Data Mining
Practical Machine Learning Tools and Techniques

Slides for Chapter 6 of Data Mining by I. H. Witten and E. Frank
Implementation:
Real machine learning schemes

- Decision trees
  - From ID3 to C4.5 (pruning, numeric attributes, ...)
- Classification rules
  - From PRISM to RIPPER and PART (pruning, numeric data, ...)
- Extending linear models
  - Support vector machines and neural networks
- Instance-based learning
  - Pruning examples, generalized exemplars, distance functions
- Numeric prediction
  - Regression/model trees, locally weighted regression
- Clustering: hierarchical, incremental, probabilistic
  - Hierarchical, incremental, probabilistic
- Bayesian networks
  - Learning and prediction, fast data structures for learning
Industrial-strength algorithms

- For an algorithm to be useful in a wide range of real-world applications it must:
  - Permit numeric attributes
  - Allow missing values
  - Be robust in the presence of noise
  - Be able to approximate arbitrary concept descriptions (at least in principle)
- Basic schemes need to be extended to fulfill these requirements
Decision trees

- Extending ID3:
  - to permit numeric attributes: straightforward
  - to deal sensibly with missing values: trickier
  - stability for noisy data: requires pruning mechanism

- End result: C4.5 (Quinlan)
  - Best-known and (probably) most widely-used learning algorithm
  - Commercial successor: C5.0
Numeric attributes

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

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Windy</th>
<th>Play</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

If outlook = sunny and humidity = high then play = no
If outlook = rainy and windy = true then play = no
If outlook = overcast then play = yes
If humidity = normal then play = yes
If none of the above then play = yes
Weather data (again!)

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Windy</th>
<th>Play</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Overcast</td>
<td>Sunny</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>Rainy</td>
<td>Sunny</td>
<td>80</td>
<td>90</td>
<td>True</td>
</tr>
<tr>
<td>Rainy</td>
<td>Overcast</td>
<td>83</td>
<td>86</td>
<td>False</td>
</tr>
<tr>
<td>Rainy</td>
<td>Rainy</td>
<td>70</td>
<td>96</td>
<td>False</td>
</tr>
<tr>
<td>Rainy</td>
<td>Rainy</td>
<td>68</td>
<td>80</td>
<td>False</td>
</tr>
<tr>
<td>Rainy</td>
<td>Rainy</td>
<td>65</td>
<td>70</td>
<td>True</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

- Split on temperature attribute:

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

  - \[ \text{Info}([4,2],[5,3]) = \frac{6}{14} \text{info}([4,2]) + \frac{8}{14} \text{info}([5,3]) = 0.939 \text{ bits} \]

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

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

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

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

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

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

• Based on statistical significance test
  ♦ Stop growing the tree when there is no statistically significant association between any attribute and the class at a particular node

• Most popular test: *chi-squared test*

• ID3 used chi-squared test in addition to information gain
  ♦ Only statistically significant attributes were allowed to be selected by information gain procedure
Early stopping

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

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

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

- Delete node
- Redistribute instances
- Slower than subtree replacement

(Worthwhile?)
Estimating error rates

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

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

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

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

Combined using ratios 6:2:6 gives 0.51
Complexity of tree induction

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

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

- C4.5 rules slow for large and noisy datasets
- Commercial version C5.0 rules uses a different technique
  - Much faster and a bit more accurate
- C4.5 has two parameters
  - Confidence value (default 25%): lower values incur heavier pruning
  - Minimum number of instances in the two most popular branches (default 2)
**TDIDT: Top-Down Induction of Decision Trees**

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

- Common procedure: *separate-and-conquer*
- Differences:
  - Search method (e.g. greedy, beam search, ...)
  - Test selection criteria (e.g. accuracy, ...)
  - Pruning method (e.g. MDL, hold-out set, ...)
  - Stopping criterion (e.g. minimum accuracy)
  - Post-processing step
- Also: Decision list
  vs.
  one rule set for each class
Test selection criteria

- **Basic covering algorithm:**
  - keep adding conditions to a rule to improve its accuracy
  - Add the condition that improves accuracy the most

- **Measure 1: \( p/t \)**
  - \( t \) total instances covered by rule
  - \( p \) number of these that are positive
  - Produce rules that don’t cover *negative* instances, as quickly as possible
  - May produce rules with very small coverage —special cases or noise?

- **Measure 2: Information gain** \( p \left( \log(p/t) - \log(P/T) \right) \)
  - \( P \) and \( T \) the positive and total numbers before the new condition was added
  - Information gain emphasizes positive rather than negative instances

- These interact with the pruning mechanism used
Missing values, numeric attributes

- Common treatment of missing values: *for any test, they fail*
  - Algorithm must either
    - use other tests to separate out positive instances
    - leave them uncovered until later in the process
- In some cases it’s better to treat “missing” as a separate value
- Numeric attributes are treated just like they are in decision trees
Pruning rules

- Two main strategies:
  - *Incremental* pruning
  - *Global* pruning

- Other difference: pruning criterion
  - Error on hold-out set (*reduced-error pruning*)
  - Statistical significance
  - MDL principle

- Also: post-pruning vs. pre-pruning
Using a pruning set

- For statistical validity, must evaluate measure on data not used for training:
  - This requires a *growing set* and a *pruning set*
- *Reduced-error pruning*: build full rule set and then prune it
- *Incremental reduced-error pruning*: simplify each rule as soon as it is built
  - Can re-split data after rule has been pruned
- *Stratification* advantageous
Incremental reduced-error pruning

Initialize $E$ to the instance set

Until $E$ is empty do

Split $E$ into Grow and Prune in the ratio 2:1

For each class $C$ for which Grow contains an instance

Use basic covering algorithm to create best perfect rule for $C$

Calculate $w(R)$: worth of rule on Prune

and $w(R^-)$: worth of rule with final condition omitted

If $w(R^-) > w(R)$, prune rule and repeat previous step

From the rules for the different classes, select the one that’s worth most (i.e. with largest $w(R)$)

Print the rule

Remove the instances covered by rule from $E$

Continue
Measures used in IREP

- \[ \frac{[p + (N - n)]}{T} \]
  - \(N\) is total number of negatives
  - Counterintuitive:
    - \(p = 2000\) and \(n = 1000\) vs. \(p = 1000\) and \(n = 1\)

- Success rate \(\frac{p}{t}\)
  - Problem: \(p = 1\) and \(t = 1\)
    vs. \(p = 1000\) and \(t = 1001\)

- \(\frac{(p - n)}{t}\)
  - Same effect as success rate because it equals \(2\frac{p}{t} - 1\)

- Seems hard to find a simple measure of a rule’s worth that corresponds with intuition
Variations

• Generating rules for classes in order
  ♦ Start with the smallest class
  ♦ Leave the largest class covered by the default rule

• Stopping criterion
  ♦ Stop rule production if accuracy becomes too low

• Rule learner RIPPER:
  ♦ Uses MDL-based stopping criterion
  ♦ Employs post-processing step to modify rules guided by MDL criterion
Using global optimization

- **RIPPER:** *Repeated Incremental Pruning to Produce Error Reduction* (does global optimization in an efficient way)
- Classes are processed in order of increasing size
- Initial rule set for each class is generated using IREP
- An MDL-based stopping condition is used
  - *DL:* bits needed to send examples wrt set of rules, bits needed to send $k$ tests, and bits for $k$
- Once a rule set has been produced for each class, each rule is re-considered and two variants are produced
  - One is an extended version, one is grown from scratch
  - Chooses among three candidates according to *DL*
- Final clean-up step greedily deletes rules to minimize DL
• Avoids global optimization step used in C4.5rules and RIPPER
• Generates an unrestricted decision list using basic separate-and-conquer procedure
• Builds a *partial* decision tree to obtain a rule
  ♦ A rule is only pruned if all its implications are known
  ♦ Prevents *hasty generalization*
• Uses C4.5’s procedures to build a tree
Expand-subset (S):

Choose test T and use it to split set of examples into subsets

Sort subsets into increasing order of average entropy

while

  there is a subset X not yet been expanded AND all subsets expanded so far are leaves

  expand-subset(X)

if

  all subsets expanded are leaves
  AND estimated error for subtree \geq estimated error for node

  undo expansion into subsets and make node a leaf
Example
Notes on PART

● Make leaf with maximum coverage into a rule

● Treat missing values just as C4.5 does
  ♦ I.e. split instance into pieces

● Time taken to generate a rule:
  ♦ Worst case: same as for building a pruned tree
    • Occurs when data is noisy
  ♦ Best case: same as for building a single rule
    • Occurs when data is noise free
Rules with exceptions

1. Given: a way of generating a single good rule
2. Then it’s easy to generate rules with exceptions
3. Select default class for top-level rule
4. Generate a good rule for one of the remaining classes
5. Apply this method recursively to the two subsets produced by the rule (I.e. instances that are covered/not covered)
Iris data example

-> Iris setosa
   50/150

petal length >= 2.45
petal width < 1.75
-> Iris versicolor
   49/52

petal length >= 4.95
petal width < 1.55
-> Iris virginica
   2/2

sepal length < 4.95
sepal width >= 2.45
-> Iris virginica
   1/1

petal length >= 3.35
-> Iris virginica
   47/48

petal length < 4.85
sepal length < 5.95
-> Iris versicolor
   1/1
Extending linear classification

- Linear classifiers can’t model nonlinear class boundaries
- Simple trick:
  - Map attributes into new space consisting of combinations of attribute values
  - E.g.: all products of $n$ factors that can be constructed from the attributes
- Example with two attributes and $n = 3$:

$$x = w_1 a_1^3 + w_2 a_1^2 a_2 + w_3 a_1 a_2^2 + w_4 a_2^3$$
Problems with this approach

- **1\textsuperscript{st} problem: speed**
  - 10 attributes, and \( n = 5 \rightarrow 2000 \) coefficients
  - Use linear regression with attribute selection
  - Run time is cubic in number of attributes
- **2\textsuperscript{nd} problem: overfitting**
  - Number of coefficients is large relative to the number of training instances
  - *Curse of dimensionality* kicks in
Support vector machines

- **Support vector machines** are algorithms for learning linear classifiers
- Resilient to overfitting because they learn a particular linear decision boundary:
  - The *maximum margin hyperplane*
- Fast in the nonlinear case
  - Use a mathematical trick to avoid creating “pseudo-attributes”
  - The nonlinear space is created implicitly
The maximum margin hyperplane

- The instances closest to the maximum margin hyperplane are called support vectors
Support vectors

- The support vectors define the maximum margin hyperplane
  - All other instances can be deleted without changing its position and orientation

\[ x = w_0 + w_1 a_1 + w_2 a_2 \]

\[ x = b + \sum_{i \text{ is supp. vector}} \alpha_i y_i \tilde{a}(i) \cdot \tilde{a} \]
Finding support vectors

\[ x = b + \sum_{i \text{ is supp. vector}} \alpha_i y_i \hat{a}(i) \cdot \hat{a} \]

- **Support vector**: training instance for which \( \alpha_i > 0 \)
- **Determine \( \alpha_i \) and \( b \)?**—
  
  A *constrained quadratic optimization* problem
  
  - Off-the-shelf tools for solving these problems
  - However, special-purpose algorithms are faster
  - Example: Platt’s *sequential minimal optimization* algorithm (implemented in WEKA)

- **Note**: all this assumes separable data!
Nonlinear SVMs

- “Pseudo attributes” represent attribute combinations
- Overfitting not a problem because the maximum margin hyperplane is stable
  - There are usually few support vectors relative to the size of the training set
- Computation time still an issue
  - Each time the dot product is computed, all the “pseudo attributes” must be included
A mathematical trick

- Avoid computing the “pseudo attributes”
- Compute the dot product before doing the nonlinear mapping
- Example:
  \[ x = b + \sum_{i \text{ is supp. vector}} \alpha_i y_i (\vec{a}(i) \cdot \vec{a})^n \]
- Corresponds to a map into the instance space spanned by all products of \( n \) attributes
Other kernel functions

- Mapping is called a “kernel function”
- Polynomial kernel
  \[ x = b + \sum_{i \text{ is supp. vector}} \alpha_i y_i (\hat{a}(i) \cdot \hat{a})^n \]
- We can use others:
  \[ x = b + \sum_{i \text{ is supp. vector}} \alpha_i y_i K(\hat{a}(i) \cdot \hat{a}) \]
- Only requirement:
  \[ K(\tilde{x}_i, \tilde{x}_j) = \phi(\tilde{x}_i) \cdot \phi(\tilde{x}_j) \]
- Examples:
  \[ K(\tilde{x}_i, \tilde{x}_j) = (\tilde{x}_i \cdot \tilde{x}_j + 1)^d \]
  \[ K(\tilde{x}_i, \tilde{x}_j) = \exp\left(\frac{-(\tilde{x}_i - \tilde{x}_j)^2}{2\sigma^2}\right) \]
  \[ K(\tilde{x}_i, \tilde{x}_j) = \tanh(\beta \tilde{x}_i \cdot \tilde{x}_j + b) \]
Have assumed that the data is separable (in original or transformed space)
Can apply SVMs to noisy data by introducing a “noise” parameter $C$
$C$ bounds the influence of any one training instance on the decision boundary
  - Corresponding constraint: $0 \leq \alpha_i \leq C$
Still a quadratic optimization problem
Have to determine $C$ by experimentation
Sparse data

• SVM algorithms speed up dramatically if the data is *sparse* (i.e. many values are 0)
• Why? Because they compute lots and lots of dot products
• Sparse data $\Rightarrow$ compute dot products very efficiently
  • Iterate only over non-zero values
• SVMs can process sparse datasets with 10,000s of attributes
Applications

- Machine vision: e.g. face identification
  - Outperforms alternative approaches (1.5% error)
- Handwritten digit recognition: USPS data
  - Comparable to best alternative (0.8% error)
- Bioinformatics: e.g. prediction of protein secondary structure
- Text classification
- Can modify SVM technique for numeric prediction problems
Support vector regression

- Maximum margin hyperplane only applies to classification
- However, idea of support vectors and kernel functions can be used for regression
- Basic method same as in linear regression: want to minimize error
  - Difference A: ignore errors smaller than $\epsilon$ and use absolute error instead of squared error
  - Difference B: simultaneously aim to maximize flatness of function
- User-specified parameter $\epsilon$ defines “tube”
More on SVM regression

- If there are tubes that enclose all the training points, the flattest of them is used
  - Eg.: mean is used if $2\varepsilon > \text{range of target values}$
- Model can be written as:
  $$x = b + \sum_{i \text{ is supp. vector}} \alpha_i \vec{a}(i) \cdot \vec{a}$$
  - Support vectors: points on or outside tube
  - Dot product can be replaced by kernel function
  - Note: coefficients $\alpha_i$ may be negative
- No tube that encloses all training points?
  - Requires trade-off between error and flatness
  - Controlled by upper limit $C$ on absolute value of coefficients $\alpha_i$
Examples

$\varepsilon = 2$

$\varepsilon = 1$

$\varepsilon = 0.5$
The kernel perceptron

- Can use “kernel trick” to make non-linear classifier using perceptron rule
- Observation: weight vector is modified by adding or subtracting training instances
- Can represent weight vector using all instances that have been misclassified:
  - Can use instead of \( y \) is either -1 or +1
  - Now swap summation signs:
  - Can be expressed as:
  - Can replace dot product by kernel:
Comments on kernel perceptron

- Finds separating hyperplane in space created by kernel function (if it exists)
  - But: doesn't find maximum-margin hyperplane
- Easy to implement, supports incremental learning
- Linear and logistic regression can also be upgraded using the kernel trick
  - But: solution is not “sparse”: every training instance contributes to solution
- Perceptron can be made more stable by using all weight vectors encountered during learning, not just last one (voted perceptron)
  - Weight vectors vote on prediction (vote based on number of successful classifications since inception)
Multilayer perceptrons

- Using kernels is only one way to build nonlinear classifier based on perceptrons
- Can create network of perceptrons to approximate arbitrary target concepts
- *Multilayer perceptron* is an example of an artificial neural network
  - Consists of: input layer, hidden layer(s), and output layer
- Structure of MLP is usually found by experimentation
- Parameters can be found using *backpropagation*
Examples
Backpropagation

• How to learn weights given network structure?
  ♦ Cannot simply use perceptron learning rule because we have hidden layer(s)
  ♦ Function we are trying to minimize: error
  ♦ Can use a general function minimization technique called gradient descent

• Need differentiable activation function: use sigmoid function instead of threshold function

\[ f(x) = \frac{1}{1 + \exp(-x)} \]

• Need differentiable error function: can't use zero-one loss, but can use squared error

\[ E = \frac{1}{2} (y - f(x))^2 \]
The two activation functions
Gradient descent example

- Function: $x^2 + 1$
- Derivative: $2x$
- Learning rate: 0.1
- Start value: 4

*Can only find a local minimum!*
Minimizing the error I

- Need to find partial derivative of error function for each parameter (i.e. weight)

\[
\frac{dE}{dw_i} = (y - f(x)) \frac{df(x)}{dw_i} \\
\frac{df(x)}{dx} = f(x)(1 - f(x)) \\
x = \sum_i w_i f(x_i) \\
\frac{df(x)}{dw_i} = f'(x) f(x_i) \\
\frac{dE}{dw_i} = (y - f(x)) f'(x) f(x_i)
\]
Minimizing the error II

- What about the weights for the connections from the input to the hidden layer?

\[
\frac{dE}{dw_{ij}} = \frac{dE}{dx} \frac{dx}{dw_{ij}} = (y - f(x)) f'(x) \frac{dx}{dw_{ij}}
\]

\[
x = \sum_i w_i f(x_i)
\]

\[
\frac{dx}{dw_{ij}} = W_i \frac{df(x_i)}{dw_{ij}}
\]

\[
\frac{df(x_i)}{dw_{ij}} = f'(x_i) \frac{dx_i}{dw_{ij}} = f'(x_i) a_i
\]

\[
\frac{dE}{dw_{ij}} = (y - f(x)) f'(x) w_i f'(x_i) a_i
\]
Remarks

- Same process works for multiple hidden layers and multiple output units (e.g., for multiple classes)
- Can update weights after all training instances have been processed or incrementally:
  - batch learning vs. stochastic backpropagation
  - Weights are initialized to small random values
- How to avoid overfitting?
  - Early stopping: use validation set to check when to stop
  - Weight decay: add penalty term to error function
- How to speed up learning?
  - Momentum: re-use proportion of old weight change
  - Use optimization method that employs 2nd derivative
Radial basis function networks

- Another type of *feedforward network* with two layers (plus the input layer)
- Hidden units represent points in instance space and activation depends on distance
  - To this end, distance is converted into similarity: Gaussian activation function
    - Width may be different for each hidden unit
  - Points of equal activation form hypersphere (or hyperellipsoid) as opposed to hyperplane
- Output layer same as in MLP
Learning RBF networks

- Parameters: centers and widths of the RBFs + weights in output layer
- Can learn two sets of parameters independently and still get accurate models
  - Eg.: clusters from $k$-means can be used to form basis functions
  - Linear model can be used based on fixed RBFs
  - Makes learning RBFs very efficient
- Disadvantage: no built-in attribute weighting based on relevance
- RBF networks are related to RBF SVMs
Instance-based learning

- Practical problems of 1-NN scheme:
  - Slow (but: fast tree-based approaches exist)
    - Remedy: remove irrelevant data
  - Noise (but: $k$-NN copes quite well with noise)
    - Remedy: remove noisy instances
  - All attributes deemed equally important
    - Remedy: weight attributes (or simply select)
  - Doesn’t perform explicit generalization
    - Remedy: rule-based NN approach
Learning prototypes

- Only those instances involved in a decision need to be stored
- Noisy instances should be filtered out
- Idea: only use *prototypical* examples
Speed up, combat noise

- **IB2: save memory, speed up classification**
  - Work incrementally
  - Only incorporate misclassified instances
  - Problem: noisy data gets incorporated

- **IB3: deal with noise**
  - Discard instances that don’t perform well
  - Compute confidence intervals for
    - 1. Each instance’s success rate
    - 2. Default accuracy of its class
  - Accept/reject instances
    - Accept if lower limit of 1 exceeds upper limit of 2
    - Reject if upper limit of 1 is below lower limit of 2
Weight attributes

- **IB4**: weight each attribute (weights can be class-specific)
- **Weighted Euclidean distance:**
  \[
  \sqrt{(w_1(x_1 - y_1)^2 + \ldots + w_n(x_n - y_n)^2)}
  \]
- **Update weights based on nearest neighbor**
  - Class correct: increase weight
  - Class incorrect: decrease weight
  - Amount of change for \(i\)th attribute depends on \(|x_i - y_i|\)
Rectangular generalizations

- Nearest-neighbor rule is used outside rectangles
- Rectangles are rules! (But they can be more conservative than “normal” rules.)
- Nested rectangles are rules with exceptions
Generalized exemplars

- Generalize instances into *hyperrectangles*
  - Online: incrementally modify rectangles
  - Offline version: seek small set of rectangles that cover the instances
- Important design decisions:
  - Allow overlapping rectangles?
    - Requires conflict resolution
  - Allow nested rectangles?
  - Dealing with uncovered instances?
Separating generalized exemplars

Class 1

Class 2

Separation line
Generalized distance functions

- Given: some transformation operations on attributes
- $K^*$: similarity = probability of transforming instance A into B by chance
  - Average over all transformation paths
  - Weight paths according their probability *(need way of measuring this)*
- Uniform way of dealing with different attribute types
- Easily generalized to give distance between sets of instances
Numeric prediction

- Counterparts exist for all schemes previously discussed
  - Decision trees, rule learners, SVMs, etc.
- (Almost) all classification schemes can be applied to regression problems using discretization
  - Discretize the class into intervals
  - Predict weighted average of interval midpoints
  - Weight according to class probabilities
Regression trees

• Like decision trees, but:
  ♦ Splitting criterion: minimize intra-subset variation
  ♦ Termination criterion: std dev becomes small
  ♦ Pruning criterion: based on numeric error measure
  ♦ Prediction: Leaf predicts average class values of instances

• Piecewise constant functions
• Easy to interpret
• More sophisticated version: model trees
Model trees

- Build a regression tree
- Each leaf $\Rightarrow$ linear regression function
- Smoothing: factor in ancestor’s predictions
  - Smoothing formula: $p' = \frac{np + kq}{n + k}$
  - Same effect can be achieved by incorporating ancestor models into the leaves
- Need linear regression function at each node
- At each node, use only a subset of attributes
  - Those occurring in subtree
  - (+maybe those occurring in path to the root)
- Fast: tree usually uses only a small subset of the attributes
Building the tree

- **Splitting:** standard deviation reduction

  \[ SDR = sd(T) - \sum_{i} \left| \frac{T_i}{T} \right| \times sd(T_i) \]

- **Termination:**
  - Standard deviation < 5% of its value on full training set
  - Too few instances remain (e.g. < 4)

**Pruning:**
- Heuristic estimate of absolute error of LR models:
  \[ \frac{n+\nu}{n-\nu} \times \text{average\_absolute\_error} \]
- Greedily remove terms from LR models to minimize estimated error
- Heavy pruning: single model may replace whole subtree
- Proceed bottom up: compare error of LR model at internal node to error of subtree
Nominal attributes

• Convert nominal attributes to binary ones
  • Sort attribute by average class value
  • If attribute has $k$ values, generate $k – 1$ binary attributes
    • $i$ th is 0 if value lies within the first $i$, otherwise 1

• Treat binary attributes as numeric
• Can prove: best split on one of the new attributes is the best (binary) split on original
Missing values

• Modify splitting criterion:
  \[ SDR = \frac{m}{|T|} \times [sd(T) - \sum_i \frac{T_i}{T} \times sd(T_i)] \]

• To determine which subset an instance goes into, use *surrogate splitting*
  • Split on the attribute whose correlation with original is greatest
  • Problem: complex and time-consuming
  • Simple solution: always use the class

• Test set: replace missing value with average
Surrogate splitting based on class

- Choose split point based on instances with known values
- Split point divides instances into 2 subsets
  - $L$ (smaller class average)
  - $R$ (larger)
- $m$ is the average of the two averages
- For an instance with a missing value:
  - Choose $L$ if class value $< m$
  - Otherwise $R$
- Once full tree is built, replace missing values with averages of corresponding leaf nodes
Pseudo-code for M5'

• Four methods:
  ♦ Main method: MakeModelTree
  ♦ Method for splitting: split
  ♦ Method for pruning: prune
  ♦ Method that computes error: subtreeError

• We’ll briefly look at each method in turn

• Assume that linear regression method performs attribute subset selection based on error
MakeModelTree (instances)
{
    SD = sd(instances)
    for each k-valued nominal attribute
        convert into k-1 synthetic binary attributes
    root = newNode
    root.instances = instances
    split(root)
    prune(root)
    printTree(root)
}
split (node) {
    if sizeof(node.instances) < 4 or 
        sd(node.instances) < 0.05*SD 
        node.type = LEAF 
    else 
        node.type = INTERIOR 
        for each attribute 
            for all possible split positions of attribute 
                calculate the attribute's SDR 
                node.attribute = attribute with maximum SDR 
                split(node.left) 
                split(node.right) 
}
prune(node)
{
    if node = INTERIOR then
        prune(node.leftChild)
        prune(node.rightChild)
        node.model = linearRegression(node)
        if subtreeError(node) > error(node) then
            node.type = LEAF
    }

subtreeError(node)
{
    l = node.left; r = node.right
    if node = INTERIOR then
        return (sizeof(l.instances)*subtreeError(l)
            + sizeof(r.instances)*subtreeError(r))
            /sizeof(node.instances)
    else return error(node)
}
Model tree for servo data

Result of merging
Rules from model trees

- **PART algorithm** generates classification rules by building partial decision trees.
- Can use the same method to build rule sets for regression:
  - Use model trees instead of decision trees.
  - Use variance instead of entropy to choose node to expand when building partial tree.
- Rules will have linear models on right-hand side.
- Caveat: using smoothed trees may not be appropriate due to separate-and-conquer strategy.
Locally weighted regression

- Numeric prediction that combines
  - instance-based learning
  - linear regression
- “Lazy”:
  - computes regression function at prediction time
  - works incrementally
- Weight training instances
  - according to distance to test instance
  - needs weighted version of linear regression
- Advantage: nonlinear approximation
- But: slow
Design decisions

• Weighting function:
  ♦ Inverse Euclidean distance
  ♦ Gaussian kernel applied to Euclidean distance
  ♦ Triangular kernel used the same way
  ♦ etc.

• *Smoothing parameter* is used to scale the distance function
  ♦ Multiply distance by inverse of this parameter
  ♦ Possible choice: distance of \( k \) th nearest training instance (makes it data dependent)
Discussion

- Regression trees were introduced in CART
- Quinlan proposed model tree method (M5)
- M5’: slightly improved, publicly available
- Quinlan also investigated combining instance-based learning with M5
- CUBIST: Quinlan’s commercial rule learner for numeric prediction
- Interesting comparison: neural nets vs. M5
Clustering: how many clusters?

- How to choose $k$ in $k$-means? Possibilities:
  - Choose $k$ that minimizes cross-validated squared distance to cluster centers
  - Use penalized squared distance on the training data (e.g., using an MDL criterion)
  - Apply $k$-means recursively with $k = 2$ and use stopping criterion (e.g., based on MDL)

- Seeds for subclusters can be chosen by seeding along direction of greatest variance in cluster (one standard deviation away in each direction from cluster center of parent cluster)
- Implemented in algorithm called $X$-means (using Bayesian Information Criterion instead of MDL)
Incremental clustering

- Heuristic approach (COBWEB/CLASSIT)
- Form a hierarchy of clusters incrementally
- Start:
  - tree consists of empty root node
- Then:
  - add instances one by one
  - update tree appropriately at each stage
  - to update, find the right leaf for an instance
  - May involve restructuring the tree
- Base update decisions on *category utility*
Clustering weather data

<table>
<thead>
<tr>
<th>ID</th>
<th>Outlook</th>
<th>Temp.</th>
<th>Humidity</th>
<th>Windy</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>B</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>True</td>
</tr>
<tr>
<td>C</td>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>D</td>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>E</td>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
</tr>
<tr>
<td>F</td>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>True</td>
</tr>
<tr>
<td>G</td>
<td>Overcast</td>
<td>Cool</td>
<td>Normal</td>
<td>True</td>
</tr>
<tr>
<td>H</td>
<td>Sunny</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>I</td>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
</tr>
<tr>
<td>J</td>
<td>Rainy</td>
<td>Mild</td>
<td>Normal</td>
<td>False</td>
</tr>
<tr>
<td>K</td>
<td>Sunny</td>
<td>Mild</td>
<td>Normal</td>
<td>True</td>
</tr>
<tr>
<td>L</td>
<td>Overcast</td>
<td>Mild</td>
<td>High</td>
<td>True</td>
</tr>
<tr>
<td>M</td>
<td>Overcast</td>
<td>Hot</td>
<td>Normal</td>
<td>False</td>
</tr>
<tr>
<td>N</td>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>True</td>
</tr>
</tbody>
</table>
Clustering weather data

<table>
<thead>
<tr>
<th>ID</th>
<th>Outlook</th>
<th>Temp.</th>
<th>Humidity</th>
<th>Windy</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>B</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>True</td>
</tr>
<tr>
<td>C</td>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>D</td>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>E</td>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
</tr>
<tr>
<td>F</td>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>True</td>
</tr>
<tr>
<td>G</td>
<td>Overcast</td>
<td>Cool</td>
<td>Normal</td>
<td>True</td>
</tr>
<tr>
<td>H</td>
<td>Sunny</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>I</td>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
</tr>
<tr>
<td>J</td>
<td>Rainy</td>
<td>Mild</td>
<td>Normal</td>
<td>False</td>
</tr>
<tr>
<td>K</td>
<td>Sunny</td>
<td>Mild</td>
<td>Normal</td>
<td>True</td>
</tr>
<tr>
<td>L</td>
<td>Overcast</td>
<td>Mild</td>
<td>High</td>
<td>True</td>
</tr>
<tr>
<td>M</td>
<td>Overcast</td>
<td>Hot</td>
<td>Normal</td>
<td>False</td>
</tr>
<tr>
<td>N</td>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>True</td>
</tr>
</tbody>
</table>

Consider splitting the best host if merging doesn’t help
Oops! $a$ and $b$ are actually very similar
Example: the iris data (subset)
Clustering with cutoff

Diagram showing a tree structure with species names at various nodes including 'Versicolor', 'Virginica', and 'Setosa'.
Category utility

• Category utility: quadratic loss function defined on conditional probabilities:

\[
CU(C_1, C_2, \ldots, C_k) = \frac{\sum_i \Pr[C_j] \sum_i \sum_j (\Pr[a_i = v_{ij} | C_j]^2 - \Pr[a_i = v_{ij}]^2)}{k}
\]

• Every instance in different category \(\Rightarrow\) numerator becomes

\[
n - \sum_i \sum_j \Pr[a_i = v_{ij}]^2
\]

\[\text{maximum}\]
Numeric attributes

- Assume normal distribution:
  \[ f(a) = \frac{1}{\sqrt{(2\pi)\sigma}} \exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right) \]
- Then:
  \[ \sum_j Pr[a_i = v_{ij}]^2 \equiv \int f(a_i)^2 \, da_i = \frac{1}{2\sqrt{\pi} \sigma_i} \]
- Thus
  \[ CU(C_1, C_2, \ldots, C_k) = \frac{\sum_l Pr[C_l] \sum_i \sum_j (Pr[a_i = v_{ij}|C_i]^2 - Pr[a_i = v_{ij}]^2)}{k} \]

becomes
  \[ CU(C_1, C_2, \ldots, C_k) = \frac{\sum_j Pr[C_j] \frac{1}{2\sqrt{\pi}} \sum_i \left(\frac{1}{\sigma_i} - \frac{1}{\bar{\sigma}}\right)}{k} \]

- Prespecified minimum variance
  - *acuity* parameter
Probability-based clustering

• Problems with heuristic approach:
  ♦ Division by $k$?
  ♦ Order of examples?
  ♦ Are restructuring operations sufficient?
  ♦ Is result at least local minimum of category utility?

• Probabilistic perspective \[\Rightarrow\] seek the most likely clusters given the data

• Also: instance belongs to a particular cluster with a certain probability
Finite mixtures

- Model data using a *mixture* of distributions
- One cluster, one distribution
  - governs probabilities of attribute values in that cluster
- *Finite mixtures*: finite number of clusters
- Individual distributions are normal (usually)
- Combine distributions using cluster weights
Two-class mixture model

Data

<table>
<thead>
<tr>
<th>A</th>
<th>51</th>
<th>B</th>
<th>62</th>
<th>B</th>
<th>64</th>
<th>A</th>
<th>48</th>
<th>A</th>
<th>39</th>
<th>A</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>43</td>
<td>A</td>
<td>47</td>
<td>A</td>
<td>51</td>
<td>B</td>
<td>64</td>
<td>B</td>
<td>62</td>
<td>A</td>
<td>48</td>
</tr>
<tr>
<td>B</td>
<td>62</td>
<td>A</td>
<td>52</td>
<td>A</td>
<td>52</td>
<td>A</td>
<td>51</td>
<td>B</td>
<td>64</td>
<td>B</td>
<td>64</td>
</tr>
<tr>
<td>B</td>
<td>64</td>
<td>B</td>
<td>64</td>
<td>B</td>
<td>62</td>
<td>B</td>
<td>63</td>
<td>A</td>
<td>52</td>
<td>A</td>
<td>42</td>
</tr>
<tr>
<td>A</td>
<td>45</td>
<td>A</td>
<td>51</td>
<td>A</td>
<td>49</td>
<td>A</td>
<td>43</td>
<td>B</td>
<td>63</td>
<td>A</td>
<td>48</td>
</tr>
<tr>
<td>A</td>
<td>42</td>
<td>B</td>
<td>65</td>
<td>A</td>
<td>48</td>
<td>B</td>
<td>65</td>
<td>B</td>
<td>64</td>
<td>A</td>
<td>41</td>
</tr>
<tr>
<td>A</td>
<td>46</td>
<td>A</td>
<td>48</td>
<td>B</td>
<td>62</td>
<td>B</td>
<td>66</td>
<td>A</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>45</td>
<td>A</td>
<td>49</td>
<td>A</td>
<td>43</td>
<td>B</td>
<td>65</td>
<td>B</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>45</td>
<td>A</td>
<td>46</td>
<td>A</td>
<td>40</td>
<td>A</td>
<td>46</td>
<td>A</td>
<td>48</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \mu_A = 50, \quad \sigma_A = 5, \quad p_A = 0.6 \quad \mu_B = 65, \quad \sigma_B = 2, \quad p_B = 0.4 \]
Using the mixture model

• Probability that instance \( x \) belongs to cluster \( A \):

\[
Pr[A | x] = \frac{Pr[x | A] Pr[A]}{Pr[x]} = \frac{f(x; \mu_A, \sigma_A) p_A}{Pr[x]}
\]

with

\[
f(x; \mu, \sigma) = \frac{1}{\sqrt{(2\pi)\sigma}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)
\]

• Probability of an instance given the clusters:

\[
Pr[x|\text{the}\_\text{clusters}] = \sum_i Pr[x|\text{cluster}_i] Pr[\text{cluster}_i]
\]
Learning the clusters

• Assume:
  ♦ we know there are $k$ clusters

• Learn the clusters ⇒
  ♦ determine their parameters
  ♦ I.e. means and standard deviations

• Performance criterion:
  ♦ probability of training data given the clusters

• EM algorithm
  ♦ finds a local maximum of the likelihood
EM algorithm

- EM = Expectation-Maximization
  - Generalize $k$-means to probabilistic setting
- Iterative procedure:
  - E “expectation” step:
    Calculate cluster probability for each instance
  - M “maximization” step:
    Estimate distribution parameters from cluster probabilities
- Store cluster probabilities as instance weights
- Stop when improvement is negligible
More on EM

- Estimate parameters from weighted instances

\[ \mu_A = \frac{w_1 x_1 + w_2 x_2 + \ldots + w_n x_n}{w_1 + w_2 + \ldots + w_n} \]

\[ \sigma_A = \frac{w_1 (x_1 - \mu)^2 + w_2 (x_2 - \mu)^2 + \ldots + w_n (x_n - \mu)^2}{w_1 + w_2 + \ldots + w_n} \]

- Stop when log-likelihood saturates

- Log-likelihood:

\[ \sum_i \log \left( p_A Pr[x_i | A] + p_B Pr[x_i | B] \right) \]
Extending the mixture model

- More than two distributions: easy
- Several attributes: easy—assuming independence!
- Correlated attributes: difficult
  - Joint model: bivariate normal distribution with a (symmetric) covariance matrix
  - $n$ attributes: need to estimate $n + n(n+1)/2$ parameters
More mixture model extensions

- Nominal attributes: easy if independent
- Correlated nominal attributes: difficult
  - Two correlated attributes \( \Rightarrow \nu_1 \nu_2 \) parameters
- Missing values: easy
- Can use other distributions than normal:
  - “log-normal” if predetermined minimum is given
  - “log-odds” if bounded from above and below
  - Poisson for attributes that are integer counts
- Use cross-validation to estimate \( k \)!
Bayesian clustering

- Problem: many parameters $\Rightarrow$ EM overfits
- *Bayesian approach*: give every parameter a prior probability distribution
  - Incorporate prior into overall likelihood figure
  - Penalizes introduction of parameters
- Eg: Laplace estimator for nominal attributes
- Can also have prior on number of clusters!
- Implementation: NASA’s AUTOCLASS
Discussion

● Can interpret clusters by using supervised learning
  ♦ post-processing step
● Decrease dependence between attributes?
  ♦ pre-processing step
  ♦ E.g. use principal component analysis
● Can be used to fill in missing values
● Key advantage of probabilistic clustering:
  ♦ Can estimate likelihood of data
  ♦ Use it to compare different models objectively
• Naïve Bayes assumes: attributes conditionally independent given the class
• Doesn’t hold in practice but classification accuracy often high
• However: sometimes performance much worse than e.g. decision tree
• Can we eliminate the assumption?
Enter Bayesian networks

- Graphical models that can represent any probability distribution
- Graphical representation: directed acyclic graph, one node for each attribute
- Overall probability distribution factorized into component distributions
- Graph’s nodes hold component distributions (conditional distributions)
Network for the weather data

Data Mining: Practical Machine Learning Tools and Techniques (Chapter 6)
Network for the weather data

Data Mining: Practical Machine Learning Tools and Techniques (Chapter 6)
Computing the class probabilities

• Two steps: computing a product of probabilities for each class and normalization
  ♦ For each class value
    • Take all attribute values and class value
    • Look up corresponding entries in conditional probability distribution tables
    • Take the product of all probabilities
  ♦ Divide the product for each class by the sum of the products (normalization)
Why can we do this? (Part I)

- Single assumption: values of a node’s parents completely determine probability distribution for current node

\[ Pr[\text{node}|\text{ancestors}] = Pr[\text{node}|\text{parents}] \]

Means that node/attribute is conditionally independent of other ancestors given parents
Why can we do this? (Part II)

- Chain rule from probability theory:

\[
Pr[a_1, a_2, \ldots, a_n] = \prod_{i=1}^{n} Pr[a_i | a_{i-1}, \ldots, a_1]
\]

Because of our assumption from the previous slide:

\[
Pr[a_1, a_2, \ldots, a_n] = \prod_{i=1}^{n} Pr[a_i | a_{i-1}, \ldots, a_1] = \prod_{i=1}^{n} Pr[a_i | a_i \text{'s parents}]
\]
Learning Bayes nets

- Basic components of algorithms for learning Bayes nets:
  - Method for evaluating the goodness of a given network
    - Measure based on probability of training data given the network (or the logarithm thereof)
  - Method for searching through space of possible networks
    - Amounts to searching through sets of edges because nodes are fixed
Problem: overfitting

- Can’t just maximize probability of the training data
  - Because then it’s always better to add more edges (fit the training data more closely)
- Need to use cross-validation or some penalty for complexity of the network

**AIC measure:**

\[
AIC_{score} = -LL + K
\]

**MDL measure:**

\[
MDL_{score} = -LL + \frac{K}{2} \log N
\]

\(LL\): log-likelihood (log of probability of data), \(K\): number of free parameters, \(N\): #instances

Another possibility: Bayesian approach with prior distribution over networks
Searching for a good structure

- Task can be simplified: can optimize each node separately
  - Because probability of an instance is product of individual nodes’ probabilities
  - Also works for AIC and MDL criterion because penalties just add up
- Can optimize node by adding or removing edges from other nodes
- Must not introduce cycles!
The K2 algorithm

- Starts with given ordering of nodes (attributes)
- Processes each node in turn
- Greedily tries adding edges from previous nodes to current node
- Moves to next node when current node can’t be optimized further
- Result depends on initial order
Some tricks

- Sometimes it helps to start the search with a naïve Bayes network.
- It can also help to ensure that every node is in a Markov blanket of class node.
  - Markov blanket of a node includes all parents, children, and children's parents of that node.
  - Given values for Markov blanket, node is conditionally independent of nodes outside blanket.
  - I.e. node is irrelevant to classification if not in Markov blanket of class node.
Other algorithms

- Extending K2 to consider greedily adding or deleting edges between any pair of nodes
  - Further step: considering inverting the direction of edges
- TAN (Tree Augmented Naïve Bayes):
  - Starts with naïve Bayes
  - Considers adding second parent to each node (apart from class node)
  - Efficient algorithm exists
Likelihood vs. conditional likelihood

- In classification what we really want is to maximize probability of class given other attributes
  
  *Not* probability of the instances

- But: no closed-form solution for probabilities in nodes’ tables that maximize this

- However: can easily compute conditional probability of data based on given network

- Seems to work well when used for network scoring
Data structures for fast learning

- Learning Bayes nets involves a lot of counting for computing conditional probabilities
- Naïve strategy for storing counts: hash table
  - Runs into memory problems very quickly
- More sophisticated strategy: *all-dimensions (AD) tree*
  - Analogous to *kD*-tree for numeric data
  - Stores counts in a tree but in a clever way such that redundancy is eliminated
  - Only makes sense to use it for large datasets
AD tree example

<table>
<thead>
<tr>
<th>humidity</th>
<th>windy</th>
<th>play</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>true</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>high</td>
<td>true</td>
<td>no</td>
<td>2</td>
</tr>
<tr>
<td>high</td>
<td>false</td>
<td>yes</td>
<td>2</td>
</tr>
<tr>
<td>high</td>
<td>false</td>
<td>no</td>
<td>2</td>
</tr>
<tr>
<td>normal</td>
<td>true</td>
<td>yes</td>
<td>2</td>
</tr>
<tr>
<td>normal</td>
<td>true</td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td>normal</td>
<td>false</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>normal</td>
<td>false</td>
<td>no</td>
<td>0</td>
</tr>
</tbody>
</table>

- any value (14 instances)
- humidity = normal (7 instances)
  - windy = true (6 instances)
    - play = no (5 instances)
  - play = no (1 instance)
- windy = true (3 instances)
  - play = no (1 instance)
Building an AD tree

- Assume each attribute in the data has been assigned an index
- Then, expand node for attribute $i$ with the values of all attributes $j > i$
  - Two important restrictions:
    - Most populous expansion for each attribute is omitted (breaking ties arbitrarily)
    - Expansions with counts that are zero are also omitted
- The root node is given index zero
Discussion

• We have assumed: discrete data, no missing values, no new nodes
• Different method of using Bayes nets for classification: *Bayesian multinets*
  ♦ I.e. build one network for each class and make prediction using Bayes’ rule
• Different class of learning methods for Bayes nets: testing conditional independence assertions
• Can also build Bayes nets for regression tasks