Tree Learning

COMP9417 Machine Learning and Data Mining

Last revision: 21 Mar 2018

Material derived from slides for the book
"Machine Learning" by T. Mitchell
McGraw-Hill (1997)
http://www-2.cs.cmu.edu/~tom/mlbook.html

Material derived from slides by Andrew W. Moore
http:www.cs.cmu.edu/~awm/tutorials

Material derived from slides by Eibe Frank
http://www.cs.waikato.ac.nz/ml/weka

Material derived from slides for the book
"Machine Learning" by P. Flach
Cambridge University Press (2012)
http://cs.bris.ac.uk/~flach/mlbook

Aims

This lecture will enable you to describe decision tree learning, the use of entropy and the problem of overfitting. Following it you should be able to:

- define the decision tree representation
- list representation properties of data and models for which decision trees are appropriate
- reproduce the basic top-down algorithm for decision tree induction (TDIDT)

- define entropy in the context of learning a Boolean classifier from examples
- describe the inductive bias of the basic TDIDT algorithm

- define overfitting of a training set by a hypothesis
- describe developments of the basic TDIDT algorithm: pruning, rule generation, numerical attributes, many-valued attributes, costs, missing values
- describe regression and model trees

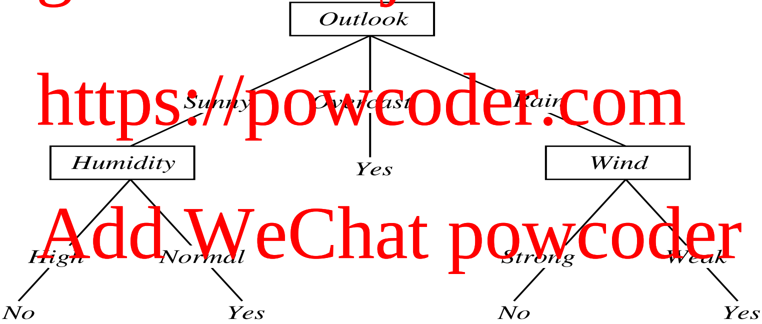
Brief History of Decision Tree Learning Algorithms

- late 1950's – Bruner et al. in psychology work on modelling *concept acquisition*
- early 1960s – Hunt et al. in computer science work on *Concept Learning Systems (CLS)*
- late 1970s – Quinlan's *Iterative Dichotomizer 3 (ID3)* based on CLS is efficient at learning on then-large data sets
- early 1990s – ID3 adds features, develops into C4.5, becomes the "default" machine learning algorithm
- late 1990s – C5.0, commercial version of C4.5 (available from SPSS and `www.rulequest.com`)
- current – widely available and applied; influential techniques

Why use decision trees?

- Decision trees are probably the single most popular data mining tool
  - Easy to understand
  - Easy to implement
  - Easy to use
  - Computationally cheap (efficient, even on big data)
- There are some drawbacks, though — e.g., high variance
- They do *classification*, i.e., predict a categorical output from categorical and/or real inputs, or *regression*

Decision Tree for $PlayTennis$



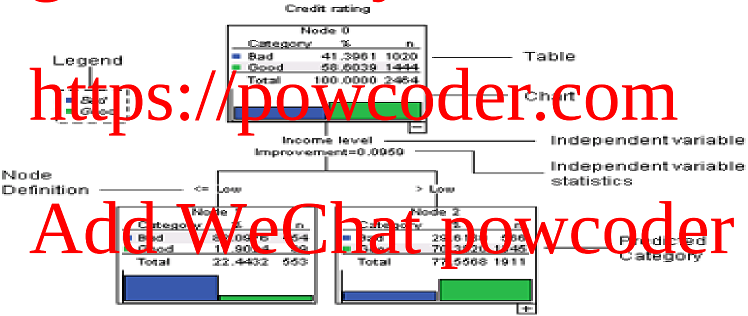
A Tree to Predict C-Section Risk

Learned from medical records of 1000 women

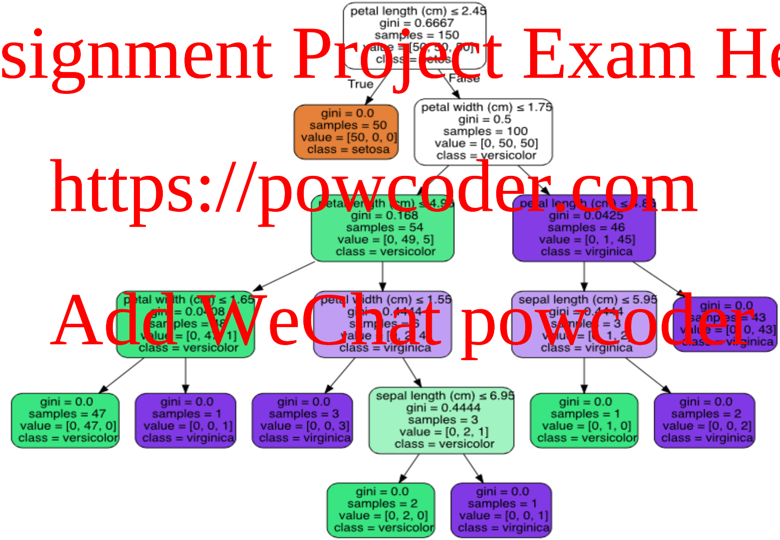
Negative examples are C-sections


```
[833+,167-] .83+ .17-
Fetal_Presentation = 1: [822+,116-] .88+ .12-
| Previous_Csection = 0: [767+,81-] .90+ .10-
| | Primiparous = 0: [399+,13-] .97+ .03-
| | Primiparous = 1: [368+,68-] .84+ .16-
| | | Fetal_Distress = 0: [334+,47-] .88+ .12-
| | | | Birth_Weight < 3349: [201+,10.6-] .95+ .05-
| | | | Birth_Weight >= 3349: [133+,36.4-] .78+ .22-
| | | Fetal_Distress = 1: [34+,21-] .62+ .38-
| Previous_Csection = 1: [55+,35-] .61+ .39-
Fetal_Presentation = 2: [3+,29-] .11+ .89-
Fetal_Presentation = 3: [8+,22-] .27+ .73-
```

Decision Tree for Credit Rating



Decision Tree for Fisher's Iris data



Decision Trees

Decision tree representation:

- Each internal node tests an attribute
- Each branch corresponds to attribute value
- Each leaf node assigns a classification

How would we represent the following expressions ?

- $\wedge, \vee,$ XOR
- $(A \wedge B) \vee (C \wedge \neg D \wedge E)$
- M of N

Decision Trees

$X \wedge Y$

```
X = t:
| Y = t: true
| Y = f: no
X = f: no
```


$X \vee Y$

```
X = t: true
X = f:
| Y = t: true
| Y = f: no
```

Decision Trees

2 of 3

```
X = t:
| Y = t: true
| Y = f:
| | Z = t: true
| | Z = f: false
X = f:
| Y = t:
| | Z = t: true
| | Z = f: false
| Y = f: false
```


So in general decision trees represent a *disjunction of conjunctions* of constraints on the attributes values of instances.

When are Decision Trees the Right Model?

- With Boolean values for the instances $\mathbf{X}$ and class Y, the representation adopted by decision-trees allows us to represent Y as a Boolean function of the $\mathbf{X}$

- Given d input Boolean variables, there are 2^d possible input values for these variables. Any specific function assigns $Y = 1$ to some subset of these, and $Y = 0$ to the rest

- Any Boolean function can be trivially represented by a tree. Each function assigns $Y = 1$ to some subset of the 2^d possible values of $\mathbf{X}$. So, for each combination of values with $Y = 1$, have a path from root to a leaf with $Y = 1$. All other leaves have $Y = 0$

When are Decision Trees the Right Model?

- This is nothing but a re-representation of the truth-table, and will have 2^d leaves. More compact trees may be possible, by taking into account what is common between one or more rows with the same Y value

- But, even for Boolean functions, there are some functions for which compact trees may not be possible (the parity and majority functions are examples)

- In general, although possible in principle to express any Boolean function, our search and prior restrictions may not allow us to find the correct tree in practice.
- BUT: If you want readable models that combine logical tests with a probability-based decision, then decision trees are a good start

When to Consider Decision Trees?

- Instances described by a mix of numeric features and discrete attribute–value pairs
- Target function is discrete valued (otherwise use regression trees)
- Disjunctive hypothesis may be required
- Possibly noisy training data
- Interpretability is an advantage

Examples are extremely numerous, including:

- Equipment or medical diagnosis
- Credit risk analysis
- Modeling calendar scheduling preferences
- etc.

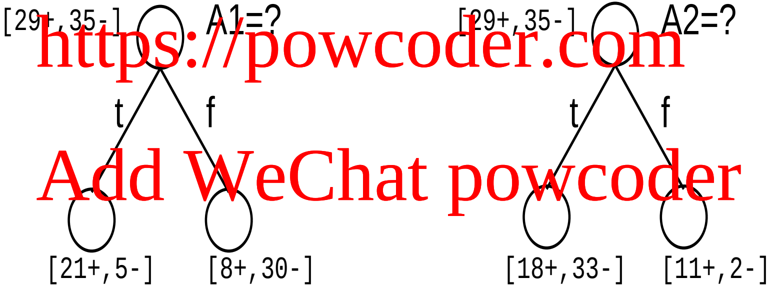
Top-Down Induction of Decision Trees (TDIDT)

Main loop:

- $A \leftarrow$ the "best" decision attribute for next $node$
- Assign A as decision attribute for $node$
- For each value of A, create new descendant of $node$
- Sort training examples to leaf nodes
- If training examples perfectly classified, Then STOP, Else iterate over new leaf nodes

Essentially this is the "ID3" algorithm (Quinlan, 1986) — the first efficient symbolic Machine Learning algorithm.

Which attribute is best?



Bits

You are watching a set of independent random samples of X

You observe that X has four possible values

$P(X = A) = \frac{1}{4}$	$P(X = B) = \frac{1}{4}$	$P(X = C) = \frac{1}{4}$	$P(X = D) = \frac{1}{4}$

So you might see: BAACBADCDADDDA...

You transmit data over a binary serial link. You can *encode* each reading with two bits (e.g. A = 00, B = 01, C = 10, D = 11)

0100001001001110110011111100...

Fewer Bits

Someone tells you that the probabilities are not equal

$P(X = A) = \frac{1}{2}$	$P(X = B) = \frac{1}{4}$	$P(X = C) = \frac{1}{8}$	$P(X = D) = \frac{1}{8}$

It's possible . . .

. . . to invent a *coding* for your transmission that only uses 1.75 bits on average per symbol. How ?

Fewer Bits

Someone tells you that the probabilities are not equal

$P(X = A) = \frac{1}{2}$	$P(X = B) = \frac{1}{4}$	$P(X = C) = \frac{1}{8}$	$P(X = D) = \frac{1}{8}$

It's possible ...

... to invent a *coding* for your transmission that only uses 1.75 bits per symbol on average. How ?

A	0
B	10
C	110
D	111

(This is just one of several ways)

Fewer Bits

Suppose there are three equally likely values

$P(X = A) = \frac{1}{3}$	$P(X = B) = \frac{1}{3}$	$P(X = C) = \frac{1}{3}$

Here's a naive coding, costing 2 bits per symbol

A	00
B	01
C	10

Can you think of a coding that would need only 1.6 bits per symbol on average ?

Fewer Bits

Suppose there are three equally likely values:

$P(X = A) = \frac{1}{3}$	$P(X = B) = \frac{1}{3}$	$P(X = C) = \frac{1}{3}$

Using the same approach as before, we can get a coding costing 1.6 bits per symbol on average . . .

A	0
B	10
C	11

This gives us, on average $\frac{1}{3} \times 1$ bit for A and $2 \times \frac{1}{3} \times 2$ bits for B and C, which equals $\frac{5}{3} \approx 1.6$ bits.

Is this the best we can do ?

Fewer Bits

Suppose there are three equally likely values

$P(X = A) = \frac{1}{3}$	$P(X = B) = \frac{1}{3}$	$P(X = C) = \frac{1}{3}$

From information theory, the optimal number of bits to encode a symbol with probability p is $-\log_2\ p\ \ldots$

So the best we can do for this case is $-\log_2 \frac{1}{3}$ bits for each of A, B and C, or 1.5849625007211563 bits per symbol

General Case

Suppose X can have one of m values ... $V_1, V_2, \ldots V_m$

$P(X = V_1) = p_1$	$P(X = V_2) = p_2$	$\ldots$	$P(X = V_m) = p_m$

What's the smallest possible number of bits, on average, per symbol, needed to transmit a stream of symbols drawn from X's distribution ? It's

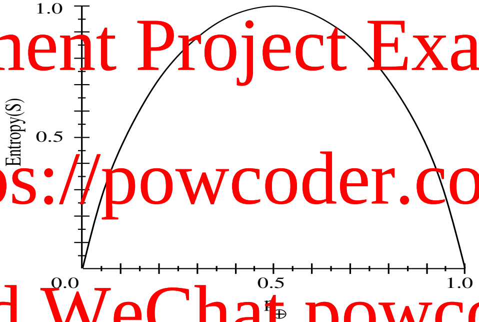
$$\begin{aligned} H(X) &= -p_1 \log_2 p_1 - p_2 \log_2 p_2 - \ldots - p_m \log_2 p_m \\ &= -\sum_{j=1}^{m} p_j \log_2 p_j \end{aligned}$$

$H(X) =$ the *entropy* of X

General Case

"High entropy" means X is very uniform and boring
"Low entropy" means X is very varied and interesting

Entropy



Where:

S is a sample of training examples

$p_\oplus$ is the proportion of positive examples in S

$p_\ominus$ is the proportion of negative examples in S

Entropy

Entropy measures the "impurity" of S

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

A "pure" sample is one in which all examples are of the same class.

Entropy

$Entropy(S)$ = expected number of bits needed to encode class ($\oplus$ or $\ominus$) of randomly drawn member of S (under the optimal, shortest-length code)

Why ?

Information theory: optimal length code assigns $-\log_2 p$ bits to message having probability p.

So, expected number of bits to encode $\oplus$ or $\ominus$ of random member of S:

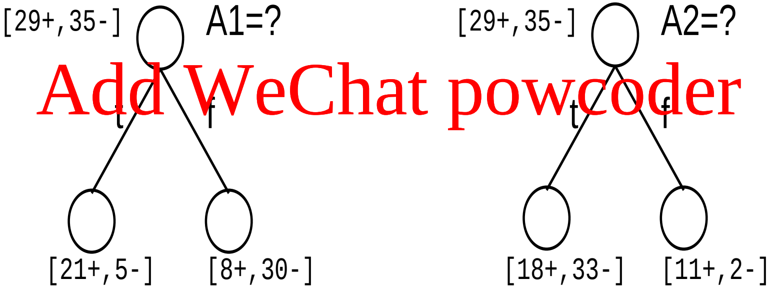
$$p_{\oplus}(-\log_2 p_{\oplus}) + p_{\ominus}(-\log_2 p_{\ominus})$$

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

Information Gain

- $Gain(S, A)$ = expected reduction in entropy due to sorting on A

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$



Information Gain

$$
\begin{aligned}
Gain(S, A1) &= Entropy(S) - \left(\frac{|S_t|}{|S|}Entropy(S_t) + \frac{|S_f|}{|S|}Entropy(S_f)\right) \\
&= 0.9936 - \\
&= ((\frac{26}{64}(-\frac{21}{26}\log_2(\frac{21}{26}) - \frac{5}{26}\log_2(\frac{5}{26}))) + \\
&\quad (\frac{38}{64}(-\frac{8}{38}\log_2(\frac{8}{38}) - \frac{30}{38}\log_2(\frac{30}{38})))) \\
&= 0.9936 - (\ 0.2869 + 0.4408\ ) \\
&= 0.2658
\end{aligned}
$$

Information Gain

$$\begin{aligned} Gain(S, A2) &= 0.9936 - (\ 0.7464 + 0.0828\ ) \\ &= 0.1643 \end{aligned}$$

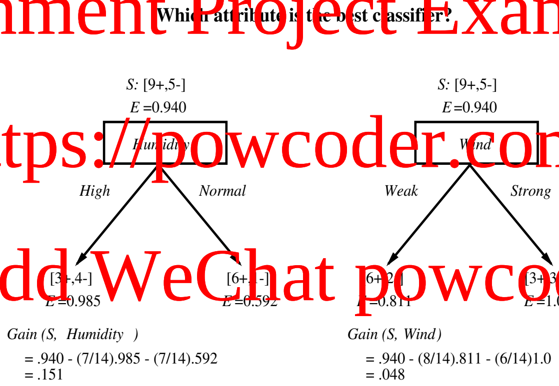
Information Gain

So we choose A1, since it gives a larger expected reduction in entropy.

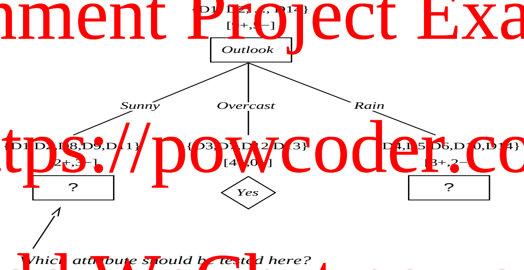
Training Examples

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Information gain once more



Information gain once more



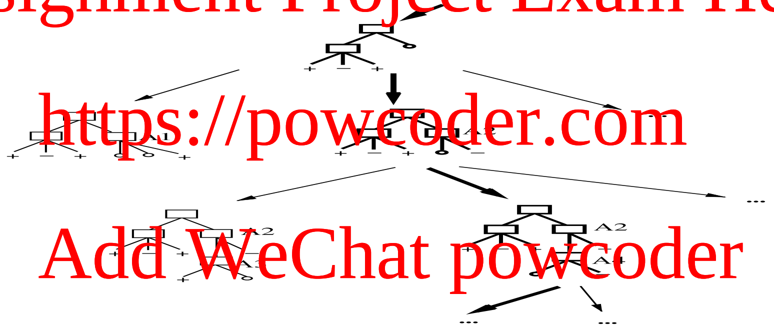
$S_{sunny} = \{D1,D2,D8,D9,D11\}$

$Gain\ (S_{sunny}\ , Humidity) = .970 - (3/5)\ 0.0 - (2/5)\ 0.0 = .970$

$Gain\ (S_{sunny}\ , Temperature) = .970 - (2/5)\ 0.0 - (2/5)\ 1.0 - (1/5)\ 0.0 = .570$

$Gain\ (S_{sunny}\ , Wind) = .970 - (2/5)\ 1.0 - (3/5)\ .918 = .019$

Hypothesis Space Search by ID3



Hypothesis Space Search by ID3

- This can be viewed as a graph-search problem
- Each vertex in the graph is a decision tree
- Suppose we only consider the two-class case ($\omega = \omega_1$ or ω_2), and all the features x_i are Boolean, so each vertex is a binary tree
- A pair of vertices in the graph have an edge if the corresponding trees differ in just the following way: one of the leaf-nodes in one vertex has been replaced by a non-leaf node testing a feature that has not appeared earlier (and 2 leaves)
- This is the full space of all decision trees (is it?). We want to search for a single tree or a small number of trees in this space. How should we do this?

Hypothesis Space Search by ID3

- Usual graph-search technique: greedy or beam search, starting with the vertex corresponding to the “empty tree” (single leaf node)
- Greedy choice: which one to select? The neighbour that results in the greatest increase in $P(D|T)$
  - How?
  - Suppose T is changed to T'. Simply use the ratio of $P(D|T')/P(D|T)$
  - Most of the calculation will cancel out: so, we will only need to do the local computation at the leaf that was converted into a non-leaf node
- RESULT: set of trees with (reasonably) high posterior probabilities given D: we can now use these to answer questions like $P(y' = \omega_1 | \ldots)$? or even make a *decision* or a *classification* that $y' = \omega_1$, given input data $\mathbf{x}$

Hypothesis Space Search by ID3

- Hypothesis space is complete! (contains all finite discrete-valued functions w.r.t attributes)
    - Target function surely in there...
- Outputs a single hypothesis (which one?)
    - Can't play 20 questions...
- No back tracking
    - Local minima...
- Statistically-based search choices
    - Robust to noisy data...
- Inductive bias: approx "prefer shortest tree"

Inductive Bias in ID3

Note H is the power set of instances X

$\rightarrow$Unbiased?

Not really...

- Preference for short trees, and for those with high information gain attributes near the root
- Bias is a *preference* for some hypotheses, rather than a *restriction* of hypothesis space H

- an incomplete search of a complete hypothesis space *versus* a complete search of an incomplete hypothesis space (as in learning conjunctive concepts)
- Occam's razor: prefer the shortest hypothesis that fits the data

Occam's Razor

William of Ockham (c. 1287-1347)

Entities should not be multiplied beyond necessity

Why prefer short hypotheses?

Argument in favour:

- Fewer short hypotheses than long hypotheses

$\rightarrow$ a short hyp that fits data unlikely to be coincidence

$\rightarrow$ a long hyp that fits data might be coincidence

Occam's Razor

Argument opposed:

- There are many ways to define small sets of hypotheses
  - e.g., all trees with a prime number of nodes that use attributes beginning with 'Z'

- What's so special about small sets based on *size* of hypothesis??

Look back to linear classification lecture to see how to make this work using Minimum Description Length (MDL)

Why does overfitting occur?

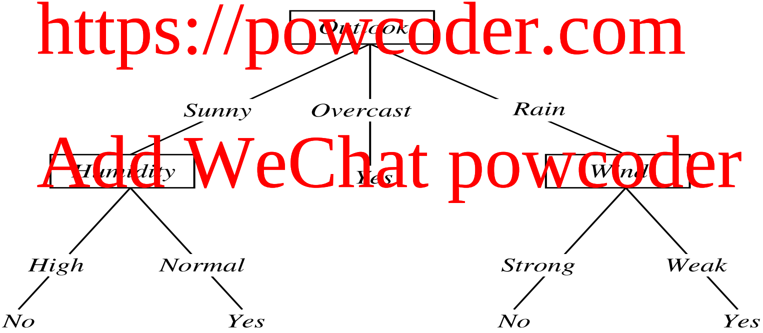
- Greedy search can make mistakes. We know that it can end up in local minima — so a sub-optimal choice earlier might result in a better solution later (*i.e.* pick a test whose posterior gain (or information gain) is less than the best one
- But there is also another kind of problem. We know that training error is an optimistic estimate of the true error of the model, and that this optimism increases as the training error decreases
    - We will see why this is the case later (lectures on Evaluation)
    - Suppose we have two models h_1 and h_2 with training errors e_1 and e_2 and optimism o_1 and o_2. Let the true error of each be $E_1 = e_1 + o_1$ and $E_2 = e_2 + o_2$
    - If $e_1 < e_2$ and $E_1 > E_2$, then we will say that h_1 has overfit then training data
- So, a search method based purely on training data estimates may end overfitting the training data

Overfitting in Decision Tree Learning

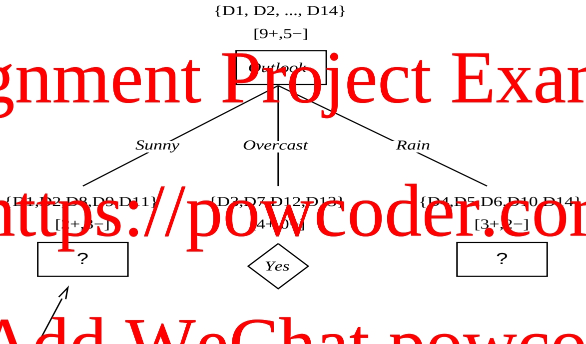
Consider adding noisy training example #15:

$$Sunny, Hot, Normal, Strong, PlayTennis = No$$

What effect on earlier tree?



Overfitting in Decision Tree Learning



Which attribute should be tested here?

S_{sunny} = {D1,D2,D8,D9,D11}

$Gain\ (S_{sunny}, Humidity) = .970 - (3/5)\ 0.0 - (2/5)\ 0.0 = .970$

$Gain\ (S_{sunny}, Temperature) = .970 - (2/5)\ 0.0 - (2/5)\ 1.0 - (1/5)\ 0.0 = .570$

$Gain\ (S_{sunny}, Wind) = .970 - (2/5)\ 1.0 - (3/5)\ .918 = .019$

Overfitting in General

Consider error of hypothesis h over

- training data: $error_{train}(h)$
- entire distribution $\mathcal{D}$ of data: $error_{\mathcal{D}}(h)$

Definition

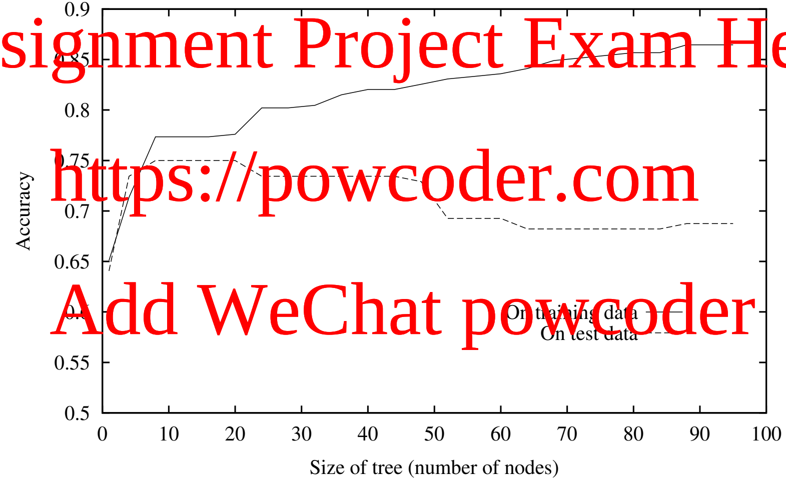
Hypothesis $h \in H$ **overfits** training data if there is an alternative hypothesis $h' \in H$ such that

$$error_{train}(h) < error_{train}(h')$$

and

$$error_{\mathcal{D}}(h) > error_{\mathcal{D}}(h')$$

Overfitting in Decision Tree Learning



Avoiding Overfitting

How can we avoid overfitting? **Pruning**

- **pre-pruning** stop growing when data split not statistically significant
- **post-pruning** grow full tree, then remove sub-trees which are overfitting

Post-pruning avoids problem of "early stopping"

How to select "best" tree:

- Measure performance over training data ?

- Measure performance over separate validation data set ?
- MDL: minimize $size(tree) + size(misclassifications(tree))$?

Avoiding Overfitting

Pre-pruning

- Can be based on statistical significance test
- Stops growing the tree when there is no statistically significant association between any attribute and the class at a particular node
- For example, in ID3: chi-squared test plus information gain
  - only statistically significant attributes were allowed to be selected by information gain procedure

Avoiding Overfitting

Pre-pruning

- Simplest approach: stop growing the tree when fewer than some lower-bound on the number of examples at a leaf

- In `C4.5`, this parameter is the `m` parameter
- In `sklearn`, this parameter is `min_samples_leaf`
- In `sklearn`, the parameter `min_impurity_decrease` enables stopping when the this falls below a lower-bound

Avoiding Overfitting

Early stopping

- Pre-pruning may suffer from early stopping: may stop the growth of tree prematurely
- Classic example: XOR/Parity-problem
  - No individual attribute exhibits a significant association with the class
  - Target structure only visible in fully expanded tree
  - Prepruning won't expand the root node
- But: XOR-type problems not common in practice
- And: pre-pruning faster than post-pruning

Avoiding Overfitting

Post-pruning

- Builds full tree first and prunes it afterwards
  - Attribute interactions are visible in fully-grown tree
- Problem: identification of subtrees and nodes that are due to chance effects

- Two main pruning operations:
  - Subtree replacement
  - Subtree raising

- Possible strategies: error estimation, significance testing, MDL principle
- We examine two methods: Reduced-error Pruning and Error-based Pruning

Reduced-Error Pruning

Split data into *training* and *validation* set

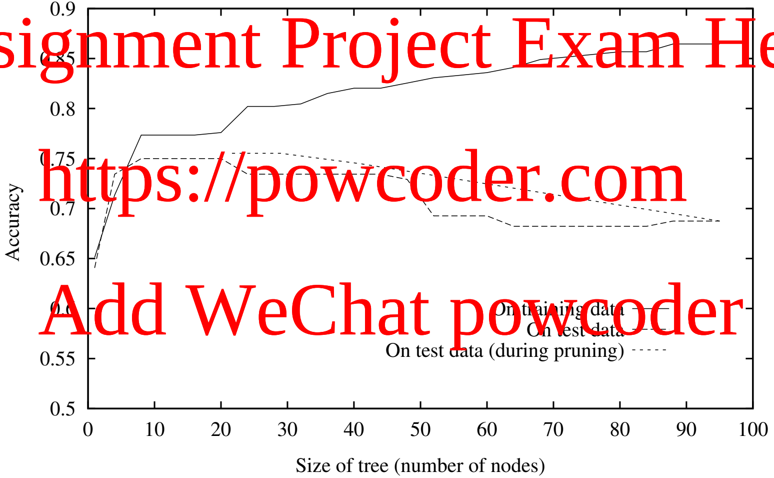
Do until further pruning is harmful:

- Evaluate impact on *validation* set of pruning each possible node (plus those below it)
- Greedily remove the one that most improves *validation* set accuracy

Reduced-Error Pruning

- **Good** produces smallest version of most accurate subtree
- **Not so good** reduces effective size of training set

Effect of Reduced-Error Pruning



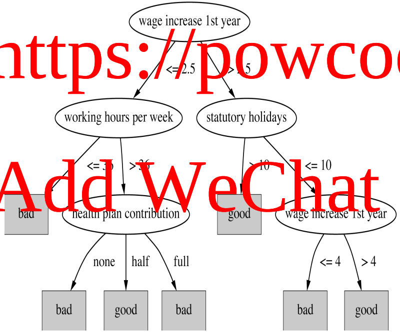
Error-based pruning (C4.5 / J48 / C5.0)

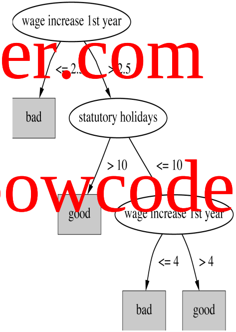
Quinlan (1993) describes the successor to ID3 – C4.5

- many extensions – see below
- post-pruning using training set
- includes sub-tree replacement and sub-tree raising
- also: pruning by converting tree to rules
- commercial version – C5.0 – is widely used
  - RuleQuest.com
  - now free
- Weka version – J48 – also widely used

Pruning operator: Sub-tree replacement

Bottom-up:
tree is considered for replacement once all its sub-trees have been considered





Error-based pruning: error estimate

Goal is to improve estimate of error on unseen data using all and only data from training set

But how can this work ?

Make the estimate of error **pessimistic** !

Error-based pruning: error estimate

- Apply pruning operation if this does not increase the estimated error
- C4.5's method: using upper limit of standard confidence interval derived from the training data
  - Standard Bernoulli-process-based method
  - **Note**: statistically motivated, but not statistically valid
  - **However**: works well in practice !

Error-based pruning: error estimate

- The error estimate for a tree node is the weighted sum of error estimates for all its subtrees (possibly leaves).
- Upper bound error estimate e for a node (simplified version):

$$e = f + Z_c \cdot \sqrt{\frac{f \cdot (1 - f)}{N}}$$

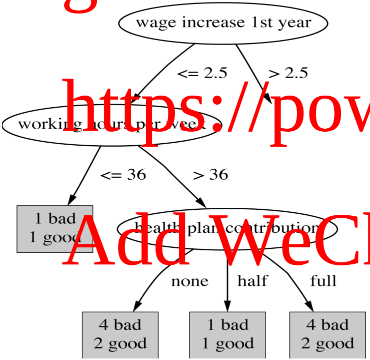
- f is actual (empirical) error of tree on examples at the tree node
- N is the number of examples at the tree node

- Z_c is a constant whose value depends on *confidence* parameter c
- C4.5's default value for confidence $c = 0.25$
- If $c = 0.25$ then $Z_c = 0.69$ (from standardized normal distribution)

Error-based pruning: error estimate

- How does this method implement a pessimistic error estimate ?
- What effect will the c parameter have on pruning ?
- As $c \uparrow$, $z \downarrow$
- See example on next slide (note: values not calculated using exactly the above formula)

Error-based pruning: error estimate



- health plan contribution: node measures $f = 0.36$, $e = 0.46$
- sub-tree measures:
  - none: $f = 0.33$, $e = 0.47$
  - half: $f = 0.5$, $e = 0.72$
  - full: $f = 0.33$, $e = 0.47$
- sub-trees combined $6 : 2 : 6$ gives 0.51
- sub-trees estimated to give *greater* error so prune away

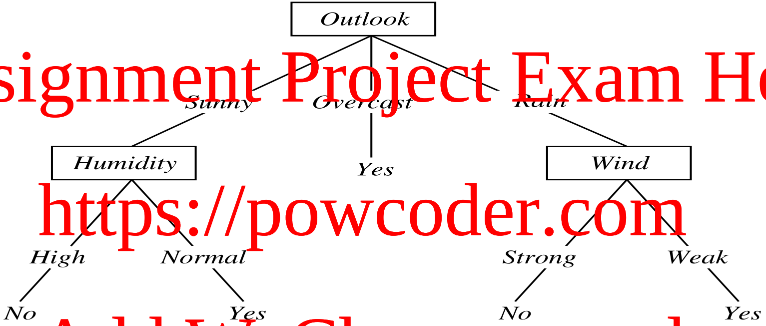
Rule Post-Pruning

This method was introduced in Quinlan's C4.5

- Convert tree to equivalent set of rules
- Prune each rule independently of others
- Sort final rules into desired sequence for use

For: simpler classifiers, people prefer rules to trees
Against: does not scale well, slow for large trees & datasets

Converting A Tree to Rules



IF $(Outlook = Sunny) \wedge (Humidity = High)$
THEN $PlayTennis = No$

IF $(Outlook = Sunny) \wedge (Humidity = Normal)$
THEN $PlayTennis = Yes$

...

Rules from Trees (Rule Post-Pruning)

Rules can be simpler than trees but just as accurate, e.g., in C4.5Rules:

- path from root to leaf in (unpruned) tree forms a *rule*
  - i.e., tree forms a *set of rules*

- can simplify rules independently by deleting conditions
  - i.e., rules can be generalized while maintaining accuracy
- greedy rule simplification algorithm
  - drop the condition giving lowest estimated error (as for pruning)
  - continue while estimated error does not increase

Rules from Trees

Select a 'good' subset of rules within a class (C4.5Rules):

- goal: remove rules not useful in terms of accuracy
- find a subset of rules which minimises an *MDL* criterion
- trade-off accuracy and complexity of rule-set
- stochastic search using simulated annealing

Sets of rules can be ordered by class (C4.5Rules):

- order classes by increasing chance of making false *positive* errors
- set as a default the class with the most training instances not covered by any rule

Continuous Valued Attributes

Decision trees originated for **discrete** attributes only. Now: continuous attributes.

Can create a discrete attribute to test continuous value:

- $Temperature = 82.5$
- $(Temperature > 72.3) \in \{t, f\}$

- Usual method: continuous attributes have a binary split
- Note:

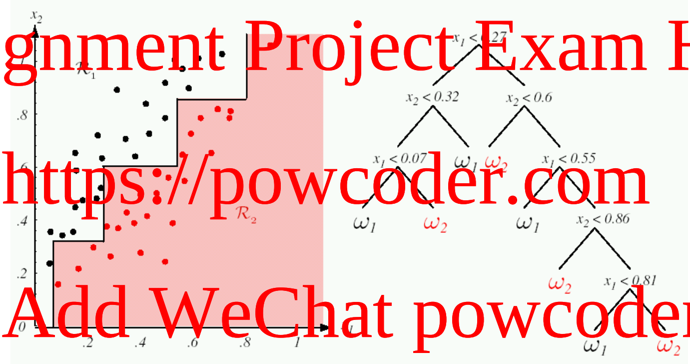
  - discrete attributes – one split exhausts all values
  - continuous attributes – can have many splits in a tree

Continuous Valued Attributes

- Splits evaluated on all possible split points
- More computation: $n-1$ possible splits for n values of an attribute in training set
- Fayyad (1991)
  - sort examples on continuous attribute
  - find midway boundaries where class changes, e.g. for *Temperature* $\frac{(48+60)}{2}$ and $\frac{(80+90)}{2}$
- Choose best split point by info gain (or evaluation of choice)
- Note: C4.5 uses actual values in data

Temperature:	40	48	60	72	80	90
PlayTennis:	No	No	Yes	Yes	Yes	No

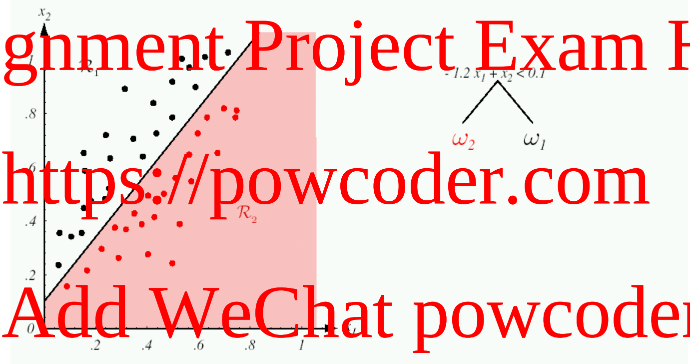
Axis-parallel Splitting



Fitting data that is not a good "match" to the possible splits in a tree.

"Pattern Classification" Duda, Hart, and Stork, (2001)

Splitting on Linear Combinations of Features



Reduced tree size by allowing splits that are a better "match" to the data.

"Pattern Classification" Duda, Hart, and Stork, (2001)

Attributes with Many Values

Problem:

- If attribute has many values, $Gain$ will select it
- Why ? more likely to split instances into "pure" subsets
  - Maximised by singleton subsets
- Imagine using *Date* = March 21, 2018 as attribute
- High gain on training set, useless for prediction

Attributes with Many Values

One approach: use $GainRatio$ instead

$$GainRatio(S, A) \equiv \frac{Gain(S, A)}{SplitInformation(S, A)}$$

$$SplitInformation(S, A) \equiv -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

where S_i is subset of S for which A has value v_i

Attributes with Many Values

Why does this help ?

- sensitive to how broadly and uniformly attribute splits instances
- actually the entropy of S w.r.t. values of A
  - i.e., the information of the partition itself
- therefore higher for many-valued attributes, especially if mostly uniformly distributed across possible values

Attributes with Costs

Consider

- medical diagnosis, $BloodTest$ has cost \$150
- robotics, $Width\_from\_1ft$ has cost 23 sec.

How to learn a consistent tree with low expected cost?

Attributes with Costs

One approach: evaluate information gain *relative to* cost:

- Example

$$\frac{Gain^2(S, A)}{Cost(A)}.$$

Preference for decision trees using lower-cost attributes.

Attributes with Costs

Also: class (misclassification) costs, instance costs, ...

SEE5 / C5.0 can use a misclassification cost matrix.

Can give *false positives* a different cost to *false negatives*

Forces a different tree structure to be learned to minimise *asymmetric* misclassification costs – can help if class distribution is skewed – why ?

Unknown Attribute Values

What if some examples missing values of A?

Use training example anyway, sort through tree. Here are 3 possible approaches

- If node n tests A, assign most common value of A among other examples sorted to node n
- assign most common value of A among other examples with same target value
- assign probability p_i to each possible value v_i of A
  - assign fraction p_i of example to each descendant in tree

Note: need to classify new (unseen) examples in same fashion

Windowing

Early implementations – training sets too large for memory

As a solution ID3 implemented *windowing*:

1. select subset of instances – the *window*
2. construct decision tree from all instances in the window
3. use tree to classify training instances *not* in window
4. if all instances correctly classified then halt, else
5. add selected misclassified instances to the window
6. go to step 2

Windowing retained in C4.5 because it can lead to *more accurate* trees. Related to *ensemble learning*.

Non-linear Regression with Trees

Despite some nice properties of Neural Networks, such as generalization to deal sensibly with unseen input patterns and robustness to losing neurons (prediction performance can degrade gracefully), they still have some problems:

- Back-propagation is often difficult to scale – large nets need lots of computing time; may have to be partitioned into separate modules that can be trained independently, e.g. NetTalk, DeepBind
- Neural Networks are not very *transparent* – hard to understand the representation of what has been learned

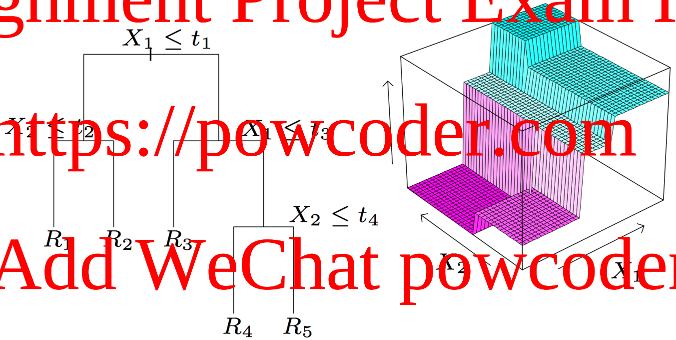
Possible solution: exploit success of tree-structured approaches in ML

Regression trees

- Differences to decision trees:
  - Splitting criterion: minimizing intra-subset variation
  - Pruning criterion: based on numeric error measure
  - Leaf node predicts average class values of training instances reaching that node

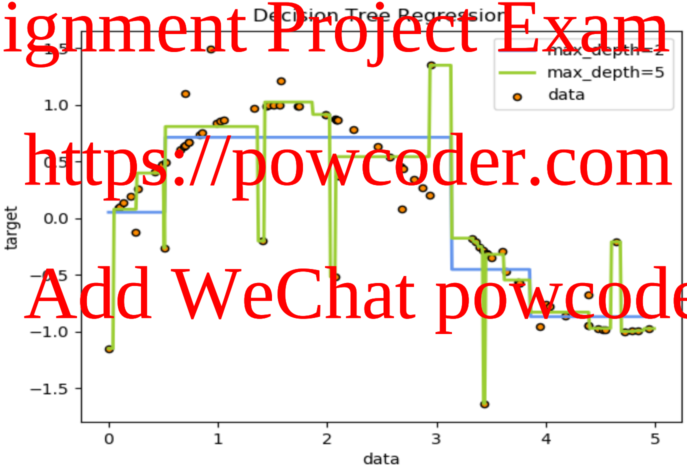
- Can approximate piecewise constant functions
- Easy to interpret
- More sophisticated version: model trees

A Regression Tree and its Prediction Surface

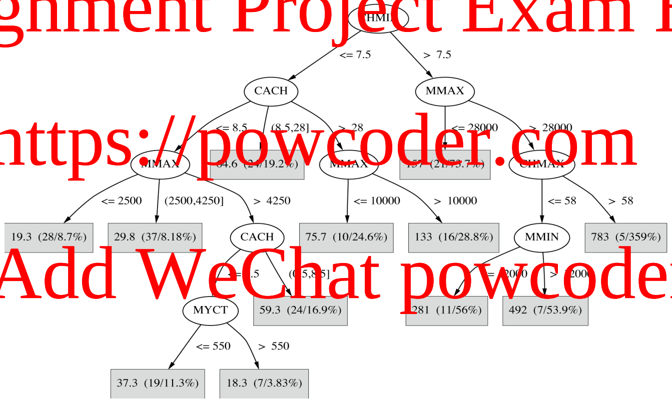


"Elements of Statistical Learning" Hastie, Tibshirani & Friedman (2001)

Regression Tree on sine dataset



Regression Tree on CPU dataset



Tree learning as variance reduction

- The variance of a Boolean (i.e., Bernoulli) variable with success probability $\dot{p}$ is $\dot{p}(1-\dot{p})$, which is half the Gini index. So we could interpret the goal of tree learning as minimising the class variance (or standard deviation, in case of $\sqrt{\text{Gini}}$) in the leaves.
- In regression problems we can define the variance in the usual way:

$$\mathrm{Var}(Y) = \frac{1}{|Y|}\sum_{y \in Y}(y - \overline{y})^2$$

If a split partitions the set of target values Y into mutually exclusive sets $\{Y_1, \ldots, Y_l\}$, the weighted average variance is then

$$\mathrm{Var}(\{Y_1, \ldots, Y_l\}) = \sum_{j=1}^{l}\frac{|Y_j|}{|Y|}\mathrm{Var}(Y_j) = \ldots = \frac{1}{|Y|}\sum_{y \in Y} y^2 - \sum_{j=1}^{l}\frac{|Y_j|}{|Y|}\overline{y}_j^2$$

The first term is constant for a given set Y and so we want to maximise the weighted average of squared means in the children.

Learning a regression tree

Imagine you are a collector of vintage Hammond tonewheel organs. You have been monitoring an online auction site, from which you collected some data about interesting transactions:

#	Model	Condition	Leslie	Price
1.	B3	excellent	no	4513
2.	T202	fair	yes	625
3.	A100	good	no	1051
4.	T202	good	no	270
5.	M102	good	yes	870
6.	A100	excellent	no	1770
7.	T202	fair	no	99
8.	A100	good	yes	1900
9.	E112	fair	no	77

Learning a regression tree

From this data, you want to construct a regression tree that will help you determine a reasonable price for your next purchase.

There are three features, hence three possible splits:

Model = [A100, B3, E112, M102, T202]

$$[1051, 1770, 1900][4513][77][870][99, 270, 625]$$

Condition = [excellent, good, fair]

$$[1770, 4513][270, 870, 1051, 1900][77, 99, 625]$$

Leslie = [yes, no] $[625, 870, 1900][77, 99, 270, 1051, 1770, 4513]$

The means of the first split are 1574, 4513, 77, 870 and 331, and the weighted average of squared means is $3.21 \cdot 10^6$. The means of the second split are 3142, 1023 and 267, with weighted average of squared means $2.68 \cdot 10^6$; for the third split the means are 1132 and 1297, with weighted average of squared means $1.55 \cdot 10^6$. We therefore branch on Model at the top level. This gives us three single-instance leaves, as well as three A100s and three T202s.

Learning a regression tree

For the A100s we obtain the following splits:

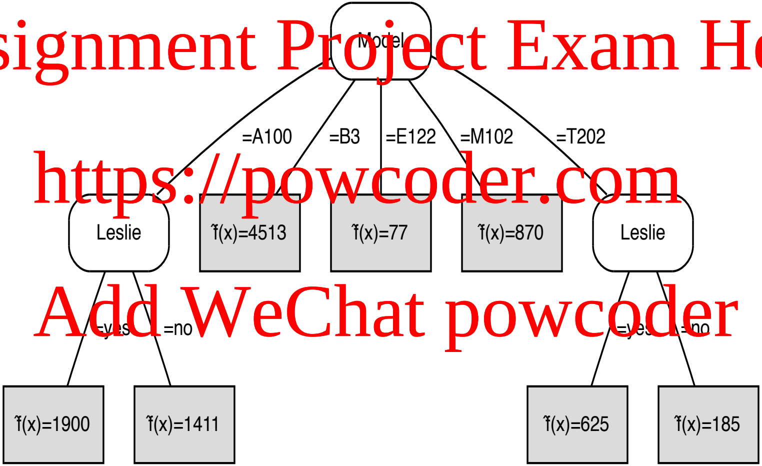
Condition = [excellent, good, fair] $[1770][1051, 1900][]$
Leslie = [yes, no] $[1900][1051, 1770]$

Without going through the calculations we can see that the second split results in less variance (to handle the empty child, it is customary to set its variance equal to that of the parent). For the T202s the splits are as follows:

Condition = [excellent, good, fair] $[][270][99, 625]$
Leslie = [yes, no] $[625][99, 270]$

Again we see that splitting on Leslie gives tighter clusters of values. The learned regression tree is depicted on the next slide.

A regression tree



A regression tree learned from the Hammond organ dataset.

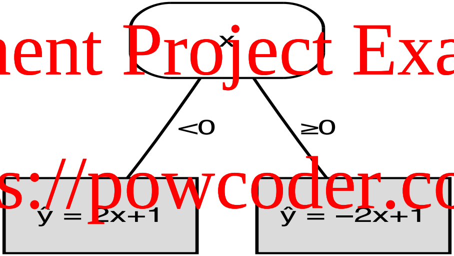
Model trees

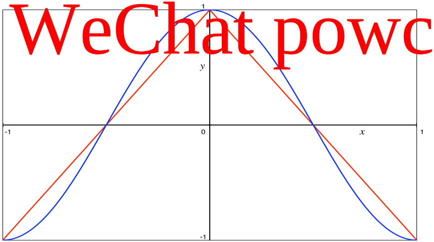
- Like regression trees but with linear regression functions at each node
- Linear regression applied to instances that reach a node after full tree has been built
- Only a subset of the attributes is used for LR
  - Attributes occurring in subtree (+maybe attributes occurring in path to the root)
- Fast: overhead for Linear Regression (LR) not large because usually only a small subset of attributes is used in tree

Two uses of features

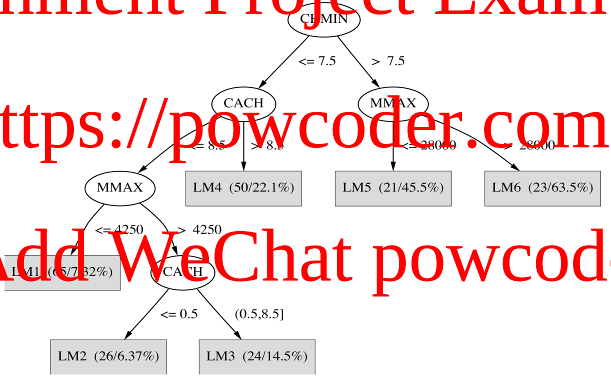
Suppose we want to approximate $y = \cos \pi x$ on the interval $-1 \leq x \leq 1$. A linear approximation is not much use here since the best fit would be $y = 0$. However, if we split the x axis in two intervals $-1 \leq x < 0$ and $0 \leq x \leq 1$, we could find reasonable linear approximations on each interval. We can achieve this by using x both as a splitting feature and as a regression variable (next slide).

A small model tree





Model Tree on CPU dataset



Smoothing

- Naïve prediction method – output value of LR model at corresponding leaf node
- Improve performance by *smoothing* predictions with *internal* LR models
  - Predicted value is weighted average of LR models along path from root to leaf

- Smoothing formula: $p' = \frac{np+kq}{n+k}$ where
  - p' prediction passed up to next higher node
  - p prediction passed to this node from below
  - q value predicted by model at this node
  - n number of instances that reach node below
  - k smoothing constant

- Same effect can be achieved by incorporating the internal models into the leaf nodes

Building the tree

- Splitting criterion: *standard deviation reduction*

$$SDR = sd(T) - \sum_i \frac{|T_i|}{|T|} \times sd(T_i)$$

where $T_1, T_2, \ldots$ are the sets from splits of data at node.

- Termination criteria (important when building trees for numeric prediction):

  - Standard deviation becomes smaller than certain fraction of sd for full training set (e.g. 5%)
  - Too few instances remain (e.g. less than four)

Pruning the tree

- Pruning is based on estimated absolute error of LR models
- Heuristic estimate:

$$\frac{n+v}{n-v} \times \text{average\_absolute\_error}$$

where n is number of training instances that reach the node, and v is the number of parameters in the linear model

- LR models are pruned by greedily removing terms to minimize the estimated error
- Model trees allow for heavy pruning: often a single LR model can replace a whole subtree
- Pruning proceeds bottom up: error for LR model at internal node is compared to error for subtree

Discrete (nominal) attributes

- Nominal attributes converted to binary attributes and treated as numeric
  - Nominal values sorted using average class value for each one
  - For k values, $k-1$ binary attributes are generated
    - the ith binary attribute is 0 if an instance's value is one of the first i in the ordering, 1 otherwise
- Best binary split with original attribute provably equivalent to a split on one of the new attributes

Summary – decision trees

- Decision tree learning is a practical method for many classifier learning tasks – still a "Top 10" data mining algorithm – see `sklearn.tree.DecisionTreeClassifier`
- TDIDT family descended from ID3 searches complete hypothesis space - the hypothesis is there, somewhere...
- Uses a search or *preference* bias, search for optimal tree is, in general, not tractable
- Overfitting is inevitable with an expressive hypothesis space and noisy data, so pruning is important
- Decades of research into extensions and refinements of the general approach, e.g., for numerical prediction, logical trees
- Often the "try-first" machine learning method in applications, illustrates many general issues
- Performance can be improved with use of "ensemble" methods

Summary – regression and model trees

- Regression trees were introduced in CART — R's implementation is close to CART, but see `sklearn.tree.DecisionTreeRegressor` for a basic version
- Quinlan proposed the M5 model tree inducer
- M5′: slightly improved version that is publicly available (M5Pin Weka is based on this)
- Quinlan also investigated combining instance-based learning with M5
- CUBIST: Quinlan's rule learner for numeric prediction `www.rulequest.com`
- Interesting comparison: Neural nets vs. model trees — both do *non-linear regression*
- other methods also can learn non-linear models