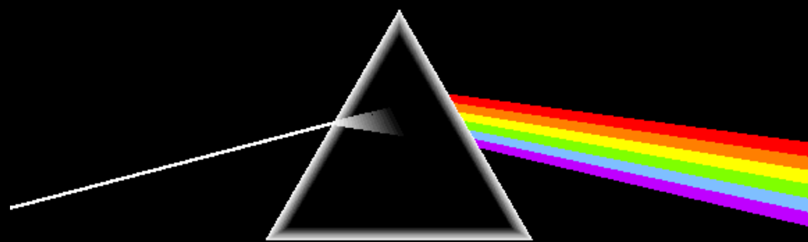
- Second rule for recommending "hard lenses":
  (built from instances not covered by first rule)

```
If age = young and astigmatism = yes
   and tear production rate = normal
   then recommendation = hard
```

- These two rules cover all "hard lenses":
  - Process is repeated with other two classes

Pseudo-code for PRISM

```
For each class C
  Initialize E to the instance set
  While E contains instances in class C
    Create a rule R with an empty left-hand side that predicts class C
    Until R is perfect (or there are no more attributes to use) do
      For each attribute A not mentioned in R, and each value v,
        Consider adding the condition A = v to the left-hand side of R
        Select A and v to maximize the accuracy p/t
          (break ties by choosing the condition with the largest p)
      Add A = v to R
    Remove the instances covered by R from E
```



Rules vs. decision lists

- PRISM with outer loop removed generates a decision list for one class
  - Subsequent rules are designed for rules that are not covered by previous rules
  - But: order doesn't matter because all rules predict the same class
- Outer loop considers all classes separately
  - No order dependence implied
- Problems: overlapping rules, default rule required

Separate and conquer

- Methods like PRISM (for dealing with one class) are *separate-and-conquer* algorithms:
  - First, identify a useful rule
  - Then, separate out all the instances it covers
  - Finally, "conquer" the remaining instances
- Difference to divide-and-conquer methods:
  - Subset covered by rule doesn't need to be explored any further

Mining association rules

- Naïve method for finding association rules:
  - Use separate-and-conquer method
  - Treat every possible combination of attribute values as a separate class
- Two problems:
  - Computational complexity
  - Resulting number of rules (which would have to be pruned on the basis of support and confidence)
- But: we can look for high support rules directly!

Item sets

- Support: number of instances correctly covered by association rule
    - The same as the number of instances covered by *all* tests in the rule (LHS and RHS!)
- *Item*: one test/attribute-value pair
- *Item set* : all items occurring in a rule
- Goal: only rules that exceed pre-defined support

  ⇒ Do it by finding all item sets with the given minimum support and generating rules from them!

Weather data

Outlook	Temp	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
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

Item sets for weather data

One-item sets	Two-item sets	Three-item sets	Four-item sets
Outlook = Sunny (5)	Outlook = Sunny Temperature = Hot (2)	Outlook = Sunny Temperature = Hot Humidity = High (2)	Outlook = Sunny Temperature = Hot Humidity = High Play = No (2)
Temperature = Cool (4)	Outlook = Sunny Humidity = High (3)	Outlook = Sunny Humidity = High Windy = False (2)	Outlook = Rainy Temperature = Mild Windy = False Play = Yes (2)
…	…	…	…

- In total: 12 one-item sets, 47 two-item sets, 39 three-item sets, 6 four-item sets and 0 five-item sets (with minimum support of two)

Generating rules from an item set

- Once all item sets with minimum support have been generated, we can turn them into rules
- Example:

```
Humidity = Normal, Windy = False, Play = Yes (4)
```

- Seven (2^N-1) potential rules:

```
If Humidity = Normal and Windy = False then Play = Yes    4/4
If Humidity = Normal and Play = Yes then Windy = False    4/6
If Windy = False and Play = Yes then Humidity = Normal    4/6
If Humidity = Normal then Windy = False and Play = Yes    4/7
If Windy = False then Humidity = Normal and Play = Yes    4/8
If Play = Yes then Humidity = Normal and Windy = False    4/9
If True then Humidity = Normal and Windy = False
   and Play = Yes                                         4/12
```

Rules for weather data

- Rules with support > 1 and confidence = 100%:

	Association rule		Sup.	Conf.
1	Humidity=Normal Windy=False	⇒Play=Yes	4	100%
2	Temperature=Cool	⇒Humidity=Normal	4	100%
3	Outlook=Overcast	⇒Play=Yes	4	100%
4	Temperature=Cold Play=Yes	⇒Humidity=Normal	3	100%
	...	...	...	...
58	Outlook=Sunny Temperature=Hot	⇒Humidity=High	2	100%

- In total:
  3 rules with support four
  5 with support three
  50 with support two

Example rules from the same set

- Item set:

```
Temperature = Cool, Humidity = Normal, Windy = False, Play = Yes (2)
```

- Resulting rules (all with 100% confidence):

```
Temperature = Cool, Windy = False ⇒Humidity = Normal, Play = Yes
Temperature = Cool, Windy = False, Humidity = Normal ⇒Play = Yes
Temperature = Cool, Windy = False, Play = Yes ⇒Humidity = Normal
```

due to the following "frequent" item sets:

```
Temperature = Cool, Windy = False                      (2)
Temperature = Cool, Humidity = Normal, Windy = False   (2)
Temperature = Cool, Windy = False, Play = Yes          (2)
```

Generating item sets efficiently

- How can we efficiently find all frequent item sets?
- Finding one-item sets easy
- Idea: use one-item sets to generate two-item sets, two-item sets to generate three-item sets, …
  - If (A B) is frequent item set, then (A) and (B) have to be frequent item sets as well!
  - In general: if X is frequent *k*-item set, then all (*k*-1)-item subsets of X are also frequent

  ⇒ Compute *k*-item set by merging (*k*-1)-item sets

Example

- Given: five three-item sets

  `(A B C), (A B D), (A C D), (A C E), (B C D)`

- Lexicographically ordered!
- Candidate four-item sets:

  ```
  (A B C D)       OK because of (A C D) (B C D)
  (A C D E)       Not OK because of (C D E)
  ```

- Final check by counting instances in dataset!
- $(k-1)$-item sets are stored in hash table

Generating rules efficiently

- We are looking for all high-confidence rules
  - Support of antecedent obtained from hash table
  - But: brute-force method is (2^N-1)
- Better way: building $(c + 1)$-consequent rules from c-consequent ones
  - Observation: $(c + 1)$-consequent rule can only hold if all corresponding c-consequent rules also hold
- Resulting algorithm similar to procedure for large item sets

Example

- 1-consequent rules:

```
If Outlook = Sunny and Windy = False and Play = No
   then Humidity = High (2/2)
```

```
If Humidity = High and Windy = False and Play = No
   then Outlook = Sunny (2/2)
```

Corresponding 2-consequent rule:

```
If Windy = False and Play = No
   then Outlook = Sunny and Humidity = High (2/2)
```

- Final check of antecedent against hash table!

Association rules: discussion

- Above method makes one pass through the data for each different size item set
  - Other possibility: generate (k+2)-item sets just after (k+1)-item sets have been generated
  - Result: more (k+2)-item sets than necessary will be considered but less passes through the data
  - Makes sense if data too large for main memory
- Practical issue: generating a certain number of rules (e.g. by incrementally reducing min. support)

Other issues

- Standard ARFF format very inefficient for typical *market basket data*
  - Attributes represent items in a basket and most items are usually missing
  - Data should be represented in sparse format
- Instances are also called *transactions*
- Confidence is not necessarily the best measure
  - Example: milk occurs in almost every supermarket transaction
  - Other measures have been devised (e.g. lift)

Linear models: linear regression

- Work most naturally with numeric attributes
- Standard technique for numeric prediction
  - Outcome is linear combination of attributes

$$x = w_0 + w_1 a_1 + w_2 a_2 + \ldots + w_k a_k$$

- Weights are calculated from the training data
- Predicted value for first training instance $\mathbf{a}^{(1)}$

$$w_0 a_0^{(1)} + w_1 a_1^{(1)} + w_2 a_2^{(1)} + \ldots + w_k a_k^{(1)} = \sum_{j=0}^{k} w_j a_j^{(1)}$$

(assuming each instance is extended with a constant attribute with value 1)

Minimizing the squared error

- Choose $k+1$ coefficients to minimize the squared error on the training data
- Squared error:
- $$\sum_{i=1}^{n} \left(x^{(i)} - \sum_{j=0}^{k} w_j a_j^{(i)} \right)^2$$
- Derive coefficients using standard matrix operations
- Can be done if there are more instances than attributes (roughly speaking)
- Minimizing the *absolute error* is more difficult

Classification

- *Any* regression technique can be used for classification
  - Training: perform a regression for each class, setting the output to 1 for training instances that belong to class, and 0 for those that don't
  - Prediction: predict class corresponding to model with largest output value (*membership value*)
- For linear regression this is known as *multi-response linear regression*
- Problem: membership values are not in [0,1] range, so aren't proper probability estimates

Linear models: logistic regression

- Builds a linear model for a transformed target variable
- Assume we have two classes
- Logistic regression replaces the target

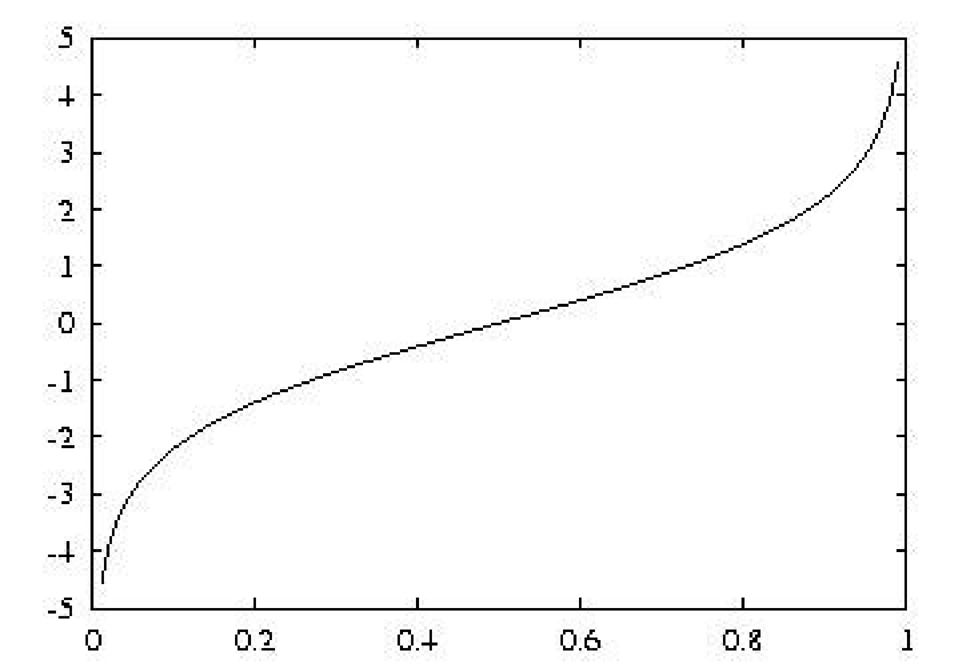
$$P[1|a_{1,}a_{2,}....,a_k]$$

by this target

$$\log\left(\frac{P[1|a_{1,}a_{2,}....,a_k]}{(1-P[1|a_{1,}a_{2,}....,a_k])}\right)$$

- *Logit transformation* maps [0,1] to (-∞, +∞)

Logit transformation

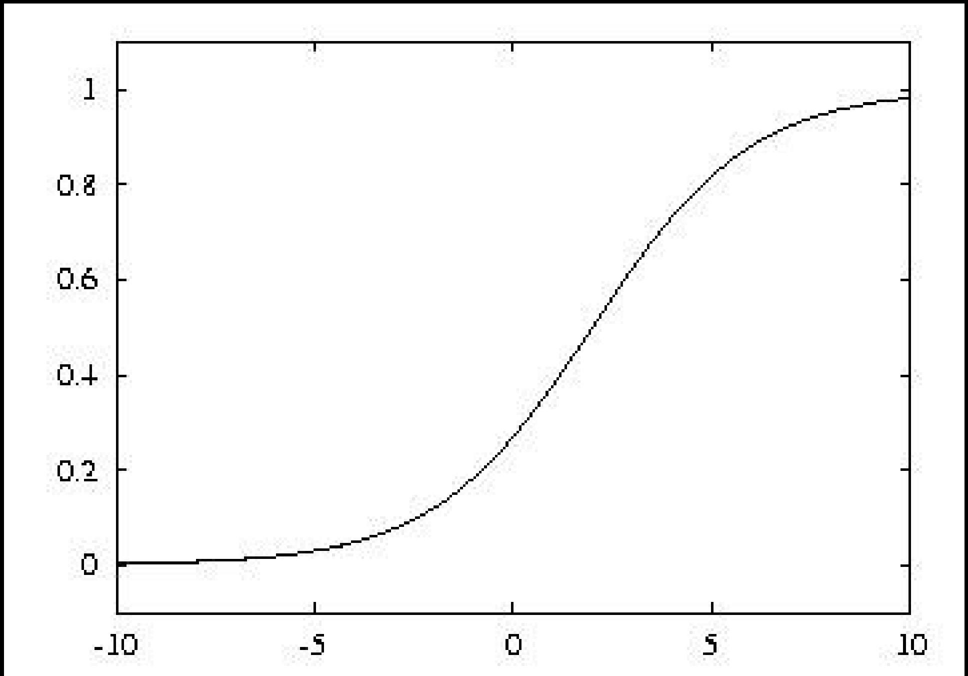


- Resulting model:

$$Pr[1|a_1, a_2, \ldots, a_k] = \frac{1}{(1+e^{-w_0 - w_1 a_1 - \ldots - w_k a_k})}$$

Example logistic regression model

- Model with $w_0 = 0.5$ and $w_1 = 1$:



- Parameters are found from training data using *maximum likelihood*

Maximum likelihood

- Aim: maximize probability of training data wrt parameters
- Can use logarithms of probabilities and maximize *log-likelihood* of model:

$$\sum_{i=1}^{n} (1-x^{(i)})\log(1-Pr[1|a_1^{(i)},a_2^{(i)},\ldots,a_k^{(i)}])+ x^{(i)}\log Pr[1|a_1^{(i)},a_2^{(i)},\ldots,a_k^{(i)}]$$

where the $x^{(i)}$ are either 0 or 1

- Weights w_i need to be chosen to maximize log-likelihood (relatively simple method: *iteratively re-weighted least squares*)

Multiple classes

- Can perform logistic regression independently for each class (like multi-response linear regression)
- Problem: probability estimates for different classes won't sum to one
- Better: train coupled models by maximizing likelihood over all classes
- Alternative that often works well in practice: *pairwise classification*

Pairwise classification

- Idea: build model for each pair of classes, using only training data from those classes
- Problem? Have to solve $k(k-1)/2$ classification problems for k-class problem
- Turns out not to be a problem in many cases because training sets become small:
  - Assume data evenly distributed, i.e. $2n/k$ per learning problem for n instances in total
  - Suppose learning algorithm is linear in n
  - Then runtime of pairwise classification is proportional to $(k(k-1)/2)\times(2n/k) = (k-1)n$

Linear models are hyperplanes

- Decision boundary for two-class logistic regression is where probability equals 0.5:

$$Pr[1|a_{1,} a_{2,} \ldots, a_k] = 1/(1+\exp(-w_0 - w_1 a_1 - \ldots - w_k a_k)) = 0.5$$

which occurs when $-w_0 - w_1 a_1 - \ldots - w_k a_k = 0$

- Thus logistic regression can only separate data that can be separated by a hyperplane
- Multi-response linear regression has the same problem. Class 1 is assigned if:

$$w_0^{(1)} + w_1^{(1)} a_1 + \ldots + w_k^{(1)} a_k > w_0^{(2)} + w_1^{(2)} a_1 + \ldots + w_k^{(2)} a_k$$

$$\Leftrightarrow (w_0^{(1)} - w_0^{(2)}) + (w_1^{(1)} - w_1^{(2)}) a_1 + \ldots + (w_k^{(1)} - w_k^{(2)}) a_k > 0$$

Linear models: the perceptron

- Don't actually need probability estimates if all we want to do is classification
- Different approach: learn separating hyperplane
- Assumption: data is *linearly separable*
- Algorithm for learning separating hyperplane: *perceptron learning rule*
- Hyperplane: $0 = w_0 a_0 + w_1 a_1 + w_2 a_2 + \ldots + w_k a_k$
  where we again assume that there is a constant attribute with value 1 (*bias*)
- If sum is greater than zero we predict the first class, otherwise the second class

The algorithm

```
Set all weights to zero
Until all instances in the training data are classified correctly
  For each instance I in the training data
    If I is classified incorrectly by the perceptron
      If I belongs to the first class add it to the weight vector
      else subtract it from the weight vector
```

- Why does this work?
  Consider situation where instance *a* pertaining to the first class has been added:

$$(w_0+a_0)a_0+(w_1+a_1)a_1+(w_2+a_2)a_2+\ldots+(w_k+a_k)a_k$$

This means output for *a* has increased by:

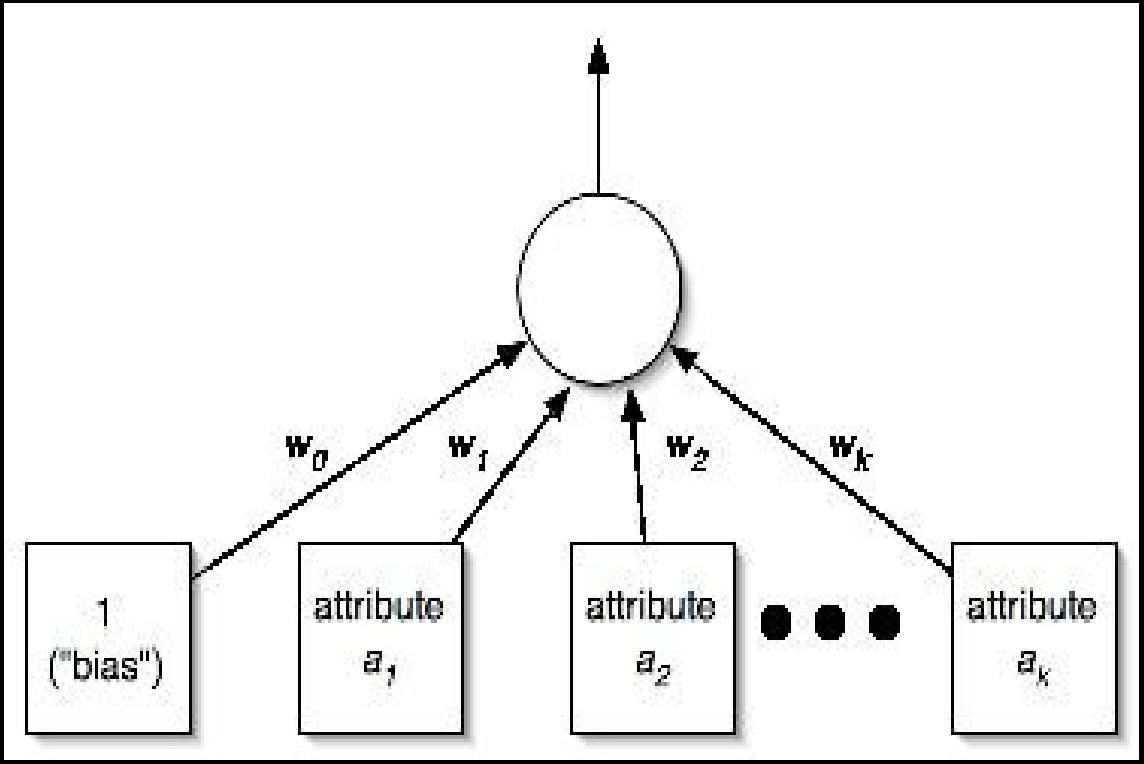
$$a_0 a_0+a_1 a_1+a_2 a_2+\ldots+a_k a_k$$

This number is always positive, thus the hyperplane has moved into the correct direction (and we can show output decreases for instances of other class)

Perceptron as a neural network

Output layer

Input layer



Linear models: Winnow

- Another *mistake-driven* algorithm for finding a separating hyperplane
  - Assumes binary data (i.e. attribute values are either zero or one)
- Difference: *multiplicative* updates instead of *additive* updates
  - Weights are multiplied by a user-specified parameter $\alpha > 1$ (or its inverse)
- Another difference: user-specified threshold parameter θ
  - Predict first class if $w_0 a_0 + w_1 a_1 + w_2 a_2 + \ldots + w_k a_k > \theta$

The algorithm

```
while some instances are misclassified
  for each instance a in the training data
    classify a using the current weights
    if the predicted class is incorrect
      if a belongs to the first class
        for each a_i that is 1, multiply w_i by alpha
        (if a_i is 0, leave w_i unchanged)
      otherwise
        for each a_i that is 1, divide w_i by alpha
        (if a_i is 0, leave w_i unchanged)
```

- Winnow is very effective in homing in on relevant features (*it is attribute efficient*)
- Can also be used in an on-line setting in which new instances arrive continuously (like the perceptron algorithm)

Balanced Winnow

- Winnow doesn't allow negative weights and this can be a drawback in some applications
- *Balanced Winnow* maintains two weight vectors, one for each class:

```
while some instances are misclassified
  for each instance a in the training data
    classify a using the current weights
    if the predicted class is incorrect
      if a belongs to the first class
        for each a_i that is 1, multiply w_i^+ by alpha and divide w_i^- by alpha
          (if a_i is 0, leave w_i^+ and w_i^- unchanged)
      otherwise
        for each a_i that is 1, multiply w_i^- by alpha and divide w_i^+ by alpha
          (if a_i is 0, leave w_i^+ and w_i^- unchanged)
```

- Instance is classified as belonging to the first class (of two classes) if:

$$(w_0^+ - w_0^-)a_0 + (w_1^+ - w_2^-)a_1 + \ldots + (w_k^+ - w_k^-)a_k > \theta$$

Instance-based learning

- Distance function defines what's learned
- Most instance-based schemes use *Euclidean distance*:

$$\sqrt{(a_1^{(1)}-a_1^{(2)})^2+(a_2^{(1)}-a_2^{(2)})^2+\ldots(a_k^{(1)}-a_k^{(2)})^2}$$

  $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(2)}$: two instances with k attributes
- Taking the square root is not required when comparing distances
- Other popular metric: *city-block metric*
  - Adds differences without squaring them

Normalization and other issues

- Different attributes are measured on different scales ⇒need to be *normalized*:

$$a_i = \frac{v_i - \min v_i}{\max v_i - \min v_i}$$

  v_i: the actual value of attribute i

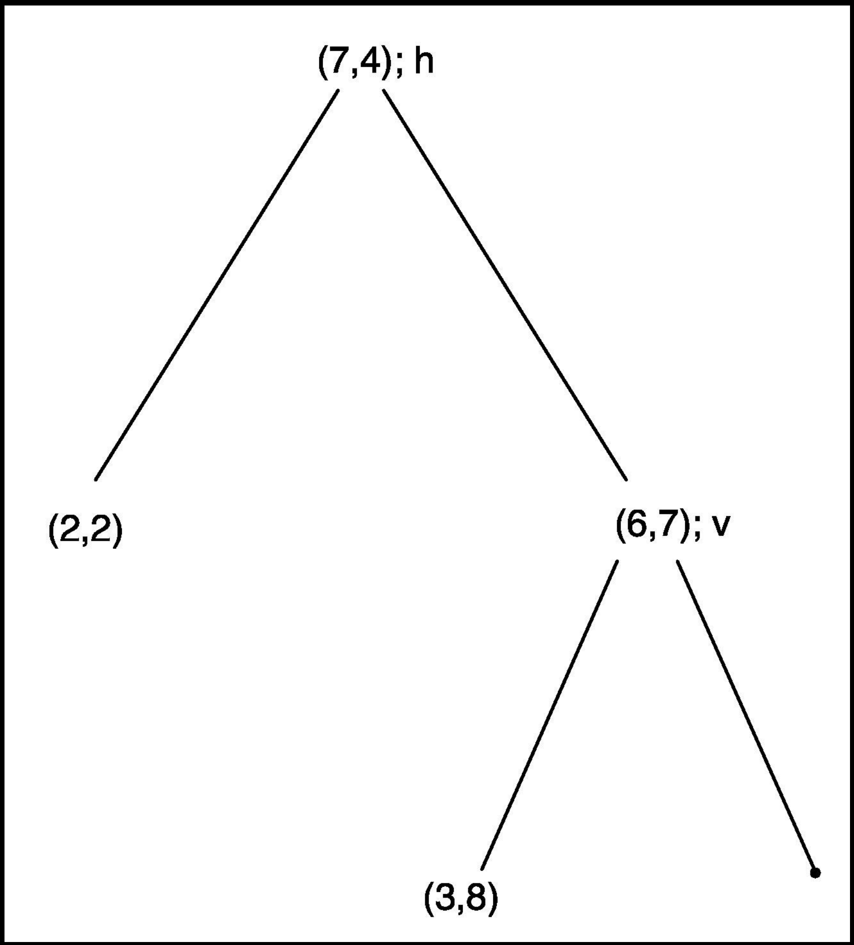
- Nominal attributes: distance either 0 or 1
- Common policy for missing values: assumed to be maximally distant (given normalized attributes)

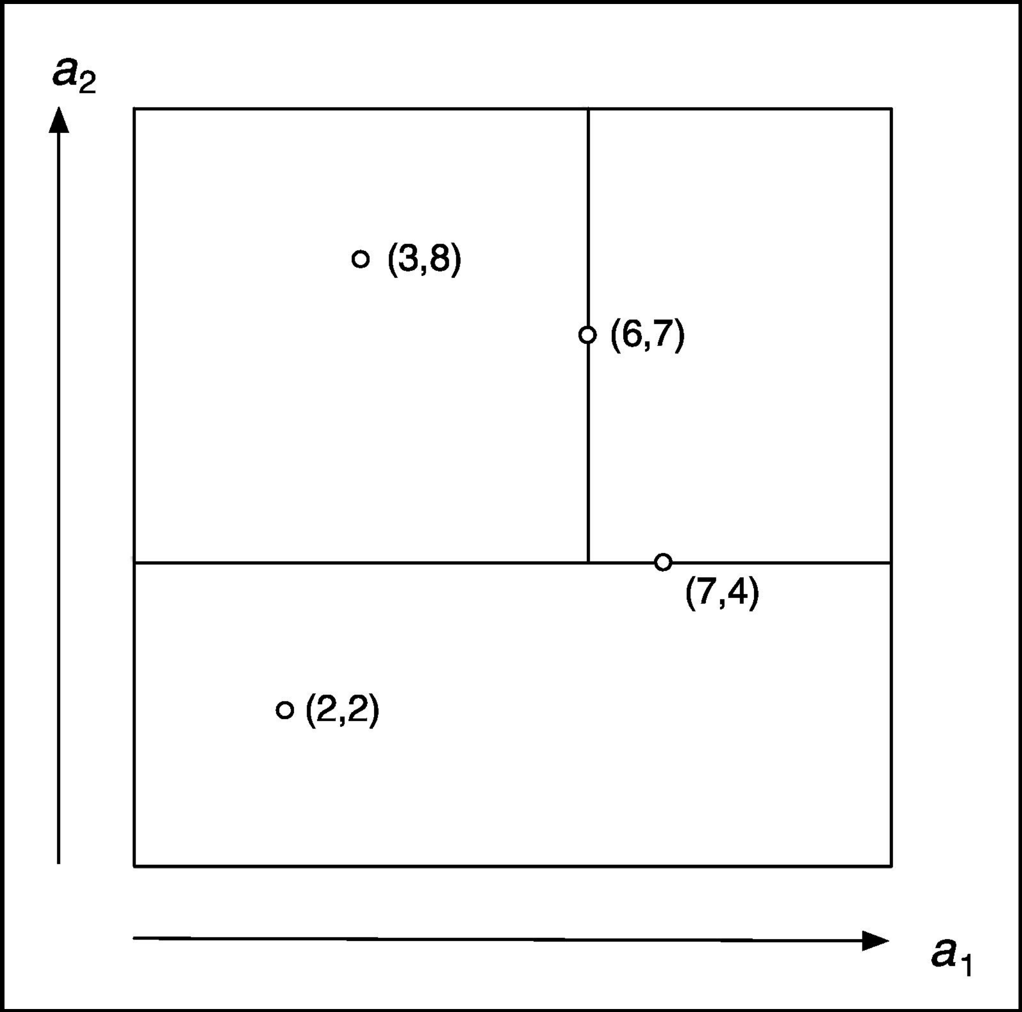
Finding nearest neighbors efficiently

- Simplest way of finding nearest neighbour: linear scan of the data
  - Classification takes time proportional to the product of the number of instances in training and test sets
- Nearest-neighbor search can be done more efficiently using appropriate data structures
- We will discuss two methods that represent training data in a tree structure:

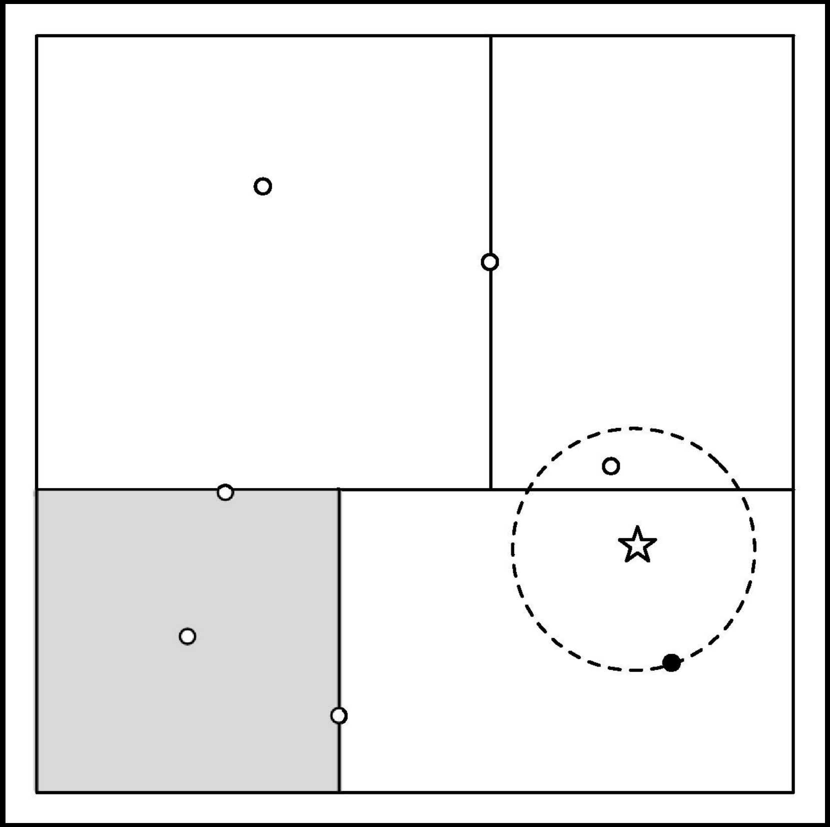
kD-trees and *ball trees*

*k*D-tree example





Using *k*D-trees: example



More on *k*D-trees

- Complexity depends on depth of tree, given by logarithm of number of nodes
- Amount of backtracking required depends on quality of tree (“square” vs. “skinny” nodes)
- How to build a good tree? Need to find good split point and split direction
  - Split direction: direction with greatest variance
  - Split point: median value along that direction
- Using value closest to mean (rather than median) can be better if data is skewed
- Can apply this recursively

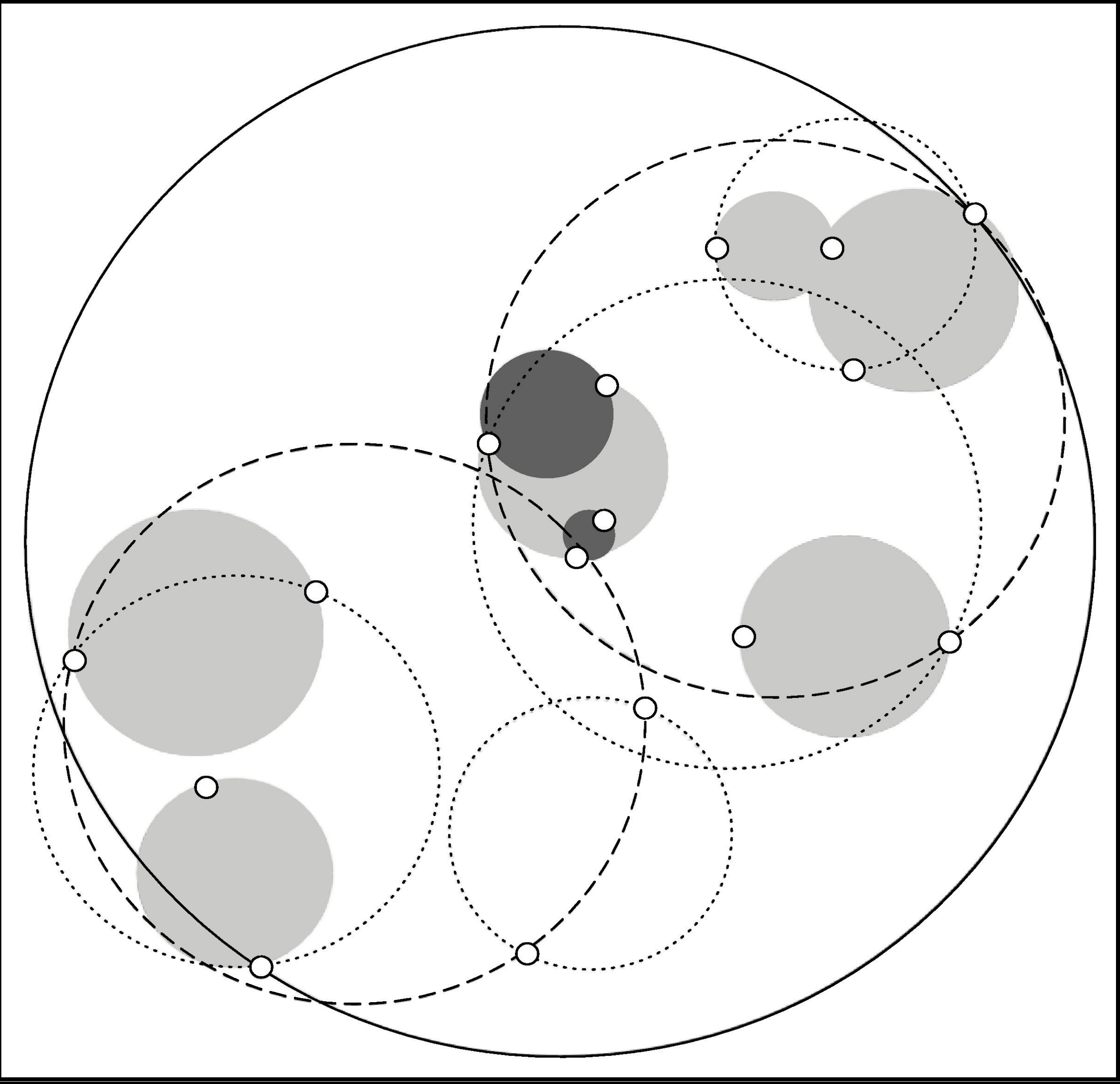
Building trees incrementally

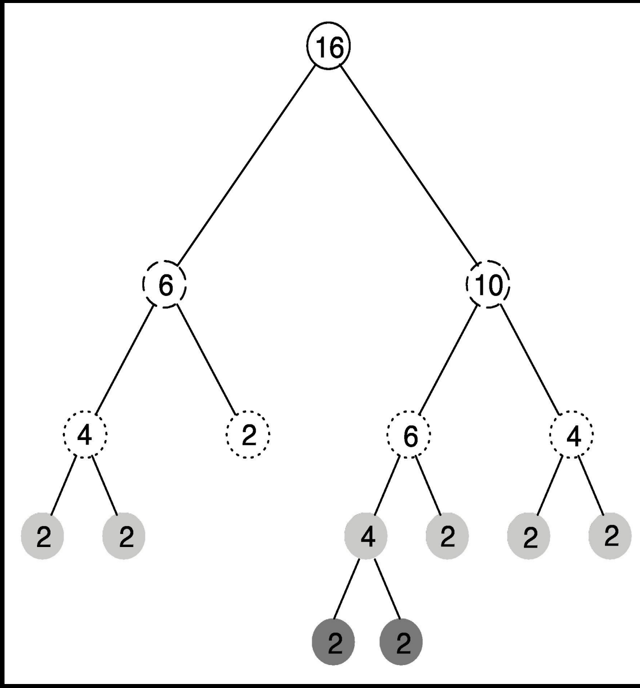
- Big advantage of instance-based learning: classifier can be updated incrementally
  - Just add new training instance!
- Can we do the same with *k*D-trees?
- Heuristic strategy:
  - Find leaf node containing new instance
  - Place instance into leaf if leaf is empty
  - Otherwise, split leaf according to the longest dimension (to preserve squareness)
- Tree should be re-built occasionally (i.e. if depth grows to twice the optimum depth)

Ball trees

- Problem in *k*D-trees: corners
- Observation: no need to make sure that regions don't overlap
- Can use balls (hyperspheres) instead of hyperrectangles
  - A *ball tree* organizes the data into a tree of *k*-dimensional hyperspheres
  - Normally allows for a better fit to the data and thus more efficient search

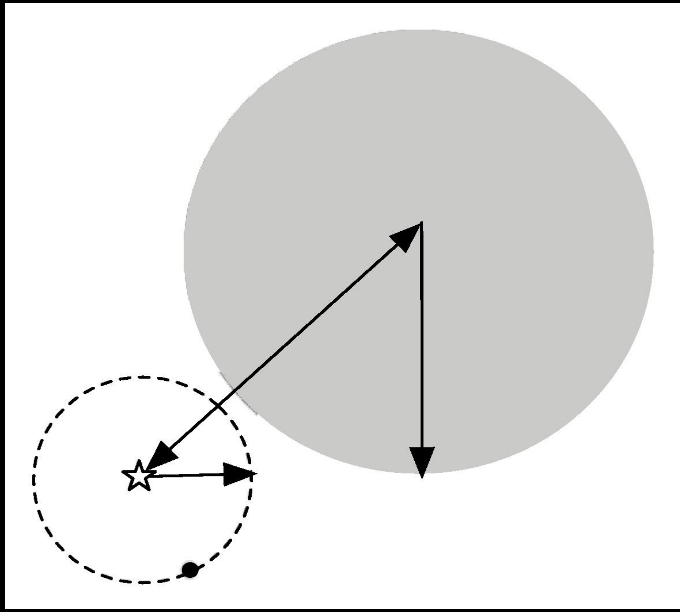
Ball tree example





Using ball trees

- Nearest-neighbor search is done using the same backtracking strategy as in *k*D-trees
- Ball can be ruled out from consideration if: distance from target to ball's center exceeds ball's radius plus current upper bound



Building ball trees

- Ball trees are built top down (like *k*D-trees)
- Don't have to continue until leaf balls contain just two points: can enforce minimum occupancy (same in *k*D-trees)
- Basic problem: splitting a ball into two
- Simple (linear-time) split selection strategy:
  - Choose point farthest from ball's center
  - Choose second point farthest from first one
  - Assign each point to these two points
  - Compute cluster centers and radii based on the two subsets to get two balls

Discussion of nearest-neighbor learning

- Often very accurate
- Assumes all attributes are equally important
  - Remedy: attribute selection or weights
- Possible remedies against noisy instances:
  - Take a majority vote over the *k* nearest neighbors
  - Removing noisy instances from dataset (difficult!)
- Statisticians have used *k*-NN since early 1950s
  - If $n \rightarrow \infty$ and $k/n \rightarrow 0$, error approaches minimum
- *k*D-trees become inefficient when number of attributes is too large (approximately > 10)
- Ball trees (which are instances of *metric trees*) work well in higher-dimensional spaces

More discussion

- Instead of storing all training instances, compress them into regions
- Example: hyperpipes (from discussion of 1R)
- Another simple technique (Voting Feature Intervals):
  - Construct intervals for each attribute
    - Discretize numeric attributes
    - Treat each value of a nominal attribute as an “interval”
  - Count number of times class occurs in interval
  - Prediction is generated by letting intervals vote (those that contain the test instance)

Clustering

- Clustering techniques apply when there is no class to be predicted
- Aim: divide instances into “natural” groups
- As we've seen clusters can be:
  - disjoint vs. overlapping
  - deterministic vs. probabilistic
  - flat vs. hierarchical
- We'll look at a classic clustering algorithm called *k-means*
  - *k-means* clusters are disjoint, deterministic, and flat

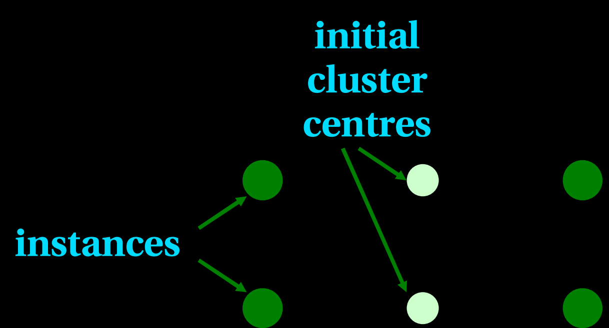
The *k*-means algorithm

To cluster data into *k* groups:
(*k* is predefined)

1. Choose *k* cluster centers
   - e.g. at random
2. Assign instances to clusters
   - based on distance to cluster centers
3. Compute *centroids* of clusters
4. Go to step 1
   - until convergence

Discussion

- Algorithm minimizes squared distance to cluster centers
- Result can vary significantly
  - based on initial choice of seeds
- Can get trapped in local minimum
  - Example:

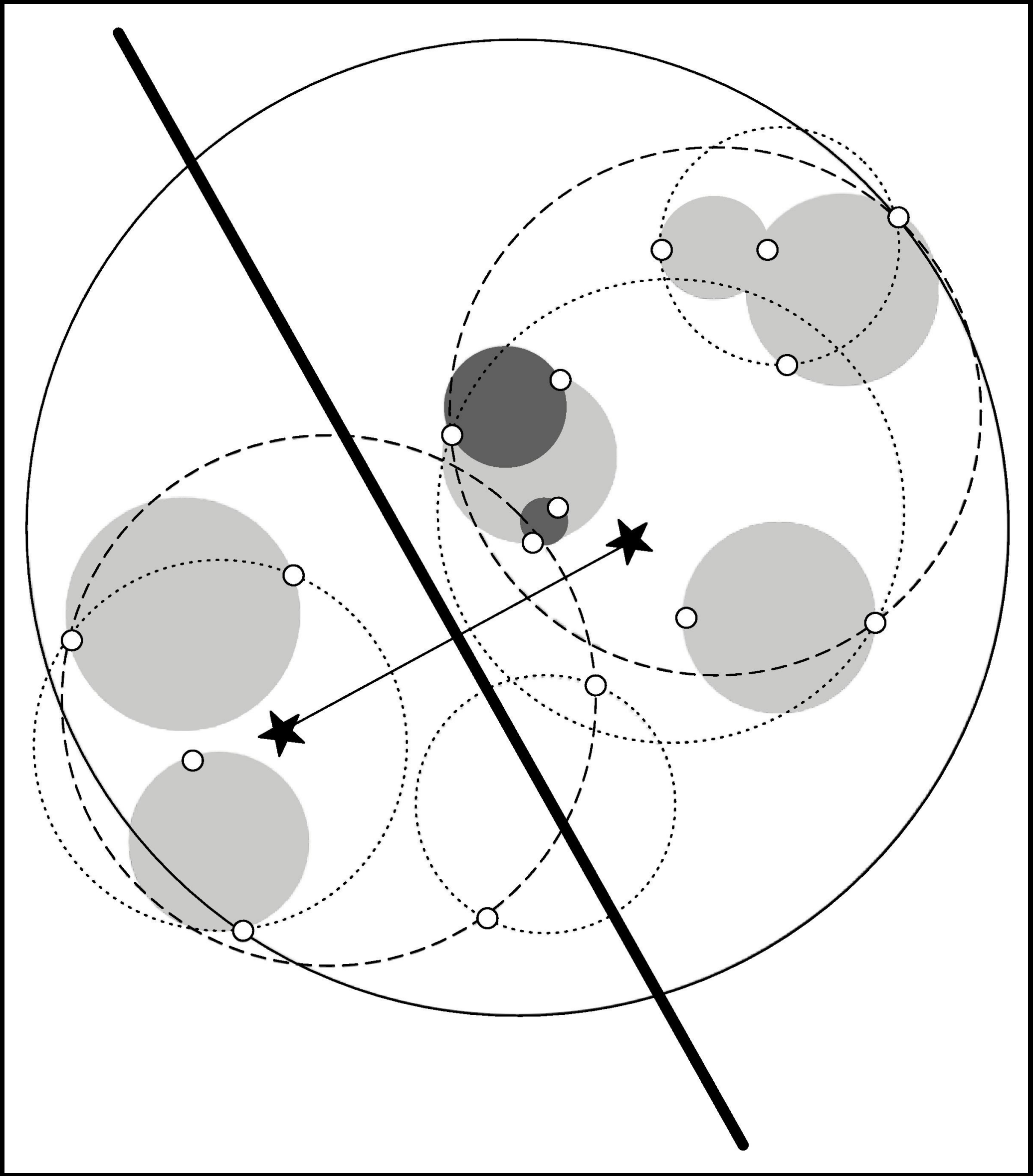


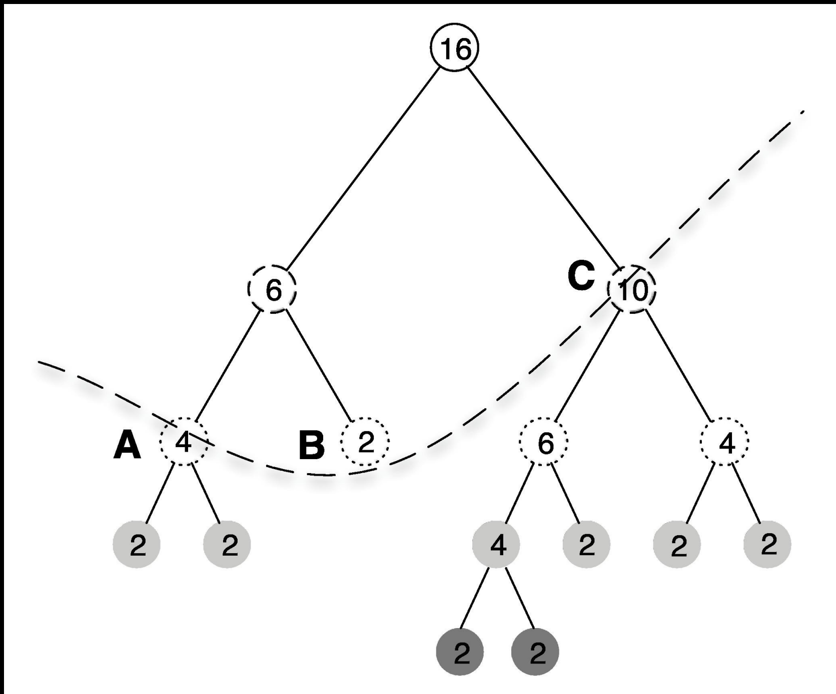
- To increase chance of finding global optimum: restart with different random seeds
- Can we applied recursively with $k = 2$

Faster distance calculations

- Can we use *k*D-trees or ball trees to speed up the process? Yes:
  - First, build tree, which remains static, for all the data points
  - At each node, store number of instances and sum of all instances
  - In each iteration, descend tree and find out which cluster each node belongs to
    - Can stop descending as soon as we find out that a node belongs entirely to a particular cluster
    - Use statistics stored at the nodes to compute new cluster centers

Example





Comments on basic methods

- Bayes' rule stems from his "Essay towards solving a problem in the doctrine of chances" (1763)
  - Difficult bit in general: estimating prior probabilities (easy in the case of naïve Bayes)
- Extension of naïve Bayes: Bayesian networks (which we'll discuss later)
- Algorithm for association rules is called APRIORI
- Minsky and Papert (1969) showed that linear classifiers have limitations, e.g. can't learn XOR
  - But: combinations of them can (→ multi-layer neural nets, which we'll discuss later)