Tree Learning

A story on uncertainty in machine learning

Based also on the book by Peter Flach, University of Bristol and
Ray Mooney, University of Texas, Lecture

Machine learning settings

	Predictive model	*Descriptive model*
Supervised learning	classification, regression	subgroup discovery
Unsupervised learning	predictive clustering	descriptive clustering, association rule discovery

Table 1.1. An overview of different machine learning settings. The rows refer to whether the training data is labelled with a target variable, while the columns indicate whether the models learned are used to predict a target variable or rather describe the given data.

Machine learning settings

- Logical approach - Decision and regression trees, rules
- Probabilistic methods – Bayesian methods
- Linear methods – Linear discriminant, SVM, perceptron, logistic regression
- Distance-based methods – Lazy Learning (kNN), clustering

Machine learning settings

- Logical approach - Decision and regression trees, rules
- Probabilistic methods – Bayesian methods
- Linear methods – Linear discriminant, SVM, perceptron, logistic regression
- Distance-based methods – Lazy Learning (kNN), clustering

Learning Trees

- Supervised method – data classified into classes, i.e. data contains a target attribute
- Classifier is a tree that represents a hypotheses in a disjunctive normal form

- Finite number of classes >= 2 (for a decision tree), continuous (for regression trees)

- Finding a minimal decision tree (nodes, leaves, or depth) is an NP-hard optimization problem. Heuristic algorithm can be used to build a tree
- Want to pick a feature that creates subsets of examples that are relatively "pure" in a single class so they are "closer" to being leaf nodes.

Tree Induction Pseudocode

DTree(*examples*, *features*) returns a tree
  If all *examples* are in one category, return a leaf node with that category label.
  Else if the set of *features* is empty, return a leaf node with the category label that
      is the most common in examples.
  Else pick a feature F and create a node R for it
    For each possible value v_i of F:
      Let $examples_i$ be the subset of examples that have value v_i for F
      Add an out-going edge E to node R labeled with the value v_i.
      If $examples_i$ is empty
        then attach a leaf node to edge E labeled with the category that
          is the most common in *examples*.
        else call DTree($examples_i$, $features - \{F\}$) and attach the resulting
          tree as the subtree under edge E.
    Return the subtree rooted at R.

Tree Induction Pseudocode

DTree(*examples*, *features*) returns a tree

If all *examples* are in one category, return a leaf node with that category label.
Else if the set of *features* is empty, return a leaf node with the category label that
is the most common in examples. Else

pick a feature F and create a node R for it
For each possible value v_i of F:
Let $examples_i$ be the subset of examples that have value v_i for F
Add an out-going edge E to node R labeled with the value v_i.
If $examples_i$ is empty
then attach a leaf node to edge E labeled with the category that
is the most common in *examples*.
else call DTree($examples_i$, $features - \{F\}$) and attach the resulting
tree as the subtree under edge E.
Return the subtree rooted at R.

Tree Induction Pseudocode

DTree(*examples*, *features*) returns a tree

If all *examples* are in one category, return a leaf node with that category label.
Else if the set of *features* is empty, return a leaf node with the category label that
is the most common in examples. Else

pick/construct a feature F and create a node R for it
For each possible value v_i of F:
Let $examples_i$ be the subset of examples that have value v_i for F
Add an out-going edge E to node R labeled with the value v_i.
If $examples_i$ is empty
then attach a leaf node to edge E labeled with the category that
is the most common in *examples*.
else call DTree($examples_i$, $features - \{F\}$) and attach the resulting
tree as the subtree under edge E.
Return the subtree rooted at R.

Tree Induction Pseudocode

DTree(*examples*, *features*) returns a tree

If all *examples* are in one category, return a leaf node with that category label.
Else if the set of *features* is empty, return a leaf node with the category label that
is the most common in examples. Else

pick/construct a feature F and create a node R for it
For each possible value v_i of F:
Let $examples_i$ be the subset of examples that have value v_i for F
Add an out-going edge E to node R labeled with the value v_i.
If $examples_i$ is empty
then attach a leaf node to edge E labeled with the category that
is the most common in *examples*.
else call DTree($examples_i$, $features - \{F\}$) and attach the resulting
tree as the subtree under edge E.
Return the subtree rooted at R.

Tree Induction Pseudocode

DTree(*examples*, *features*) returns a tree

If all *examples* are in one category, return a leaf node with that category label.
Else if the set of *features* is empty, return a leaf node with the category label that
is the most common in examples. Else

pick/construct a feature F and create a node R for it
For each possible value v_i of F:
Let $examples_i$ be the subset of examples that have value v_i for F
Add an out-going edge E to node R labeled with the value v_i.
OR add a label v_i. to an existing edge
If $examples_i$ is empty
then attach a leaf node to edge E labeled with the category that
is the most common in *examples* (mode or mean).
else call DTree($examples_i$, $features - \{F\}$) and attach the resulting
tree as the subtree under edge E.
Return the subtree rooted at R.

Decision Tree Induction

Impurity. Picking a Good Split Feature

Impurity for classes $D_1, \ldots, D_l$

$$\text{Imp}(\{D_1, \ldots, D_l\}) = \sum_{j=1}^{l} \frac{|D_j|}{|D|} \text{Imp}(D_j)$$

Minority class $\min(\dot{p}, 1-\dot{p})$ – this is sometimes referred to as the error rate, as it measures the proportion of misclassified examples if the leaf was labelled with the majority class; the purer the set of examples, the fewer errors this will make. This impurity measure can equivalently be written as $1/2 - |\dot{p} - 1/2|$.

Gini index $2\dot{p}(1-\dot{p})$ – this is the expected error if we label examples in the leaf randomly: positive with probability $\dot{p}$ and negative with probability $1-\dot{p}$. The probability of a false positive is then $\dot{p}(1-\dot{p})$ and the probability of a false negative $(1-\dot{p})\dot{p}$. [3]

entropy $-\dot{p}\log_2\dot{p} - (1-\dot{p})\log_2(1-\dot{p})$ – this is the expected information, in bits, conveyed by somebody telling you the class of a randomly drawn example; the purer the set of examples, the more predictable this message becomes and the smaller the expected information.

Entropy

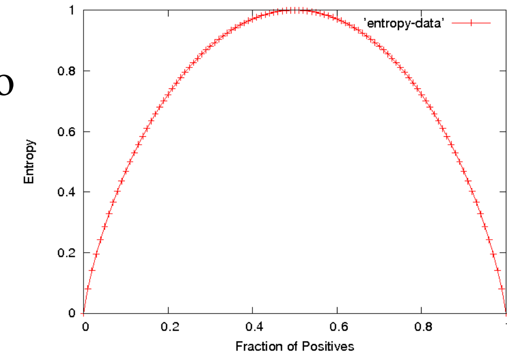
- Entropy (disorder, impurity) of a set of examples, S, relative to a binary classification is:

$$Entropy(S) = -p_1 \log_2(p_1) - p_0 \log_2(p_0)$$

where p_1 is the fraction of positive examples in S and p_0 is the fraction of negatives.

- If all examples are in one category, entropy is zero (we define $0 \cdot \log(0)=0$)
- If examples are equally mixed (p_1=p_0=0.5), entropy is a maximum of 1.
- Entropy can be viewed as the number of bits required on average to encode the class of an example in *S* where data compression (e.g. Huffman coding) is used to give shorter codes to more likely cases.
- For multi-class problems with c categories, entropy generalizes to

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2(p_i)$$



- Information Gain

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

Hypothesis Space Search

- Performs ***batch learning*** that processes all training instances at once rather than ***incremental learning*** that updates a hypothesis after each example.
- Performs hill-climbing (greedy search) that may only find a locally-optimal solution. Guaranteed to find a tree consistent with any conflict-free training set (i.e. identical feature vectors always assigned the same class), but not necessarily the simplest tree.
- Finds a single discrete hypothesis, so there is no way to provide confidences or create useful queries.

Continuous features

Use binary split of the current interval using the same impurity measure as for discrete attributes

64	65	68	69	70	71	72	75	80	81	83	85
yes	no	yes	yes	yes	no	no yes	yes yes	no	yes	yes	no

(Repeated values have been collapsed together.) There are only 11 possible positions for the breakpoint—8 if the breakpoint is not allowed to separate items of the same class. The information gain for each can be calculated in the usual way. For example, the test *temperature* < 71.5 produces four *yes*'s and two *no*'s, whereas *temperature* > 71.5 produces five *yes*'s and three *no*'s, and so the information value of this test is

$$\text{info}([4,2],[5,3]) = (6/14) \times \text{info}([4,2]) + (8/14) \times \text{info}([5,3]) = 0.939 \text{ bits}.$$

Missing feature values

- Remove the instance
- Replace with the most common (mode, mean) value
- Replace with the most common (mode, mean) value w.r.t. a class
- Decision trees: use weighted Impurity measure (add relative increment to each atribute value

Bias in Decision-Tree Induction

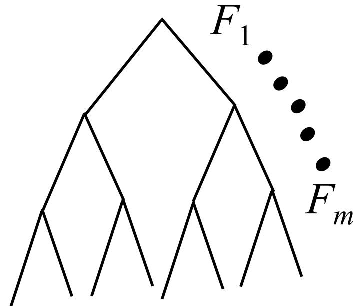
- Information-gain gives a bias for trees with minimal depth.
- Implements a search (preference) bias instead of a language (restriction) bias.

History of Decision-Tree Research

- Hunt and colleagues use exhaustive search decision-tree methods (CLS) to model human concept learning in the 1960's.
- In the late 70's, Quinlan developed ID3 with the information gain heuristic to learn expert systems from examples.
- Simulataneously, Breiman and Friedman and colleagues develop CART (Classification and Regression Trees), similar to ID3.
- In the 1980's a variety of improvements are introduced to handle noise, continuous features, missing features, and improved splitting criteria. Various expert-system development tools results.
- Quinlan's updated decision-tree package (C4.5) released in 1993.
- Weka includes Java version of C4.5 called J48.

Computational Complexity

- Worst case builds a complete tree where every path test every feature. Assume *n* examples and *m* features.



Maximum of *n* examples spread across all nodes at each of the *m* levels

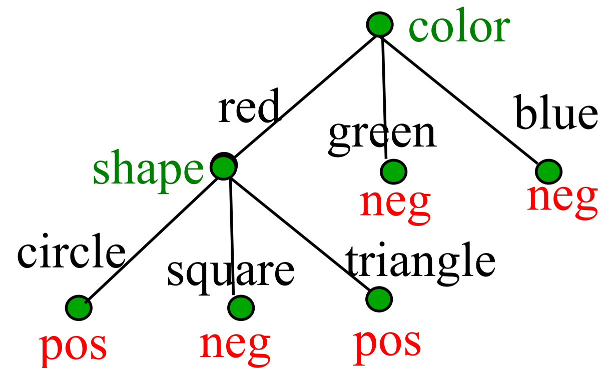
- At each level, *i*, in the tree, must examine the remaining *m*-*i* features for each instance at the level to calculate info gains.

$$\sum_{i=1}^{m} i \cdot n = O(nm^2)$$

- However, learned tree is rarely complete (number of leaves is ≤ *n*). In practice, complexity is linear in both number of features (*m*) and number of training examples (*n*).

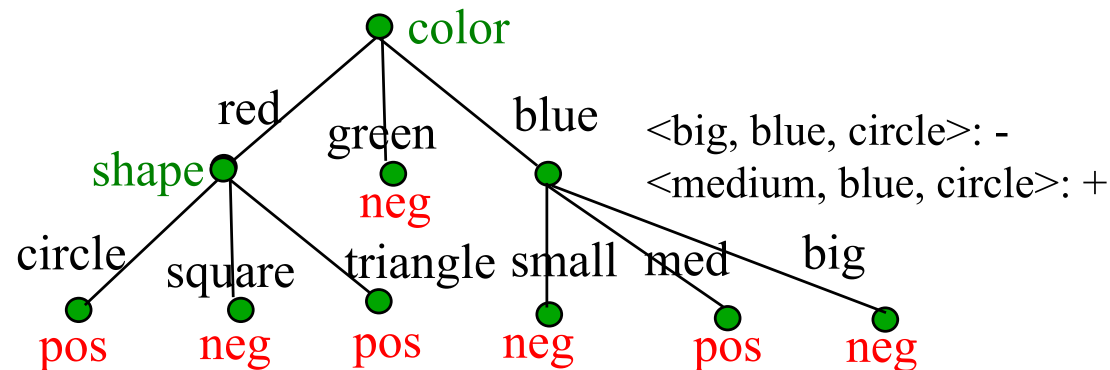
Overfitting Noise in Decision Trees

- Category or feature noise can easily cause overfitting.
  - Add noisy instance <medium, blue, circle>: pos (but really neg)



Overfitting Noise in Decision Trees

- Category or feature noise can easily cause overfitting.
  - Add noisy instance <medium, blue, circle>: pos (but really neg)



- Noise can also cause different instances of the same feature vector to have different classes. Impossible to fit this data and must label leaf with the majority class.
  - <big, red, circle>: neg (but really pos)
- Conflicting examples can also arise if the features are incomplete and inadequate to determine the class or if the target concept is non-deterministic.

Overfitting Prevention (Pruning) Methods

- Two basic approaches for decision trees
  - Prepruning: Stop growing tree as some point during top-down construction when there is no longer sufficient data to make reliable decisions.
  - Postpruning: Grow the full tree, then remove subtrees that do not have sufficient evidence.
- Label leaf resulting from pruning with the majority class of the remaining data, or a class probability distribution.
- Method for determining which subtrees to prune:
  - Cross-validation: Reserve some training data as a hold-out set (***validation set***, ***tuning set***) to evaluate utility of subtrees.
  - Statistical test: Use a statistical test on the training data to determine if any observed regularity can be dismisses as likely due to random chance.
  - Minimum description length (MDL): Determine if the additional complexity of the hypothesis is less complex than just explicitly remembering any exceptions resulting from pruning.

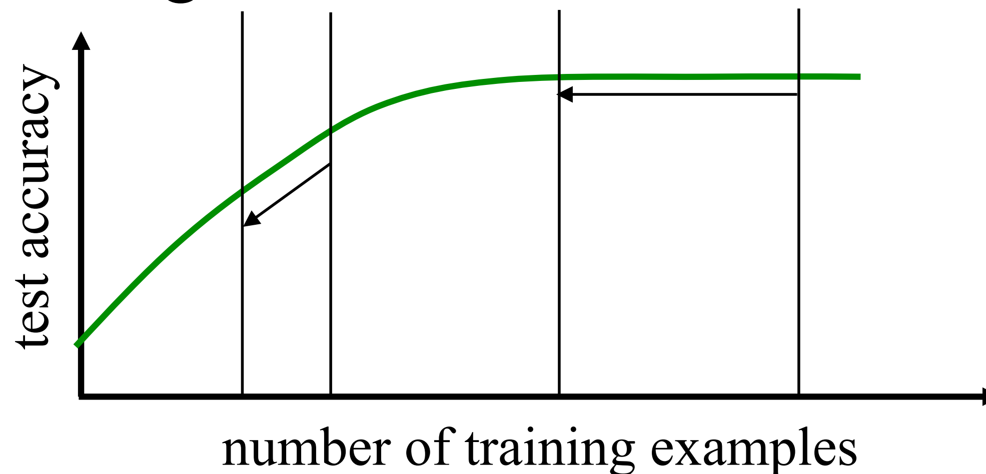
Reduced Error Pruning

- A post-pruning, cross-validation approach.

Partition training data in “grow” and “validation” sets.
Build a complete tree from the “grow” data.
Until accuracy on validation set decreases do:
    For each non-leaf node, n, in the tree do:
        Temporarily prune the subtree below n and replace it with a
          leaf labeled with the current majority class at that node.
        Measure and record the accuracy of the pruned tree on the validation set.
    Permanently prune the node that results in the greatest increase in accuracy on
      the validation set.

Issues with Reduced Error Pruning

- The problem with this approach is that it potentially “wastes” training data on the validation set.
- Severity of this problem depends where we are on the learning curve:



Cross-Validating without Losing Training Data

- If the algorithm is modified to grow trees breadth-first rather than depth-first, we can stop growing after reaching any specified tree complexity.
- First, run several trials of reduced error-pruning using different random splits of grow and validation sets.
- Record the complexity of the pruned tree learned in each trial. Let C be the average pruned-tree complexity.
- Grow a final tree breadth-first from all the training data but stop when the complexity reaches C.
- Similar cross-validation approach can be used to set arbitrary algorithm parameters in general.

Additional Decision Tree Issues

- Features with costs
- Misclassification costs
- Incremental learning
- Mining large databases that do not fit in main memory

C4.5

- Based on ID3 algorithm, author Ross Quinlan
- In all (or most of) non-commercial and commercial data mining tools
- Weka: C4.5 ver.8 -> j48

Scheme of C4.5 algorithm:

Run several time and choose the best tree

Inner:Take L% of learning data randomly

Call ID3 (pre-pruning, see –m parameter)

Prune the tree (post-pruning, -cf)

Take T% of unseen learning data for validation

If validation criterion holds, exit

Otherwise add L.increment to L and go to Inner