WestVirginiaUniversity
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2005

Ensemble learning for ranking interesting attributes

Erik W. Sinsel
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation

Sinsel, Erik W., "Ensemble learning for ranking interesting attributes" (2005). *Graduate Theses, Dissertations, and Problem Reports*. 1685.
https://researchrepository.wvu.edu/etd/1685

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Ensemble Learning for Ranking Interesting Attributes

Erik W. Sinsel

Thesis submitted to the College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Bojan Cukic, Ph.D., Chair
Frances Van Scoy, Ph.D.
John Atkins, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia

2005

Machine learning knowledge representations, such as decision trees, are often incomprehensible to humans. They can also contain errors specific to the representation type and the data used to generate them. By combining larger, less comprehensible decision trees, it is possible to increase their accuracy as an ensemble compared to the best individual tree. The thesis examines an ensemble learning technique and presents a unique knowledge elicitation technique which produces an ordered ranking of attributes by their importance in leading to more desirable classifications. The technique compares full branches of decision trees, finding the set difference of shared attributes. The combination of this information from all ensemble members is used to build an importance table which allows attributes to be ranked ordinally and by relative magnitude. A case study utilizing this method is discussed and its results are presented and summarized.

Acknowledgments

My special thanks to Dr. Bojan Cukic for advising me, to Dr. Tim Menzies for giving me my first research position, and to my family and friends for hearing the often-repeated excuse for missing countless events: “I have to work on my thesis.”

Great appreciation is also due the West Virginia University College of Engineering and Mineral Resources, and especially the Lane Department of Computer Science and Electrical Engineering for endless support.

List of Figures

2.1 Graphical Representation of a Decision Tree 12
2.2 Sample C4.5 Error Rate Summary Table 13
2.3 Sample of C4.5 Decision Tree Syntax 14
2.4 Decision Rules Derived from Figure 2.3 14

3.1 C4.5 Decision Tree Without Level Spacing 36
3.2 Adding Children in a Full Binary Tree 45
3.3 A Perfectly Incomplete Full Binary Tree 46

4.1 Example Expert COCOMO Input Screen 56
4.2 Example Expert COCOMO Output Screen 57
4.3 Rel. Importance and Rel. Standardized Beta, 10K-50K SLOC 62
4.4 Rel. Importance and Rel. Standardized Beta, 50K-100K SLOC 63
4.5 Rel. Importance and Rel. Standardized Beta, 100K-250K SLOC 63
4.6 Rel. Importance and Rel. Standardized Beta, 250K-500K SLOC 64

List of Tables

2.1 A sample training set. 11
2.2 Empirical Results of Fischer's Combined Classifiers 28
2.3 Empirical Results of Ho's Random Subspace Method 30

3.1 Example Importance Table 38

4.1 Example of an Expert COCOMO 2.0 Joint Risk Matrix 55
4.2 Expert COCOMO 2.0 Risk Categories 56
4.3 C4.5 Tree Error by Expert COCOMO 2.0 Test Set Size 59
4.4 Importance and Beta, 10-50K SLOC 65
4.5 Importance and Beta, 50-100K SLOC 65
4.6 Importance and Beta, 100-250K SLOC 66
4.7 Importance and Beta, 250-500K SLOC 66

A.1 Importance Table for 10-50K SLOC 75
A.2 Importance Table for 50-100K SLOC 76
A.3 Importance Table for 100-250K SLOC 77
A.4 Importance Table for 250-500K SLOC 78

B.1 Valid Values for Expert COCOMO 2.0 Attributes 79
B.2 COCOMO 2.0 Attribute Descriptions 81

Contents

Abstract

Acknowledgments **iii**

List of Figures **iv**

List of Tables **v**

Table of Contents **vi**

Definitions **viii**

1 The Problem **1**

2 Related Work **8**
- 2.1 Machine Learning and Data Mining . . . 8
- 2.2 Decision Trees . . . 10
- 2.3 Knowledge Discovery in Databases . . . 16
- 2.4 Ensemble Learning . . . 20
  - 2.4.1 Generating Ensembles . . . 22
  - 2.4.2 Combination Techniques . . . 23
  - 2.4.3 Combination Functions . . . 25
  - 2.4.4 Empirical Results of Ensemble Learning . . . 27
- 2.5 Interestingness . . . 31

3 The Approach **33**
- 3.1 Introduction . . . 33
- 3.2 Analysis Technique . . . 34
  - 3.2.1 Overview . . . 34
  - 3.2.2 Experimental Setup . . . 35
  - 3.2.3 Implementation . . . 40
  - 3.2.4 Proof of $l = \frac{1}{2}(n+1)$ . . . 43
  - 3.2.5 Proof of Worst-Case Total Rule Size . . . 45
  - 3.2.6 Complexity Analysis . . . 48
  - 3.2.7 Total Execution Time . . . 51

4 Case Study: Expert COCOMO 2.0 **52**
4.1 Introduction . . . 52
4.2 COCOMO . . . 53
4.3 Experimental Design . . . 57
4.4 Results and Conclusions . . . 61
4.5 Possible Improvements . . . 67

5 Summary and Conclusions **69**

APPENDICES **75**

A Case Study 1: Importance Tables **75**

B COCOMO 2.0 Factors **79**

Definitions

tree	:	connected, undirected, acyclic graph
depth of a node	:	the number of edges in the path from the root to the node
height of a tree	:	the maximum depth in the tree
full binary tree	:	a tree where all but one node have degree either one or three, and exactly one node has degree two (the root)
complete binary tree	:	full binary tree where a node at level k may have children only if all nodes at level $k-1$ already have two children
path	:	the ordered list of nodes in a traversal from the root node to the destination node, including the root node
n	:	the number of nodes in a tree
l	:	the number of terminal leaf nodes in a tree
h	:	the height of the tree
R_i	:	the ith rule of the tree, which is a path where tests at nodes are conjuncted
$\|x\|$	:	the arity of set-representable variable x
A	:	the set of all attributes present in the model examined
V_a	:	the set of all valid values for attribute $a \in A$

Chapter 1

The Problem

With ever-increasing computer processing power and data storage capabilities comes an ever-increasing amount of stored data to process [26]. From catalogues of deep-space objects to consumer databases, current *data warehouses* are designed to hold on the order of many terabytes of data. For example, the NASA Earth Observing System of satellites and observing instruments is projected to generate some 50 terabytes of data per hour when fully employed [13]. Traditional methods of data analysis can be overwhelmed by such massive amounts of data. In addition, this data is expressed at such a low level of of knowledge that it is not directly *actionable* [32].

The process known as *knowledge discovery in databases* (KDD) has been devised and enumerated specifically to address this problem of analyzing such large amounts of data, which has been described as "the most nagging and fundamental problem of KDD" [34]. KDD involves a process which integrates human subjectivity with computer-assisted data mining techniques in order to transform data into knowledge which is both useful and understandable.

Definition 1 *"Knowledge discovery in databases is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data."* [13]

The last three components of the KDD process are of particular interest. Novel patterns are those which are *interesting* to the user. Findings which simply reinforce a strongly held prior belief may be of much less interest than those which represent previously unknown or under-represented relationships. Useful patterns are those which are *actionable*, or capable of being implemented in a real-world situation. A strong relationship between a desired outcome and a fixed attribute is not a particularly useful discovery. Usefulness could be discussed in terms of the *utility* given by the discovered knowledge. Finally, an understandable pattern is one which is expressed in such a way as to be well understood by the human end user who may not be a KDD expert.

Within this KDD process, the step which provides an enumeration of patterns discovered over the data is called *data mining* (DM). This step may be constrained by processing limitations, and produces some *concept description* capable either of *prediction* or *classification* [13, pp. 12-13]. While prediction involves using existing data to predict values in future data, the classification aspect of data mining considered here is a function of mapping the *feature vectors* of data onto output classification classes. A feature vector includes a set of descriptive attributes and their instance values, while the output classification is some discrete description class to which the instance feature vector belongs. An example may be the classification of riskiness of health insurance policy holders, based on a feature vector which contains such attributes as age, weight, cigarette smoking, occupation, income, education, and so on. For each instance of policy-holder data, there would be a risk classification.

The input space X of the data mining algorithm is the feature space of all possible combinations of attribute values, where $X = A \times V$, the cross product of the attributes and their values, and each $x \in X$ is a feature vector. If the output space is C, the set of all possible classifications, then the goal of the data mining step of the KDD process is then to develop a hypothesis which is a mapping of feature vectors onto classifications, $h = X \rightarrow C$, and which is expressed as some concept description, such as a decision tree. The set H of all possible hypotheses is the *hypothesis space*.

The hypothesis creation step in the knowledge elicitation process is circumscribed by the characteristics of the particular data mining algorithm used, and by its form of knowledge representation. It is this circumscription of the knowledge representation which the author addresses through the combination of multiple hypotheses h from within H in an attempt to provide a closer approximation of the underlying concept.

As Fayyad notes, the resulting concept descriptions provided by the data mining step of the KDD process may not, on their own, satisfy our definition of the KDD process in that their descriptions may not provide particularly useful or interesting knowledge [13]. Useful knowledge is that which is actionable, i.e. it allows actions to be taken by the data user within the particular domain in question, using the extracted knowledge in an attempt to produce the desired output classification. For example, the knowledge that reducing cholesterol in the blood tends to reduce health insurance risk is actionable. Interesting knowledge is that which is additive from the point of view of the end user. What is interesting is that the knowledge is new to the end user, and what is useful is that the knowledge provides a set of actionable changes which may be taken to gain the desired

outcome. Taken together, these two aspects of knowledge provide *domain utility* to the end user by providing a strategy for potentially reducing health insurance costs (not to mention perhaps avoiding serious health complications).

While there are several methods to perform the data mining step of the KDD process, the thesis will focus primarily on the popular *decision tree* (DT) method. Standard decision trees utilize univariate node splits on features within the feature vector, ending with leaf nodes which are labeled with an output classification. The tree may then be viewed as a composition of rules given in the form of conjunctions in propositional logic, leading to some implied classification. The tree structure is thus a concept simplification more easily absorbed by human minds than a list of conjunctions of rules in propositional logic. This becomes especially important as the users of KDD are increasingly becoming those who are charged with making operational decisions but who are not KDD experts [34]. KDD literature also warns against directly using information obtained through the use of a data mining methodology without involving a "human in the loop" to provide necessary interpretation of the results [13] [33, pp. 14] . Decision trees are a form of knowledge representation which help satisfy this human-interaction requirement by presenting knowledge in a form readily absorbed by human users.

While many machine learning packages, including some decision tree inducers like Ross Quinlan's C4.5 program, can generate decision rules in the simple to understand format of propositional logic, these rules have limited usefulness in a proactive decision making context. Trees and rules are useful for answering instance-based questions such as "Given these values for my parameters, what outcome is most likely to occur?" However, they are inadequate to answer such directive proactive

questions as "What parameters are most important for me to focus on if I wish to move from outcome A to more desirable outcome B?" This aspect of automated knowledge learners reduces their utility in resource allocation decision making where planning a future course of action is of paramount importance. This loss of utility could be avoided by providing a method of presenting the discovered knowledge in such a way that the end-user can utilize the knowledge as a whole in a proactive decision making process, rather than simply as a descriptive or single-instance classification.

In general, decision trees learned by such machine learners as C4.5 become more complex as they become more accurate, a likelihood which increases with the complexity of the underlying relationship to be modeled [36]. This is seen by considering that given a particular representation with a particular error rate, increasing the accuracy necessarily means producing a structure which misclassifies fewer cases. This can be done either by creating a new structure with better accuracy, or more commonly by simply creating a new branch in the tree which correctly classifies one or more previously misclassified cases. This latter case provides an increase in accuracy paid for with an increase in representation complexity. Thus, with decision trees, there is what the author refers to as the *accuracy-complexity constraint* in the knowledge representation.

Definition 2 Accuracy-complexity constraint - *The typical direct relationship in decision tree classifiers between predictive accuracy and tree complexity.*

While more accurate representations may also generally imply more complexity, human beings generally have more difficulty understanding knowledge representations which are more complex,

and less difficulty understanding less complex ones. According to Ross Quinlan, author of the C4.5 machine learner, these more complex trees are "suspect" as an explanation of the underlying relationship [29]. Rather than relying solely on predictive accuracy, Quinlan seems to imply that for a model to adequately explain a relationship, a human must be capable of understanding that knowledge representation. This requirement leads to what the author refers to as the *complexity-understandability constraint* on knowledge representations.

Definition 3 Complexity-understandability constraint - *The inverse relationship in decision tree classifiers between tree complexity and human understandability.*

Taken transitively, the accuracy-complexity constraint and the complexity-understandability constraint lead to an *accuracy-understandability constraint*.

Definition 4 Accuracy-understandability constraint - *The more accurate the decision trees become, the less understandable they are likely to be.*

Because of this accuracy-complexity constraint when modeling complex relationships within highly dimensional feature sets, DT's may often be unable to satisfy the definition of a KDD process. The presence of dozens of conjunctions of sometimes-overlapping, sometimes-disjoint feature sets results in a concept description incapable of being either understandable or actionable [31]. This difficulty is present precisely due to the requirement of involving the human KDD user in the modeling process, without which trees could be permitted to grow sufficiently complex to achieve the desired

accuracy. A solution to this problem could thus lie in the development of methods which reduce the complexity of the knowledge representation while maintaining adequate accuracy, resulting in a representation which is understandable enough to the human user to provide actionable analysis.

In this thesis, we will present a unique method for addressing this accuracy-understandibility constraint on the KDD process. The approach first involves creating an ensemble of decision trees. Each tree is then converted into a disjunction of decision rules which are the branches of the tree. Rules with classifications differing by two or more ordinal positions are compared according to shared attributes. When two rules contain the same attribute, the set of values of the rule with the worse classification is subtracted from the set of values from the rule with the better classification. This set of values in the better rule which are not in the worse rule, for all attributes existing in both rules is then added to a frequency table, called the importance table. Each tree's importance table is then added to a master ensemble importance table. This table provides a relative measure of the importance of each attribute's value in leading to classifications with a better outcome from a rule with a worse outcome. A case study will be presented which utilize this method, and its results will be presented.

Chapter 2

Related Work

2.1 Machine Learning and Data Mining

Machine learning (ML) from data is a discipline primarily concerned with developing computer-intensive techniques which aid in the task of building knowledge representations which, given a set of example cases with associated classifications will correctly map unseen cases to those classes. This inductive learning from already classified examples is referred to in ML literature as *supervised learning* [30]. Data Mining (DM) is also often used synonymously with machine learning but ML subsumes DM as ML is the general task of learning problem solving strategies given some input data. For example, solving a chess problem using a computer learning algorithm would be considered machine learning, while not falling within the domain of data mining. Data mining, while often used synonymously with Knowledge Discovery in Databases (KDD) is actually a methodology which is utilized within the overall KDD process [24]. The KDD process will be covered in more detail in Section 2.3.

Machine learning algorithms can create classifiers in many different forms, including decision trees and rules, Bayes networks, and artificial neural networks (ANNs) as the most popular. Other popular methods include nonlinear regression, example-based reasoning, inductive logic programming, as well as other approaches. While algorithms like Bayes networks and ANNs can sometimes consistently outperform induced decision trees, they both require specific knowledge on the part of the user which would violate the problem statement considered in Section 1. For example, Bayes network learning requires estimating the posterior probabilities. It also assumes that features are independent of one another, something which may not be practical beyond an academic problem formulation. ANNs require training to customize the algorithm to a particular data set, with possible required modifications to such things as the internal weightings. In essence, both approaches require modification or training before being deployed in a real-world situation.

As decision trees are a popular approach for decision combination in the ensemble learning approach utilized here, and decision tree induction problems generally require little specific KDD training on the part of the user, the research presented here will focus on decision trees. However, the approach is not limited to this knowledge representation, and can be used with any technique which allows the knowledge to be translated into the intermediate representation of decision rules involving feature ranges and ordinal or categorical outcome classes.

2.2 Decision Trees

A decision tree (DT) is a knowledge representation utilizing a simple binary decision test branching formalism which lacks the power of some more sophisticated representations, yet is still capable of powerful practical problem solving [29]. In the world of machine learning, a family of decision tree learners exists whose members are referred to as being Top Down Induction of Decision Tree (TDIDT) algorithms [29, pp. 350]. The TDIDT family is characterized primarily by a non-incremental induction which utilizes frequency information in the training samples yet does not consider the order in which samples are given. The TDIDT tool is supplied with a *training set* of cases where membership in some classification is known, and then the task of the induction algorithm is to develop a tree-based decision-test classification scheme which allows prediction of a sample case's class by finding a tree branch consistent with the case's feature vector.

An example training set is given in Figure 2.1. This data set consists of four attributes and a single categorical classification. The attributes represent some weather conditions important when deciding whether to play golf, and the classification is then "N" for "no play", and "P" for "play".

In a decision tree, each non-leaf node represents a binary test on an attribute a with child branches representing certain mutually exclusive subsets of V_a, the set of all valid values for a. Each leaf node contains a classification c which is predicted given the conjunction of the internal nodes along the full path to that leaf from the root of the tree. The decision tree can then be thought of as a disjunction of branches, where each leaf node represents the consequent implied by the satisfaction of the conjunction of all tests in the branch connecting the root node to that particular leaf node.

No.	Attributes				Class
	Outlook	Temperature	Humidity	Windy	
1	sunny	hot	high	false	N
2	sunny	hot	high	true	N
3	overcast	hot	high	false	P
4	rain	mild	high	false	P
5	rain	cool	normal	false	P
6	rain	cool	normal	true	N
7	overcast	cool	normal	true	P
8	sunny	mild	high	false	N
9	sunny	cool	normal	false	P
10	rain	mild	normal	false	P
11	sunny	mild	normal	true	P
12	overcast	mild	high	true	P
13	overcast	hot	normal	false	P
14	rain	mild	high	true	N

Table 2.1: A sample training set. [29]

A possible decision tree generated from the data of Figure 2.1 is shown in Figure 2.1. Quinlan's ID3-based TDIDT machine learners, including C4.5, produce such decision trees by utilizing an estimate of information gained by attribute tests at each node [29]. An initial *window* subset of the data is selected and the attribute test found to contribute the most information to the classification task is selected as the root, and then this process is repeated for the subsequent branches of the current node as the window is widened to eventually include the entire training set. Algorithms in the ID3 family are not designed to seek an optimal decision tree, but rather are designed to accept large training sets with a large number of attributes and produce a "reasonably good" decision tree without extensive computation requirements [29, pp. 352].

A feature vector consists of a conjunction of attribute-value tuples (a, v) where $a \in A$, the set of all valid attributes, and $v \subseteq V_A$, the set of all valid values for attribute A. Each branch of a decision tree may be thought of as being a *decision rule*, consisting of a feature vector and $c \in C$, the

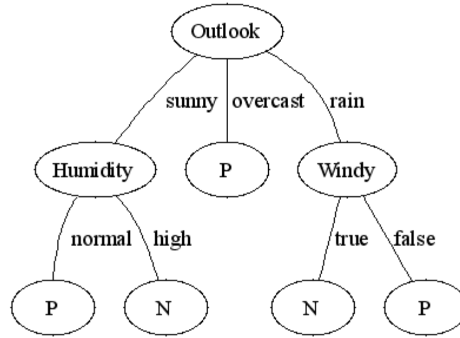

Figure 2.1: Graphical Representation of a Decision Tree [29, pp. 352]

implied classification selected from the set of all valid classifications. A decision rule R_i has the following general form:

$$R_i = a_1 \in v_1 \wedge a_2 \in v_2 \wedge \ldots \wedge a_k \in v_k \rightarrow c \tag{2.1}$$

where $k = |A|$, the number of attributes, and no attribute appears in R_i more than once. This definition symbolizes TREE's *simple disjunctive normal form*, discussed in Section 3. A decision tree T with l leaf nodes then consists of the disjunction of all individual decision rules which represent its branches:

$$T = R_1 \vee R_2 \vee R_3 \vee \ldots \vee R_l \tag{2.2}$$

Given Equation 2.1, the accuracy of a decision rule can be defined as the conditional probability that c is satisfied if R is satisfied, and then from Equation 2.2 the error of the tree can be defined as the sum of the errors of the disjuncted rules. The overall actual error rate of a C4.5 decision tree can then be expressed as the sum of weighted error rates at all of the leaf nodes, as shown in Equation 2.3:

$$E(T) = \sum_{i=1}^{n}(\frac{E(i)}{T(i)} \times \frac{T(i)}{T}) = \frac{1}{T}\sum_{i=1}^{n}E(i) \tag{2.3}$$

where $E(i)$ is the number of erroneously classified cases at node i, $T(i)$ is the total number of cases classified as belonging at node i, and T is the total number of cases.

Tree error is simply then the total number of incorrectly classified cases divided by the total number of training cases. At the end of C4.5's decision tree output is a summary table giving actual and predicted error rate statistics for the given decision tree. An example from an Expert COCOMO Case Study tree is given in Figure 2.2.

```
Evaluation on training data (10000 items):

Before Pruning   After Pruning
---------------- ---------------------------
Size Errors Size Errors Estimate

2069 278( 2.8%)  2055 278( 2.8%) ( 4.1%) <<
```

Figure 2.2: Sample C4.5 Error Rate Summary Table

The graphical decision tree of Figure 2.1 is shown in C4.5 ASCII text format in Figure 2.3. The parenthecized values after each leaf node is the accuracy of that particular rule, measured as the ratio of cases classified correctly by the branch terminating at that leaf node and the number of cases which were consistent with the branch but did not have the same classification. For example, "`windy = true :  N (2.0)`" indicates that this rule correctly classified two cases which satisfied the branch node tests, and there were no cases which satisfied the node tests but did not match the resulting classification of `N`.

```
outlook = sunny :
| humidity = high :  N (4.0/1.0)
| humidity = normal :  P (2.0)
outlook = overcast :  P (6.0/1.0)
outlook = rain :
| windy = true :  N (2.0)
| windy = false :  P (3.0)
```

Figure 2.3: Sample of C4.5 Decision Tree Syntax

Structure in a C4.5 decision tree is indicated by both depth and indentation level, and is recursive. Tree traversal is done in a typical "depth-first" fashion until a leaf node is reached, at which point the recursive descent stops and begins again at the last node visited which has untraversed edges. In Figure 2.3, a horizontal bar followed by three spaces ("`|   `") indicates a level of depth. Therefore, `humidity = high` is located at $d = 1$. Conversion of C4.5 ASCII trees to a linked list structure is covered in more detail in Section 3.

The decision tree in Figures 2.1 and 2.3 could be rewritten as decision rules in the form of Equation 2.1:

$$\begin{aligned}
R_1 &: \text{outlook} = \text{sunny} \wedge \text{humidity} = \text{high} \rightarrow \text{N} \\
R_2 &: \text{outlook} = \text{sunny} \wedge \text{humidity} = \text{normal} \rightarrow \text{P} \\
R_3 &: \text{outlook} = \text{overcast} \rightarrow \text{P} \\
R_4 &: \text{outlook} = \text{rain} \wedge \text{windy} = \text{true} \rightarrow \text{N} \\
R_5 &: \text{outlook} = \text{rain} \wedge \text{windy} = \text{false} \rightarrow \text{P}
\end{aligned}$$

Figure 2.4: Decision Rules Derived from Figure 2.3

Decision tree induction is typically *unstable*, in that the generated classifier will likely undergo significant changes due to small changes in the training data used to create it [9]. This means that ambiguous cases, those which have identical feature vectors but contradictory classifications, and cases which contain faulty or imprecise measurements referred to as *noise* in the data, can cause

errors in the resulting decision tree. Section 3 will present methods which have been devised in part to address this sensitivity to noise in unstable learning algorithms such as those in the TDIDT family of machine learners.

2.3 Knowledge Discovery in Databases

According to Usama Fayyad, the unifying goal of the knowledge discovery in databases (KDD) process is extracting knowledge from large databases [13]. The KDD process integrates data mining (DM) methods in a cycle which begins with data collection and culminates in the interpretation and use of the elicited knowledge by the end user [13]. While the DM step is generally of most importance to the academic investigator, the industrial user is concerned with the entire KDD process, especially with the practical utility of the output [24].

The particular methodologies used in the process can vary with domain and end-user requirements, while remaining a component in the overall KDD process. Fayyad provides a more rotund definition:

> "*Knowledge discovery in databases* is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data." [13]

The first two components of the definition are fairly straightforward. *Valid* patterns are simply those which can classify unseen cases within a desired misclassification rate. *Novel* patterns are those which present the user with some new information, or with desired reinforcement for existing views which lack adequate substantiation. It is important to note that this particular aspect of the KDD process necessitates human interpretation, and is thus fundamentally subjective. Discovering a strong covariance between two particular features known to exhibit such behaviors would have little novelty value.

Potentially useful patterns are those which are would provide utility for the end user [26]. Closely

related to novelty, utility is derived by being able to implement some change based on the discovered knowledge which would result in achieving some desirable outcome. To continue the example, a strong covariance between two features which are completely immutable would, regardless of statistical significance and novelty, be of no utility whatsoever. In other words, a necessary condition for a pattern to be useful is that it is *actionable*.

As several authors note, it is essential that the KDD system allow for discrimination between interesting patterns and uninteresting ones, which relies on the goal of the end-user as being a necessary and beneficial bias within the knowledge discovery process [38] [24]. KDD systems are increasingly user-oriented systems working interactively with the end-user, as opposed to autonomous decision making systems [1]. As a result, there are both objective and subjective measures of interestingness. Among the objective measures are standard approaches of classical statistics. The subjective measures are bifurcated, being measured by their unexpectedness or their actionability.

Unexpectedness is a feature of the descriptive nature of data mining, providing some explanatory information, while actionability is a feature of the predictive nature of data mining [31]. Neither of these aspects alone is sufficient to provide useful knowledge. For example, a physician may find that a particular patient's blood glucose fluctuations are most influenced by his body weight. While significant weight loss is an achievable long-term goal, this information is of little use for short-term avoidance of dangerous levels of blood glucose, as it is not actionable. However, if the physician finds that late evening carbohydrate intake is the next most significant factor in avoiding such dangerous situations, and this is new information regarding this particular patient, then the physician can take the immediate action of prescribing a diet which implements this new

knowledge. Clearly, unexpected patterns are not useful if not actionable, and actionable patterns are of little usefulness if they provide only that information which was already known to the user, and presumably already integrated into the decision framework prior to using the KDD process. Therefore, in order to be useful, the pattern must be both interesting and actionable.

Understandable patterns are those which are interpretable by a human end-user who may not be a KDD expert, or who may have no particular expertise in the knowledge representation used [34]. This has a particular impact on such representations as the weightings in neural networks, complex statistical data analysis methods, and large decision trees representing dozens of decision rules. Recent trends in KDD indicate that more emphasis is being placed on an understandable representation of discovered knowledge, even at the cost of the formal neatness of more logically provable representations [1] [24]. Kodratoff goes as far as to say that “the results of KDD have to be directly usable, even if they are particular, imprecise, and unproved” [24]. Recalling the notion of the utility of the discovered knowledge to the end user, formally precise representations which are difficult to understand may actually provide less utility than less precise representations which are much more easily understood and implemented.

While the real-world users of KDD insist that comprehensibility and usability are most important, academics insist that predictive accuracy is most important, and all but ignore the other two aspects of the process [24]. One reason for this, as Kodratoff notes, is that the former two aspects are largely user-dependent and subjective, while accuracy can be tested empirically and objectively. Involving a “human in the loop”, however, necessitates some intermediate approach involving both of these types of measures. Interactive KDD environments support both “human-assisted com-

puter discovery" as well as "computer-assisted human discovery" [34]. As Uthur notes, the ability of a human user to understand the learned patterns (if displayed in an easily understood format) allows for a faster application of the KDD process than either the computer or the human alone could achieve. According to Uthur, the "complete solution" is an environment which successfully integrates the KDD process as a support in the analyst's existing decision framework. In the end, the final step of the KDD process lies in the strategic use of the knowledge discovered to achieve some competitive advantage [34] .

2.4 Ensemble Learning

Combining an ensemble of classifiers produced by machine learning algorithms is a relatively new technique which has yielded experimental successes in reducing classification errors on unseen cases [11][15][35][17]. In ensemble learning a group of classifiers, called an ensemble, have their decisions combined in some way in an attempt to provide better classification on new cases than any expected individual classifier would produce alone [10][11][15][20][35][36]. When the classifiers used are decision trees, this ensemble is often referred to as a *decision forest* [27].

Ensemble learning is a two-stage process, consisting of the generation of multiple classifiers perhaps with some manipulation of the data prior to training, followed by the application of some combination function to combine the classifications of the ensemble of classifiers. Ho categorizes ensemble combination techniques as being either *decision optimization* or *coverage optimization* techniques [20]. In decision optimization, a select group of highly specialized classifiers are used to attempt to find an optimal combination of their decisions. By contrast, coverage optimization is characterized by the use of complementary generic classifiers in an attempt to provide sufficient coverage of the hypothesis space to provide an optimal classification. As discussed in Section 1, coverage optimization involves the combination of multiple hypotheses $h \in H$.

Dietterich argues that there are three reasons why good ensembles can be created [10]:

Statistical ML involves searching hypothesis space H for an approximation h_i of the true classification function h^*. If there is little training data relative to a large hypothesis space H, then it may be difficult for any particular h_i to closely approximate h^*. This is an effect of the *curse of dimensionality*, which refers to the exponential growth in data required to estimate h relative to the number of variables involved [3, pp. 94].

Computational Finding optimal decision trees is an NP-hard problem [8]. Even in cases where the statistical problem is overcome by a relative wealth of data there is great difficulty for current algorithms to find an optimal solution, requiring significant computing resources compared to the creation of an ensemble of close approximations. In addition, many ML algorithms may get stuck in a local optimum, and the use of ensembles of approximations may allow the algorithm to closer approximate h^* than it could achieve seeking an optimal solution [15].

Representational The true model may not be representable through traditional ML concept descriptions, even in cases where the previous two barriers are surmounted. In ensemble learning, the output of the classification algorithm can be seen as mapping the data to an intermediate space, over which the combination algorithm operates to make the final decision [11]. If the problem, or data used, does not allow any particular classifier to provide a sufficiently close approximation, it is possible that the combination of several nearby classifiers can more closely approximate the true decision function than would be expected by the selection of a single classifier.

Finally, the possible reduction in classification errors provides perhaps the strongest justification for the use of ensemble learning in these cases where it is appropriate. If the errors made by the

individual classifiers used are relatively independent, and their individual error rates are on average less than 50%, then their combination can result in a lower error rate in classifying new cases than the best individual classifier in the ensemble.

2.4.1 Generating Ensembles

In order to provide sufficiently generalized coverage of the hypothesis space to address the representational limitation of single classifiers, ensembles are created, generally using some variation in the type or amount of data which is used as input to the classifier algorithm. The intent of this manipulation is to allow the use of a single data set to generate multiple classifiers which vary sufficiently in the types of errors they make. Typically this is achieved either by restricting which cases are selected for input, or by selecting a subset of the full attribute set A onto which each case in the data set is projected before being used as input. Ho refers to the former method of varying particular cases selected as *training set subsampling*, while referring to the latter method which he pioneered as the *random subspace method* [19].

Bootstrapping is a training set subsampling technique which seeks to vary the particular cases selected as input to the classifier generator. Boostrapping, also often referred to as bagging for Bootstrap AGGregatING, involves pseudorandom sampling with replacement possibly up to a training set size equal to the size of the full data set [10]. Through bagging it is hoped that by varying the cases which are excluded from the training set, the classifiers produced will exhibit sufficiently different errors to allow mutually complementary combination. Even slight changes in the data set used as input can often cause classification differences in the resulting classifiers created with unstable learning algorithms. Data set manipulation is a technique found to work well with

such unstable learning algorithms [15][10]. Duin has found in his ensemble learning research that data set manipulation with the same classifier produced a more accurate ensemble than using the same data set with different classifier algorithms [11]. Wang has also found that in constructing neural net classifiers, such manipulations of the input data set produced more performance improvement than perterbations of other values such as weightings or learning rates [35].

In Ho's random subspace method a particular number of features is selected (Ho chooses half of the total, selected pseudrandomly) and all cases are projected onto this feature subspace. The resulting data set is then used as input to C4.5 in order to produce a single decision tree classifier. This process is then repeated to produce an ensemble of decision trees, each classifier produced using a pseudorandomly selected feature subspace. Fischer also utilized Ho's random subspace method and found that this technique outperformed both similar ensembles with unmodified data sets and the single best classifier [15].

2.4.2 Combination Techniques

Ensemble combination can be effective provided that individual classifiers have less than 50% error rates, and that the classifiers tend to make diverse errors [17], meaning that their errors should be relatively independent. Classifiers with higher than 50% error will simply propagate their errors when combined, and classifiers making the same errors, regardless of error rate, will reinforce those erroneous classifications in direct correlation with the number of classifiers containing the same error. For this reason it may be beneficial to combine classifiers of multiple types, such as decision trees and neural networks, using a combination method such as majority vote, or simply use multiple algorithms which generate the same type of classifier.

While multiple classifier types can be combined, such as decision trees and neural networks, it is necessary to normalize their classification outputs before they can be combined [11]. For example, a continuous-range classifier may require mapping into a discrete categorical space to permit combination with other categorical classifiers. As mentioned earlier, Duin [11] found that modifications to the input data set used as input to the same classification algorithm to produce an ensemble outperformed using the same data set as input to multiple algorithms, therefore the author has chosen to restrict his investigation to the simpler method of combining the results from the same classification algorithm.

Duin presents 3 classes of combination [11]:

1) Parallel combining where the same classification algorithm is applied to different feature sets and then combined with some single combination rule. The different feature sets typically represent wholly different types of data, such as sound and vision, and are not merely subsets of an original feature set.

2) Stacked combining where the classifiers are produced over the same feature space but may be of different types (e.g. neural nets and decision trees).

3) Combining weak classifiers (like DTs). Often these weak classifiers are first produced utilizing one of the previously mentioned data set manipulations.

Ho also discusses combination approaches in the context of the construction of decision forests

[20]. Ho describes parallel combining as an inter-tree combination of decision trees, perhaps utilizing his random subspace data set manipulation, which differs significantly with Duin's parallel combining, but is consistent with Duin's description of combining weak classifiers. In addition Ho also discusses serial combination, which is the intra-tree combination of classification information [20].

While all authors surveyed utilized Ho's parallel approach of decision combination (Duin's third type), the author has devised an approach which seeks to integrate both inter- and intra-tree decision combination utilizing this parallel combination technique, by attempting to consolidate the internal classification information inside each decision tree produced before aggregating that information across the entire ensemble. Duin reports that decision trees can be combined in almost any way to achieve some improvement over their individual estimates [11].

2.4.3 Combination Functions

There are two types of decision combination functions [11, pp. 21]:

1) Fixed combination function: a well-defined function which is independent of the particular ensemble used.

2) Trained combination function: a training strategy which, when operated over a given ensemble, provides a specialized combination function specific to that ensemble or domain.

Among common fixed combination functions are: maximum, median, mean, minimum, product, majority vote [11]. Fixed combination functions tend to apply a mathematical rule to the aggregate

classification decisions of each member of the ensemble. The aggregation is typically either discrete, such as with majority vote where each classifier's vote for the correct classification is aggregated and the majority decision is used, or real-valued, as in using the sum of the estimated posterior probabilities or confidences for each class given by each classifier, with the classification with the maximum total estimated confidence selected as the correct classification.

Trained combination functions directly utilize posterior probabilities as an intermediate feature space on which some combination technique operates. Such techniques include Bayes algorithms, nearest mean, and nearest neighbor [11]. However, techniques which rely on estimates of posterior probability can in practice be very sensitive to errors in these estimations [11]. Duin notes that combining classifications given by techniques utilizing posterior probability estimation techniques which make dissimilar errors can possibly counteract the problems imposed by making such estimates. In addition, while trained combination strategies have been shown in cases to outperform generic fixed combination functions, they require training of the combination function which may require more expert knowledge within the KDD process than simpler fixed functions.

Although they require no training and much less expert knowledge, even simple fixed combination functions have been shown to produce accuracy gains over single classifiers. Kittler combined ensembles generated from the same dataset using various fixed techniques and found improvement using even the simple fixed combination functions of sum and majority vote [23]. In addition, Duin found fixed majority vote obtained the best error rates in 5 of the 6 data sets used, and in the sixth the best method was trained Nearest Mean. Ho used a simple majority vote and found that using the same classifier method on a modified feature set produced better results than combining

different types of classifiers produced over the same feature set [19]. Fischer also found better accuracy utilizing a majority vote combination of an ensemble of decision trees compared to the best single decision tree produced [15].

2.4.4 Empirical Results of Ensemble Learning

Fischer utilized decision forests of C4.5 trees to classify diatoms (unicellular algae). Microscopic images were analyzed to obtain data on various features of the subject in question which might be used as a basis of discriminating between various classes of diatoms. The task of identifying diatoms is inherently difficult, as it is estimated that some 100,000 distinct species exist and that another 100,000 may yet be discovered [15]. The 149 features used as input to C4.5 were: moment invariants (11), fourier descriptors (126), scalar shape descriptors (5), symmetry (1), geometric properties (4), diatom specific features (2).

Fischer employs three experimental approaches. First single decision trees were constructed using the full data sets. The second approach utilized the bagging bootstrapping method to augment the data sets. In the third method, 100 random subsets of exactly one half of the available features were used for constructing the decision trees. The second and third methods then utilized a majority voting technique for making the final decision.

In calculating classification accuracy, Fischer considered whether the technique correctly classified a case within its top six ranked candidates. The actual implementation of Fischer's technique provided these selections to the domain user who would then make the final determination. Figure 2.2 shows the classification accuracy in percent correct classifications for first rank, as well as

for the accuracy plateau rank. The accuracy of the best single decision tree was always significantly less than the random subspace ensemble, with only minor gains in accuracy when considering successive classification decisions. Both ensemble techniques achieved higher accuracy in their first choices, with significant gains in accuracy when considering successive alternate choices.

	data set 1		data set 2	
method	rank 1 %	max acc %, rank	rank 1 %	max acc %, rank
single best	≈ 91	93.33, 3	65.53	≈ 70, 1
bagging	92.5	99.17, 2	77.13	95.74, 5
subspace	≈ 95	100, 3	79.26	96.81, 5

Table 2.2: Empirical Results of Fischer's Combined Classifiers

Duin and Ho perform similar experiments involving combination of different feature sets. Duin provides a more comprehensive application of different trained and fixed combining rules applied to individual feature sets and the entire feature set. In his experiment different standard image analysis techniques were applied to images of handwritten numerals found on Dutch utility maps. In all there were 649 features measured. There were 10 classifiers used utilizing such techniques as decision trees, Bayes analysis, nearest mean, neural networks. There were also 6 fixed combining techniques: maximum, median, mean, minimum, product, majority, and 4 trained combiners: Bayes-normal-2, Bayes-normal-1, nearest mean, and nearest neighbor.

Duin [11] found that the best results came from combining across differing feature sets rather than combining the 10 different classifiers produced from one feature set. This is in agreement with Ho's findings regarding combining across differing feature sets. In both types of analyses, fixed combining rules tended to produce the lowest error more often than trained combining rules.

The worst results overall in classification error were exhibited with decision trees where the single best error rate using only decision trees was 102%, achieved combining decision trees produced from the individual feature sets using the nearest mean combination function. This result however is a significant improvement from the 480% error for the decision tree method applied to the entire feature set. In addition, most combination methods resulted in significant reductions in error, with all but two combination functions reducing error by more than half. Decision tree, however, was not the only approach which exhibited such large errors. The best neural network approach showed 896% error using the entire feature set. Unlike decision tree, however, neural network combination using trained combination functions achieved error rates below 50%.

The decision tree was not the only method which performed poorly when applied to the image processing metrics used in Duin's experiment. It is noteworthy that all combination techniques yielded significant reductions in the error rates of the resulting combined decision tree classification, even when the inputs trees exhibited such large errors.

Ho [19] compares his random subspace method for constructing ensembles with both single C4.5 decision trees and with bootstrapping methods of subsampling. The decision trees constructed are fully split, such that if there are no ambiguities in the training data, each training case is correctly classified. To populate the decision forest, Ho constructed 100 trees pseudorandomly selecting half of the full feature set for each tree. The trees were fully split such that they would produce no errors given unambiguous data. The risk of overtraining posed by fully-split trees was addressed through the generation of an ensemble of such trees. He then combined the class conditional probabilities of each tree using a mean fixed combination function. The final classification of a combined forest

was then the class which had the maximum mean conditional probability.

Ho conducted his experiments using four data sets which contained no missing values. The number of features in the data sets varied from a low of 9 to a high of 180. Ho found that his method of selecting half of the features to use for each tree of the ensemble did not demonstrate significant improvement over a single tree when applied to the data set of 9 features. Although Ho does not provide specific numerical data, the approximate misclassification rates of the various methods can be determined from his supplied graphs, and is given in Figure 2.3. In all trials, using a combined ensemble of trees produced gains in accuracy over the use of the best single C4.5 tree, and in all trials the random subspace method produced the highest levels of accuracy.

		Ensemble Technique			
data set	features	single	bootstrap	boost	random subspace
dna	180	92.5	$\approx$ 94.5	$\approx$ 95	$\approx$ 95.5
letter	16	$\approx$ 89	$\approx$ 93	$\approx$ 93	>96
satimage	36	$\approx$ 86	$\approx$ 90	$\approx$ 90	$\approx$ 91
shuttle	9	$\approx$ 99.96	$\approx$ 99.96	NA	> 99.98

Table 2.3: Empirical Results of Ho's Random Subspace Method [19]

In conclusion, Duin has reported that almost any of the combination functions increase performance given lower than 50% error in the average member of the ensemble with simple majority vote showing consistent benefits. Kittler, Ho, Fischer, and Duin have found improvement using either sum or majority vote. Consistent with the problem statement which requires a human end user in the loop who is not expected to be a KDD or data analysis expert, the author has chosen to restrict his explorations to using a simple fixed combination function which is both easy to explain and understand, and is easy to implement. This approach will be outlined in Section 3.2.

2.5 Interestingness

The definition of KDD requires that discovered knowledge be "interesting", meaning that this must supply the user with some new or surprising information [26]. This implies that the information may also contradict existing beliefs. Finding a statistical deviation is not enough to satisfy this KDD requirement. The task of discovering interesting knowledge involves selecting particular pieces from the potentially large number of the deviations discovered. These pieces are judged on how well they meet the entire definition of the KDD process. Thus, while much research has been focused on refining strictly mathematical measures of interestingness, many of which are defined in [18], these measures fail to meet the requirements of comprehension and actionability stated in the definition of the KDD process in Section 2.3.

Interestingness is closely related to the other requirements of the KDD process, especially that of actionability. While an actionable item of knowledge may not be interesting to the end user, an interesting item of knowledge must be actionable to be interesting in any practical sense. Knowing that if one could change some immutable factor one would see desirable results is less than useful information to the real-world user facing short-term decision making, and while interesting in an academic sense is not interesting from the point of view of the KDD process. The goal of the KDD process, in industry, is not merely to find interesting knowledge, but to do so in a manner which provides the user some competitive advantage [26]

In addition to being actionable, to be interesting knowledge must also be understandable, which is to say the end user must comprehend the given representation to an extent which allows actions to be taken as a result of the discovered knowledge. This implies that it is preferable to have more

easily understood representations even if that may result in some loss of accuracy. James emphasizes that in some cases "the presentation of a classification rule is as important as its performance and a comprehensible classifier is preferred as long as its error rate is reasonable" [21, pp. 183]. And Simoudis says, "the quality of each rule is measured by its discriminating power, its generality, and its interestingness." [32] And so it is necessary to include both accuracy and comprehensibility in knowledge representations, without relying purely on subjective measures of comprehension.

While there may be no psychologically well-founded definition of comprehensibility [16], this does not mean that to address comprehensibility we must rely exclusively on subjective measures of comprehension. Matheus suggests using a user-defined utility measure to weight the interestingness of discovered knowledge to combine objective and subjective measures of interestingness [26]. For example, the user could weight the relative institutional costs of changing certain features, where not all costs are purely objective. Findings could then be weighted by the net benefit of taking advantage of the discovered knowledge. In situations of limited resources, discovered knowledge could then be ranked such that preference is given to those items of knowledge which have highest domain or institutional utility.

In conclusion, measures of interestingness for discovered knowledge must be interesting to the end user who may not be a KDD expert versed in mathematical descriptions of information. We may define interestingness by the general statement,

> ... a pattern is interesting relative to some belief system if it "affects" this sytem, and the more it "affects" it, the more interesting the pattern is [31].

Chapter 3

The Approach

3.1 Introduction

To address the problem presented by the accuracy-comprehensibility constraint on presenting discovered knowledge in a human-comprehensible representation we developed an approach utilizing an *ensemble* of multiple classifiers, combined in such a way that the user is presented with an "importance table" showing the feature ranges found to be the most significant in changing the classification from one outcome to a significantly different, and generally preferential one. The purpose of this table is to provide a scalable representation of elicited knowledge which is not circumscribed by the accuracy-comprehensibility constraint when modeling complex representations in highly-dimensional and large data sets.

3.2 Analysis Technique

3.2.1 Overview

The following lessons learned from the literature review of machine learning, knowledge discovery, and ensemble learning give structure to the author's experimental approach.

Knowledge Requires Understanding Discoveries qualify as knowledge only insofar as they are understood by the human end-user.

Knowledge Must Be Actionable Knowledge discoveries which are interesting yet cannot be acted upon in the target domain have little practical interest to the end-user seeking aid in the decision making process.

Ensembles Work Coverage optimization using generic classifiers to approximate h^* can improve predictive accuracy over single trees while remaining consistent with the requirements of the definition of KDD.

Ensembles Can Be Homogenous Parallel combination (as from Ho) using multiple classifiers of the same type has been used to improve predictive accuracy while not requiring the ability to successfully combine classifiers of different types.

Simple Combination Functions Work Fixed combination functions, while often giving slightly less predictive accuracy than trained functions, still improve accuracy while remaining consistent with the problem statement which requires the KDD methodology be able to be understood and implemented by an end-user not expert in data analysis or KDD methodology.

For the purposes of the experiment, it was necessary to simulate the environment described in the problem statement where a data glut exists and some end-user who is a non-expert in data analysis seeks to use this data as an aid in decision making and resource allocation. To this end, a popular software cost risk estimation model was obtained and pseudo-randomly exercised to produce simulated data sets which could be used as surrogates for data collected through a real-world KDD process.

3.2.2 Experimental Setup

The COCOMO 2.0 Heuristic Risk Estimator, was obtained in the form of C source code and modified to allow command-line parameter input for all atttributes. The inputs to the COCOMO 2.0 Risk model were allowed to vary within their full valid ranges, with the exception of Source Lines Of Code (SLOC) which was allowed to vary within certain specified ranges of interest, and was allowed to take on only a certain number of evenly-spaced values within that range. The output from the COCOMO 2.0 Risk model is a numerical risk estimate which is also classified linguistically which allowed direct translation into input for C4.5. Both risk scales for Expert COCOMO 2.0 can be found in Figure 4.2 on page 56

Once datasets were generated for each particular virtual scenario, the popular C4.5 decision tree inducer created by Ross Quinlan [28] was used to produce decision trees from these datasets. Source code for C4.5 was obtained and modified to execute on a personal computer with a Microsoft Windows NT[1] operating system (the native environment of the supplied version of C4.5 was a Sun

[1]Windows NT is a registered Trademark of Microsoft Corp.

```
outlook = sunny :
humidity = high :  N (4.0/1.0)
humidity = normal :  P (2.0)
outlook = overcast :  P (6.0/1.0)
outlook = rain :
windy = true :  N (2.0)
windy = false :  P (3.0)
```

Figure 3.1: C4.5 Decision Tree Without Level Spacing

Solaris[2] operating system, and no Makefile compiler file was supplied to allow compilation in other environments). No internal C4.5 heuristics were modified, however the tree output parameters were modified so that the visual alignment string "`| `" would not be printed to designate each successive level of depth at a node. Due to this modification, which alternately could have been done with a text file post-processor if access to C4.5's source code had not been possible, the example C4.5 decision tree in Figure 2.3 is saved to file as shown in Figure 3.1. Because the tree is stored in a consistent recursive format, no level designation characters are required.

We write the decision tree analysis program (TREE) which accepts a command-line parameter list of C4.5 trees to analyze as a single ensemble. If a simplified tree is included in the same file as the original, the simplified tree is used. Each tree is converted into *disjunctive normal form* [37, pp. 114]. In this form, a decision tree is expressed as a disjunction of *feature vectors* [12]. Each feature vector is simplified such that each attribute appears only once, followed by the set of values which are valid within that vector. The result of this simplification will be referred to as *simple disjunctive normal form*.

Recall from Section 2.2 that a feature vector consists of a conjunction of a non-empty set of

[2] Solaris is a registered Trademark of Sun Microsystems Inc.

attribute-value pairs (a, v) where $a \in A$, the set of all valid attributes, and $v \in V_A$, the set of valid values for attribute A. Recall also that each branch of a decision tree, or each path from root to leaf node, can be thought of as a feature vector and its implied classification at the leaf, and that each branch can be thought of as a decision rule. As each tree is read and transformed into simple disjunctive normal form, a combinatorial intra-tree comparison is executed between all R rules, where R is the number of leaf nodes l. This results in an upper-bound of l-choose-2 combinations, or $C_2^l = l \times (l-1)/2$ comparisons. The algorithmic complexity of TREE will be addressed in Section 3.2.6.

Search space is reduced by comparing only those feature vectors of significantly different classification values. For example, two feature vectors, both of classification “High Risk” would not be compared. Feature vectors of classifications “High Risk” and ”Medium Risk”, separated by one ordinal position only, would also not be compared, as the stated goal of the investigation is to present a mechanism for finding the most significant factors. If a factor was found to be significant only within one ordinal position, then it would possess limited domain utility, and the significance of the result could undermine its usefulness.

The decision of what constitutes a significant difference is subjective, and domain specific, yet this is precisely what the problem statement gives as being a necessary aspect of real-world use of the KDD process. The end-user would be necessarily involved in setting parameters for what would be considered an interesting change. For the COCOMO 2.0 Heuristic Risk Model, a difference in risk of two ordinal positions or more was considered by this author to be significant.

When two rules are are found to differ significantly in outcome, an attribute-by-attribute comparison is made between them, searching for differences in the values of attributes they both include. If the aim of the study is to discover what is important in discovering what can determine a low risk project instead of a high risk one, the set of values from the high risk rule would be subtracted from the values of the low risk rule, for shared attributes. This is the equivalent of asking the question: "What values of an attribute, *ceteris parabus*, will change my risk classification from that of the worse rule to that of the better rule?" An *importance table* is a frequency table of all such instances of an attribute's value existing in this set difference. The importance table answers the question: "How often, *ceteris parabus*, can an attribute's value possibly determine whether the classification

Attribute	1	2	3	4	5	6
SLOC	0	216	187	96	103	188
PREC	70	71	60	75	93	0
FLEX	72	46	42	47	57	0
ARCH	84	29	27	32	51	0
TEAM	75	86	157	238	295	0
PMAT	360	265	587	896	1355	0
RELY	8073	7221	6476	3736	5653	0
DATA	0	48	36	32	74	0
CPLX	14700	13845	12675	10643	8600	7624
RUSE	0	9912	9512	8237	4316	4486
DOCU	50	32	25	21	77	0
TIME	0	0	16418	14482	9214	10170
STOR	0	0	5369	4349	3012	2722
PVOL	0	356	262	141	265	0
ACAP	12150	10497	27575	32439	34214	0
PCAP	12556	11468	27855	33213	34552	0
PCON	45	12	19	42	105	0
AEXP	2356	2548	4352	6557	7078	0
PEXP	621	532	1045	1382	2077	0
LTEX	1872	2003	4659	5974	6780	0
TOOL	3139	4382	7523	9062	10122	0
SITE	108	66	52	57	62	109
SCED	12464	9144	31954	35102	39280	0

Table 3.1: Example Importance Table

is of the ‘better’ class as opposed to the ‘worse’ class?”

If R_2 has the more desirable classification, and R_1 has the less desirable classification, then the *changes set*, $\Delta_{2,1}$, is the set difference v_a2/v_a1 for all attributes a which exist in both R_2 and R_1:

$$\Delta_{1,2} = \{(a, v) : a \in R_1 \wedge a \in R_2, v_{a2} \backslash v_{a1}\} \tag{3.1}$$

Once a particular tree’s importance table has been completed, the values are added to the ensemble importance table of the same dimensions. This step implements a simple sum fixed combination function, although not quite in the same inter-tree parallel combination approach as the authors surveyed, since it is neither a sum of posterior probabilities nor is it a sum of votes for particular classifications. Since the operation involved is a simple sum, the single-tree importance table could be eliminated saving a small amount of memory and $\bigcirc(A \times V)$ execution time. However, maintaining a separate table allows for easy future modification of either function. For example, the individual importance table may be weighted by the tree’s overall estimated predictive accuracy, giving less weight to knowledge gleaned from less accurate trees, and the addition of a tree’s importance table to the ensemble importance table could be weighted by the overall accuracy of that particular tree. Figure 3.1 shows a typical single-tree importance table using a tree generated by C4.5 using Expert COCOMO 2.0 data.

3.2.3 Implementation

Step 1: Parse the C4.5 Trees

The first step of TREE involves reading in C4.5 ASCII trees and converting them into a linked list of rules.

In this stage, the decision test at each C4.5 node is fully preserved. So, a node with test "`ACAP > 4`" would be preserved in a record data structure with three values: attribute, operation, value. For simplicity, inclusive operators such as ">=" are transformed into exclusive operators like ">" and the test value is incremented in the least significant digit. For the models studied in the experiment these values were integers, but TREE was designed to increment or decrement the least significant digit of strings expressing decimal point values. Note that this assumes the decision tree will express precision in floating point comparisons, such that "`>= 3.2`" and "`>= 3.20`" are both possible and not equivalent comparisons.

Since C4.5 stores trees in a consistent depth-first representation (right- or left-recursive is indistinguishable so long as the vertical order of the expansion is consistent), a depth-first recursive algorithm can be used to efficiently convert the C4.5 file into a set of rules in a single pass through the tree file without look-ahead.

Step 2: Rule Simplification

In the second stage, the rules are transformed into simple disjunctive normal form (see page 36). The first part of this process involves sorting rules by attribute name. This is accomplished by a

simple bubble sort applied to each rule.

In Step 1, rules are stored as a linked list of records which contain three atoms: {attribute, operator, value}. The operator can take on the following values: “>”, “=”, “<”. This is the logical equivalent of expressing a set of values in the ordinal list of valid values for the attribute. For example, {“ACAP”, “>”, “2” } expresses the set defined by the range [3, max(V_{ACAP})]. Resolving this tuple with {“ACAP”, “<”, “5” } then involves finding the set intersection between these two implied sets.

To express the conjunction of multiple ranges, the tuple is transformed into an attribute range record of three atoms: {attribute, v_{min}, v_{max}}. No branch of the decision tree produced by C4.5 can contain unsatisifiable contradictions between decision tests as it then has no cases classified and is pruned with 100% confidence. Each additional decision test tuple of the same attribute then either expresses the same values set, a superset, or a subset of the current set intersection. A superset results in no change to the set intersection, as does an equivalent set. The only possible change made by the additional record in the intersection is a restriction in one or both of the range endpoints. And so the previous two records are resolved as follows:

$$\{\text{“}\texttt{ACAP}'', \text{“}>'', \text{“}2''\} \cap \{\text{“}\texttt{ACAP}'', \text{“}<'', \text{“}5''\} = \{\text{“}ACAP'', 3, 4\} \tag{3.2}$$

At the end of this step, each rule will contain at most $|A|$ attributes, with each attribute appearing at most once in any given rule.

Step 3: Rule Comparison

After rules are transformed into simple disjunctive normal form, a combinatorial comparison takes place between all rules. The useful comparisons have been declared to be those between rules without outcome classifications which differ by more than one ordinal position, however comparisons involving all classifications are still performed to determine the ordinal distance between classes.

When two rules are found to differ in classification by more than one ordinal position, TREE traverses the linked list of the destination, or "better" rule, and for each attribute found there it searches for that attribute in the linked list of the origin, or "worse" rule. Since both rules are sorted alphabetically by attribute name, the position pointer in the origin rule need only be advanced to perform each search, and will never need to restart or revisit any entries. If an attribute lexicographically greater than the search parameter is found in the origin, the search is terminated and the current pointer in the destination linked list remains in that position for the next search.

If a match is found, TREE creates a changes set record, which is an attribute range record as described in the previous step, by subtracting from the values set in the destination rule the values set of the origin rule. This is accomplished by pair wise comparisons of the minimums and maximums of the values set ranges. The larger of the two minimums and the smaller of the two maximums is written to the changes set attribute range values. The change is then written to the tree's importance table by incrementing the matrix entry corresponding to each value in the changes set for that particular attribute. The changes set is included as a potentially unnecessary step in this process to preserve the possibility of future alterations to the combination function with minimal source code modification.

Step 4: Update Ensemble Importance Table

The final step is a pair-wise addition of identical matrix indecies, adding each tree importance table entry to its corresponding ensemble importance table entry. This step requires a simple nested loop. As with the changes set of the last step, a potentially unnecessary tree importance table is preserved to allow easy future modification of the combination function.

Since the only information retained between processing of each tree is the ensemble importance table, the only scalability limit on the number of trees which can be processed is total execution time and the size of required variable storage in the importance table (`MAX_INT` or equivalent system-dependent value). The storage restriction could be addressed through programming strategies. The execution time constraint, while reduced to a low polynomial, would still be significant if processing large numbers of very complex trees.

3.2.4 Proof of $l = \frac{1}{2}(n+1)$

It is possible to prove inductively that the relationship between the number of total nodes and the number of leaf nodes in a full binary tree is completely independent of height and tree structure, and that this relationship is $l = \frac{1}{2}(n+1)$.

Consider the base case of $n = 1$, or a lone root node. Trivially, in this case $n_1 = l_1 = 1$, as the root is both the only node, and is a leaf node.

Now consider the next largest full binary tree, $n = 3$. This results from the addition of two nodes, and the net addition of one leaf node as the root changes from a leaf to an internal node.

Adding two nodes then results in the following ratio:

$$\frac{n_3}{l_3} = \frac{n_1 + \Delta n_1}{l_1 + \Delta l_1} = \frac{1+2}{1+1} = \frac{3}{2} \tag{3.3}$$

Now consider the first non-trivial case, where $n = 5$. We must choose one of the two descendants of the root to populate. Regardless of choice, we add two more nodes, and one more net leaf node. Again, the addition is in the same ratio:

$$\frac{n_5}{l_5} = \frac{n_3 + \Delta n_3}{l_3 + \Delta l_3} = \frac{3+2}{2+1} = \frac{5}{3} = \frac{n_5}{\frac{1}{2}(n_5+1)} \tag{3.4}$$

Consider the case of $n = k$. From the final denominator of Equation 3.4 we can state the inductive hypothesis of l as a function of n:

$$l_k = \frac{1}{2}(k+1) \tag{3.5}$$

Now consider the inductive case of $n = k + 2$, the next largest binary tree. Substituting into Equation 3.5:

$$\begin{aligned} l_k + \Delta l &= l_{k+2} \\ \frac{1}{2}(k+1) + \Delta l &= \frac{1}{2}((k+2)+1) \\ \frac{k+1}{2} + 1 &= \frac{k+2+1}{2} \\ \frac{k+3}{2} &= \frac{k+3}{2} \end{aligned} \tag{3.6}$$

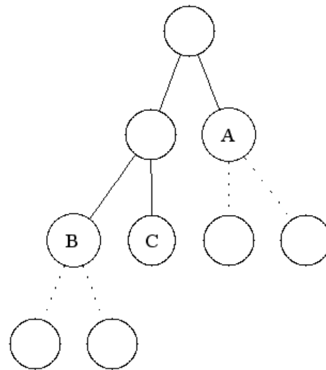

Figure 3.2: Adding Children in a Full Binary Tree

3.2.5 Proof of Worst-Case Total Rule Size

Consider the construction of a full binary tree such that $|R|$ is the sum of the paths to the leaf nodes. In a trivial case of a root with two children, $|R| = 4$, since each leaf forms a path of size 2 with the root. With this tree of size $n = 3$, there is no difference in $|R|$ between adding two children to one leaf node or the other. In both cases, adding two children results in the subtraction of a path of size 2, replacing it with two paths of size 3. The result is then $|R| = 4 - 2 + 6 = 8$.

The tree just described is shown in Figure 3.2. Nodes B and C are those added from the previous step. Now consider adding two more children, which must be added either at A, B, or C. If children are added to A, the following total path change results:

$$\Delta|R| = 2(|A| + 1) - |A| = |A| + 2 \tag{3.7}$$

where $|A|$ represents the path length of node A. The path lengths $|B|$ and $|C|$ are equivalent, and

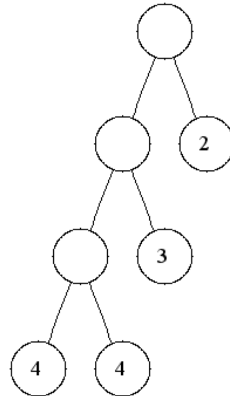

Figure 3.3: A Perfectly Incomplete Full Binary Tree. Numeric labels at leaf nodes indicated —R— at that node.

adding children at B results in the following change:

$$\Delta|R| = 2(|B| + 1) - |B| = |B| + 2 \tag{3.8}$$

However, $|B| = |A| + 1$. By substitution in Equation 3.8, the change from adding to B or C is:

$$\Delta|R| = 2(|A| + 1 + 1) - (|A| + 1) = |A| + 3 \tag{3.9}$$

Therefore, the maximum size of $|R|$ comes from creating a tree by always adding new children to a node of maximal depth, resulting in a tree which has two nodes at each level below the root. The author terms this tree structure a *perfectly incomplete binary tree.*

We can now determine the size of $|\bar{R}|$, the mean path size. In a perfectly incomplete binary tree, there is a leaf node at every depth below the root, and two at the maximal depth, h. Such a tree is shown in Figure 3.3, where the leaf nodes are labeled with their path lengths. The total path size $|R|$ is found by taking the sum of all such leaf path weights. Since at each depth d below the

root, $|R| = d + 1$, and there is one additional path at the maximal depth which has size $h + 1$, we can express the total path length of the worst-case tree:

$$\begin{aligned} |R| &= \sum_{d=1}^{h}(d+1) + (h+1) \\ &= \sum_{d=1}^{h}(d) + h + (h+1) \\ |R| &= \frac{1}{2}h(h+1) + 2h + 1 \end{aligned} \tag{3.10}$$

For a perfectly incomplete full binary tree, $l = h + 1$, as there is a leaf node at every depth below the root, with an additional leaf node at the maximum depth. Substituting into Equation 3.10:

$$\begin{aligned} |R| &= \frac{1}{2}(l-1)(l-1+1) + 2(l-1) + 1 \\ |R| &= \frac{1}{2}l(l-1) + 2l - 1 \end{aligned} \tag{3.11}$$

It is then possible to find the mean rule length $|\bar{R}|$, by dividing Equation 3.11 by l, the number of rules:

$$\begin{aligned} |\bar{R}| &= \frac{\frac{1}{2}l(l-1) + 2l - 1}{l} \\ |\bar{R}| &= \frac{1}{2}(l-1) + 2 - \frac{1}{l} \end{aligned} \tag{3.12}$$

We can then substitute Equation 3.5 into Equation 3.12 to express $|\bar{R}|$ as a function of n:

$$\begin{aligned} |\bar{R}| &= \frac{1}{2}(\frac{1}{2}(n+1) - 1) + 2 - \frac{1}{\frac{1}{2}(n+1)} \\ &= \frac{1}{4}(n+1) - \frac{1}{2} + 2 - \frac{2}{n+1} \\ |\bar{R}| &= \frac{1}{4}(n+1) + \frac{3}{2} - \frac{2}{n+1} \end{aligned} \tag{3.13}$$

3.2.6 Complexity Analysis

The execution analysis of TREE will be expressed in terms of tree size, quantified as the number of nodes n. From Equation 3.5, the size of a full binary tree may be expressed either in terms of nodes or leaf nodes, and is irrespective of the height or structure of the tree, and the relationship $n = 2l - 1$ holds for all such trees.

Step 1: Parse the C4.5 Trees

Since each line of the ASCII tree contains exactly one node of the tree, and each node requires at most one line, the execution time of this conversion is on the order of $\Theta(n)$.

$$T_1(n) \in \Theta(n) \tag{3.14}$$

Step 2: Rule Simplification

The implementation of the set intersection of multiple tuples involves first sorting the list of decision test tuples by attribute. A standard bubble sort is utilized on the linked list, requiring $\bigcirc(m^2)$ on m objects. The sort is two nested loops, the outer executes m times, while the inner executes m minus the number of completed outer loops. This gives a sum of the first m integers as the number of executions of the innermost statements: $\sum_{i=1}^{k} i = \frac{1}{2}n(n+1)$.

In this case however , n is the number of nodes for the entire tree, which is not sorted. Rather, each rule is sorted, and if $|\bar{R}|$ is the number of nodes in each rule, the total sort time would then be $\bigcirc(l \times |\bar{R}|)$. As proven in Section 3.2.4, regardless of tree structure, for a full binary tree $l = \frac{1}{2}(n+1)$. Sorting the l rules is then done on the order of $\bigcirc(\frac{1}{2}(n+1) \times |\bar{R}|)$. Substituting Equation 3.13, we

can express the sorting execution time as a function of n:

$$\begin{aligned} T_2(n) &= \bigcirc(\frac{1}{2}(n+1)\times(\frac{1}{4}(n+1)+\frac{3}{2}-\frac{2}{n+1})) \\ T_2(n) &\in \bigcirc(\frac{1}{8}(n+1)^2+\frac{3}{4}(n+1)-1) \end{aligned} \tag{3.15}$$

Bubble sort is an inefficient sorting technique, and is not generally advised for lists of more than one hundred items. Its use in this case for reasons of simplicity is justified by considering that the sort is not actually being conducted on n, but rather is being done with l rules of varying size. This gives a range for the general sort time for any particular rule, which has the minimum of a complete binary tree, $\Omega((\log_2(\frac{n+1}{2}))^2)$, and the maximum of a perfectly incomplete binary tree, $\bigcirc((\frac{n+1}{2})^2)$. An absolute upper bound could then be placed on the sort procedure by assuming all rules have length $\frac{n+1}{2}$, where the number of rules is $l = \frac{n+1}{2}$, which would give an unreachable upper bound for sorting the entire tree as $\frac{n+1}{2} \times (\frac{n+1}{2})^2 = (\frac{n+1}{2})^3$.

Step 3: Rule Comparison

Since every rule is compared to every other, the minimum execution time for this step is binomial and is C(l, 2):

$$\begin{aligned} T_3(n) &= \bigcirc(\frac{l!}{2!(l-2)!}) \\ T_3(n) &= \bigcirc\frac{1}{2}l(l-1)) \end{aligned}$$

and, by substitution of Equation 3.5:

$$\begin{aligned}
T_3(n) &= \bigcirc(\frac{1}{2}(\frac{1}{2}(n+1))(\frac{1}{2}(n+1)-1)) \\
&= \bigcirc(\frac{1}{4}(n+1)(\frac{1}{2}n+\frac{1}{2}-1)) \\
&= \bigcirc(\frac{1}{4}(n+1)\frac{1}{2}(n-1)) \\
&= \bigcirc(\frac{1}{8}(n+1)(n-1)) \\
T_3(n) &\in \bigcirc(\frac{1}{8}(n^2-1)) \qquad (3.16)
\end{aligned}$$

Step 4: Update Ensemble Importance Table

The ensemble importance table is a master table which contains the sum of frequency counts of changes found in each tree. The dimensions of all tables are the same, which is the number of attributes by the maximum number of values possible from all attributes. For example, the COCOMO 2.0 trees contained 23 attributes, with a maximum of 6 values, resulting in a 23×6 matrix. This execution time is independent of tree size, and so is only a small constant addition to the total execution time:

$$T_4(n) = \Theta(c) \qquad (3.17)$$

3.2.7 Total Execution Time

The total execution cost from these four stages is then:

$$\begin{aligned}
T(n) &= T_1(n) + T_2(n) + T_3(n) + T_4(n) \\
&= \Theta(n) + \bigcirc(\frac{1}{8}(n+1)^2 + \frac{3}{4}(n+1) - 1) + \bigcirc(\frac{1}{8}(n^2-1)) + \Theta(c) \\
&= \bigcirc(n) + \bigcirc(\frac{1}{8}(n+1)^2 + \frac{3}{4}(n+1)) + \bigcirc(\frac{1}{8}(n^2-1)) \\
&= \bigcirc(n) + \bigcirc(n^2) + \bigcirc(n^2) \\
T(n) &\in \bigcirc(n^2) \qquad (3.18)
\end{aligned}$$

Chapter 4

Case Study: Expert COCOMO 2.0

4.1 Introduction

Accurately predicting the costs of software projects has been notoriously difficult [7]. The specific difficulty with this task is that estimating the costs of software projects, and the related risk of cost overrun, is significantly determined by organization-specific attributes [22], and the estimate is done prior to the start of the project, at a time when many if not most relevant factors are still unknown [2]. Not only does this make data collection of known factors difficult, no model can include all factors which affect the effort of a software project [22]. As late as 1998 the Air Force Research Laboratory reported that estimation models' predictive accuracy of software project cost, schedule, and effort on U.S. Department of Defense software projects were found to be within 25 percent of actuals only around 50 percent of the time [14].

In addition to often having large inaccuracies, estimates are merely attempts to identify the outcome which has a mean probability of being correct. This means that the output of a software

effort prediction tool includes an implicit range of possible values centered around the estimate (in the case of normally distributed errors). In addition, errors in measurement and errors in model behavior are additive [22]. Therefore, project managers are faced with a situation where they face a seemingly impossible task of trying to estimate the cost, and cost risk, of a particular software project before that project has begun. While the estimates of such software cost tools cannot generally be well-trusted at this stage of unique projects, project managers could benefit from knowing which specific aspects of their software development environment could most effectively reduce the risk of cost overruns, without needing to be able to specifically quantify such risks or risk reductions. The method outlined in Section 3 seeks to provide such resource allocation information in order to provide this domain utility to the software project manager.

4.2 COCOMO

Software cost tools have often been parametric in nature, created by empirically investigating what factors of influence should be measured as cost metrics, and using statistical techniques to form equations involving these metrics [5]. Such models, often derived through a statistical regression to determine the coefficient values, resemble the form $\mathit{Effort} = A \cdot (\mathit{Size})^B$ [6]. Barry Boehm developed the parametric software cost tool COCOMO, the COnstructive COst MOdel[1], in 1981, by utilizing the project information of 64 software projects of varied types and industry [4]. In Boehm's COCOMO family, A includes the effect of *cost drivers* while B includes *scale factors*. As of USC COCOMO II's Post-Architectural model, this amounted to 17 cost drivers and 5 scale factors [6]. An abbreviated explanation of these attributes can be found in Appendix B, and on

[1]COCOMO, the original model, is currently referred to as COCOMO81. The COCOMO project is located at http://sunset.usc.edu/research/COCOMOII/index.html

the WWW[2].

The basic mathematical form of the COCOMO II cost model is:

$$PM = \alpha \cdot (KSLOC^{(1.01+\sum_{i=1}^{5} SF_i)}) \cdot (\prod_{j=1}^{17} EM_j) \quad (4.1)$$

where PM is effort in person-months, $KSLOC$ is thousands of source lines of code, and α is a constant [6]. The SF_i are the *scale factors* which affect effort as an exponential function of size, and the EM_j are the cost drivers. All attributes in the COCOMO model except source lines of code can take on a maximum of six values, ranked ordinally as the integers 1-6. The direction of "goodness" varies, and most attributes have a range of less than six values.

COCOMO II was adapted by Ray Madachy to estimate software project cost risk, defined as being the probability of a loss multiplied by the cost of the loss [25]. In the domain of software development, cost is associated with effort in person-months. Madachy utilized the COCOMO II cost model along with risk heuristics in a tool[3] he named "Expert COCOMO 2.0 with Risk Heuristics" [25]. These risk heuristics include what Madachy terms joint risk matrices between select pairs composed of the various scale factors and cost drivers (the EM_j of Equation 4.1). For example, if the time schedule cost driver SCED has a value less than the estimated nominal time, and programmer capability PCAP is low, then this indicates a high risk contribution. In addition, if applications experience AEXP of the developers is low, then the relationship of AEXP with SCED also has a high risk contribution. Figure 4.1 gives one such joint risk matrix. The figure gives the

[2]Descriptions of the COCOMO II scale factors and cost drivers can also be found at http://sunset.usc.edu/research/COCOMOII/expert_cocomo/drivers.html

[3]As of the time of writing, Expert COCOMO 2.0 can be found and executed at http://sunset.usc.edu/research/COCOMOII/expert_cocomo/expert_cocomo2000.html

internal weightings of joint risks and illustrates their non-linearity.

	Attribute 1					
Attribute 2	Very Low	Low	Nominal	High	Very High	Extra High
Very Low				1	2	4
Low					1	2
Nominal						1
High						
Very High						

Table 4.1: Example of an Expert COCOMO 2.0 Joint Risk Matrix

Expert COCOMO 2.0 implements a rule base of 94 rules as its risk heuristics, and covers 600 risk conditions [25]. The risks are divided into 6 categories: schedule, product, platform, personnel, process, and reuse. The joint risk matrices are representations of the 94 risk rules which are included within at least one and possibly multiple risk categories.

As input the tool asks for the standard COCOMO 2.0 scale factors and cost drivers, as well as the estimated source lines of code, as shown in Figure 4.1, a screen capture of the input screen of the online tool. Expert COCOMO 2.0 outputs a normalized total risk estimate on a scale of 0-100, with 100 as the highest risk, as well as the individual risk estimates for the 6 categories, and for the 94 individual joint risk pairs from which the total risk was computed. To achieve a risk level of 100 would require each individual cost factor within the given category as being rated at its riskiest level within each joint risk matrix (each joint risk factor would have to be 4).

Madachy categorizes his risk output according to Figure 4.2. These risk categories were used directly to prepare data for C4.5, by considering the upper range, with the exception of 100, as being an exclusive range. Therefore, 5 would be classified as Medium risk, 15 as High risk, and 50

Platform Attributes

Execution Time Constraint		N	H	VH	EH
Main Storage Constraint		N	H	VH	EH
Platform Volatility	L	N	H	VH	

Personnel Attributes

Analyst Capability	VL	L	N	H	VH
Programmer Capability	VL	L	N	H	VH
Personnel Continuity	VL	L	N	H	VH
Applications Experience	VL	L	N	H	VH
Platform Experience	VL	L	N	H	VH
Language and Toolset Experience	VL	L	N	H	VH

Project Attributes

Use of Software Tools	VL	L	N	H	VH	
Multisite Development	VL	L	N	H	VH	XH
Required Development Schedule	VL	L	N	H	VH	

Submit Reset

For more information e-mail: madachy@usc.edu

Figure 4.1: Example Expert COCOMO Input Screen

as Very High risk.

Risk	
Output	Category
0-5	Low
5-15	Medium
15-50	High
50-100	Very High

Table 4.2: Expert COCOMO 2.0 Risk Categories [25]

The output screen resulting from executing the WWW Expert COCOMO 2.0 tool with the inputs shown in Figure 4.1 can be seen in Figure 4.2.

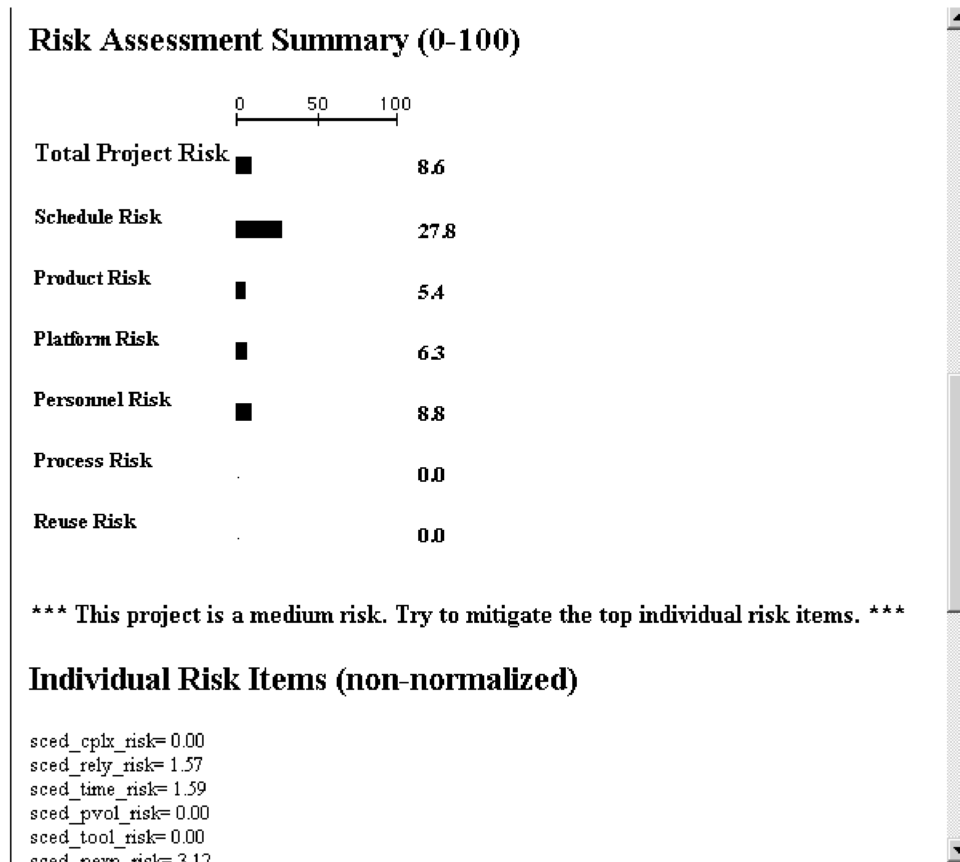

Figure 4.2: Example Expert COCOMO Output Screen

4.3 Experimental Design

Expert COCOMO 2.0 was selected to produce simulated software project data primarily for two reasons. Firstly, it is a mathematically-based deterministic model and as such would allow validation of the experimental results. Secondly, the model is designed to be used by a project manager for prediction of project cost risk and as such is perfectly aligned with the goal of the experimental technique which is to aid in resource allocation. However, the COCOMO family of software project cost models are based on relatively small sets of project data, and any conclusions reached from our analysis relate to the Expert COCOMO 2.0 model used to generate the data sets analyzed.

We received the source code for Expert COCOMO 2.0 from Ray Madachy, in the form of a C-language program. The program was modified to execute a given number of times in a pseudo-random fashion, printing to file the set of inputs followed by the numerical total risk output. Given the existing linguistic classifications in Expert COCOMO 2.0, shown in Figure 4.2, preparing the data file for use by C4.5 required only minor data processing. The model was designed to accept command-line input, allowing either pseudo-random generation of input values, or the explicit input of specific attribute values:

Randomized	cocomo -r <# of cases> <min SLOC> <max SLOC>
Semi-Randomized	cocomo [-attr value]*

The modified Expert COCOMO 2.0, referred to here as COCOMO* first "seeds" the random number generator by using the current system time, and proceeds through # of cases iterations of Expert COCOMO 2.0, supplying it with pseudo-randomly generated input values. Because SLOC is the only continuous Expert COCOMO 2.0 input and has range $\{10000, \infty\}$, it is necessary to specify the range over which to vary it. Because software projects are generally classified into linguistic size categories, such as small, medium, and large, the experimental approach involved using COCOMO* to produce data files with SLOC in four specific SLOC ranges, starting at COCOMO's minimum project size: 10K-50K, 50K-100K, 100K-250K, 250K-500K.

To determine a suitable number of cases, C4.5 was executed with data sets of varying sizes produced using COCOMO*. Figure 4.3 gives the average predicted error rates in the C4.5 trees created from these data sets. A sample size of 10,000 cases was selected as giving acceptable error rates with significantly smaller tree sizes than larger sample sizes.

Samples	Tree Nodes	Predicted Error (%)
5000	1147	5.5
10000	2035	4.2
15000	2877	3.9
25000	4451	3.6
50000	7927	3.1
100000	14339	2.8

Table 4.3: C4.5 Tree Error by Expert COCOMO 2.0 Test Set Size

Next, 10 data files of 10,000 cases each was created for each SLOC range. Since it was possible to stochastically generate as many cases as desired, bagging was not necessary to ensure sufficiently large data sets.

Once 10,000 cases were generated for a particular project size range, the SLOC values were categorized into 5 equally-sized bins. For example, the 10K-50K range contained the following bins: 10K-18K, 18K-26K, 26K-34K, 34K-42K, 42K-50K, where the upper bound is exclusive except for 50K. The rationale for this change is that it is impossible to guide such large software projects so closely as to control their lines of code with the arbitrary specificity that could be achieved in a stochastic simulation. This approach is similar to the ranges of other COCOMO factors, such as Programmer Experience (PEXP), which gives a small number of experience ranges measured in years, and not experience measured in months or weeks. Although C4.5 can process continuous variables, and TREE could easily be modified to accomodate thousands of possible values for an attribute, it would eventually be necessary to present this list of frequency counts of changes in a manner more easily processed by the human mind. In essence a summary of the summary would need to be created. The author chose to institute this change prior to using C4.5 in order to simplify node splits on SLOC, and to simplify the programming in TREE, as well as to reduce the number

of samples required.

Once 10 data sets were created for each SLOC range, C4.5 was used to create a decision tree for each data set. Using an automation script, the following command was issued to create each tree:

```
> c4.5 -f <data set> -c 100 > <data set>.t
```

The `-f <data set>` parameter merely tells C4.5 what file stem to use to look for the necessary input files. The `-c 100` parameter instructs C4.5 to not attempt to prune the decision tree produced unless it has 100% confidence in that pruning. In effect, this parameter restricts C4.5 from increasing the error rate of the decision tree in order to reduce its size. The `> <data set>.t` parameter redirects the ASCII tree normally printed to STDOUT to a text file for later processing by TREE.

The set of 10 trees created for each SLOC range were used as an input ensemble to TREE. An automation script was used to issue the following command for each of the four SLOC ranges:

```
> tree <t1> <t2> <t3> <t4> <t5> <t6> <t7> <t8> <t9> <t10>
```

where `<t`$_n$`>` is the file name (with extension) of each tree in the ensemble. TREE displays summary information for each tree processed, and produces an ASCII text file named TREE.txt which contains the importance table generated from the input ensemble. The automation script renamed this file to correspond to the SLOC range used in each execution.

4.4 Results and Conclusions

A total of four importance tables were produce in our experiment. The raw importance tables for each SLOC range can be found in Appendix A. As stated in Section 1, a requirement of the KDD process as well as the goal of our research is to present interesting knowledge in a representation which is easily understood by a human who can utilize this knowledge to take actions with domain utility. Therefore, two more easily understood representations will be used for our findings: a histogram of relative magnitude and a table of rankings.

Because Expert COCOMO 2.0 is fundamentally based on a regression equation, a linear regression was performed with SPSS[4] on a single 10,000 case data set from each SLOC range to determine the standardized beta coefficient for each attribute. These beta coefficients will be used to evaluate the effectiveness of our approach.

For each SLOC range, the relative importance was found for each attribute value by expressing each as a percentage of the maximum importance in the table. Similarly, each standardized beta coefficient was expressed as a percentage of the absolute magnitude of the maximum coefficient. This transformation preserves relative magnitude while allowing comparison between the two metrics. In addition, the highest magnitude among all values for each attribute in the importance table was ranked, giving a ranking scale from highest to lowest of 1 to 23. The beta coefficient was similarly ranked, allowing a second method of comparison of the relative magnitude of importance found for each attribute by both methods.

[4]SPSS is a trademark of SPSS Inc.

Each histogram shows the relative magnitudes of both the importance of each attribute value, as well as the relative magnitude of the standardized beta coefficient for each attribute. As mentioned previously, not all attributes have the same range of valid values, and so not all attributes have histogram plots for all possible vaues in the range of 1 to 6. With each histogram is a table giving the comparison between the rankings and relative magnitudes found by our method and by standardized beta coefficients from linear regression in SPSS.

As can be seen in Figures 4.3 through 4.6, the ranks of attributes as given by their importance values is consistent across project size. The comparison between ranking methods can be seen most clearly in Tables 4.4 through 4.7. The rankings given by TREE's importance tables correlate closely with the standardized beta values, with correlations between importance rank and beta coefficient rank for all SLOC ranges at 0.96 or greater.

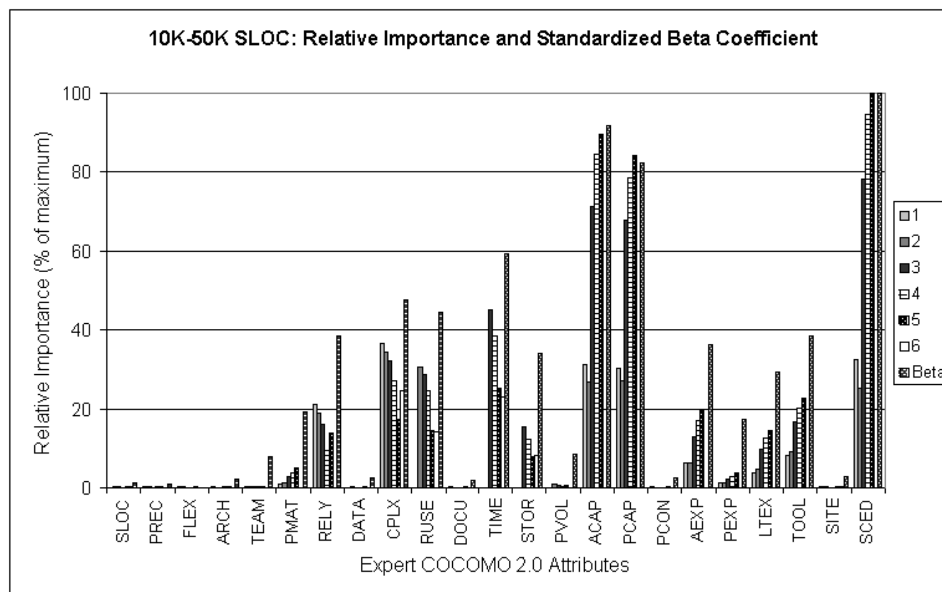

Figure 4.3: Relative Importance and Relative Standardized Beta, 10K-50K SLOC

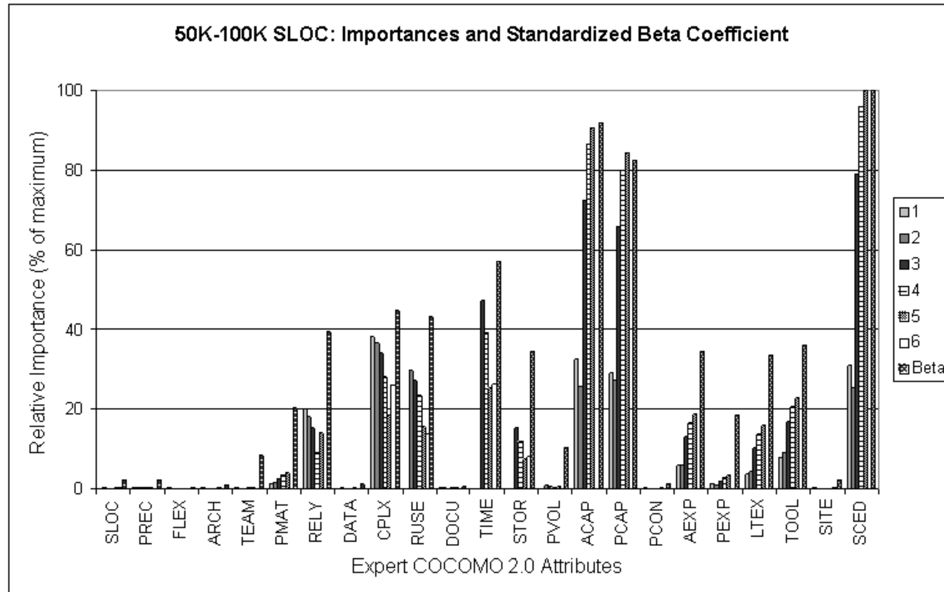

Figure 4.4: Relative Importance and Relative Standardized Beta, 50K-100K SLOC

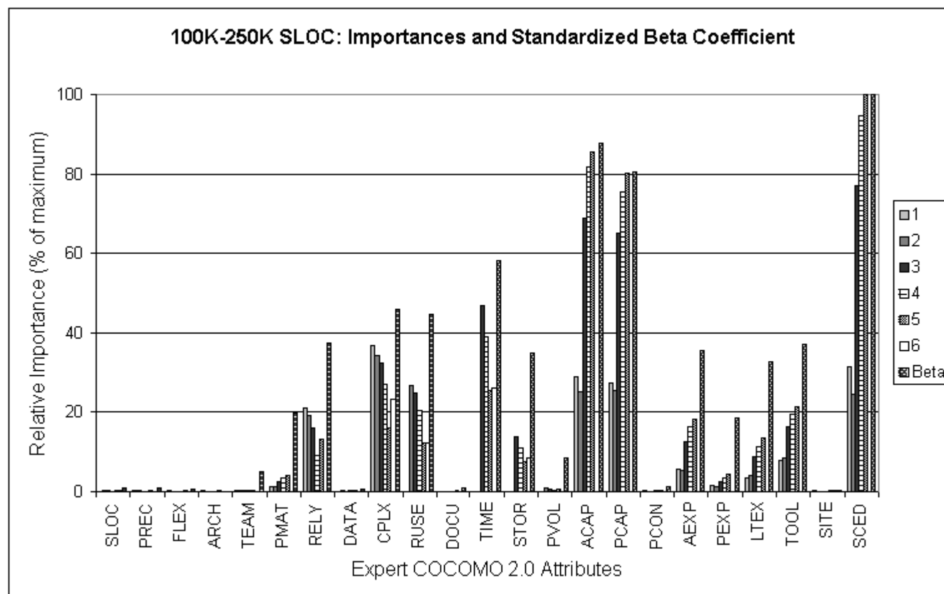

Figure 4.5: Relative Importance and Relative Standardized Beta, 100K-250K SLOC

Both rank and relative magnitude values are useful in interpreting TREE's results. The rank for each attribute gives a clear measure of which attributes are most important, while the relative

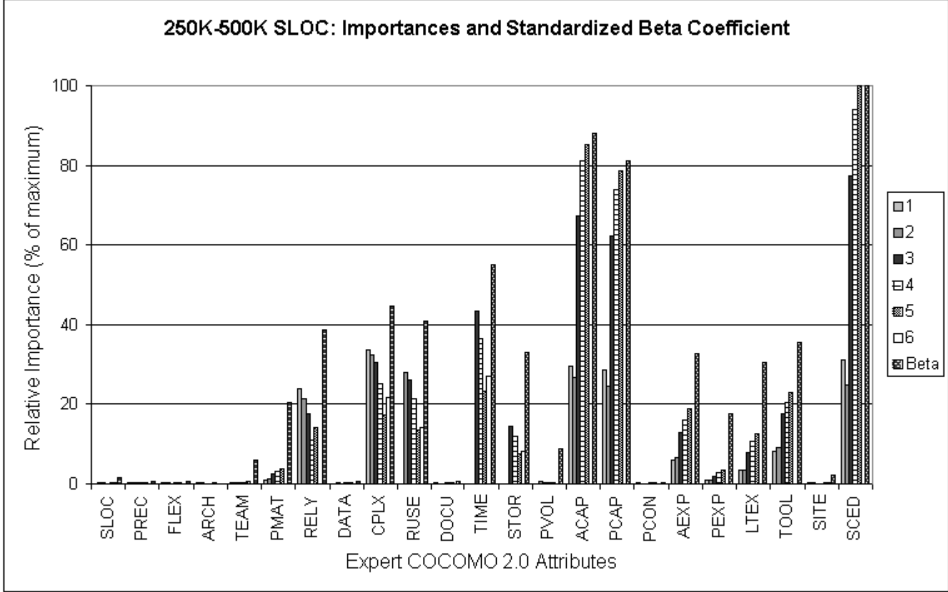

Figure 4.6: Relative Importance and Relative Standardized Beta, 250K-500K SLOC

magnitude value shows how much more or less important some attributes are compared to others. As can be seen in the graphs and tables of TREE's importance findings for Expert COCOMO 2.0, attributes such as documentation (DOCU), development flexibility (FLEX), team cohesion (TEAM), and even the size of the program within the general program size range (SLOC) are ranked in importance much lower than the consistently-highest ranked attributes: development schedule (SCED), analyst capability (ACAP), programmer capability (PCAP), and execution time constraints (TIME). As can be seen by their relative magnitude values, these attributes have little impact on the project risk output. Only these last four attributes are consistently above fifty percent of maximum importance.

	Rank		Relative	
Attribute	Imp	Beta	Imp	Beta
SCED	1	1	100.0	100.0
ACAP	2	2	89.5	91.8
PCAP	3	3	84.2	82.2
TIME	4	4	45.0	59.3
CPLX	5	5	36.4	47.8
RUSE	6	6	30.8	44.5
TOOL	7	7	22.7	38.6
RELY	8	8	21.2	38.4
AEXP	9	9	20.0	36.2
STOR	10	10	15.5	34.0
LTEX	11	11	14.4	29.3
PMAT	12	12	4.9	19.4
PEXP	13	13	3.8	17.2
PVOL	14	14	0.9	8.5
TEAM	15	15	0.4	8.0
SLOC	16	21	0.3	1.2
PREC	17	22	0.3	1.0
DATA	18	18	0.3	2.6
PCON	19	17	0.3	2.6
SITE	20	16	0.3	2.9
DOCU	21	20	0.3	1.9
FLEX	22	23	0.2	0.1
ARCH	23	19	0.2	2.3

Table 4.4: Importance and Beta by Attribute for SLOC range: 10-50K

	Rank		Relative	
Attribute	Imp	Beta	Imp	Beta
SCED	1	1	100.0	100.0
ACAP	2	2	90.7	91.9
PCAP	3	3	84.3	82.5
TIME	4	4	47.0	57.3
CPLX	5	5	38.1	44.8
RUSE	6	6	29.6	43.1
TOOL	7	8	22.7	36.1
RELY	8	7	20.0	39.4
AEXP	9	9	18.7	34.5
LTEX	10	11	15.8	33.4
STOR	11	10	15.4	34.3
PMAT	12	12	4.0	20.5
PEXP	13	13	3.5	18.6
PVOL	14	14	0.8	10.3
TEAM	15	15	0.4	8.4
SLOC	16	16	0.4	2.2
DOCU	17	22	0.3	0.5
DATA	18	19	0.3	1.1
PREC	19	17	0.3	2.1
ARCH	20	21	0.2	1.0
SITE	21	18	0.2	2.1
FLEX	22	23	0.2	0.3
PCON	23	20	0.2	1.1

Table 4.5: Importance and Beta by Attribute for SLOC range: 50-100K

	Rank		Relative	
Attribute	Imp	Beta	Imp	Beta
SCED	1	1	100.0	100.0
ACAP	2	2	85.4	87.6
PCAP	3	3	80.1	80.6
TIME	4	4	46.8	58.3
CPLX	5	5	36.8	46.0
RUSE	6	6	26.6	44.8
TOOL	7	8	21.5	37.2
RELY	8	7	21.1	37.5
AEXP	9	9	18.4	35.5
STOR	10	10	13.8	34.9
LTEX	11	11	13.5	32.8
PEXP	12	13	4.4	18.4
PMAT	13	12	4.2	19.7
PVOL	14	14	0.8	8.6
TEAM	15	15	0.5	5.0
DATA	16	21	0.4	0.6
SLOC	17	17	0.4	1.0
SITE	18	22	0.3	0.2
PCON	19	16	0.3	1.2
FLEX	20	20	0.2	0.6
DOCU	21	18	0.2	1.0
PREC	22	19	0.2	1.0
ARCH	23	23	0.2	0.0

Table 4.6: Importance and Beta by Attribute for SLOC range: 100-250K

	Rank		Relative	
Attribute	Imp	Beta	Imp	Beta
SCED	1	1	100.0	100.0
ACAP	2	2	85.3	88.1
PCAP	3	3	78.5	81.2
TIME	4	4	43.5	54.9
CPLX	5	5	33.6	44.8
RUSE	6	6	27.8	41.0
RELY	7	7	23.8	38.6
TOOL	8	8	22.9	35.7
AEXP	9	10	18.7	32.6
STOR	10	9	14.3	33.0
LTEX	11	11	12.6	30.4
PMAT	12	12	3.9	20.4
PEXP	13	13	3.3	17.6
PVOL	14	14	0.7	8.9
TEAM	15	15	0.5	6.0
SLOC	16	17	0.3	1.5
FLEX	17	18	0.3	0.6
PREC	18	21	0.3	0.5
DATA	19	20	0.3	0.5
ARCH	20	23	0.3	0.0
PCON	21	22	0.3	0.2
SITE	22	16	0.3	2.2
DOCU	23	19	0.2	0.6

Table 4.7: Importance and Beta by Attribute for SLOC range: 250-500K

4.5 Possible Improvements

Due to its overall low-polynomial runtime, the algorithm presented here was not modified in a few identified ways which could have reduced the constant multiple factor of that runtime. In addition, many variations of combination function could have been performed and their results compared. This was not within the scope of the investigation, but the possibility was identified and will be discussed here.

A reduction in Step 2, rule sorting, could be achieved by implementing a more efficient sorting technique. Bubble sort is an easily implemented yet inefficient sorting technique. It's execution time is less than that of the cost of Step 3, and this runtime was deemed acceptable by the author. The reduction of the sorting step runtime to that of a quicksort might be a significant reduction when coupled to a simultaneous reduction in the cost of Step 3, rule comparison.

In Step 3, all rule classifications are compared, and those with what the author designates a significant difference have their rules compared as described in Section 3. This full combinatorial comparison is not necessary if the list of rules is sorted by classification. Again, the use of a sorting technique like quicksort could keep this additional sorting cost below $\bigcirc(n^2)$. Once sorted, $|C|$ pointers can be used to keep track of the first instance of each classification, if any such rules exist. The comparison time is then reduced to a sum of comparisons between groups which have significantly different classifications.

Finally, the importances determined by TREE may be modified by some weighting function. An obvious possibility is a function of the accuracy of the rules being compared. Another obvious pos-

sibility is adding a tree accuracy weighting to the ensemble combination function. Each tree could have its values multiplied by a tree accuracy weighting. In the case where trees vary significantly by error rate, this would place less emphasis on knowledge gleaned from less accurate trees.

Chapter 5

Summary and Conclusions

To be considered useful, that is to be considered to have domain utility, knowledge must be both understandable and actionable. While automated learning techniques may discover interesting features of the data on which they operate, they do not always employ knowledge representations which satisfy these knowledge requirements. Decision trees are one such representation which rely on the extraordinary human ability to comprehend the knowledge contained in graph-based structures. However, the human mind is quickly overcome by the complexity of even moderately large trees, especially when the branches of those trees contain differing sets of attributes used in the node decision tests. This unfortunate constraint on machine learning with decision trees is compounded by considering that typically the tree accuracy improves relative to tree size, as incorrectly classified cases can be correctly classified by increasing tree complexity.

We have described this conundrum with the introduction of two new definitions of the constraints on the use of decision trees (page 5). The *accuracy-complexity constraint*, combined transitively with the *complexity-understandability constraint*, leads to an *accuracy-understandability constraint*.

We have presented a novel ensemble learning technique which provides a knowledge representation which is completely scalable, can summarize the knowledge contained in decision trees as an importance ranking of attributes, and which can be used as a means of guidance for activities like resource allocation and data collection.

Our importance table displays the frequency count of the instances where an attribute value, *ceteris paribus*, would lead to a better outcome classification than the outcome of another rule containing the same attribute. This gives an indication of the relative importance of the attributes in the underlying data set. Although the case study presented involved the use of a data-generating model, this approach would not be intended for situations where a well-defined model already exists. The intended application of the approach would be in a domain where data can be collected and analyzed within the KDD process, yet for which there exists no clearly defined, accurate model, and where descriptive statistics fails in its assumptions. It is our opinion that the technique presented here would in such a case provide beneficial information which can help guide organizational decision makers.

We have presented a case study which utilized simulated data sets created by a popular regression-based model. The output of our approach has been presented and compared with a standardized linear regression beta coefficient. Our approach has yielded a correlation of importance rankings with the standardized beta coefficient rankings of 0.96 or greater for all trials. It is our belief that our approach can be utilized just as effectively in problem domains for which such a statistical analysis would be either less effective, or would require a level of statistical expertise to achieve required accuracy which would violate the given requirements of the KDD process. By contrast,

our approach requires no expert guidance and produces a knowledge representation which is easily understood by human users without requiring specific expert knowledge to interpret. Our technique results in an ordinal ranking and a relative magnitude for each attribute which can be used to guide such efforts as resource allocation, metrics data collection, and model validation.

Bibliography

[1] Agnar Aamodt. Knowledge acquisition and learning by experience - the role of case-specific knowledge. In Gheorghe Tecuci and Yves Kodratoff, editors, *Machine Learning and Knowledge Acquisition*, chapter 8, pages 197–243. Academic Press Ltd, 1995.

[2] Tarek K. Abdel-Hamid. Adapting, correcting, and perfecting software estimates: A maintenance metaphor. *IEEE Computer*, 26(3), March 1993.

[3] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[4] Barry Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[5] Barry Boehm and Philip N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Cost Engineering*, 14(10), October 1988.

[6] Barry W. Boehm, Bradford Clark, Ellis Horowitz, J. Christopher Westland, Raymond J. Madachy, and Richard W. Selby. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, 1:57–94, 1995.

[7] Sunita Chulani, Barry Boehm, and Bert Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transactions on Software Engineering*, 25(4), 1999.

[8] J. R. B. Crockett and J. A. Herrera. Decision tree reduction. *Journal of the Association for Computing Machinery*, 37(4):815–842, 1990.

[9] Thomas G. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18:97–136, 1998.

[10] Thomas G. Dietterich. Ensemble methods in machine learning: First international workshop. In Jan van Leeuwen Gerhard Goos, Juris Hartmanis, editor, *Multiple Classifier Systems*, volume 1857 of *Lecture Notes in Computer Science*, pages 1–15, Berlin, 2000. Springer-Verlag.

[11] Robert P.W. Duin and David M.J. Tax. Experiments with classifier combining rules. In Jan van Leeuwen Gerhard Goos, Juris Hartmanis, editor, *Multiple Classifier Systems*, volume 1857 of *Lecture Notes in Computer Science*, pages 16–29, Berlin, 2000. Springer-Verlag.

[12] Piatetsky-Shapiro Gregory Smyth Padhraic Fayyad, Usama M. From data mining to knowledge discovery: An overview. In Usama M. Fayyad, editor, *Advances in Knowledge Discovery and Data Mining*, chapter 1. American Association for Artificial Intelligence and The MIT Press, Menlo Park, California, 1996.

[13] Usama M. Fayyad. From data mining to knowledge discovery: An overview. In Usama M. Fayyad, editor, *Advances in Knowledge Discovery and Data Mining*, chapter 1, pages 1–31. The AAAI Press and The MIT Press, Menlo Park, California, 1996.

[14] Daniel V. Ferens. The conundrum of software estimation models. In *Proceedings of the 1998 IEEE National Aerospace and Electronics Conference*, pages 320–328. IEEE Press, July 1998.

[15] Stefan Fischer and Horst Bunke. Automatic identification of diatoms using decision forests. In Springer-Verlag, editor, *Machine Learning and Data Mining in Pattern Recognition: Second International Workshop*, volume 2123 of *Lecture Notes in Computer Science*, pages 173–183, Berlin, 2001.

[16] Brian R. Gaines. Transforming rules and trees. In Usama M. Fayyad, editor, *Advances in Knowledge Discovery and Data Mining*, chapter 8, pages 205–226. American Association for Artificial Intelligence and The MIT Press, Menlo Park, California, 1996.

[17] Giorgio Giacinto and Fabio Roli. Dynamic classifier selection. In Jan van Leeuwen Gerhard Goos, Juris Hartmanis, editor, *Multiple Classifier Systems*, volume 1857 of *Lecture Notes in Computer Science*, pages 177–189, Berlin, 2000. Springer-Verlag.

[18] Robert J. Hilderman and Howard J. Hamilton. Evaluation of interestingness measures for ranking discovered knowledge. In Jaime G. Carbonell and Jörg Siekmann, editors, *Advances in Knowledge Discovery and Data Mining: 5th Pacific Asia Conference*, volume 2035 of *Lecture Notes in Computer Science*, pages 247–259, Berlin, 2001. Springer-Verlag.

[19] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, August 1998.

[20] Tin Kam Ho. Complexity of classification problems and comparative advantages of combined classifiers. In Jan van Leeuwen Gerhard Goos, Juris Hartmanis, editor, *Multiple Classifier Systems*, volume 1857 of *Lecture Notes in Computer Science*, pages 97–107, Berlin, 2000. Springer-Verlag.

[21] Mike James. *Classification Algorithms*. William Collins Sons & Co. Ltd, London, 1985.

[22] Barbara Kitchenham and Stephen Linkman. Estimates, uncertainty and risk. *IEEE Software*, 1997.

[23] J. Kittler and F.M. Alkoot. Relationship of sum and vote fusion strategies. In Jan van Leeuwen Gerhard Goos, Juris Hartmanis, editor, *Multiple Classifier Systems*, volume 2096 of *Lecture Notes in Computer Science*, pages 339–347, Berlin, 2001. Springer-Verlag.

[24] Yves Kodratoff. Comparing machine learning and knowledge discovery in databases: An application to knowledge discovery in texts. In Vangelis Karkaletsis Georgios Paliouras and Constantine D. Spyropoulos, editors, *Machine Learning and its Applications: Advanced Lectures*, volume 2049 of *Lecture Notes in Computer Science*, pages 1–21, Berlin, 2001. Springer-Verlag.

[25] Raymond J. Madachy. Heuristic risk assessment using cost factors. *IEEE Software*, 14(3):51–59, May 1997.

[26] Christopher J. Matheus and Gregory Piatetsky-Shapiro. Selecting and reporting what is interesting. In Usama M. Fayyad, editor, *Advances in Knowledge Discovery and Data Mining*, chapter 1, pages 495–516. American Association for Artificial Intelligence and The MIT Press, Menlo Park, California, 1996.

[27] Patrick M. Murphy and Michael J. Pazzani. Exploring the decision forest: An empirical investigation of occam's razor in decision tree induction. *Journal of Artificial Inteligence Research*, 1:257–275, 1994.

[28] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1993.

[29] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[30] Jude W. Shavlik, Raymond J. Mooney, and Geoffrey G. Towell. Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning*, 6(2):111–143, March 1991.

[31] Avi Silberschatz and Alexander Tuzhilin. What makes patterns interesting in knowledge discovery systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):970–974, December 1996.

[32] Evangelos Simoudis, Brian Livezey, and Randy Kerber. Integrating inductive and deductive reasoning for data mining. In Usama M. Fayyad, editor, *Advances in Knowledge Discovery and Data Mining*, number 3, chapter 14, pages 353–373. The AAAI Press and The MIT Press, Menlo Park, California, Fall 1996.

[33] Gheorghe Tecuci and Yves Kodratoff. *Machine Learning and Knowledge Acquisition*. Academic Press, 1995.

[34] Ramasamy Uthurusamy. From data mining to knowledge discovery: Current challenges and future directions. In Usama M. Fayyad, editor, *Advances in Knowledge Discovery and Data Mining*, chapter 23, pages 561–569. The AAAI Press and The MIT Press, Menlo Park, California, 1996.

[35] Wenjia Wang, Phillis Jones, and Derek Partridge. Diversity between neural networks and decision trees for building multiple classifier systems. In Jan van Leeuwen Gerhard Goos, Juris Hartmanis, editor, *Multiple Classifier Systems*, volume 1857 of *Lecture Notes in Computer Science*, pages 240–249, Berlin, 2000. Springer-Verlag.

[36] Geoffrey I. Webb. Further experimental evidence against the utility of occam's razor. *Journal of Artificial Intelligence Research*, 4:397–417, June 1996.

[37] Sholom M. Weiss. *Computer Systems That Learn*. Morgan Kauffman, San Fransisco, CA, 1990.

[38] Nobuhiro Yugami, Yuiko Ohta, and Seishi Okamoto. Fast discovery of interesting rules. In Jaime G. Carbonell and Jörg Siekmann, editors, *Knowledge Discovery and Data Mining: Current Issues and New Applications*, volume 1805 of *Lecture Notes in Computer Science*, pages 17–28, Berlin, 2000. Springer-Verlag.

Appendix A

Case Study 1: Importance Tables

Attribute	1	2	3	4	5	6
SLOC	0	1435	915	573	792	1499
PREC	1394	1010	682	706	1187	0
FLEX	1092	858	593	613	921	0
ARCH	1069	678	618	715	1005	0
TEAM	1092	798	856	1190	1740	0
PMAT	4608	5787	12074	16892	21547	0
RELY	93073	83557	70298	42181	61586	0
DATA	0	1317	641	631	1326	0
CPLX	160255	150959	141705	119696	75673	107800
RUSE	0	135270	126377	108764	64226	62362
DOCU	853	607	412	585	1104	0
TIME	0	0	197708	169151	111323	101896
STOR	0	0	68351	53928	34002	36362
PVOL	0	3867	2383	1433	2220	0
ACAP	137410	117473	313370	371199	393566	0
PCAP	133157	119329	298984	345663	370228	0
PCON	758	534	447	653	1296	0
AEXP	27344	27171	56839	75432	88061	0
PEXP	4891	5589	9976	12944	16827	0
LTEX	15965	20606	42599	55660	63365	0
TOOL	35739	40421	73941	88939	99850	0
SITE	983	753	663	657	811	1277
SCED	142823	111008	343691	415834	439808	0

Table A.1: Importance Table for 10-50K SLOC

Attribute	1	2	3	4	5	6
SLOC	0	1016	623	461	877	1556
PREC	1345	933	694	712	1060	0
FLEX	882	579	400	413	620	0
ARCH	1055	682	469	524	717	0
TEAM	956	612	634	956	1637	0
PMAT	5697	6189	11125	14723	17690	0
RELY	87493	78837	67443	39111	61627	0
DATA	0	853	441	542	1404	0
CPLX	167215	159925	149247	123357	80935	113487
RUSE	0	129790	119538	102493	68088	60603
DOCU	1057	723	565	825	1443	0
TIME	0	0	206208	171634	110599	115021
STOR	0	0	67466	52354	33215	35415
PVOL	0	3674	2467	1643	2722	0
ACAP	142976	112653	318243	379211	397551	0
PCAP	127670	119267	288482	351083	369818	0
PCON	761	358	268	456	784	0
AEXP	25667	26229	58150	72465	81934	0
PEXP	5267	4676	8551	12140	15448	0
LTEX	16899	19775	44904	60351	69437	0
TOOL	33747	39781	73934	89966	99596	0
SITE	759	530	520	572	667	1035
SCED	134986	110937	346470	420930	438438	0

Table A.2: Importance Table for 50-100K SLOC

Attribute	1	2	3	4	5	6
SLOC	0	1212	757	616	1061	1637
PREC	1018	745	618	657	999	0
FLEX	989	638	584	667	1090	0
ARCH	789	528	450	506	811	0
TEAM	967	753	943	1360	2110	0
PMAT	5240	6100	10994	15087	18816	0
RELY	95670	87068	72241	41686	60475	0
DATA	0	1454	672	784	1675	0
CPLX	166663	154721	146329	121776	73145	105729
RUSE	0	120562	112034	92841	54819	56079
DOCU	616	407	335	550	1072	0
TIME	0	0	211958	177066	114670	118034
STOR	0	0	62299	49204	33871	37898
PVOL	0	3633	2441	1430	2224	0
ACAP	130245	113720	310974	369588	386682	0
PCAP	123196	115173	295208	340929	362698	0
PCON	976	556	540	877	1448	0
AEXP	25032	24891	56717	74509	83152	0
PEXP	7588	5460	11455	15447	19924	0
LTEX	15769	17866	39432	51711	61169	0
TOOL	35199	38905	73425	88589	97473	0
SITE	731	476	475	682	994	1525
SCED	141770	110675	348977	428067	452572	0

Table A.3: Importance Table for 100-250K SLOC

Attribute	1	2	3	4	5	6
SLOC	0	1460	1013	630	725	1580
PREC	1437	925	774	826	1290	0
FLEX	1506	1025	705	785	993	0
ARCH	1321	767	652	653	1023	0
TEAM	1255	955	1230	1688	2322	0
PMAT	4902	5624	11454	15123	17953	0
RELY	109588	98646	81020	50400	65178	0
DATA	0	1148	551	737	1412	0
CPLX	154631	148713	140805	116411	79568	99612
RUSE	0	128052	119363	98144	62150	65537
DOCU	955	536	494	730	1141	0
TIME	0	0	199967	167869	107499	124071
STOR	0	0	65918	54999	34699	37837
PVOL	0	3210	2068	1317	2047	0
ACAP	135427	122393	309784	373530	392257	0
PCAP	131194	113357	286029	340160	361155	0
PCON	781	563	488	726	1291	0
AEXP	27829	30248	58775	73767	86188	0
PEXP	4563	4365	8383	12339	15357	0
LTEX	15592	16189	36542	49401	57894	0
TOOL	38314	41237	80947	93465	105248	0
SITE	1015	747	640	563	808	1179
SCED	142595	113599	355355	432774	459913	0

Table A.4: Importance Table for 250-500K SLOC

Appendix B

COCOMO 2.0 Factors

Attribute	Range
SLOC	10,000-inf
PREC	1-5
FLEX	1-5
ARCH	1-5
TEAM	1-5
PMAT	1-5
RELY	1-5
DATA	2-5
CPLX	1-6
RUSE	2-6
DOCU	1-5
TIME	3-6
STOR	3-6
PVOL	2-5
ACAP	1-5
PCAP	1-5
PCON	1-5
AEXP	1-5
PEXP	1-5
LTEX	1-5
TOOL	1-5
SITE	1-6
SCED	1-5

Table B.1: Valid Values for Expert COCOMO 2.0 Attributes

Acronym	Definition	Low-end	Medium	High-end
acap	analyst capability	worst 15%	55%	best 10%
flex	development flexibility	development process rigorously defined	some guidelines, which can be relaxed	only general goals defined
aexp	applications experience	2 months	1 year	6 years
cplx	product complexity	e.g. simple read/write statements	e.g. use of simple interface widgets	e.g. performance-critical embedded systems
data	database size (DB bytes/Program SLOC)	10	100	1000
docu	documentation	many life-cycle phases not documented		extensive reporting for each life-cycle phase
ltex	language and toolset experience	2 months	1 year	6 years
pcap	programmer capability	worst 15%	55%	best 10%
pcon	personnel continuity (% turnover/year)	48%	12%	3%
pexp	platform experience	2 months	1 year	6 years
pmat	process maturity	CMM level 1	CMM level 3	CMM level 5
prec	precedentedness	never built this kind of software before	somewhat new	thoroughly familiar
pvol	platform volatility (freq. of major changes/freq. of minor changes)	12 months / 1 month	6 months / 2 weeks	2 weeks / 2 days
rely	required reliability	errors mean slight inconvenience	errors are easily recoverable	errors can risk human life
resl	architecture/risk resolution	few interfaces defined, few risks eliminated	most interfaces defined and risks eliminated	all interfaces defined and risks eliminated
ruse	required reuse	none	across program	across multiple product lines

sced	dictated development schedule	deadlines moved closer to 75% of original estimate	no change	deadlines moved back to 160% of original estimate
site	multi-site development	some contact: phone, mail	some email	interactive multi-media
stor	main storage constraints (% of available RAM)	N/A	50%	95%
team	team cohesion	very difficult interactions	basically cooperative	seamless interactions
time	execution time constraints	N/A	50%	95%
tool	use of software tools	edit, code, debug		well integrated with lifecycle

Table B.2: COCOMO 2.0 Attribute Descriptions