Decision Tree

A Project Report submitted in partial fulfillment of the Requirement for the degree of

M.Sc. in Mathematics and Computing

Submitted By: Kaushik Malakar (9925219)

Under the Guidance of: Dr. A. Goswami

Date of Submission: Dec. 3, 2003

Department of Mathematics
Indian Institute of Technology
Kharagpur – 721 302

To my parents, Mr. Madhusudan Malakar and Swapna Malakar, who brought me up despite the stress and complexities of their lives and devoted themselves to my education;

To my teachers, who taught me the art of reacting to a changing environment; and

To millions of the poor and downtrodden people of my country and the world, whose sacrifice and tolerance paved the royal road of my education, and whose love and emotion, smile and tears inspired me to speak their thoughts in my words.

Kaushik Malakar

Department of Mathematics
Indian Institute of Technology
Kharagpur -721 302, INDIA

CERTIFICATE

This is to certify that the project entitled, **"Decision Tree"** submitted by Kaushik Malakar, (Roll number 9925219, Department of Mathematics), to Indian Institute of Technology, Kharagpur - 721302, is a bonafide record of the work carried out under my supervision and guidance.

Dr. A. Goswami
Professor, Department of Mathematics
Indian Institute of Technology
Kharagpur – 721 302

ACKNOWLEDGEMENTS

***A**cknowledgement is not a mere formality but a genuine opportunity to express the sincere thanks to all those without whose active support and encouragement this project wouldn't have been successful.*

***I** express my deep sense of regards and indebtedness to my guide Dr. A. Goswami for his valuable guidance, continuous encouragement and wholehearted support, which were of immense help to me in completing the project. In spite of his hectic schedule he has always extended his help and invaluable suggestions.*

***I** express my thanks to Dr .S. Nanda , HOD, Department of Mathematics and to all the faculty members and the staff of the Department of Mathematics for their continuous help*

***W**ords are not enough to express my indebtedness and gratitude towards my parents to whom I owe every success and achievements of my life. Their constant support and encouragement under all odds that has brought me where I stand today.*

***F**inally I thank all my friends for their love, cheerful encouragements and valuable criticism.*

Table of Content

Section Description Page No

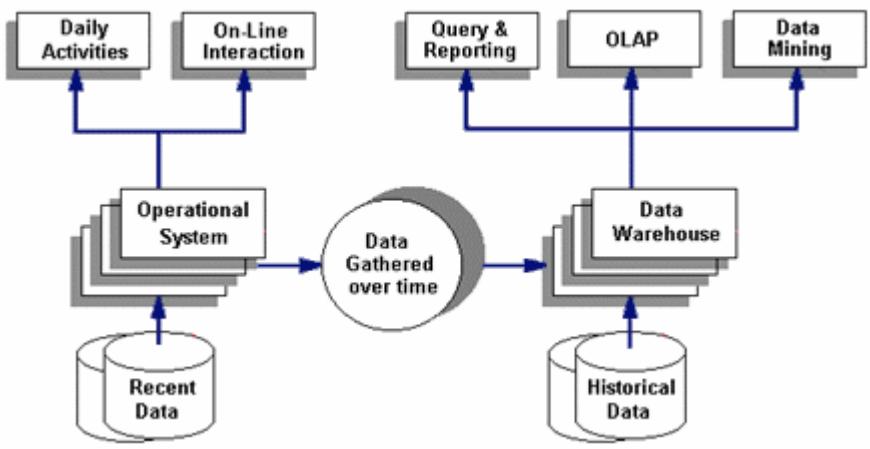
Certificate……..…..3
Acknowledgement ……..……4
Table of Content ……..…5
1. Introduction ……..…..6
1.1. What is Data Mining ……..…8
1.2. Data Mining Functions ……..……..10
1.3. Data Mining Techniques ……..10
1.4. Overview of Decision Trees ……..…10
1.4.1. Appropriate Problems for Decision Tree learning …….11
1.4.2. Decision Tree representation ……..……11
2. ID3: Decision Tree Learning Algorithm ……14
2.1. Data Description ……..……16
2.2. Attribute Selection ……16
2.3. The Basic ID3 Algorithm ……19
2.4. Comments on the ID3 Algorithm ……..……19
2.5. Decision Tree Rules and Pruning ……..……21
2.5.1. Rule Generation ……..21
2.5.2. Rule Simplification Overview ……21
2.5.3. Contingency Table ……..22
2.5.4. Test for Independence ……..…22
2.6. Example: Factor effecting Sunburns……..……23
2.7. Occam's Razor ……..……29
2.8. Limitations with ID3 ……..…30
3. SLIQ ……..…..31
3.1. Overview ……..……..31
3.2. Pre-sorting and Breadth-first growth ……..……32
3.3. Processing Node Splits ……..……32
3.4. Updating the Class List ……..……34
3.5. An Optimization ……..……..34
3.6. Subsetting for Categorical Attribute ……..……34
4. SPRINT ……..……36
4.1. Serial Algorithm ……..…36
4.1.1. Data Structures ……..……38
4.1.2. Finding Split Points ……..……39
4.1.3. Performing the Split ……..……40
4.2. Parallel Classification ……..……...41
4.2.1. Data Placement and Workload Balancing ……...41
4.2.2. Finding split Points ……..…42
4.2.3. Performing the Split ……..…43
5. Performance Result ……..……44
6. Conclusion ……..…48
References ……..…50

1.0 INTRODUCTION

Over the past three decades, computers have been used to capture details of business transactions such as banking and credit card records, retail sales, manufacturing warranty, telecommunications, etc. The data from these transactional systems have thumb prints of the key trends that impact various aspects of each business -- e.g. products that sell together, sources of profits, factors that affect manufacturing quality, etc. This data is gathered over time and stored in a separate database called a *data warehouse*.

While operational data deals with daily activities, the warehouse data is historical in nature and is used to obtain perspective on the business trends. In time, the insights gathered from the analysis of historical data are used to improve business decisions.

Data Mining is the "automatic" extraction of patterns of information from historical data, enabling companies to focus on the most important aspects of their business -- telling them what they did not know and had not even thought of asking.



Industry surveys clearly indicate that over 80% of Fortune 500 companies view data mining as a critical factor for business success by the year 2000. Most such companies now collect and refine massive quantities of data in data warehouses.

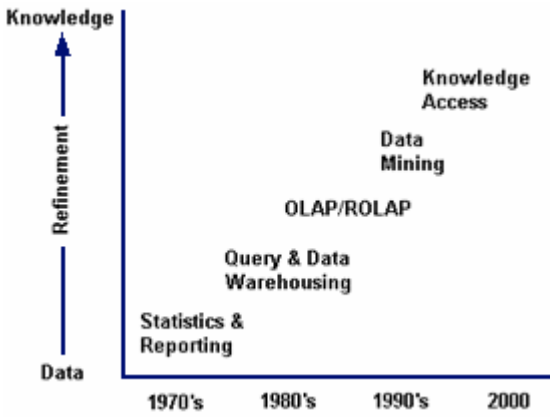
These companies realize that to succeed in a fast paced world, business users need to be able to get information on demand. And, they need to be pleasantly surprised by unexpected, but useful, information. There is never enough time to think of all the important questions -- the computer should do this itself. It can provide the winning edge in business by *exploring the database itself* and brings back invaluable nuggets of information.

Many organizations now view information as one of their most valuable assets and data mining allows a company to make full use of these information assets.

Decision Support is a broad term referring to the use of information as a strategic corporate asset, enabling companies to utilize their databases to make better decisions. Decision support systems have traditionally relied on three types of analyses:

- ***Query and Reporting:*** where a user asks a question, e.g. "what were the sales for a specific product?"
- ***OLAP* (On Line Analytical Processing):** which amounts to the processing of queries along multiple dimensions such as state, month, etc? For instance: "categorize sales by month by brand, by store-chain."
- ***Data Mining:*** which provides influence factors and relationships in data, e.g. “what impacts sales in New York for a given product?"

We can view the progress of the field over the last 30 years in terms of a series of steps, each providing better and more refined information.



With statistics and reports just summaries of data were available to business users. And, the data could only be obtained by request from an analyst, e.g. sales summaries per quarter. With data warehouses, some query and reporting could be performed by business users on their own, e.g. product and store performance reports, etc.

With OLAP, multi-dimensional summary questions could be addressed by business users, e.g. the total of sales by product, by channel, by month. With data mining, analysts and a sophisticated subset of business users could gain insight into the influence factors and trends in data. But often significant analysis was needed before key questions could be answered.

With knowledge access, almost all the relevant patterns in the data are found beforehand, and stored for use by business users such as marketing analysts, bank branch managers, store managers, etc. Business users get the interesting patterns of change every week or month or can query the pattern-base at will.

Because large databases often provide too much of a good thing, approaches based on Query and OLAP usually encounter a problem known as "The Maze of a Million Graphs" -- a user can build a million pie charts and yet not see the forest for the trees because there is so much data. Data mining, on the other hand, draws its power from the ability to search through the data with its own initiative, discovering key patterns by itself.
Although the three approaches above are useful, they share a common trait in that the user has to perform analysis to gain knowledge; this is called the *Data Analysis Paradigm*. A novel and unique approach to empowering business users with refined information is the *Knowledge Access Paradigm* pioneered by Information Discovery, Inc. With the knowledge access paradigm data analysis is performed beforehand and the user just looks up the pre-mined knowledge on demand.

1.1 WHAT IS DATA MINING?

The past two decades has seen a dramatic increase in the amount of information or data being stored in electronic format. This accumulation of data has taken place at an explosive rate. It has been estimated that the amount of information in the world doubles every 20 months and the size and number of databases are increasing even faster. The increase in use of electronic data gathering devices such as point-of-sale or remote sensing devices has contributed to this explosion of available data.

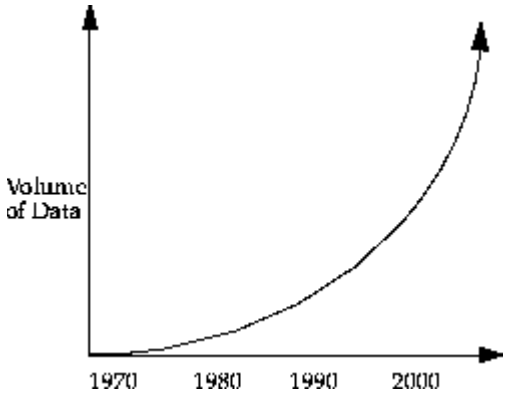


Figure: The Growing Base of Data

Data storage became easier as the availability of large amounts of computing power at low cost i.e. the cost of processing power and storage is falling, made data cheap. There was also the introduction of new machine learning methods for knowledge representation based on logic programming etc. in addition to traditional statistical analysis of data. The new methods tend to be computationally intensive hence a demand for more processing power.

Having concentrated so much attention on the accumulation of data the problem was what to do with this valuable resource? It was recognized that information is at the heart of business operations and that decision-makers could make use of the data stored to gain valuable insight into the

business. Database Management systems gave access to the data stored but this was only a small part of what could be gained from the data. Traditional on-line transaction processing systems, OLTPs, are good at putting data into databases quickly, safely and efficiently but are not good at delivering meaningful analysis in return. Analyzing data can provide further knowledge about a business by going beyond the data explicitly stored to derive knowledge about the business. This is where Data Mining or Knowledge Discovery in Databases (KDD) has obvious benefits for any enterprise.

The term data mining has been stretched beyond its limits to apply to any form of data analysis.

Data mining refers to "using a variety of techniques to identify nuggets of information or decision-making knowledge in bodies of data, and extracting these in such a way that they can be put to use in the areas such as decision support, prediction, forecasting and estimation. The data is often voluminous, but as it stands of low value as no direct use can be made of it; it is the hidden information in the data that is useful"

Some of the stages/processes identified in data mining and knowledge discovery by Usama Fayyad & Evangelos Simoudis, two of leading exponents of this area are:

- **Selection** - selecting or segmenting the data according to some criteria e.g. all those people who own a car, in this way subsets of the data can be determined.

- **Preprocessing** - this is the data cleansing stage where certain information is removed which is deemed unnecessary and may slow down queries for example unnecessary to note the sex of a patient when studying pregnancy. Also the data is reconfigured to ensure a consistent format as there is a possibility of inconsistent formats because the data is drawn from several sources e.g. sex may recorded as f or m and also as 1 or 0.

- **Transformation** - the data is not merely transferred across but transformed in that overlays may added such as the demographic overlays commonly used in market research. The data is made useable and navigable.

- **Data mining** - this stage is concerned with the extraction of patterns from the data. A pattern can be defined as given a set of facts(data) *F*, a language *L*, and some measure of certainty *C* a pattern is a statement *S* in *L* that describes relationships among a subset *Fs* of *F* with a certainty *c* such that *S* is simpler in some sense than the enumeration of all the facts in *Fs*.

- **Interpretation and evaluation** - the patterns identified by the system are interpreted into knowledge which can then be used to support human decision-making e.g. prediction and classification tasks,

summarizing the contents of a database or explaining observed phenomena.

Data mining research has drawn on a number of other fields such as inductive learning, machine learning and statistics etc.

1.2 DATA MINING FUNCTIONS

Data mining methods may be classified by the function they perform or according to the class of application they can be used in. Some of the main techniques used in data mining are described in this section.

1. Classification
2. Associations
3. Sequential/Temporal patterns
4. Clustering/Segmentation

1.3 DATA MINING TECHNIQUES

The various data mining techniques which are used in current days are:

- *Cluster Analysis*
- *Induction*
  - *Decision trees*
  - *Rule induction*
- *Neural networks*
- *On-line Analytical processing*
- *Data Visualization*

1.4 OVERVIEW OF DECISION TREES

Robust to noisy data and capable of learning disjunctive expressions, **decision tree learning**, a method for approximating discrete-valued target functions, is one of the most widely used and practical methods for inductive inference. Decision tree is the process of representing the induced knowledge by selecting the best attribute to obtain the compact tree with the high predictive accuracy.

- **Input:** a database of training records
  - Class label Attribute(s)
  - Predictor Attributes
- **Goal**
  - To build a concise model of the distribution of class label in terms of predictor attributes
- **Applications**
  - Scientific experiments, medical diagnosis, fraud detection, etc.

1.4.1 APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

Decision tree learning is generally best suited to problems with the following characteristics:

- Instances are represented by **attribute-value pairs**.
  - Instances are described by a fixed set of attributes (e.g., temperature) and their values (e.g., hot).
  - The easiest situation for decision tree learning occurs when each attribute takes on a small number of disjoint possible values (e.g., hot, mild, cold).
  - Extensions to the basic algorithm allow handling real-valued attributes as well (e.g., a floating point temperature).
- The target function has **discrete output values**.
  - A decision tree assigns a classification to each example.
    - Simplest case exists when there are only two possible classes (**Boolean classification**).
    - Decision tree methods can also be easily extended to learning functions with more than two possible output values.
  - A more substantial extension allows learning target functions with real-valued outputs, although the application of decision trees in this setting is less common.
- Disjunctive descriptions may be required.
  - Decision trees naturally represent disjunctive expressions.
- The training data may contain errors.
  - Decision tree learning methods are robust to errors - both errors in classifications of the training examples and errors in the attribute values that describe these examples.
- The training data may contain missing attribute values.
  - Decision tree methods can be used even when some training examples have unknown values (e.g., humidity is known for only a fraction of the examples).

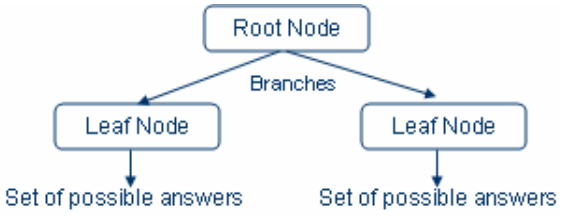
1.4.2 DECISION TREE REPRESENTATION

Decision tree is one of the most attractive classification models. Decision trees classify **instances** by sorting them down the tree from the **root node** to some **leaf node**, which provides the classification of the instance. Each node in the tree specifies a **test** of some **attribute** of the instance, and each **branch** descending from that node corresponds to one of the possible **values** for this attribute, i.e. decision tree has

- A set of internal nodes which are decisions, where each node tests the value of an attribute and branches on all possible values.
- A set of leaves which are labels, where each leaf gives a class value.

An instance is classified by starting at the root node of the decision tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute. This process is then repeated at the node on this branch and so on until a leaf node is reached. The decision tree is represented diagrammatically as:

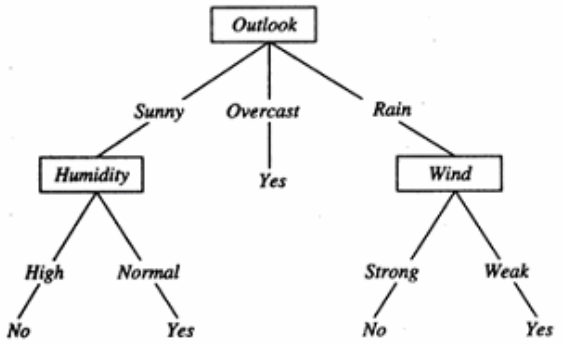
- Each non-leaf node is connected to a test that splits its set of possible answers into subsets corresponding to different test results.
- Each branch carries a particular test result's subset to another node.
- Each node is connected to a set of possible answers.



Given *m* attributes, a decision tree may have a maximum height of *m*.

For Example:

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



There are a large number of algorithms to construct decision trees

- E.g.: SLIQ, CART, C4.5, SPRINT
- Most are main memory algorithms
- Tradeoff between supporting large databases, performance and constructing more accurate decision trees

Some TDIDT (Top Down Induction of Decision Trees) Systems:

- ID3 (Quinlan 79)
- CART (Brieman 84)
- Assistant (Cestnik 87)
- C4.5 (Quinlan 93)
- See5 (Quinlan 97)
- ...
- Orange (Demšar, Zupan 98-03)

2.0 ID3: Decision Tree Learning Algorithm

Induction is the inference of information from data and inductive learning is the model building process where the environment i.e. database is analyzed with a view to finding patterns. Similar objects are grouped in classes and rules formulated whereby it is possible to predict the class of unseen objects. This process of classification identifies classes such that each class has a unique pattern of values which forms the class description. The nature of the environment is dynamic hence the model must be adaptive i.e. should be able learn.

Generally it is only possible to use a small number of properties to characterize objects so we make abstractions in that objects which satisfy the same subset of properties are mapped to the same internal representation.

Inductive learning where the system infers knowledge itself from observing its environment has two main strategies:

- **Supervised learning** - this is learning from examples where a teacher helps the system construct a model by defining classes and supplying examples of each class. The system has to find a description of each class i.e. the common properties in the examples. Once the description has been formulated the description and the class form a classification rule which can be used to predict the class of previously unseen objects. This is similar to discriminate analysis as in statistics.

- **Unsupervised learning** - this is learning from observation and discovery. The data mine system is supplied with objects but no classes are defined so it has to observe the examples and recognize patterns (i.e. class description) by itself. This system results in a set of class descriptions, one for each class discovered in the environment. Again this similar to cluster analysis as in statistics.

Induction is therefore the extraction of patterns. The quality of the model produced by inductive learning methods is such that the model could be used to predict the outcome of future situations in other words not only for states encountered but rather for unseen states that could occur. The problem is that most environments have different states, i.e. changes within, and it is not always possible to verify a model by checking it for all possible situations.

Given a set of examples the system can construct multiple models some of which will be simpler than others. The simpler models are more likely to be correct if we adhere to Occams razor, which states that if there are multiple explanations for a particular phenomenon it makes sense to choose the simplest because it is more likely to capture the nature of the phenomenon.

Here in this section we will try to understand one of the most primitive and simplest classifier (decision tree) used in data mining. This classification algorithm is known as ID3. J. Ross Quinlan originally developed ID3 at the University of Sydney. He first presented ID3 in 1975 in a book, *Machine Learning*, vol. 1, no. 1. ID3 is based off the **Concept Learning System (CLS)** algorithm. Very simply, ID3 builds a decision tree from a fixed set of examples. The resulting tree is used to classify future samples. The example has several attributes and belongs to a class (like yes or no). The leaf nodes of the decision tree contain the class name whereas a non-leaf node is a decision node. The decision node is an attribute test with each branch (to another decision tree) being a possible value of the attribute. ID3 uses information gain to help it decide which attribute goes into a decision node. The advantage of learning a decision tree is that a program, rather than a knowledge engineer, elicits knowledge from an expert. The basic CLS algorithm over a set of training instances C:

Step 1: If all instances in C are positive, then create YES node and halt.
If all instances in C are negative, create a NO node and halt.
Otherwise select a feature, F with values v_1, ..., v_n and create a decision node.

Step 2: Partition the training instances in C into subsets C_1, C_2, ..., C_n according to the values of V.

Step 3: Apply the algorithm recursively to each of the sets C_i.

Note: *The trainer (the expert) decides which feature to select*.

ID3 improves on CLS by adding a feature selection heuristic. ID3 searches through the attributes of the training instances and extracts the attribute that best separates the given examples. If the attribute perfectly classifies the training sets then ID3 stops; otherwise it recursively operates on the n (where n = number of possible values of an attribute) partitioned subsets to get their "best" attribute. The algorithm uses a greedy search, that is, it picks the best attribute and never looks back to reconsider earlier choices.

ID3 is a non-incremental algorithm, meaning it derives its classes from a fixed set of training instances. An incremental algorithm revises the current concept definition, if necessary, with a new sample. The classes created by ID3 are inductive, that is, given a small set of training instances, the specific classes created by ID3 are expected to work for all future instances. The distribution of the unknowns must be the same as the test cases. Induction classes cannot be proven to work in every case since they may classify an infinite number of instances. Note that ID3 (or any inductive algorithm) may misclassify data.

General Form

Until each leaf node is populated by as homogeneous a sample set as possible:

- Select a leaf node with an inhomogeneous sample set.
- Replace that leaf node by a test node that divides the inhomogeneous sample set into minimally inhomogeneous subsets, according to an entropy calculation.

Specific Form

- Examine the attributes to add at the next level of the tree using an entropy calculation.
- Choose the attribute that minimizes the entropy.

It is computationally impractical to find the smallest possible decision tree, so we use a procedure that tends to build small trees.

2.1 DATA DESCRIPTION

The sample data used by ID3 has certain requirements, which are:

- **Attribute-value description** - the same attributes must describe each example and have a fixed number of values.
- **Predefined classes** - an example's attributes must already be defined, that is, they are not learned by ID3.
- **Discrete classes** - classes must be sharply delineated. Continuous classes broken up into vague categories such as a metal being "hard, quite hard, flexible, soft, quite soft" are suspect.
- **Sufficient examples** - since inductive generalization is used (i.e. not provable) there must be enough test cases to distinguish valid patterns from chance occurrences.

2.2 ATTRIBUTE SELECTION

How does ID3 decide which attribute is the best? The central focus of the ID3 algorithm is selecting which attribute to test at each node in the tree. A statistical property, called Entropy is used. This idea is borrowed from information theory.

Procedure:

1. See how the attribute distributes the instances.

2. Minimize the average entropy.
    - Calculate the average entropy of each test attribute and choose the one with the lowest degree of entropy.

♦ Review of Log_2

On a calculator: $$\log_2(x) = \frac{\ln(x)}{\ln(2)} = \frac{\log_{10}(x)}{\log_{10}(2)}$$

e.g.) $(0)\log_2(0) = -\infty$ (the formula is no good for a probability of 0)
e.g.) $(1)\log_2(1) = 0$
e.g.) $\log_2(2) = 1$
e.g.) $\log_2(4) = 2$
e.g.) $\log_2(1/2) = -1$
e.g.) $\log_2(1/4) = -2$
e.g.) $(1/2)\log_2(1/2) = (1/2)(-1) = -1/2$

♦ Entropy Formulae

Entropy, a measure from information theory, characterizes the (im) purity, or homogeneity, of an arbitrary collection of examples.

Given:

- n_b, the number of positive instances in branch *b*.
- n_{bc}, the total number of instances in branch *b* of class *c*.
- n_t, the total number of instances in all branches.

♦ Probability

$$P_b = \text{Probability an instance on a branch } b \text{ is positive}$$

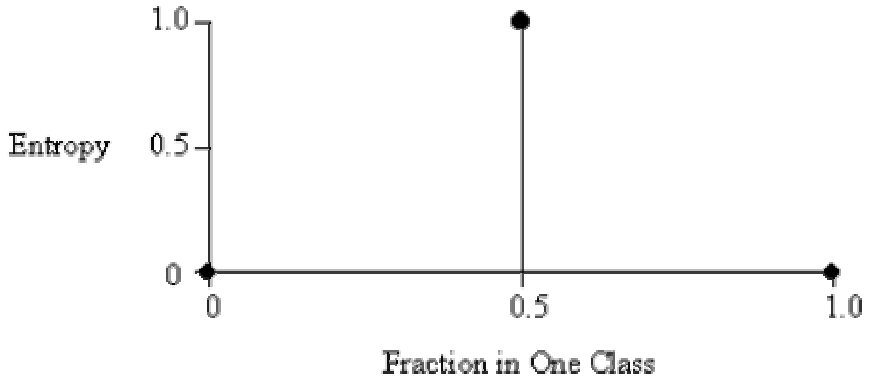
$$= \frac{\text{number of positive instances on branch}}{\text{total number of instances on branch}} = \frac{n_{bc}}{n_b}$$

- If all the instances on the branch are positive, then $P_b = 1$ (homogeneous positive)
- If all the instances on the branch are negative, then $P_b = 0$ (homogeneous negative)

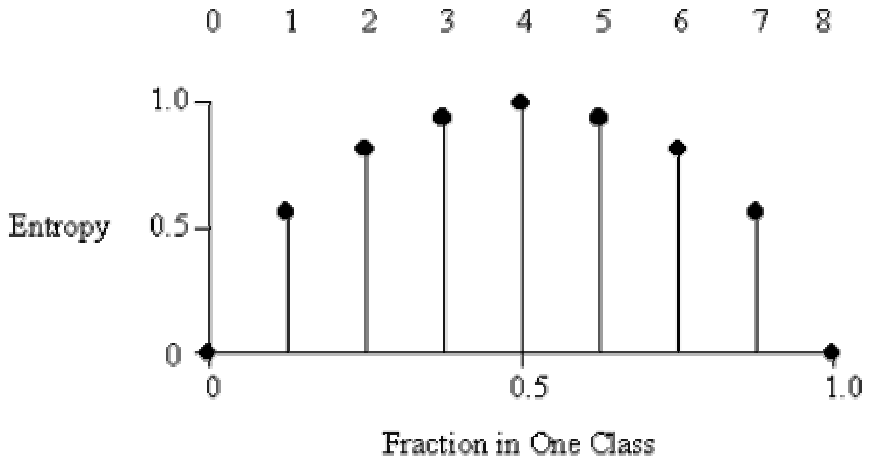
♦ Entropy

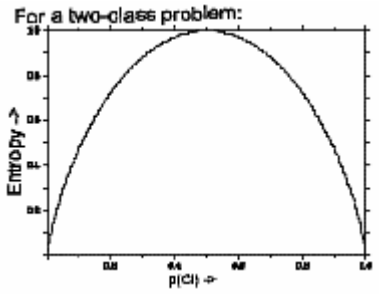
$$\text{Entropy} = \sum_c -\left(\frac{n_{bc}}{n_b}\right)\log_2\left(\frac{n_{bc}}{n_b}\right)$$

- As you move from perfect balance and perfect homogeneity, entropy varies smoothly between zero and one.
    - The entropy is zero when the set is perfectly homogeneous.
    - The entropy is one when the set is perfectly inhomogeneous.



The disorder in a set containing members of two classes A and B, as a function of the fraction of the set belonging to class A. In the diagram above, the total number of instances in both classes combined is two. In the diagram below, the total number of instances in both classes is eight.





♦ Average Entropy

$$\text{Average Entropy} = \sum_{b} \left(\frac{n_b}{n_t}\right) \times \left[\sum_{c} -\left(\frac{n_{bc}}{n_b}\right) \log_2 \left(\frac{n_{bc}}{n_b}\right) \right]$$

2.3 THE BASIC ID3 ALGORITHM

- **ID3(Examples, Target, Attributes)**
- Create a root node
- If all Examples have the same Target value, give the root that label
- Else if Attributes is empty label, the root according to the most common value
- Else begin
  - Calculate the information gain for each attribute, according to the average entropy formula
  - Select the attribute, **A**, with the lowest average entropy (highest information gain) and make this the attribute tested at the root
  - For each possible value, **v**, of this attribute
    - Add a new branch below the root, corresponding to **A** = **v**
    - Let Examples(**v**) be those examples with **A** = **v**
    - If Examples(**v**) is empty, make the new branch a leaf node labeled with the most common value among Examples
    - Else let the new branch be the tree created by **ID3**(Examples(**v**), Target, Attributes - {**A**})
- end
- Return root

2.4 COMMENTS ON THE ID3 ALGORITHM

Unlike the version space candidate-elimination algorithm,

- ID3 searches a *completely* expressive hypothesis space (ie. one capable of expressing any finite discrete-valued function), and thus avoids the difficulties associated with restricted hypothesis spaces.
- ID3 searches *incompletely* through this space, from simple to complex hypotheses, until its termination condition is met (eg. until it finds a hypothesis consistent with the data).
- ID3's inductive bias is based on the ordering of hypotheses by its search strategy (ie. follows from its search strategy).
- ID3's *hypothesis space* introduces no additional bias.

Approximate inductive bias of ID3: shorter trees are preferred over larger trees.

A closer approximation to the inductive bias of ID3: shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

By viewing ID3 in terms of its search space and search strategy, we can get some insight into its capabilities and limitations:

- ID3's hypothesis space of all decision spaces is a *complete* space of finite discrete-valued functions, relative to the available attributes.
    - Because every finite discrete-valued function can be represented by some decision tree, ID3 avoids one of the major risks of methods that search incomplete hypothesis spaces (such as version space methods that consider only conjunctive hypotheses): that the hypothesis space might not contain the target function.
- ID3 maintains only a *single current hypothesis* as it searches through the space of decision trees.
    - This contrasts, for example, with the earlier version space candidate-elimination method, which maintains the set of *all* hypotheses consistent with the available training examples.
    - However, by determining only a single hypothesis, ID3 loses the capabilities that follow from explicitly representing all consistent hypotheses.
        - For example, it does not have the ability to determine how many alternative decision trees are consistent with the available training data, or to pose new instance queries that optimally resolve among these competing hypotheses.
- ID3, in its pure form, performs *no backtracking* in its search (**greedy algorithm**).
    - Once it selects an attribute to test at a particular level in the tree, it never backtracks to reconsider this choice; it is susceptible to the usual risks of hill-climbing search without backtracking: converging to locally optimal solutions that are not globally optimal.
        - In the case of ID3, a locally optimal solution corresponds to the decision tree it selects along the single search path it explores. However, this locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search.

2.5 DECISION TREE RULES & PRUNING

The rule generation is a vital step in any classification problem. This helps to make the result easily understandable by common persons. The pruning process helps in reducing the conditions in a rule by eliminating the unnecessary conditions.

2.5.1 RULE GENERATION:

Once a decision tree has been constructed, it is a simple matter to convert it into an equivalent set of rules.

Converting a decision tree to rules before pruning has three main advantages:

1. Converting to rules allows distinguishing among the different contexts in which a decision node is used.
    - Since each distinct path through the decision tree node produces a distinct rule, the pruning decision regarding that attribute test can be made differently for each path.
    - In contrast, if the tree itself were pruned, the only two choices would be:
        1. Remove the decision node completely, or
        2. Retain it in its original form.
2. Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves.
    - We thus avoid messy bookkeeping issues such as how to reorganize the tree if the root node is pruned while retaining part of the sub tree below this test.
3. Converting to rules improves readability.
    - Rules are often easier for people to understand.

To generate rules, trace each path in the decision tree, from root node to leaf node, recording the test outcomes as antecedents and the leaf-node classification as the consequent.

2.5.2 RULE SIMPLIFICATION OVERVIEW

Once a rule set has been devised:

1. Eliminate unnecessary rule antecedents to simplify the rules.
    - Construct contingency tables for each rule consisting of more than one antecedent.
        - Rules with only one antecedent cannot be further simplified, so we only consider those with two or more.
    - To simplify a rule, eliminate antecedents that have no effect on the conclusion reached by the rule.
    - A conclusion's independence from an antecedents is verified using a test for independency, which is

- **A chi-square test** if the expected cell frequencies are greater than 10.
- **Yates' Correction for Continuity** when the expected frequencies are between 5 and 10.
- **Fisher's Exact Test** for expected frequencies less than 5.

2. Eliminate unnecessary rules to simplify the rule set.
   - Once individual rules have been simplified by eliminating redundant antecedents, simplify the entire set by eliminating unnecessary rules.
   - Attempt to replace those rules that share the most common consequent by a **default rule** that is triggered when no other rule is triggered.
     - In the event of a tie, use some heuristic tie breaker to choose a default rule.

2.5.3 CONTINGENCY TABLES

The following is a **contingency table**, a tabular representation of a rule.

	C_1	**C_2**	**Marginal Sums**
R_1	x_1	x_2	$R_{1T} = x_1 + x_2$
R_2	x_3	x_4	$R_{2T} = x_3 + x_4$
Marginal Sums	$C_{1T} = x_1 + x_3$	$C_{2T} = x_2 + x_4$	$T = x_1 + x_2 + x_3 + x_4$

R_1 and R_2 represent the Boolean states of an antecedent for the conclusions C_1 and C_2 (C_2 is the negation of C_1). x_1 through x_4 represent the frequencies of each antecedent-consequent pair. R_{1T}, R_{2T}, C_{1T}, C_{2T} are the **marginal sums** of the rows and columns, respectively.

The marginal sums and T, the total frequency of the table, are used to calculate expected cell values in step 3 of the test for independence.

2.5.4 TEST FOR INDEPENDENCE

Given a contingency table of dimensions r by c (rows x columns):

1. Calculate and fix the sizes of the marginal sums.
2. Calculate the total frequency, T, using the marginal sums.
3. Calculate the expected frequencies for each cell.
   The general formula for obtaining the expected frequency of any cell in a contingency table is given by:

$$e_n = \frac{R_{n_T} \cdot C_{n_T}}{T}$$

   Where R_{nT} and C_{nT} are the corresponding row and column totals for contingency table cell n.
4. Select the test to be used to calculate χ^2 based on the highest expected frequency, m:

if	then use
$m > 10$	Chi-Square Test
$5 \leq m \leq 10$	Yates' Correction for Continuity
$m < 5$	Fisher's Exact Test

5. Calculate χ^2 using the chosen test.
6. Calculate the degrees of freedom (df).
   df = (r - 1)(c - 1)
7. Use a chi-square table with χ^2 and df to determine if the conclusions are independent from the antecedent at the selected level of significance, α.
   - Assume $\alpha = 0.05$ unless otherwise stated.
   - If $\chi^2 > \chi^2_\alpha$
     - Reject the null hypothesis of independence and accept the alternate hypothesis of dependence.
       - We keep the antecedents because the conclusions are dependent upon them.
   - If $\chi^2 \leq \chi^2_\alpha$
     - Accept the null hypothesis of independence.
       - We discard the antecedents because the conclusions are independent from them.

- **Chi-Square Test**

$$\chi^2 = \sum_i \frac{(o_i - e_i)^2}{e_i}$$

- **Yates' Correction for Continuity**

$$\chi^2(\text{corrected}) = \sum_i \frac{(|o_i - e_i| - 0.5)^2}{e_i}$$

2.6 EXAMPLE: *FACTORS AFFECTING SUNBURN*

Given Data:

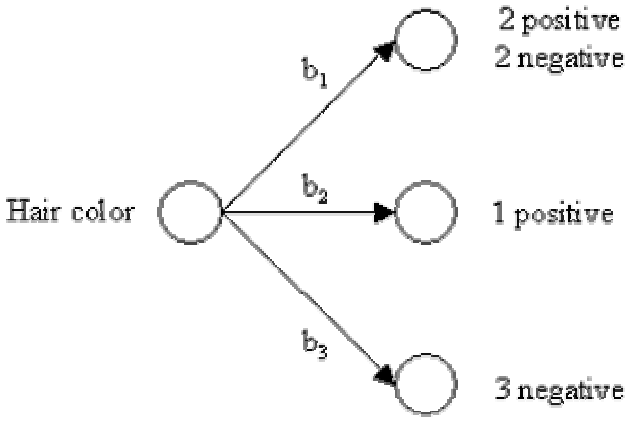
Independent Attributes / Condition Attributes — Dependent Attributes / Decision Attributes

Name	Hair	Height	Weight	Lotion	Result
Sarah	blonde	average	light	no	sunburned (positive)
Dana	blonde	tall	average	yes	none (negative)
Alex	brown	short	average	yes	none
Annie	blonde	short	average	no	sunburned

Emily	red	average	heavy	no	sunburned
Pete	brown	tall	heavy	no	none
John	brown	average	heavy	no	none
Katie	blonde	short	light	yes	none

Phase 1: From Data to Tree

1. Perform average entropy calculations on the complete data set for each of the four attributes:



b_1 = blonde Average Entropy = 0.50
b_2 = red
b_3 = brown

Sample average entropy calculation for the attribute "hair color"

$$= \sum_b \left(\frac{n_b}{n_t}\right) \times \left[\sum_c -\left(\frac{n_{bc}}{n_b}\right)\log_2\left(\frac{n_{bc}}{n_b}\right)\right]$$

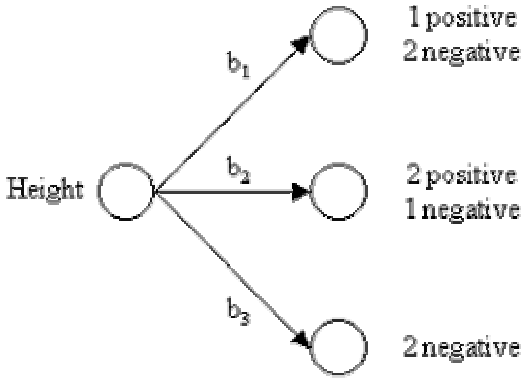
$$= \left(\frac{4}{8}\right) \times \left[-\left(\frac{2}{4}\right)\log_2\left(\frac{2}{4}\right) - \left(\frac{2}{4}\right)\log_2\left(\frac{2}{4}\right)\right]$$

$$+ \left(\frac{1}{8}\right) \times \left[(1)\log_2\left(\frac{1}{1}\right)\right]$$

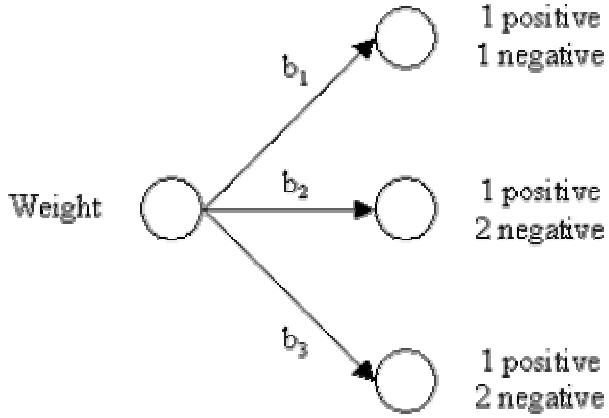
$$+ \left(\frac{3}{8}\right) \times \left[\left(\frac{3}{3}\right)\log_2\left(\frac{3}{3}\right)\right]$$

$$= \left(\frac{4}{8}\right) \times \left[-\left(\frac{2}{4}\right)\log_2\left(\frac{2}{4}\right) - \left(\frac{2}{4}\right)\log_2\left(\frac{2}{4}\right)\right]$$

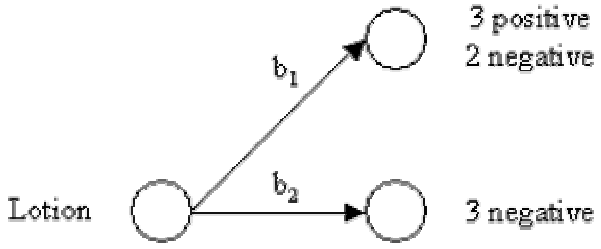
$$= 0.50$$



b_1 = short Average Entropy = 0.69
b_2 = average
b_3 = tall



b_1 = light Average Entropy = 0.94
b_2 = average
b_3 = heavy

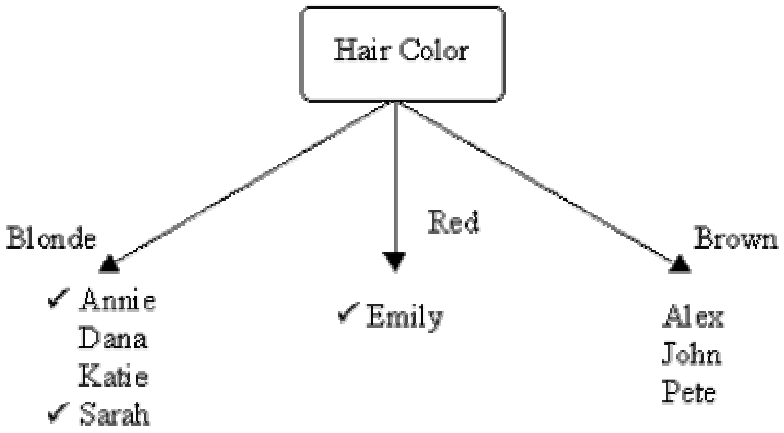


b_1 = no Average Entropy = 0.61
b_2 = yes

Results:

Attribute	Average Entropy
Hair Color	**0.50**
Height	0.69
Weight	0.94
Lotion	0.61

The attribute "hair color" is selected as the first test because it minimizes the entropy.



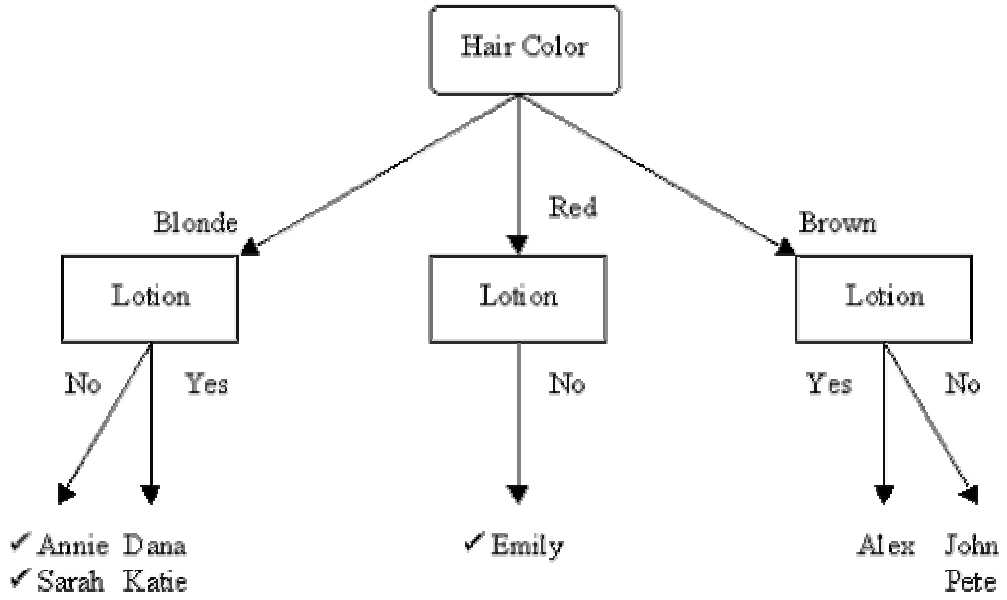
Similarly, we now choose another test to separate out the sunburned individuals from the blonde haired inhomogeneous subset, {Sarah, Dana, Annie, and Katie}.

Results:

Attribute	Average Entropy
Height	0.50
Weight	1.00
Lotion	**0.00**

The attribute "lotion" is selected because it minimizes the entropy in the blonde hair subset.

Thus, using the "hair color" and "lotion" tests together ensures the proper identification of all the samples.



This is the completed decision tree.

Phase 2: From Tree to Rules

We may now establish rules from the decision tree.

Rule	If	then
1	the person's hair color is blonde the person uses no lotion	The person gets sunburned.
2	the person's hair color is blonde the person uses lotion	Nothing happens.
3	the person's hair color is red	The person gets sunburned.
4	the person's hair color is brown	Nothing happens.

Phase 3: Simplify Rules

Once a rule set has been generated, simplify it.

[The training data has been multiplied by a factor of 4 to permit the use of the chi-square test below]

Assume a significance level of $\alpha = 0.05$

a) **Eliminate Unnecessary Rule Antecedents**

1. Consider the two antecedents in Rule #1: blonde hair and lotion.

1. Blonde

Actual:

	No Change	Sunburned	Marginal Sum
Blonde	8	8	16
Not Blonde	12	4	16
Marginal Sum	20	12	32

Expected:

Sample expected value calculation for contingency cell 1:

$$e_1 = \frac{R_{1_T} \cdot C_{1_T}}{T} = \frac{(16)(20)}{32} = 10$$

	No Change	Sunburned
Blonde	10	6
Not Blonde	10	6

Sample χ^2 calculation, where o_i and e_i are the observed (actual) and expected values for cell i, respectively, and $1 \leq i \leq 4$:

$$\chi^2 = \sum_i^4 \frac{(o_i - e_i)^2}{e_i}$$

$$= \frac{(8-10)^2}{10} + \frac{(8-6)^2}{6} + \frac{(12-10)^2}{10} + \frac{(4-6)^2}{6}$$

$$= \frac{4}{10} + \frac{4}{6} + \frac{4}{10} + \frac{4}{6} = \frac{32}{15} = 2.13$$

df = (r - 1)(c - 1) = (2 - 1)(2 - 1) = 1

From the chi-square table, $\chi^2_\alpha = 3.84$

Since $\chi^2 < \chi^2_\alpha$, we accept the null hypothesis of independence, H_0.

We thus conclude, according to the training examples, that sunburn is independent from blonde hair, and thus we may eliminate this antecedent from Rule #1 and Rule #2.

2. Lotion

For argument's sake we will also examine the lotion antecedent for independence.

Actual:

	No Change	Sunburned	Marginal Sum
Lotion	12	0	12
No Lotion	8	12	20
Marginal Sum	20	12	32

Expected:

	No Change	Sunburned
Lotion	7.5	4.5
No Lotion	12.5	7.5

χ^2= 11.52

df = 1

From the chi-square table, χ^2_α= 3.84

Since $\chi^2 > \chi^2_\alpha$, we reject the null hypothesis of independence, H_0, and accept the alternate hypothesis of dependence, H_a.

Therefore, according to the training examples, sunburn is clearly dependent upon the use of lotion, so we cannot eliminate this antecedent.

b) **Eliminate Unnecessary Rules** (under construction)

The tentative rule set is as follows:

Rule	If	then
1	the person uses no lotion	the person gets sunburned.
2	the person uses lotion	nothing happens.
3	the person's hair color is red	the person gets sunburned.
4	the person's hair color is brown	nothing happens.

ID3 prefers trees that are shorter and place the attribute tests that minimize entropy the most near the top of the tree. This yields short hypotheses that satisfy Occam's Razor.

2.7 OCCAM'S RAZOR

It is a logical principle attributed to the mediaeval philosopher William Occam (or Ockham). The principle states that one should not make more assumptions than the minimum needed. This principle is often called the

principle of parsimony. It underlies all scientific modeling and theory building. It admonishes us to choose from a set of otherwise equivalent models of a given phenomenon the simplest one. In any given model, Occam's razor helps us to "shave off" those concepts, variables or constructs that are not really needed to explain the phenomenon. By doing that, developing the model will become much easier, and there is less chance of introducing inconsistencies, ambiguities and redundancies.

Though the principle may seem rather trivial, it is essential for model building because of what is known as the "under determination of theories by data". For a given set of observations or data, there is always an infinite number of possible models explaining those same data. This is because a model normally represents an infinite number of possible cases, of which the observed cases are only a finite subset. The non-observed cases are inferred by postulating general rules covering both actual and potential observations.

For example, through two data points in a diagram you can always draw a straight line, and induce that all further observations will lie on that line. However, you could also draw an infinite variety of the most complicated curves passing through those same two points, and these curves would fit the empirical data just as well. Only Occam's razor would in this case guide you in choosing the "straight" (i.e. linear) relation as best candidate model. A similar reasoning can be made for *n* data points lying in any kind of distribution.

Those who support Occam's Razor claim that smaller hypotheses will be less likely to over fit data. However, there has been research that suggests that adhering to Occam's Razor does not yield the best hypotheses.

2.8 LIMITATIONS WITH ID3

Sometimes decision trees conform to their training set too tightly. This occurs when coincidences or errors in the training set yield a tree that will not correctly classify other instances. In such cases the tree is not general enough and the data is said to be over-fitted. Also, ID3 is designed to operate on discrete-valued functions and does not deal with continuous-valued functions. Furthermore, it does not handle situations in which attributes do not have specified. Moreover, in some cases, ID3 is not computationally efficient. To deal with these limitations Quinlan proposed an extended algorithm: C4.5.

3.0 SLIQ

Classification is an important problem in the emerging field of data mining. Although classification has been studied extensively in the past, most of the classification algorithms are designed only for memory-resident data, thus limiting their suitability for data mining large data sets. We will discuss the design of a fast scalable classifier called SLIQ. SLIQ is a decision tree classifier that can handle both numerical and categorical attributes. **SLIQ** stands for **S**upervised **L**earning **I**n **Q**uest, where Quest is the Data Mining project of the IBM Almaden Research Center. It uses a novel pre-sorting technique in the tree growing phase. This sorting procedure is integrated with a breadth-first tree growing strategy to enable classification of disk-resident datasets. It uses a new tree-pruning algorithm –"Minimum description Length", which is inexpensive and results in compact and accurate tree. The combination of these techniques enables SLIQ to scale for large data sets and classify data sets irrespective of the number of classes and attributes, thus making it an attractive tool for data mining.

In data mining applications, very large training sets with several million examples are common. Our primary motivation is to study classifiers that scales well and can handle training data of this magnitude. The ability to classify large training data can also improve the classification accuracy. Decision tree classifiers are relatively fast compared to other classification methods. A decision tree can be converted into simple and easy to understand classification rules. They can also be converted to SQL queries for accessing databases.

3.1 OVERVIEW

SLIQ is a decision tree classifier that can handle both numerical and categorical attributes. SLIQ uses a pre-sorting technique in the tree-growing phase to reduce the cost of evaluating numerical attributes. This sorting technique is integrated with breadth-first tree growing strategy to enable SLIQ to classify disk-resident datasets. In addition SLIQ uses a fast subsetting algorithm for determining splits for categorical attributes. SLIQ uses a new tree-pruning algorithm –"Minimum description Length", which is inexpensive and results in compact and accurate tree.

While growing the tree, the goal at each node is to determine the split point that "best" divides the training records belonging to that leaf. The value of a split point depends upon how well it separates the classes. Several splitting indices have been proposed in the past to evaluate the goodness of the split. The gini index is used here. For a data set S containing examples from n classes, gini(S) is defined as:

$$\mathbf{gini(S) = 1 - \sum p_j^2}$$

Where p_j is the relative frequency of class j in S. If a split divides S into two subsets S_1 and S_2, the index of the divided data $gini_{split}$ (S) is given by:

$$gini_{split}(S) = (n_1/n)\, gini(S_1) + (n_2/n)\, gini(S_2)$$

3.2 PRE-SORTING AND BREADTH-FIRST GROWTH

For numerical attributes, sorting time is the dominating factor when finding the best split at a decision tree node. Therefore, the first technique used in SLIQ is to implement a scheme that eliminates the need to sort the data at each node of the decision tree. Instead the training data are sorted just once for each numerical attributes at the beginning of the tree growth phase.

To achieve this pre-sorting, the following data structure is used. We create a separate list for each attribute of the training data. Additionally, a separate list, called *class list,* is created for the class labels attached to the examples. An entry in an attribute list has two fields: one containing an attribute value, the other an index into the class list. An entry in the class list also has two fields: one containing a class label, the other a reference to a leaf node of the decision tree. The i^{th} entry of the class list corresponds to the i^{th} example in the training data. Each leaf node of the decision tree represents a partition of the training data, the partition being defined by the conjunction of the predicated on the path from the node to the root. Thus, the class list can at any time identify the partition to which an example belongs. We assume that there is enough memory to keep the class list memory-resident. Attribute lists are written to disk if necessary.

Initially, the leaf reference fields of all the entries of the class list are set to point to the root of the decision tree. Then a pass is made over the training data, distributing values of the attribute for each example across the lists. Each attribute value is also tagged with the corresponding class list index. The attribute lists for the numerical features are sorted independently.

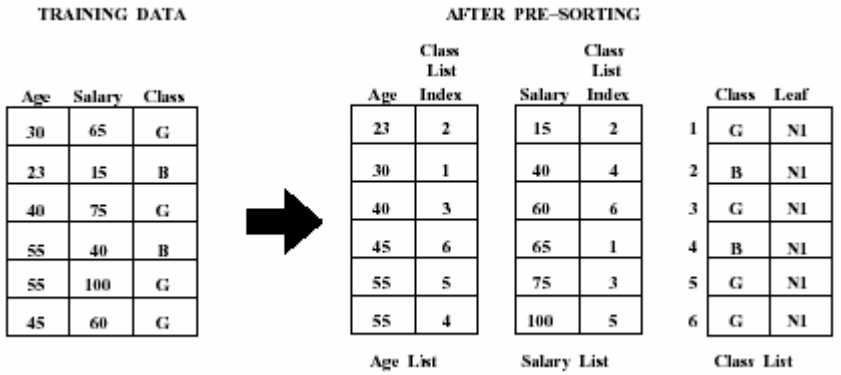


Figure: *Example of Pre-Sorting.*

3.3 PROCESSING NODE SPLITS

Rather than using a depth-first strategy used in the earlier decision tree classifiers, the tree is grown in breadth-first. Consequently, splits for all the leaves of the current tree are simultaneously evaluated in one pass over the data. To compute the gini splitting-index for an attribute at a node, we

need the frequency distribution of class values in the data partition corresponding to the node. The distribution is accumulated in a class histogram attached with each leaf node. For a numerical attribute, the histogram is a list of pairs of the form <class, frequency>. For a categorical attribute, this histogram is a list of triples of the form <attribute value, class, frequency>. Attribute lists are processed one at a time (since the attribute lists can be on disk). For each value v in the attribute list for the current attribute A, we find the corresponding entry in the class list, which yields the corresponding class and the leaf node. We now update the histogram attached with this leaf node. If A is a numerical attribute, we compute at the same time the splitting-index for the test A ≤ v for this leaf. If A is categorical attribute, we wait till the attribute list has been completely scanned and then find the subset of A with the best split. Thus, in one traversal of an attribute of an attribute list, the best split using this attribute is known for all the leaf nodes. Similarly, with one traversal of all of the attribute lists, the best overall split for al of the leaf node is known. The best split test is saved with each of the leaf nodes.

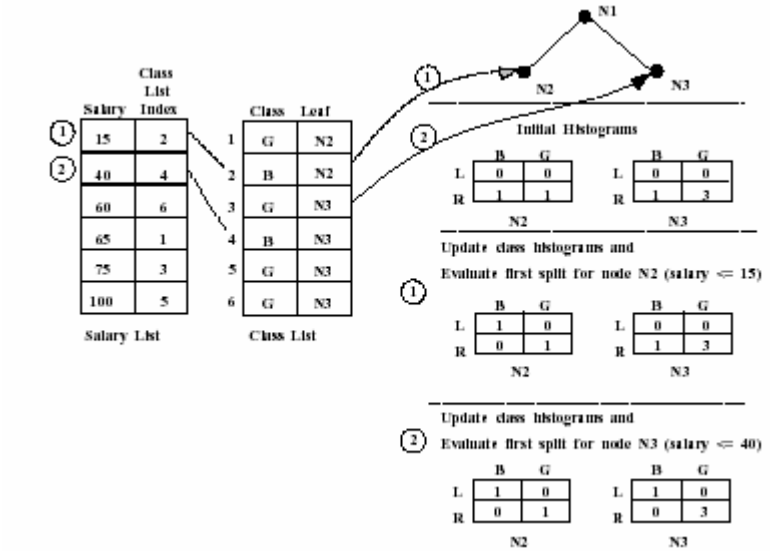


Figure: *Example of Evaluating Splits*

The above example assumes that the data has been initially split on the age attribute using the split age ≤ 35. The class histograms reflect the distribution of the points at each leaf node as a result of the split. The L values represent the distribution for examples that satisfy the test and R values represent examples that do not satisfy the test. We show how the class histogram are updated as each split is evaluated. The first value in the salary list belongs to node N2. So the first split evaluated is (salary ≤ 15) for N2. After this split, the corresponding example (salary 15, call index 2) which satisfy the predicate belongs to the left branch and the rest belong to the right branch. The class histogram of the node N2 is updated to reflect this fact. Next, the split (salary ≤ 40) is evaluated for the node N3. After the split, the corresponding example (salary 40, class index 4) belongs to the left branch and the class histogram of node N3 is updated to reflect this fact.

3.4 UPDATING THE CLASS LIST

The next step is to creat child nodes for each of the nodes and updates the class list.

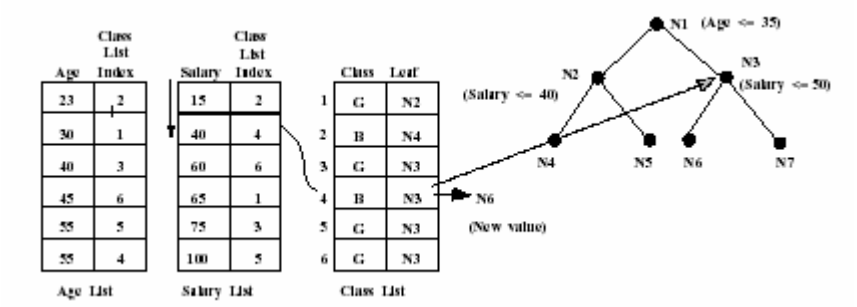


Figure: *Example of Class List Updating*

The above figure shows the class list being updated after the nodes N2 and N3 have been split on the salary attribute. The salary attribute list is being traversed and the class list entry (entry 4) corresponding to the salary value 40 is being updated. First, the leaf reference in the entry 4 of the class list is used to find the node to which the example used to belong (N3 in this case). Then, the split selected at N3 is applied to find the new child to which the example belongs (N6 in this case). The leaf reference field of the entry 4 in the class list is updated to reflect the new value.

3.5 AN OPTIMIZATION

While growing the tree, the above two steps of splitting nodes and updating labels are repeated until each leaf node becomes a pure node (i.e. it contains examples belonging to only one class) and no further splits are required. Note that some nodes may become pure earlier than other and it may be better to condense the attribute lists to discard entries corresponding to examples belonging to these pure nodes. This optimization can easily be implemented by rewriting condensed list when the savings from reading smaller lists outweigh the extra cost of writing condensed lists. The information required to make this decision is available from the previous pass over the data.

The important thing to note about pre-sorting and breadth-first growth is that these strategies allow SLIQ to scale for large data sets with no loss in accuracy. This is because the set of splits evaluated with and without pre-sorting is identical. Pre-sorting simply eliminates the task of resorting data at each node and removes the restriction that the training set be memory-resident.

3.6 SUBSETTING FOR CATEGORICAL ATTRIBUTES

The splits for categorical attribute a are of the form A∈S', where S'⊂S and S is the set of possible values of attribute A. The evaluation of all

the subsets of S can be prohibitively expensive, especially if the cardinality of S is large.

SLIQ uses a hybrid approach to overcome this issue. If the cardinality of S is less than a threshold, MAXSETSIZE, then all of the subsets of S are evaluated. Otherwise, a greedy algorithm is used to obtain the desired subset. The greedy algorithm starts with an empty subset S' and add that one element of S and S' which gives the best split. The process is repeated until there is no improvement in the splits. This hybrid approach finds the optimal subset if S is small and also performs well for large subsets.

4.0 SPRINT

Classification is an important data mining problem. Although classification is a well-studied problem, most of the current classification algorithms require that all or a portion of the entire dataset remain permanently in memory. This limits their suitability for mining over large databases. We will study a new decision-tree-based classification algorithm, called SPRINT that removes all of the memory restrictions, and is fast and scalable. **SPRINT** stands for **S**calable **P**a**R**allelizable **IN**ndution of decision **T**rees This algorithm has also been designed to be easily parallelized, allowing many processors to work together to build a single consistent model. This parallelization exhibits excellent scalability as well. The combination of these characteristics makes this algorithm an ideal tool for data mining.

The intuition is that by classifying larger datasets, we will be able to improve the accuracy of the classification model. In classification, we are given a set of example records, called a training set, where each record consists of several fields or attributes. Attributes are continuous, coming from an ordered domain, or categorical, coming from an unordered domain. One of the attributes, called the classifying attribute, indicates the class to which each example belongs. The objective of classification is to build a model of the classifying attribute based upon the other attributes. Once a model is built, it can be used to determine the class of future unclassified records. We will therefore focus on understanding a scalable and parallelizable decision-tree classifier.

The recently studied SLIQ classification algorithm addressed several issues in building a fast scalable classifier. SLIQ gracefully handles disk-resident data that is too large to fit in memory. It does not use small memory-sized datasets obtained via sampling or partitioning, but builds a single decision tree using the entire training set. However, SLIQ does require that some data per record stay memory-resident all the time. Since the size of this in-memory data structure grows in direct proportion to the number of input records, this limits the amount of data that can be classified by SLIQ.

4.1 SERIAL ALGORITHM

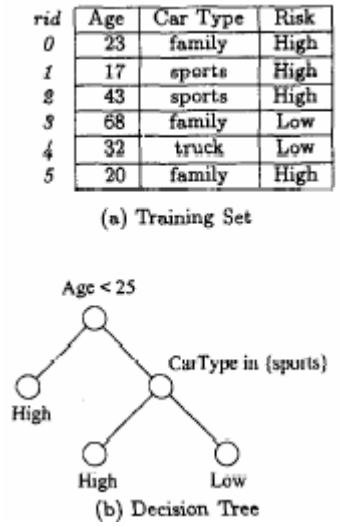
A decision tree classifier is built in two phases: a growth phase and a prune phase. In the growth phase, the tree is built by recursively partitioning the data until each partition is either “pure” (all members belong to the same class) or sufficiently small (a parameter set by the user). The form of the split used to partition the data depends on the type of the attribute used in the split. Splits for a continuous attribute A are of the form value(A) < x where x is a value in the domain of A. Splits for a categorical attribute A are of the form value(A) $\in$ X where X $\subseteq$ domain(A). We consider only binary

splits because they usually lead to more accurate trees; however, our techniques can be extended to handle multi-way splits. Once the tree has been fully grown, it is pruned in the second phase to generalize the tree by removing dependence on statistical noise or variation that may be particular only to the training set. The tree growth phase is computationally much more expensive than pruning, since the data is scanned multiple times in this part of the computation. Pruning requires access only to the fully grown decision tree. Our experience based on our previous work on SLIQ has been that the pruning phase typically takes less than 1% of the total time needed to build a classifier. We therefore focus only on the tree-growth phase. For pruning, we use the algorithm used in SLIQ, which is based on the Minimum Description Length Principle.

There are two major issues that have critical performance implications in the tree-growth phase:

1. How to find split points that define node tests.
2. Having chosen a split point, how to partition the data.

The well-known CART and C4.5.classifiers, for example, grow trees depth-first and repeatedly sort the data at every node of the tree to arrive at the best splits for numeric attributes. SLIQ, on the other hand, replaces this repeated sorting with one-time sort by using separate lists for each attribute. SLIQ uses a data structure called a class list which must remain memory resident at all times. The size of this structure is proportional to the number of input records, and this is what limits the number of input records that SLIQ can handle. SPRINT addresses the above two issues differently from previous algorithms; it has no restriction on the size of input and yet is a fast algorithm. It shares with SLIQ the advantage of a one-time sort, but uses different-data structures. In particular, there is no structure like the class list that grows with the size of input and needs to be memory-resident.



rid	Age	Car Type	Risk
0	23	family	High
1	17	sports	High
2	43	sports	High
3	68	family	Low
4	32	truck	Low
5	20	family	High

(a) Training Set

(b) Decision Tree

Figure: *Construction of Decision Tree Classifier.*

4.1.1 DATA STRUCTURES

Attribute lists: SPRINT initially creates an attribute list for each attribute in the data.

Age	Class	rid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

Car Type	Class	rid
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

Figure: *Example of attribute list.*

Entries in these lists, which we will call attribute records, consist of an attribute value, a class label, and the index of the record (rid) from which these value were obtained. Initial lists for continuous attributes are sorted by attribute value once when first created. If the entire data does not fit in memory, attribute lists are maintained on disk. The initial lists created from the training set are associated with the root of the classification tree. As the tree is grown and nodes are split to create new children, the attribute lists belonging to each node are partitioned and associated with the children. When a list is partitioned, the order of the records in the list is preserved; thus, partitioned lists never require resorting.

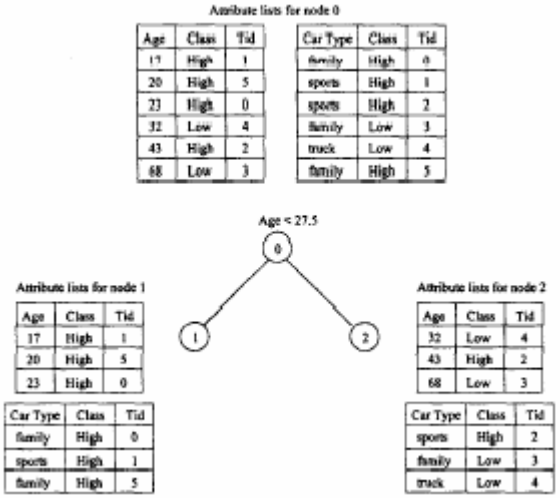


Figure: *Splitting a node attribute list*.

Histograms: For continuous attributes, two histograms are associated with each decision-tree node that is under consideration for splitting. These histograms, denoted as C_{above} and C_{below}, are used to capture the class distribution of the attribute records at a given node. As we will see, C_{below} maintains this distribution for attribute records that have already been processed, whereas C_{above} maintains it for those that have not. Categorical attributes also have a histogram associated with a node. However, only one histogram is needed and it contains the class distribution for each value of the given attribute. We call this histogram a *count matrix*. Since attribute

lists are processed one at a time, memory is required for only one set of such histograms.

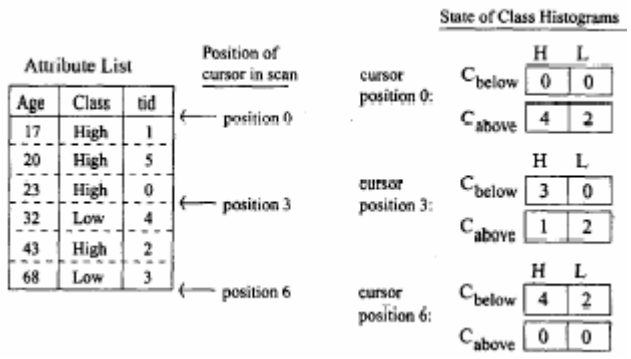


Figure*: Evaluating Continuous Split point.*

4.1.2 FINDING SPLIT POINTS

While growing the tree, the goal at each node is to determine the split point that "best" divides the training records belonging to that leaf. The value of a split point depends upon how well it separates the classes. Several splitting indices have been proposed in the past to evaluate the goodness of the split. The gini index is used here. For a data set S containing examples from n classes, gini(S) is defined as:

$$\mathbf{gini(S) = 1 - \sum p_j^2}$$

Where p_j is the relative frequency of class j in S. If a split divides S into two subsets S_1 and S_2, the index of the divided data $gini_{split}$ (S) is given by:

$$\mathbf{gini_{split}\ (S) = (n_1/n)\ gini(S_1) + (n_2/n)\ gini(S_2)}$$

The advantage of this index is that its calculation requires only the distribution of the class values in each of the partitions. To find the best split point for a node, we scan each of the node's attribute lists and evaluate splits based on that attribute. The attribute containing the split point with the lowest value for the gini index is then used to split the node. We discuss next how split points are evaluated within each attribute list.

Continuous attributes: For continuous attributes, the candidate split points are mid-points between every two consecutive attribute values in the training data. For determining the split for an attribute for a node, the histogram C_{below} is initialized to zeros whereas C_{above} is initialized with the class distribution for all the records for the node. For the root node, this distribution is obtained at the time of sorting. For other nodes this distribution is obtained when the node is created. Attribute records are read one at a time and C_{below} and C_{above} are updated for each record read. After each record is read, a split between values (i.e. attribute records) we have and have not yet seen is evaluated. Note that C_{below} and C_{above} have all the necessary information to compute the gini index. Since the lists for continuous attributes are kept in sorted order, each of the candidate split-points for an attribute are evaluated in a single sequential scan of the

corresponding attribute list. If a winning split point was found during the scan, it is saved and the C_{below} and C_{above} histograms are deallocated before processing the next attribute.

Categorical attributes: For categorical split-points, we make a single scan through the attribute list collecting counts in the count matrix for each combination of class label and attribute value found in the data. Once we are finished with the scan, we consider all subsets of the attribute values as possible split points and compute the corresponding gini index. If the cardinality of an attribute is above certain threshold, the greedy algorithm is instead used for sub setting. The important point is that the information required for computing the gini index for any subset splitting is available in the count matrix. The memory allocated for a count matrix is reclaimed after the splits for the corresponding attribute have been evaluated.

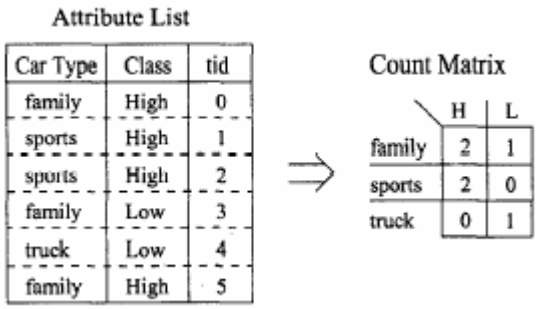


Figure: *Evaluating Categorical split point.*

4.1.3 PERFORMING THE SPLIT

Once the best split point has been found for a node, we execute the split by creating child nodes and dividing the attribute records between them. This requires splitting the node's lists for every attribute into. Partitioning the attribute list of the winning attribute (i.e. the attribute used in the winning split point - Age in our example) is straightforward. We scan the list, apply the split test, and move the records to two new attribute lists - one for each new child.

Unfortunately, for the remaining attribute lists of the node (CarType in our example); we have no test that we can apply to the attribute values to decide how to divide the records. We therefore work with the rids. As we partition the list of the splitting attribute (i.e. Age), we insert the rids of each record into a probe structure (hash table), noting to which child the record was moved. Once we have collected all the rids, we scan the lists of the remaining attributes and probe the hash table with the rid of each record. The retrieved information tells us with which child to place the record.

If the hash-table is too large for memory, splitting is done in more than one step. The attribute list for the splitting attribute is partitioned up to the attribute record for which the hash table will fit in memory; portions of attribute lists of non-splitting attributes are partitioned; and the process is repeated for the remainder of the attribute list of the splitting attribute.

If the hash-table can fit in memory (quite likely for nodes at lower levels of the tree), a simple optimization is possible. We can build the hash table out of the rids of only the smaller of the two children. Relative sizes of the two children are determined at the time the split point is evaluated.

During this splitting operation, we also build class histograms for each new leaf. As stated earlier, these histograms are used to initialize the C_{above} histograms when evaluating continuous split-points in the next pass.

4.2 PARALLELIZING CLASSIFICATION

We now try to understand the problem of building classification trees in parallel. We again focus only on the growth phase due to its data-intensive nature. The pruning phase can easily be done off-line on a serial processor as it is computationally inexpensive, and requires access to only the decision-tree grown in the training phase. In parallel tree-growth, the primary problems remain finding good split-points and partitioning the data using the discovered split points. As in any parallel algorithm, there are also issues of data placement and workload balancing that must be considered. Fortunately, these issues are easily resolved in the SPRINT algorithm. SPRINT was specifically designed to remove any dependence on data structures that are either centralized or memory-resident; because of these design goals, SPRINT parallelizes quite naturally and efficiently. In this section we will understand how to parallelize SPRINT. This algorithm assumes a shared-nothing parallel environment where each of N processors has private memory and disks. The processors are connected by a communication network and can communicate only by passing messages. Examples of such parallel machines include GAMMA, Teradata, and IBM's SP2.

4.2.1 DATA PLACEMENT AND WORKLOAD BALANCING

We recall that the main data structures used in SPRINT are the attribute lists and the class histograms. SPRINT achieves uniform data placement and workload balancing by distributing the attribute lists evenly over N processors of a shared-nothing machine. This allows each processor to work on only l/N of the total data.

The partitioning is achieved by first distributing the training-set examples equally among all the processors. Each processor then generates its own attribute-list partitions in parallel by projecting out each attribute from training-set examples it was assigned. Lists for categorical attributes are therefore evenly partitioned and require no further processing. However, continuous attribute lists must now be sorted and repartitioned into contiguous sorted sections. For this, we use parallel sorting algorithm. The result of this sorting operation is that each processor gets a fairly equal-sized sorted section of each attribute list.

Processor 0

Age	Class	rid
17	High	1
20	High	5
23	High	0

Car Type	Class	rid
family	High	0
sports	High	1
sports	High	2

Processor 1

Age	Class	rid
32	Low	4
43	High	2
68	Low	3

Car Type	Class	rid
family	Low	3
truck	Low	4
family	High	5

Figure: *Parallel Data Placement*

4.2.2 FINDING SPLIT POINTS

Finding split points in parallel SPRINT is very similar to the serial algorithm. In the serial version, processors scan the attribute lists either evaluating split points for continuous attributes or collecting distribution counts for categorical attributes. This does not change in the parallel algorithm - no extra work or communication is required while each processor is scanning its attribute-list partitions. We get the full advantage of having N processors simultaneously and independently processing l/N of the total data. The differences between the serial and parallel algorithms appear only before and after the attribute-list partitions are scanned.

Continuous attributes: For continuous attributes, the parallel version of SPRINT differs from the serial version in how it initializes the C_{below} and C_{above} class-histograms. In a parallel environment, each processor has a separate contiguous section of a "global" attribute list. Thus, a processor's C_{below} and C_{above} histograms must be initialized to reflect the fact that there are sections of the attribute list on other processors. Specifically, C_{below} must initially reflect the class distribution of all sections of an attribute-list assigned to processors of lower rank. The C_{above} histograms must likewise initially reflect the class distribution of the local section as well as all sections assigned to processors of higher rank. As in the serial version, these statistics are gathered when attribute lists for new leaves are created. After collecting statistics, the information is exchanged between all the processors and stored with each leaf, where it is later used to initialize that leaf's C_{below} and C_{above} class histograms. Once all the attribute-list sections of a leaf have been processed, each processor will have what it considers being the best split for that leaf. The processors then communicate to determine which of the N split points has the lowest cost.

Categorical attributes: For categorical attributes, the difference between the serial and parallel versions arises after an attribute-list section has been scanned to build the count matrix for a leaf. Since the count matrix built by each processor is based on "local" information only, we must exchange these matrices to get the "global" counts. This is done by choosing a coordinator to collect the count matrices from each processor. The coordinator process then sums the local matrices to get the global count-

matrix. As in the serial algorithm, the global matrix is used to find the best split for each categorical attribute.

4.2.3 PERFORMING THE SPLITS

Having determined the winning split points, splitting the attribute lists for each leaf is nearly identical to the serial algorithm with each processor responsible for splitting its own attribute-list partitions. The only additional step is that before building the probe structure, we will need to collect rids from all the processors. (Recall that a processor can have attribute records belonging to any leaf.) Thus, after partitioning the list of a leaf's splitting attribute, the rids collected during the scan are exchanged with all other processors. After the exchange, each processor continues independently, constructing a probe-structure with all the rids and using it to split the leaf's remaining attribute lists. No further work is needed to parallelize the SPRINT algorithm. Because of its design, SPRINT does not require a complex parallelization and scales quite nicely.

5.0 PERFORMANCE RESULT

The performance evaluation and comparison of ID3, SLIQ and SPRINT has been studied using a P-II, 333 MHz, 32 MB RAM machine. The operating system in which they have been tested is Microsoft Windows 98 4.10.2222A having Swap file of 2361 MB.

(a) The dataset used is of voting records drawn from the Congressional Quarterly Almanac, 98th Congress, 2nd session 1984, Volume XL: Congressional Quarterly Inc. Washington, D.C., 1985.This data set includes votes for each of the U.S. House of Representatives Congressmen on the 16 key votes identified by the CQA. The CQA lists nine different types of votes: voted for, paired for, and announced for (these three simplified to yea), voted against, paired against, and announced against (these three simplified to nay), voted present, voted present to avoid conflict of interest, and did not vote or otherwise make a position known (these three simplified to an unknown disposition).This has been collected by Jeff Schlimmer, 23 April 1987. The data set has 16 categorical attribute each having three possible values. The classification (class) attribute has two values. They are:

democrat, republican	classes
handicapped infants:	n, y, u
water project cost sharing:	n, y, u
adoption of the budget resolution:	n, y, u
physician fee freeze:	n, y, u
el salvador aid:	n, y, u
religious groups in schools:	n, y, u
anti satellite test ban:	n, y, u
aid to nicaraguan contras:	n, y, u
mx missile:	n, y, u
immigration:	n, y, u
synfuels corporation cutback:	n, y, u
education spending:	n, y, u
superfund right to sue:	n, y, u
crime:	n, y, u
duty free exports:	n, y, u
export administration act south africa:	n, y, u

No: of Records: 300

Algo.	Time	Rules
ID3	1	34
ID3-gini	1	34
SLIQ	2.8	34
SPRINT (serial)	7	34

No: of Records: 3000

Algo.	Time	Rules
ID3	11	34
ID3-gini	12	34
SLIQ	11	34
SPRINT (serial)	30	34

No: of Records: 30000

Algo.	Time	Rules
ID3	138	-
ID3-gini	140	-
SLIQ	104	34
SPRINT (serial)	1320	34

When the classifier –decision tree was generated it was tested with a data sample consisting of 135 records. Among them 125 were correctly classified while 8 were found to be misclassified giving a error of 5.925%. In this experiment the scalability issues of the algorithms were tested. ID3 was found to be incapable of using a record set of 30,000 as its training sample while SPRINT performed average, with SLIQ excelling.

(b) Here we will try to find the goodness of a classifier depending on how well (truly) it can classify unseen datasets. Here we will use three datasets, namely – Monk1, Monk2 and Monk3. The Monk's problems are a collection of three binary classification problems over a six-attribute discrete domain. Each training/test data is of the form
<name>: <value1> <value2> <value3> <value4> <value5> <value6> -> <class>

Where <name> is an ASCII-string, <value n> represents the value of attribute # n, and <class> is either 0 or 1, depending on the class this example belongs to. The attributes may take the following values:

Attribute#1: {1, 2, 3}
Attribute#2: {1, 2, 3}
Attribute#3: {1, 2}
Attribute#4: {1, 2, 3}
Attribute#5: {1, 2, 3, 4}
Attribute#6: {1, 2}

Thus, the six attributes span a space of 432=3x3x2x3x4x2 examples.
The "*true*" concepts underlying each Monk's problem are given by:

MONK-1: (attribute_1 = attribute_2) or (attribute_5 = 1)
MONK-2: (attribute_n = 1) for EXACTLY TWO choices of n (in {1,2,...,6})
MONK-3: (attribute_5 = 3 and attribute_4 = 1) or (attribute_5 != 4 and attribute_2 != 3) (With "! =" denoting inequality).
MONK-3 has 5% additional noise (misclassifications) in the training set.

The result generated by all the three algorithms is the same except the time of execution. ID3 (using entropy/gini) required the least amount of time, SLIQ (using entropy/gini) required a bit more execution time while SPRINT(using entropy/gini) took the most.

	Monk1	Monk2	Monk3
Training Records	124	169	122
Test Records	432	432	432
No. of Rules	50	95	28
True classification	376	299	408
False classification	56	133	24
Error %	12.96	30.79	5.55

Many other tests were conducted on various datasets of **mixed and continuous data attributes.** For example tests were conducted on *Remote sensing data, Binary data, crx (Credit Card Data), hypothyroid data from Garvan Institute and Golf databases*. SLIQ and SPRINT have been found to perform well, excepting the fact that SPRINT requires more execution time than SLIQ. But SPRINT can handle bigger datasets where SLIQ fails. ID3 cannot handle datasets with continuous attribute. It has been designed for only categorical datasets.

(c) One experiment was also conducted on the remote sensing data. The decision tree was used as a land cover classifier. Three bands were present in the dataset, which were used as the classification attribute to classify the land type. Various details of the data were undisclosed due to privacy. The class attribute took value from 1 to 6 while the other three attributes used were of continuous nature.

Attribute-1 ranged from 13 to 70.
Attribute-1 ranged from 12 to 78.
Attribute-1 ranged from 8 to 65.

The results are concluded in the table below:

Total no of data	198
Number of training data	88 (44%)
Number of test data	110
Class distribution in original data	14-49-71-16-30-18
Class distribution in training data	9-20-21-7-20-11
Number of rules generated	7
Error Percentage	6.36

The results found has been verified and found to be satisfactory. A small modification was also made in the SLIQ algorithm to incorporate the non-linearity. It was made possible to select an attribute more than once during a rule/decision tree generation from root to a particular leaf node of the decision tree.

The rules generated using SLIQ without any modification are:

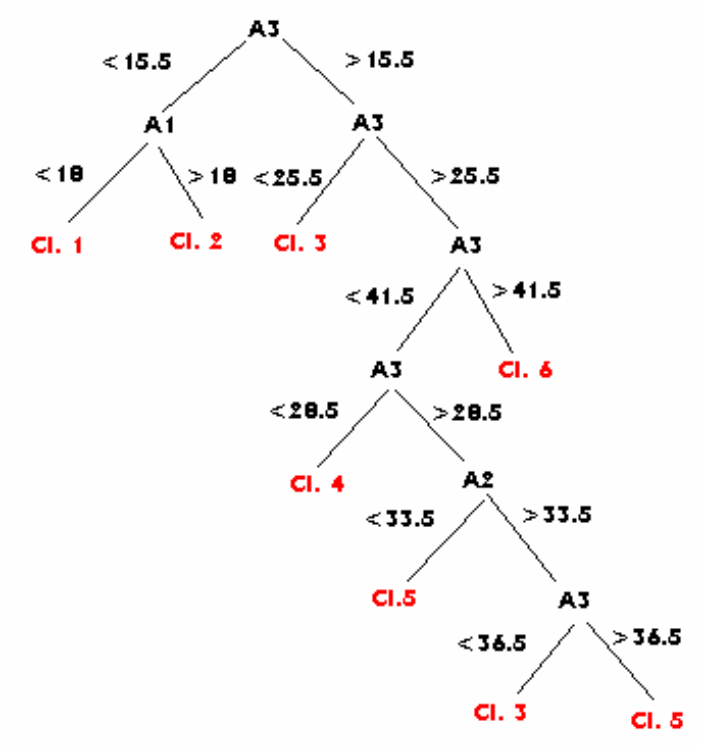
Total no of data	198
Number of training data	198(100%)
Number of test data	198
Class distribution in original data	14-49-71-16-30-18
Number of rules generated	4
Error Percentage	64.14

The rules generated using the full dataset as training sample are:

1. If A3 (<15.5) and A1 (<18) Then Class (1)
2. If A3 (<15.5) and A1 (>18) Then Class (2)

3. If A3 (>15.5) and A3 (<25.5) Then Class (3)
4. If A3 (>41.5) Then Class (6)
5. If A3 (28.5>A3>25.5) Then Class (4)
6. If A3 (41.5>A3>28.5) and A2 (<33.5) Then Class (5)
7. If A3 (36.5>A3>28.5) and A2 (>33.5) Then Class (3)
8. If A3 (41.5>A3>36.5) and A2 (>33.5) Then Class (5)

The decision tree generated is:



6.0 CONCLUSION

The past two decades has seen a dramatic increase in the amount of information or data being stored in electronic format. This accumulation of data has taken place at an explosive rate. It has been estimated that the amount of information in the world doubles every 20 months and the size and number of databases are increasing even faster. The increase in use of electronic data gathering devices such as point-of-sale or remote sensing devices has contributed to this explosion of available data.

With such enormous amount of data at our disposal, our main aim is to extract the hidden valuable information from it. It is important for us to build a classifier from such training set which can be used to take important decisions on unseen data. It is trivial that if we can use a large training set then our classifier will be of better accuracy. Decision tree has been found to be a very fast classifier than other known classifiers. Thus, our motive is to have a classifier which can handle all the major issues like scalability, fuzziness, ambiguity etc. of real world data.

The ID3 have been developed for categorical data with small datasets. It has its own limitations but it outperforms any other classifier within its domain.

The technique of creating separate attribute lists from the original data was first proposed by the SLIQ algorithm. In SLIQ, an entry in an attribute list consists only of an attribute value and a rid; the class labels are kept in a separate data-structure called a class list which is indexed by rid. In addition to the class label, an entry in the class list also contains a pointer to a node of the classification tree which indicates to which node the corresponding data record currently belongs. Finally, there is only one list for each attribute. SLIQ uses a novel pre-sorting technique in the tree-growing phase to reduce the cost of evaluating numerical attributes. This sorting technique is integrated with breadth-first tree growing strategy to enable SLIQ to classify disk-resident datasets. In addition SLIQ uses a fast subsetting algorithm for determining splits for categorical attributes. SLIQ uses a new tree-pruning algorithm –“Minimum description Length”, which is inexpensive and results in compact and accurate tree. The advantage of not having separate sets of attribute lists for each node is that SLIQ does not have to rewrite these lists during a split. Reassignment of records to new nodes is done simply by changing the tree-pointer field of the corresponding class-list entry. Since the class list is randomly accessed and frequently updated, it must stay in memory all the time or suffer severe performance degradations. The size of this list also grows in direct proportion to the training-set size. This ultimately limits the size of the training set that SLIQ can handle. SPRINT is designed to be easily parallelizable.

With the recent emergence of the field of data mining, there is a great need for algorithms for building classifiers that can handle very large databases. The recently proposed SLIQ algorithm was the first to address these concerns. Unfortunately, due to the use of a memory-resident data

structure that scales with the size of the training set, even SLIQ has an upper limit on the number of records it can process. But SPRINT removes all memory restrictions that limit existing decision-tree algorithms, and yet exhibits the same excellent scaling behavior as SLIQ. By eschewing the need for any centralized, memory-resident data structures, SPRINT efficiently allows classification of virtually any sized dataset. SPRINT does have an efficient parallelization that requires very few additions to the serial algorithm. Using measurements from actual implementations of these algorithms, we showed that SPRINT handles datasets that are too large for SLIQ to handle. Moreover, SPRINT scales nicely with the size of the dataset, even into the large problem regions where no other decision-tree classifier can compete. It has excellent scaleup, speedup, and sizeup characteristics. Given SLIQ's somewhat superior performance in problem regions where a class list can fit in memory, one can envision a hybrid algorithm combining SPRINT and SLIQ. The algorithm would initially run SPRINT until a point is reached where a class list could be constructed and kept in real memory. At this point, the algorithm would switch over from SPRINT to SLIQ exploiting the advantages of each algorithm in the operating regions for which they were intended. Since the amount of memory needed to build a class list is easily calculated, the switch over point would not be difficult to determine.

REFERENCES

1. http://www.cs.cornell.edu
2. http://www.cs.uregina.ca
3. http://www.risc.uni-linz.ac.at
4. http://www.cs.brandeis.edu
5. http://mycroft.ncsa.uiuc.edu
6. Jiawei Han and Micheline Kamber. Data Mining-Concepts and technique. Morgan Kaufmann publication.
7. Michael j. A. Berry and Gordon S. Linoff. Mastering Data Mining. Wiley publication.
8. J. Ross Quinlan. Introduction of decision tree. Machine Learning, 1:81-106, 1986.
9. J. Ross Quinlan. C4.5: programs for Machine Learning. Morgan Kaufmann, 1993.
10. Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance per-spective. IEEE Transactions on Knowledge and Data Engineering, 5(6):914-925, December 1993
11. J. Gehrke, R. Ramakrishnan, V. Ganti. “RainForest – A Framework for Fast Decision Tree Construction of Large Datasets”.
12. Blaž Zupan, Ivan Bratko “Induction of Decision Trees”.
13. John C Shafer, Rakesh Agarwal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. Research report, IBM Almaden Research Center, San Jose, California, 1996. Available from http://www.almaden.ibm.com/cs/quest.
14. Manish Mehta, Rakesh Agarwal and Jorma Rissanen. SLIQ: A fast Scalable Classifier for Data Mining. Research report, IBM Almaden Research Center, San Jose, California