Which: A Stochastic Best−First Search Learner

Zachery A. Milton

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

Tim Menzies, Ph.D., Chair
Arun Ross, Ph.D.
Katerina Goseva-Popstojanova, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2008

Keywords: Data Mining, Rule Learning, Software Defect Detetection,
Best−First Search, Stochastic, J48, Ripper, Naïve Bayes , TAR3

Copyright © 2008 Zachery A. Milton

Abstract

Rule-based learners search a version space of combinations of features. Starting at "false", features can be added using disjunctions, conjunctions and negations which can combine to some top-most "true" node that covers all possible examples. A complete search of all possible combinations can be too slow, due to the size of the space, or too error prone, due to noise in the space.

When complete search fails, stochastic search may be a useful alternative. For example, the TAR3 "treatment learner" is a stochastic rule learner that leaps around random parts of the version space. TAR favors the construction of tiny rules and, often, the rules generated by TAR are much smaller that the models generated by other methods, e.g. decision trees. Also, while older, and complete, versions of TAR did not scale well, the stochastic search of TAR3 was shown to scale linearly while providing nearly identical rules to TAR2.

The inventors of TAR3 never explored alternate forms of stochastic rule learning. In this thesis we find the TAR3's stochastic search is over−elaborate. Our new algorithm, called Which, is a stochastic best−first search that scales in the same linear manner as TAR3. WHICH returns the same rules as TAR3 but does so after generating 20%, or less, of the rules of TAR3.

Further, WHICH shows that TAR3's stochastic rule generation methods can be significantly generalized. While TAR3 uses a hard−wired rule assessment predicate, Which allows for arbitrary assessment predicates.

Also, we have found a new evaluation criteria that out-performs the current state of the high−water mark in static code defect prediction.

In summary, this research has repeated, simplified, and improved old results on stochastic rule generation. Which is simpler than TAR3, and can be easily customized to produce better learning systems that out−perform certain state−of−the−art results.

Dedication

To My Family
To My Beeb

Acknowledgments

I would like to send out my thanks to Dr. Tim Menzies. Without him this thesis would not be possible. Your dedication to my work and keeping me on track helped me greatly in this process. You have taught me a lot about the world of data mining and problem solving in general that I do not think I could have gotten elseware. It was an honor to be one of your students and one of research assistants.

I would like th thank the Lane Department of Computer Science for providing me with the analytical and programming skills required to approach an undertaking such as this. I would like to also thank the professors I have had over the years here at WVU for providing me with the knowledge I have today. I would especially like to thank Dr. John Atkins for being an amazing teacher as well as a great adviser over the years.

I would like to thank my family. Without you I would not have had the courage and abilities necessary to complete an undertaking such as this.

I would like to thank my girlfriend, Morgan Baxter. Your support and faith in me over the past few years has been unparalleled. I really appreciate your courage and patience with me as I was writing this thesis. I know it was hard not being able to be together as often as we would like, but your strength was what kept me going. Thank you.

I would like to thank my friends, namely Brian Sowers, Ricky Hussmann, Dan Frederick, Anthony Vaccaro, Bryan Szarko, Omid Jalali, and Andrew Matheny for keeping me entertained and laughing throughout this entire process. If it was not for you all, I would be a far less happy man then I am now.

Contents

1 Introduction **1**

1.1 Contribution of This Thesis . . . 3

1.2 Structure of This Document . . . 4

2 Literature Review **6**

2.1 The ARFF Format . . . 7

2.1.1 Header . . . 7

2.1.2 Body . . . 9

2.2 Sampling Policies . . . 9

2.2.1 Discretization . . . 9

2.2.2 Class Distribution Altering Methods . . . 16

2.3 Machine Learners . . . 19

2.3.1 j48 . . . 19

2.3.2 Ripper . . . 23

2.3.3 Naïve Bayes . . . 26

2.3.4 Random Forests . . . 29

2.3.5 TAR3 . . . 30

2.4 Isometric Space Evaluation of Performance Heuristics . . . 33

2.4.1 Probabilities of Detection and False Alarm . . . 33

2.4.2 Receiver Operating Characteristic Curves . . . 35

2.4.3 Isometric Space . . . 36

2.5 Cross Validation . . . 37

2.6 Evaluation Processes . . . 39

2.6.1 Sum of Differences . . . 39

2.6.2 Quartile Charts . . . 40

2.6.3 MannWhitney U−Tests . . . 41

2.7 The Pareto Distribution . . . 43

2.8 Previous Experiments . . . 44

2.8.1 Classic Machine Learners . . . 44

2.8.2 Statistical Models . . . 46

2.9 Summary . . . 47

3 Which **48**
3.1 Idea Behind Which 49
3.1.1 Best−First Search Implementation 49
3.2 Implementation 51
3.2.1 Sorted Linked List 51
3.2.2 Rule Combination 52
3.2.3 Probabilistic Selection 53
3.2.4 Scoring Functionality 55
3.2.5 Stopping Conditions 56
3.3 Finite List 57
3.4 Advantages 57

4 Experiments **59**
4.1 Evaluation Metric: "The Koru Diagram" 60
4.1.1 Special Detectors 62
4.1.2 Area Under the Curve 64
4.2 Experiments with Which's Parameters 64
4.2.1 Changing the Maximium Selection Count 65
4.2.2 Changing the Check Parameters 65
4.2.3 Changing the List Size 66
4.3 Category I: TAR3 72
4.3.1 Data 72
4.3.2 Experiments 73
4.3.3 Results 73
4.4 Category II: Multi Class Classification 76
4.4.1 Which's Heuristic 77
4.4.2 Design of Experiments 78
4.4.3 Results 78
4.5 Defect Detection 80
4.5.1 Data 80
4.5.2 Halstead and McCabe 81
4.5.3 Design of Experiments 82
4.5.4 Which's Heuristic 84
4.5.5 Evaluation Criteria 86
4.5.6 Probability of Detection vs. %Lines of Code 86
4.6 Category III: MDP Data 87
4.6.1 Experiment 87
4.6.2 Results 88
4.7 Category III: Turkey Data 96
4.7.1 Experiment 97
4.7.2 Results 97
4.8 Category III: AT&T Data 103
4.8.1 Data 103

4.8.2 Experiment 104
4.8.3 Results 106
4.9 Category IV: Micro−Sampling 111
4.9.1 Experiment 111
4.9.2 Results 112

5 Conclusion **117**
5.1 Overview 117
5.2 Findings 118
5.3 Future Work 119

A Using Which **120**

List of Figures

2.1 Example ARFF File, Data Taken from UCI Repository [3]. 7
2.2 A Histogram of an Example Bin Layout for Equal Interval Discretization. 12
2.3 Example of Equal Interval Discretization. 12
2.4 A Histogram Example Bin Layout for Equal Frequency Discretization. 14
2.5 Example of Equal Frequency Discretization. 14
2.6 Example J48 Decision Tree Using Data from Figure 2.1. 20
2.7 Example Confusion Matrix 34
2.8 Confusion Matrix for A from Confusion Matrix in Figure 2.7. 34
2.9 Illustration of Different Areas of the ROC Curve. 35
2.10 Isometrics of Precision, J48, and Balance. 37
2.11 Example Row in a Sum of Differences Chart. 39
2.12 Example Quartile Chart. 40
2.13 Example of Using MannWhitney U−Tests on Two Distributions. 42
2.14 Example Pareto Distribution. 44

3.1 Example of the Modified Best−First Search. 51
3.2 Probabilistic Selection Pseudo Code 54
3.3 Illustration of the Pseudo-Code in Figure 3.2 55

4.1 Illustration of Main Components of Koru Diagram. 60
4.2 Oracle Detection Rule 62
4.3 Graphs Displaying the Early Maxima Phenomenon 67
4.4 Graphs Displaying the Effects of Changing the Sorted List Maximum Size. Which2 is always under Which1000 in these graphs. 70
4.5 Sum of Differences Results for 2bins, 4bins, 8bins, 16bins, and 32bins comparison of Which and TAR3. 74
4.6 Overall Sum of Differences for Which vs TAR3. 75
4.7 Oracle Detection Rule 76
4.8 Quartile Chart Illustrating the Results of Table 4.4. 79
4.9 Some Example Attribute Distributions for kc3mod, mw1mod, and mc2mod. Here, we sorted the values of the attributes in each data set in ascending order. What we are measuring is the distribution of the values over the number of instances in the set. 83
4.10 MDP Data Files where Which2 > manualUp > rest. 90
4.11 Quartile Charts Repesenting Graphs in Figure 4.10 91

4.12 MannWhitney U−Tests Representing Graphs in Figure 4.10 92
4.13 MDP Data Files where Which2 > rest. This is for kc3 mod. 93
4.14 Quartile Charts Representing Graphs in Figure 4.13. 93
4.15 MannWhitney U−Tests Representing Graphs in Figure 4.13. 93
4.16 MDP Data Files where all > Which. This is the data file, mc2_mod. 94
4.17 Quartile Chart Representing Graphs in Figure 4.16. 94
4.18 MannWhitney U−Tests Representing Graphs in Figure 4.16. 94
4.19 Quartile Charts Representing the Overall Performance of the Detectors on the MDP Experiment. 95
4.20 MannWhitney U−Tests Representing the Overall Performance of the Detectors on the MDP Experiment. 95
4.21 Graphs representing the results for the Turkey experiment. 99
4.22 Quartile Charts Correlating to Figure 4.21. 100
4.23 MannWhitney U−Tests Correlating to Figure 4.21. 101
4.24 Quartile Charts Representing the Overall Performance of the Detectors on the Turkey Experiment. 101
4.25 MannWhitney U−Tests Representing the Overall Performance of the Detectors on the Turkey Experiment. 102
4.26 Plots of Selected Results from AT&T Experiment. 107
4.27 4 Result Quartile Charts Taken from AT&T Experiment. 108
4.28 4 Result MannWhitney U−Tests Taken from AT&T Experiment. 108
4.29 Overall Quartile Chart for the AT&T Experiment. 108
4.30 Overall MannWhitney U−Tests Taken for the AT&T Experiment. 109
4.31 Quartile Chart Results for Micro−sampling. 113
4.32 MannWhitney U−Tests for Micro−sampling. 114
4.33 Overall Quartile Chart for Micro−Sampling. 115
4.34 Overall MannWhitney U−Tests for Micro−Sampling. 115

A.1 Example Got Want Table. 121

List of Tables

2.1 Description of Possible Data Types for Attributes in ARFF. 8
2.2 Example Size Increases Due to Over−Sampling with Distribution Set to 50%. . . . 17
2.3 Look−up Table Generated by Naïve Bayes using the data given from Figure 2.1. . 26
2.4 Configurable Parameters for TAR3. 32

4.1 Description of Different Learners Used in Experiment of Section 4.2.3. 69
4.2 Comparison of Run−Times for Different Stack Sizes. 71
4.3 Description of the UCI [3] data sets used in the experiment under Section 4.3. . . . 72
4.4 Overal MannWhitney U−Test on Selected UCI Data Sets. 78
4.5 Statistics Taken About LOC From a Few of the Data Sets Used. 80
4.6 Halstead and McCabe Descriptions of MDP Data Sets. 81
4.7 Information About the NASA data. 87
4.8 Descriptions of Which Variants used in Section 4.6.1. 88
4.9 Information About the Turkey data. 96
4.10 Information About the Attributes Used in the AT&T Data Sets. 104
4.11 Description of Different Micro−Sampling Values used in Section 4.9.1. 112

A.1 Breif Description of the Which command−interface. 122

Chapter 1

Introduction

It can be said that most data contains two things, *clumps* [11] and *collars* [7]. Clumps are a property of data that discusses the phases, or types of data. Essentially, a clump is a class of data in the data set that all share similar classification properties. A collar, or collar variable, is a specific descriptor of the data in the set that is very higly correlated to the clumps. That is, a noticable change in the collar variable will generally mean the change of one of these clumps. It is our goal to develop a machine learner that is highly tuned to exploiting these clumps and collars in the data. It is also our goal to allow this new machine learner to have different methods of finding these clumps and collars by allowing application−specific goals to be used when searching through this data.

Also, as a result of our exploitation of specific collar variables in the data, our machine learner produces rules, or conjunctions of conditions, that are very small and easy to understand. This leads to an easy and brief explanation time on what exactly these rules are trying to tell about the data set. This reduces the time necessary to discuss the tool being used and allows for more time to be allotted for the actual task.

The TAR3 [21] machine learner is one implementation of such a tool that attempts to isolate a specific clump and exploit the collars in the data to create easy to understand, simple rules. TAR3 behaved well in that it produced these small rules and produced these rules with a linear time

scaling to the data. However, TAR3 implemented a hard−wired rule learning heuristic. In all of the experiments with the TAR3 learner, no other types of heuristics were used to create these rules. Also, the effects of such collar−variable exploitation where never fully explored in terms of learning on one set and testing on the other. While this machine learner shows a lot of promise in this type of field, we will show with our new machine learner, Which, how other learning heuristics can also perform well by exploiting these collar variables, and that this method of learning works well when training with one set and testing on another.

One specific instance of an application is software defect detection. The study of detecting defective modules in releases of software is an important area that many researchers have experimented in. This realm is important because reducing the cost and time a company needs to test software would greatly reduce the overall cost and time a project takes to complete. Detecting fault functions, files, or classes automatically could greatly reduce the amount of work software testers would need to do as well as allow them to focus on specific areas of the software instead of the entire package to test for defects.

The current machine learners used in the realm of software defect detection today are classification learners [18, 30, 32, 33]. These learners have hard coded evaluation heuristics and as a result cannot be finely tuned to the business applications they are being used with. There is a need for for a more customizable machine learner that can better adapt to the different business situations it can be used in. Classic machine learners create their classification models based on some internal heuristic and are then evaluated in the business application with an entirely different method. It is obvious that having a machine learner that can change its internal evaluation heursitic when a business application calls for will have an advantage and be able to perform better.

There is a need in software defect detection to remove the use of these classification machine learners and move towards a different paradigm, one that is created specifically for the purpose of detecting defective modules. TAR3 was created as a method of selecting specific classes and generating rules to predict only that class. It does this in a rather verbose method and our learner,

Which, was created to take its place. Which, like, TAR3 is a machine learner that attempts to create a rule that best describes the attribute combonations that predict a single class. Which is different from TAR3 in that its growing heuristic is no hard coded and is easily modifiable. Machine learners like TAR3 and Which are perfect for the area of software defect detection because they make no sacrifices in order to maximize the classification of both defective and non−defective source code as the classification learners do. Instead, they are focused on the goal at hand, to maximize the classification of just the defective modules.

There is a common notion in the world of software defect detection and that is that many development companies are just beginning to understand the power of automated defect detection via machine learners. As a result, the data on previous projects is very limited. We introduce a sampling policy called micro−sampling [34] into our experiments that in our experiments has performed just as well as the machine learner without micro−sampling. This study gives evidence that as lone as the distributions of defective and non−defective modules is kept relative close, machine learners do not require a lot of previous data to develop detectors.

1.1 Contribution of This Thesis

The contributions of this thesis are:

- The Which algorithm;
- Configurable scoring heuristics;
- Validity of lift learning in a cross−validation testing environment;
- Further evidence of a common distribution of faults in software;
- A novel evaluation method for evaluating learners in software defect detection, The Koru Diagram;

- Extensive comparisons of Which to class machine learners in the are of software defect detection; and,
- Testing of a new sampling policy called micro−sampling.

The results of the software defect detection give us results that this type of specialized machine learner is the state−of−the−art for using maching learners to detect defects. It gives useful insight onto the uses of the classic learners and their limitations in this field.

1.2 Structure of This Document

This thesis consists of five different chapters that discuss the different aspects of the research that went into this thesis.

Chapter 2 discusses the background information required to understand the experiment and theoretical work that went into this thesis. It gives the explainations of the different classic machine learners that we used in our experiments. It also talks about the different types of comparison metrics we used to evaluate superiority and inferiority of learning methods over all of the experiments we conducted. Chapter 2 discusses some of the universal concepts our different experiments all use as well as some of the discovered properties of the data we used. This chapter also discusses some of the previous types of experiments that have been done in the field of software defect detection.

Chapter 3 discusses the theory and implementation of our new machine learner, Which. It discusses in detail how Which compares and contrasts with TAR3. Chapter 3 also discusses the different implementation decisions that were made and justifies these decisions.

Chapter 4 discusses the six different overall experiments that were done with the Which machine learner. It goes into detail about the types of data that were chosen and the different evaluation methods that were chosen. Extensive results are given for each of the experiments and discussions of these results can be found in this chapter as well. In cases were results are contradictory to each other, various explanations can be found wheere these contradictions occur.

Chapter 5 consists of a summary of the work that this thesis contains. It also provides information about potential future work with the Which learner.

Chapter 2

Literature Review

This section is a collection of different ideas and experiments in the realm of machine learning. It focuses mainly on the different types of properties that software defect detection experiments and data tend to have. Section 2.1 discusses the properties of a special data format that is used throughout this thesis. Section 2.2 discusses the different sampling policies that have been used in the past and created for this work that this thesis uses in its varying experiments. Section 2.3 discusses details about the different machine learners that either this thesis has used in experiments or other experiments have used. Section 2.4 covers various comparison methods for the different heuristics of learners as well as defines a few core concepts in the performance evaluation of machine learners. Section 2.5 discusses the main method of dividing up the data sets into a train and a test set that nearly all of the experiments do. Section 2.6 discusses the different types of comparison methods that are used in the various experiments throughout this thesis. Section 2.7 explains a common phenomenon in the distribution of faults that has been discovered in several different data sets. Section 2.8 covers some of the previous work that has been done in the field of automatic defect detection.

2.1 The ARFF Format

```
@relation weather.nominal

@attribute outlook {sunny, overcast, rainy}
@attribute temperature {hot, mild, cool}
@attribute humidity {high, normal}
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,hot,high,FALSE,no
sunny,hot,high,TRUE,no
overcast,hot,high,FALSE,yes
rainy,mild,high,FALSE,yes
rainy,cool,normal,FALSE,yes
rainy,cool,normal,TRUE,no
overcast,cool,normal,TRUE,yes
sunny,mild,high,FALSE,no
sunny,cool,normal,FALSE,yes
rainy,mild,normal,FALSE,yes
sunny,mild,normal,TRUE,yes
overcast,mild,high,TRUE,yes
overcast,hot,normal,FALSE,yes
rainy,mild,high,TRUE,no
```

Figure 2.1: Example ARFF File, Data Taken from UCI Repository [3].

The Attribute−Relation File Format, or ARFF, was developed for use with the wEka [17]. This format is used by all of the machine learners that the wEka implements, as well as the Which learner described in Chapter 3. ARFF can be broken up into two specific regions, the header and the body. The "%" symbol is used as a comment in an ARFF file and can occur anywhere on a line and in both sections. ARFF files are case−insensitive. An example ARFF file can be seen in Figure 2.1.

2.1.1 Header

The header contains the meta−data that the machine learner reading the file can use to set up the required structures for reading in and understanding the data. This meta−data can also be used to check the data to make sure it is correct. The first part of the header file is a line that starts with "@releation $< description >$". This keyword is meant to describe the type of data this file

represents. The $< description >$ portion can contain spaces, but must be surrounded by quotes if it does. Following the @relation keyword, a series of "@attribute $< name > < type >$" lines appear, as can be shown in Figure 2.1. There are a few types of attributes that can be used and they are described in Table 2.1.

Numeric	Used to define values that fall into the real numbers.
String	Used to define values that are strings.
Date	Used to define dates. A date format is also required
Nominal	Used to define discrete $-$valued attributes.

Table 2.1: Description of Possible Data Types for Attributes in ARFF.

As is mentioned in Table 2.1, the date attribute type needs a date format as well. This format is by default in the ISO-8601 format of "$yyy - mm - dd'T'HH : mm : ss$." This can be altered to allow for different representations. The format described in the declaration of the date attribute must be adhered to in the body section of the ARFF file.

The nominal type, as described in Table 2.1, is used to define discrete$-$valued attributes. The word *nominal* does not actually, appear in the ARFF file, however. Instead the possible values of this attribute are declared after the attribute name and surrounded by "$\{\}$". An example of this is the variable *outlook* in Figure 2.1. Outlook can have three possible values, sunny, overcast and rainy. All discrete$-$valued attributes must be defined in this way and all possible values must be defined.

The final attribute in the ARFF file has special meaning. This attribute is considered the *class* of the data file and is what is used in the wEka as the target for prediction. The class attribute can be any of the types listed in Table 2.1. The class need not be called class, and in the example in Figure 2.1 it is called *play*.

2.1.2 Body

The body of the ARFF file is designated by a line with "@data" written on it. After this point no new meta−data can be declared. The body is just a series of comma seperated values that represent the attributes in order. Each row in the body portion of the ARFF file is called an *instance*. An instance is considered one specific case of data that was recorded. Each instance must contain one value for each of the attributes listed in the header portion and these attribute−values must be in the same order the attributes were declared in, thus ending with the class always. It is possible for "?" to appear in the body as values without being specifically declared as a possible value for an attribute in the header section. The "?" stands for an unknown. Different learners handle unknowns in different ways.

2.2 Sampling Policies

A sampling policy in terms of data mining is a method of altering the data in order to allow machine learners to perform better on a given data set. There are several different type of sampling policies. Below is an explaination of the different kinds of sampling policies that are used in the experiments of this thesis, found in Chapter 4.

2.2.1 Discretization

Discretization is the process of converting a real−valued attribute into a discrete−valued one. There are many ways to achieve this effect and [22] did several experiments to illustrate the effectiveness of the different policies. Three of the the most common methods used are equal interval, equal frequency, and FayyadIrani [12]. A comparison of these discretization methods, with others can be found in [9]. This comparison is done by using the discretization methods as a preprocessor and then running the Naïve Bayes machine learner on the data and comparing the results. More

information on the Naïve Bayes learner can be found in Section 2.3.3. There are at least two reasons for discretization. The first reason is that several machine learners only work on discretized data, or that certain types of machine learners can perform better on discretized data [25]. A second reason is that discretizing the data can simplify the search process by limiting the range of the attributes to smaller range. This second reason can not only allow the machine learner to perform better by having less space to search, but can also decrease the run time of the application.

As an example of the first reason for converting continuous attributes into discrete ones could be an example of Naïve Bayes . As is discussed in Section 2.3.3, Naïve Bayes uses a table to store the frequencies of each attribute value that appears in the data set [8]. Using continuous values could result in a very sparse look−up table, as many of the values could only appear once in the entirety of the data set. Also, using this continously ranged look−up table would result in values that have not been seen in the train set not having a frequency in the test set. As will be explained in the section on Naïve Bayes , this would result in the instance containing the new value not being classified.

Another example of the first reason for converting continuous attributes into discrete ones is an example of a decision tree machine learner, like j48 discussed in Section 2.3.1. As a brief introduction, a decision tree learner makes its classification criteria by choosing an attribute to split into partitions based on some splitting criteria. If attributes were continuous, splitting on an attribute that is continuous could lead to many of the branches to be pure in that there is only one instance of data that contains that continuous value [25]. In contrast, using a discretization method to either lump together continuous ranges as a single value for the attribute can prevent this problem from happening.

The current idea behind discretization is that the values in each bin of the discretization are continuous values. That is there will never be a bin that contains two disjoint ranges of numbers. A potential bin that is generated from a discretization policy could be $[0,10)$, whereas a bin such as $[0,5) \wedge [8,10)$ is not traditionally created. In Chapter 3 this idea of disjoint combinations of

attributes is discussed and an idea of using Which to create a discretizer is further elaborated on in Chapter 5. The three discretization policies discussed here use the idea of discrete bins of continuous attributes always containing continuous ranges.

Equal Interval

Equal interval discretization is an *unsupervised* discretization policy. This means that each attribute is discretized in isolation, or that the process of discretization is done independantly of the class or any other attributes in the data set. This type of discretization is not done in any of our experiments, but it is a part of the Which source code and an option. For further information on the use of the Which program, please see Appendix A. Equal interval discretization is considered to be the simplest discretization method [41], as it takes nothing about the data into account and the bins have a variable number of items in them [41].

Equal interval discretization is a two phase process. The first phase is to go over the entire range of the data set and find the minimum and maximum values the attribute can have. The second phase involves using Equation 2.1 with val being the value of the attribute for each instance of data to calculate which bin this value should be in. This method of discretization creates n bins each containing the same interval length. However, each bin does not necessarily have to have the same number of numbers in it. In fact, it is entirely possible for some of the bins to be completely empty [41]. A histogram of a possible configuration of bins can be seen in Figure 2.2 [41].

$$bin = \lfloor \frac{val - min}{max - min} \rfloor * n \tag{2.1}$$

Figure 2.3 provides an example of how equal interval discretization would create the bins for a simple attribute range that contains 20 numbers. In this example,the minimum value is 0 and the maximum value is 200 and $n = 10$.

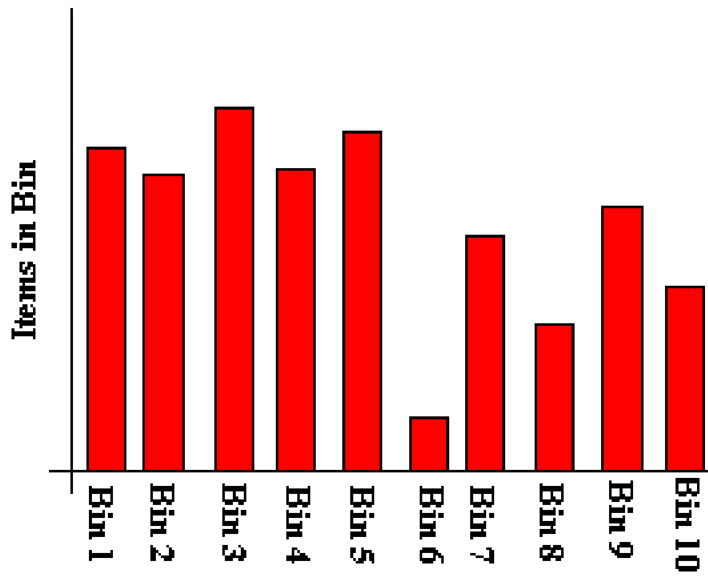

Figure 2.2: A Histogram of an Example Bin Layout for Equal Interval Discretization.

Bin Number	**Range**	**Number of Items in Bin**
1	$[0,20)$	2
2	$[20,40)$	5
3	$[40,60)$	1
4	$[60,80)$	1
5	$[80,100)$	0
6	$[100,120)$	2
7	$[120,140)$	1
8	$[140,160)$	5
9	$[160,180)$	1
10	$[180,200]$	2

$Values = 0\ 19\ 24\ 27\ 27\ 30\ 32\ 52\ 65\ 100\ 110\ 123\ 145\ 146\ 149\ 150\ 155\ 170\ 180\ 200$

Figure 2.3: Example of Equal Interval Discretization.

Equal Frequency

Equal frequency discretization is similiar in some ways to equal interval discretization. First of all equal frequency discretization is an *unsupervised* process. Also, at least in the implementations

used in this thesis, equal frequency discretization is done as a preprocessor to the actual learning, and each of the continuous attributes are discretized into the same amount of bins. Equal Frequency Discretization is the natural type of internal discretization for the Which machine learner discussed in Chapter 3 and is used in experiments in Section 4.6.1, Section 4.7, Section 4.8, and Section 4.9. Unlike equal interval discretization, equal frequency discretization creates a histogram, seen in Figure 2.4, that is completely flat. This is because all of the bins contain the exact same number of items in them.

The process of equal frequency discretization is a done in two phases. The first phase requires sorting the values of the attribues in ascending order. The second phase is to divide this sorted list into n bins with each bin containing the same *amount* of values. What this does, unlike equal interval discretization, is create a flat histogram [41]. Each of the n bins have the same amount of items in them. In the case where the number if instances is not divisible by the number of bins, one of the bins, in our implementation this is the last bin, will have the least number of items in it. Figure 2.5 is an example of the result of equal frequency discretization on the same distribution of 20 numbers that were used in Figure 2.3 on equal interval discretization.

In the example in Figure 2.5, the second and third bins both contain the value of 27. This is due to the nature of the equal frequency discretization. In the example, each bin contains exactly 2 items, regardless of their value. The two values of 27 happen to lie on a cutoff point in the binning algorithm, so they fall into different bins. In contrast to the result of the number of items in each bin varying but the range of each bin being constant in equal interval discretization, the number of items in each bin are constant but the range of each bin is varying in equal interval discretization. As will be discussed briefly in Section 4.5.3, it is entirely possible for an attribute to have an exponential distribution. Using equal frequency discretization on this type of distribtion will result in lower numbered bins having the same range, or in fact that same values with no range, and the final few bins having a very large range.

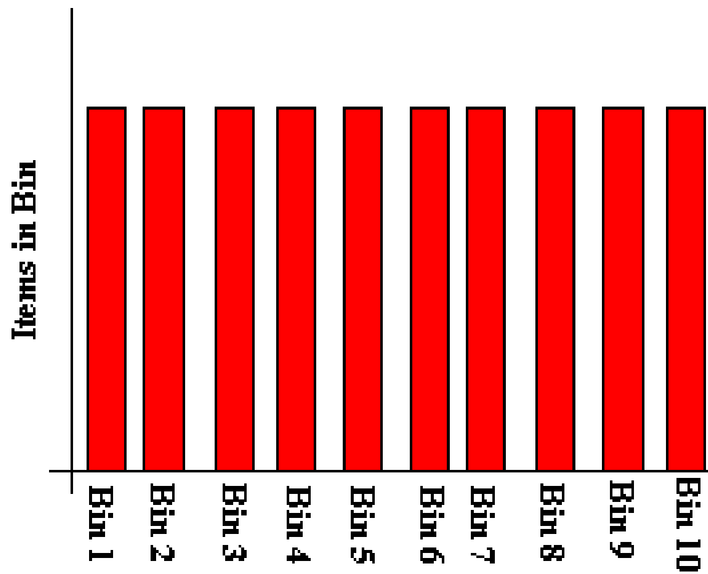

Figure 2.4: A Histogram Example Bin Layout for Equal Frequency Discretization.

Bin Number	**Range**	**Number of Items in Bin**
1	$[0, 19]$	2
2	$[24, 27]$	2
3	$[27, 30]$	2
4	$[32, 52]$	2
5	$[65, 100]$	2
6	$[110, 123]$	2
7	$[145, 146]$	2
8	$[149, 150]$	2
9	$[155, 170]$	2
10	$[180, 200]$	2

$Values =$ 0 19 24 27 27 30 32 52 65 100 110 123 145 146 149 150 155 170 180 200

Figure 2.5: Example of Equal Frequency Discretization.

FayyadIrani

Unlike the previous two discretization methods, FayyadIrani [12] is a *supervised* process. This means that the method of deciding both the number of bins and the values that go into the bins

is determined by some criteria involving the class makeup. This process is considerably more complex in terms of how it decides the bins, but is not overly complex in that it drastically alters the time required to discretize, at least in our experiments in Section 4.4. The FayyadIrani method of discretization was used in the experiments of Section 4.3 and Section 4.4. In these experiments, the implemention of FayyadIrani that was used is part of the wEka [17] distribution of data mining tools. It is under the listing of *weka.filters.supervised.attribute.Discretize*.

FayyadIrani first will sort the data in ascending order based on the continuous attribute to be discretized. It will then, using an evaluation method called information entropy, determine the best possible split to create two new bins of the data. Everything greater than the split becomes the second bin and everything below the split becomes the first bin. After doing this, the process will then take only the instances of data contained in bin one and repeat this information entropy assessment to determine the next reasonable splitting point. The same process is done on the upper bin as well. This process of splitting the bins into smaller bins is continued until one of two critera are met. The first criteria is that the bin is *pure*, or that only one value of the class is associated with all instances in the bin. The second criteria is called the *minimum description length principle* [9,25,41] and can be seen in Equation 2.2. This equation illustrates the if the gain from a split is not greater than this number, no new splits are considered. In Equation 2.2, N is the number of instances, c is the number of classes in the current bin, E is entropy of the current bin, c_1 and c_2 are the number of classes in the potential resulting bins, and E_1 and E_2 are the entropies of the potential resulting bins. The variable called *gain* in this equation is elaborated further in Section 2.3.1 with Equation 2.5.

$$gain < \frac{log(N-1)}{N} + \frac{log(3^c - 2) - cE + c_1E_1 + c_2E_2}{N} \tag{2.2}$$

Since this discretization process is done per attribute and based on the distribution of the

$(range, class)$ tuples, it can be easily seen that the number of bins for each attribute do not necessarily have to be a constant over all of them. Another interesting idea of this entropy−based discretization is that it is entirely possible for an attribute to only have one bin [22]. Having an attribute in a data file that contains only one value is completely useless as that attribute tells the learner nothing about the underlying class distribution. In fact, the wEka maching learning package [17], will not allow a data set to contain attributes to have only one value. This behavior of FayyadIrani is not to be considered a con against the discretization policy. The fact that an attribute was only allowed to have one bin most likely means that the attribute is too noisy and should not be considered in the overall data set at all.

2.2.2 Class Distribution Altering Methods

Class distribution altering methods involve the processes of removing instances from or inserting copied instances into the data set. Like discretization methods having two classes of these methods, *unsupervised* and *supervised*, these distribution altering methods also have two classes. They are *unstructured* and *structured*. The types of distribution altering methods that are used in the experiments of Chapter 4 are structured methods of class distribution altering. Unstructured methods include randomly copying or removing instances from the data set with no concept of which instances are actually being copied and reduced. These methods were not used in this thesis and further discussion of them is not in the scope of this thesis. The following consists of three sections that discuss the different types of structured distribution altering methods. Only one of these methods was used in our experiments, but we feel that over−sampling and under−sampling explainations lead to better understanding of how micro−sampling works.

Over−Sampling

Over−sampling is the process of selecting a class from a data set and increasing its distribution in that set up to a desired percentage. This is done by randomly copying instances of the class in

the data set multiple times until the desired distribution is met. In essence, this process is meant to make a rare class more common by artificially adding copies of instances of that class into the data set. This results in larger data sets and in the case where the desired class to be over sampled is very rare, these data sets can become very large and unmanageable. Table 2.2 is an example of the size increase of the data file being dependant on the original distribution. In the first example, weather, only 4 instances needed to be added to aquire the desired final distribtion of 50%. However, in the second example, pc1, 769 examples needed to be added to aquire the same desired final distribution. It can easily be inferred that if the distribution is even smaller and the original instance count is high, an even larger explosion in the size of the data set will occur. These two examples were taken from data files containing two classes for ease of computation. This explosion is even greater if there are more than two classes in the data file.

File Name	Class	Original Distribution	Original Instances	Final Instances
Weather	no	35.7%	14	18
pc1	true	6.94%	1109	1878

Table 2.2: Example Size Increases Due to Over−Sampling with Distribution Set to 50%.

[10] performs several tests using C4.5 [39], which is the C implementation for which j48, a Java implementation, is based on, with over and under−sampling. These tests show that over−sampling can actually hurt the performance of a machine learner. This is most likely caused by the fact that over−sampling creates exact copies of instances already in the data set and most machine learners use some measure of support, or how many instances are covered by this decision, as an evaluation criteria.

Under−Sampling

Under−sampling is the opposite process of over−sampling. That is instead of copying instances of the desired class to better enhance its distribution in the data set, under−sampling removes instances of the other classes at random to acheive the same effect. As stated, this has the same

desired effect as over−sampling, but achieves it by actually removing available information that the learner could have used. As a result of this removal process, the data files are smaller.

In the same paper by Chris Drummond and Robert Holte [10], their results conclude that under−sampling does not significantly lower or raise the performance of C4.5. Initially, this result may seem to mean that under−sampling has no use in the field, however, considering that under−sampling actually makes the data file to be used smaller, this process can have uses in areas where the learning application's running time is dependant on the size of the data set in some significant way.

Micro−Sampling

Micro−sampling [34] is a type of data sampling policy that is related to the sampling policy of under−sampling [10]. There is one key difference in micro−sampling, and that is the removal of the desired class as well. The idea behind micro−sampling is to always create an even $50-50$ spread amongst the classes. As a note, the micro−sampling process is only meant to be used on two classes data sets. This is because micro−sampling was meant to be a method of boosting performance of the classic machine learners in the field of software defect detection. Software defect detection is discussed heavily in Section 2.8.

Micro−sampling's approach to altering the class distribution is to make the data set a fixed size, as well as a fixed distribution. For instance, in the weather data example given in Figure 2.1, a micro−sampling of 2 would end with a data file that contained only 4 instances. In [34], they used micro−sampling sizes of $size >= 25$ for their experiments and concluded that the micro−sampled data sets performed just as well as the data sets that were left alone. The method that micro−sampling uses is to simply take n instances of the desired class as well as n instances of the other class, and create a new data set that only contains $2n$ instances. Experiments with micro−sampling can be seen Chapter 4.

2.3 Machine Learners

2.3.1 j48

J48 is a decision tree learner that is a Java implementation of Quinlan's C4.5 [39]. Decision tree learners work by the divide−and−conquer method. That is, they attempt to split up the data into smaller, more managable parts. Decision trees do this by singling out candidate attributes to split on. In other words, a decision tree learner starts out by determining which attribute is best suited to work as the root of the tree. Once this is done, the data is divided into n portions where n is the number of values this attribute can have. In the case of a continuous attribute, some form of discretization or other decision process must be made to determine the splits. In j48, this discretization is handled by creating a binary chop, I.E. finding some division that makes it a good split. If an attribute needs to be further divided, this is handled at a lower level in the tree. Once this splitting has happened, this decision of calculating the best splitting attribute is decided again on the subset of data. Figure 2.6 gives an example of running the j48 algorithm on the ARFF file illustrated in Figure 2.1. It can be seen here that the initial split was determined to be outlook, so one node is created for each possible value of outlook: sunny, overcast, and rainy. So the subsets of data that are passed to each node are the ones that contain only the specified values of outlook. There are three main processes that all decision tree learners implement in order to build their finished tress. This processes are grow, stop, and prune. J48 is found in the wEka [17] under *weka.classifiers.trees.J48*.

Grow

The method of determining a split in J48 is called *information gain*. The attribute that, when selected for a split, gives the best information gain is the one that will split. If this is the first split, this attribute will be the root of the tree. It can be seen in Figure 2.6 that outlook was the attribute that has the highest information gain. Information gain is calculated by using a formula to

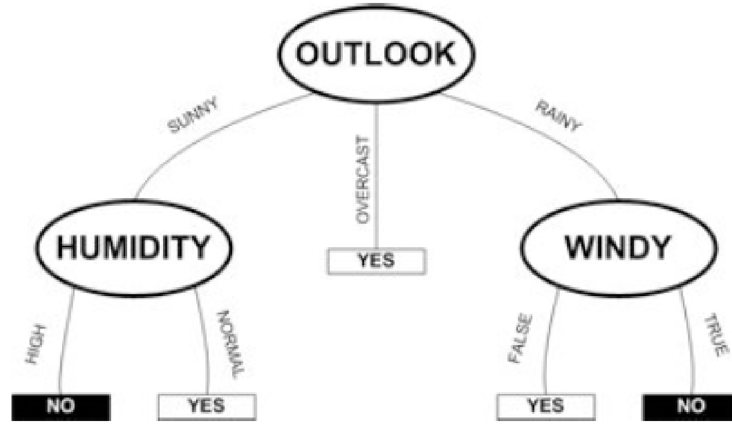

Figure 2.6: Example J48 Decision Tree Using Data from Figure 2.1.

determine the number of bits required to encode the class distribution of subset. This calculation is called *entropy*. Equation 2.3 illustrates the equation for entropy. If an attribute−value−class couple does not appear, the fraction in the log_2 will be zero. Since this is undefined, we say that if an attribute−value−class couple does not appear, its entropy is zero.

$$\begin{aligned} entropy(x_1, x_2, ..., x_n) &= -x_1 log_2(x_1) + -x_2 log_2(x_2) + ... + -x_n log_2(x_n) \\ => entropy(x_1, x_2, ..., x_n) &= -\sum_{i=1}^{n} x_i log_2(x_i) \end{aligned} \tag{2.3}$$

$$info([x_1, x_2, ..., x_n]) = entropy\left(\frac{x_1}{\sum_{i=1}^{n} x_i}, \frac{x_2}{\sum_{i=1}^{n} x_i}, ..., \frac{x_n}{\sum_{i=1}^{n} x_i}\right) \tag{2.4}$$

$$gain = info(current) - info(proposed) \tag{2.5}$$

Using the example in Figure 2.1, we can see that this data set consists of 14 insances with 5 of them being no and 9 being yes. Using Equation 2.3 and Equation 2.4, we can calculate the number of bits required to encode the natural class distribtion.

$$info([5,9]) = entropy\left(\frac{5}{14}, \frac{9}{14}\right) = \frac{5}{14}log_2(\frac{5}{14}) - \frac{9}{14}log_2(\frac{9}{14}) \approx 0.950683515\ bits$$

The entropy defined in Equation 2.3 defines the entropy for an attribute−value pair. In order to calculate the overall attribute entropy, a different equation is used. Each attribute can be broken up into n subsets where n is the number of possible attributes this value can have. If we consider the idea that the entropy of Equation 2.3 is the entropy of that one subset, S_i, then we can calculate the entropy of the entire attribute via Equation 2.6. This is a weighted sum where each component of the summation is the fraction of the entire set, S this subset contains multiplied by the number of bits required to encode this subset.

$$E(attribute) = \sum_{i=1}^{n} \frac{|S_i|}{|S|} entropy(S_i) \tag{2.6}$$

The following is an example calculation using the outlook attribute taken from Figure 2.1. The outlook attribute contains 3 values, these are sunny, overcast, and rainy. It can been seen from the example that sunny appears with 3 no instances and 2 yes instances, overcast appears with 4 yes instances and no no instances, and rainy appears with 2 no instances and 3 yes instances. Combining all of these and using Equation 2.6, it can be shown that encoding outlook requires approximately 0.624182525 bits.

$$E(outlook) = E([3,2],[0,4],[2,3])$$
$$entropy([3,2]) = -\frac{3}{5}log_2\frac{3}{5} - \frac{2}{5}log_2\frac{2}{5} \approx 0.970950594\ bits$$
$$entropy([0,4]) = -\frac{0}{4}log_2\frac{0}{4} - \frac{4}{4}log_2\frac{4}{4} \approx 0\ bits$$
$$entropy([2,3]) = -\frac{2}{5}log_2\frac{2}{5} - \frac{3}{5}log_2\frac{3}{5} \approx 0.970950594\ bits$$
$$=> \frac{5}{14} * 0.970950594\ bits + \frac{4}{14} * 0\ bits + \frac{5}{14} * 0.970950594\ bits \approx 0.624182525\ bits$$

J48 does this process for each of the attributes and then uses Equation 2.5 to calculate the information gain that would be gained by using this attribute for the next split. In the case of outlook, the example solution below demonstrates the information gain of choosing it.

$$gain = entropy(whole) - entropy(outlook) = 0.950683515\ bits - 0.624182525\ bits = 0.32650099\ bits$$

Stop

This process is continued for each of the subsets that result at each splitting point until a stopping condition is met. J48 uses two different stopping conditions. The first of these stopping conditions is met if the current node is pure. This means that the number of bits required to encode the subset of data this node is meant to split on is zero. In other words, only once class exists in this subset. The second stopping condition is met when the support for any more splits is too low. This is an over−fitting avoidance measure that means if the current subset has too few instances in it, no more splitting will occur. In the event that the first stopping condition is met, the class that is contained in the subset is reported. as a classification if that node is reached. In the event the second stopping condition is met, the majority class contained in the subset is reported as the classification if that node is reached.

Prune

Pruning is a method that is meant to avoid over−fitting as well as make the decision trees smaller. This process consists of randomly removing leaves, thus making that leaf's parent the new leaf and testing the overall error increase that is incurred from that removal. If the removal of a leaf increases the overall error of the tree by something less than a specified amount, the leaf is permanently removed. This process is recursed for all leaves of the tree and continues until the overall error of the removals becomes too great.

2.3.2 Ripper

Ripper, or Repeated incremental pruning to produce error reduction, is a rule based learner. It was originally proposed by William Cohen [6] for two reasons. Firstly, in his proposal he makes the claim that rule based learners did not scale well with increases in sample size as well as with data sets that contained a lot of noisy data [5]. A rule based learner typically produces a set of rules in the form of $x_i = a \wedge x_2 = b \wedge \ldots \wedge x_n = z \rightarrow C$ where x_1 through x_n are attributes and a through z are the values that the attributes have. C in the rule is the class that rule classifies. An instance of data is considered to be class C if it matches all clauses in the rule. A rule learner will typically create more than one rule to attempt to best classify the data. A very simple $straw-man$ rule learner is zeroR. ZeroR will only produce one rule, that rule is simply report the majority class in the data set. An extension to zeroR is oneR. OneR works similiary to zeroR but instead of simply reporting the majority class, it will select an attribute that has the best prediction rate and report a class for each one of the that attribute's possible values. As was stated about zeroR, oneR is also meant to be a $straw-man$. OneR would only perform well if the attribute that it picked was very highly correlated to the class attribute.

Rule based learning is different from the decision tree learning described in Section 2.3.1 in a few ways. Firstly as was discussed in the decision tree learning section, decision tree learning works on a divide−and−conquer method. This involves splitting up the data and individually processing those halves recursively until a final tree is created. The method most rule learners, Ripper included, use is called seperate−and−conquer. This is different than the other method because it involves creating a rule that *covers* specific instances in the data set and then removing those covered instances from that set and repeating the process until some stopping condition is met or there are no more instances to process. It is because of this method that rule learners are sometimes refered to as covering algorithms. Another difference between rule learning and decision tree learner is that in decision tree learning it is only possible to traverse down one possible path in the tree, that is each split must be unambiguous. In rule learning it is entirely possible for

rules to overlap and what class the instance is classified is up to the implementation, some learners will report one of them at random whereas others will report all rules that are fired.

The implementation of Ripper that is used in this thesis is called jRip and is a Java implementation of the Ripper machine learner that is found in the wEka [17]. Ripper can be found in the wEka under *weka.classifiers.rules.JRip*. Also, zeroR and oneR can be found in the wEka under *weka.classifiers.rules.ZeroR* and *weka.classifiers.rules.OneR* respectively.

Ripper uses a unique process of creating and testing rules in order to come up with the final rule list. It can be subdivided into two distinct parts the IREP* algorithm, which is used to grow and create rule sets, and the RIPPER_k algorithm, which creates competing rules and is meant to optimize the error rate of the rules generated by IREP*.

IREP*

IREP* is the final procedure that Cohen came up with in his initial proposal of the Ripper algorithm [6], IREP. IREP, or incremental reduced error pruning, was developed by Fürnkranz and Widmer [16]. IREP is a seperate−and−conquer rule learning algorithm that takes the training set that is given to it and divides it into two sets, the *growing−set* and the *pruning−set*. In Cohen's Ripper implementation, the growing−set $\frac{2}{3}$ of the instances and the pruning−set gets $\frac{1}{3}$ of the instances. IREP will work in seperate−and−conquer by greedily growing a rule set using the growing−set and then removing conditions from the rule that maximize the function given Equation 2.7 [6]. In Equation 2.7, P and N are the total number of positive and negative examples in the pruning−set respectively and PrunePos and PruneNeg are the number of positive and negative examples in the pruning−set covered by the rule respectively.

$$v(Rule, PrunePos, PruneNeg) = \frac{P + (N - n)}{P + N} \tag{2.7}$$

IREP* is an improvement to this process in two major ways, changing Equation 2.7 to Equation 2.8 and adding a stopping condition to the greedy growing phase of IREP. The reasons for changing the function to maximize is because the original would tend to pick rules that covered a lot of one class, even though a large amount of the other class was also covered.

$$v(Rule, PrunePos, PruneNeg) = \frac{p-n}{p+n} \tag{2.8}$$

The current stopping of IREP was to stop if a rule generated had an error rate of higher than 50%, it would stop creating rules. This is a problem because if a rule is created that has low coverage, the variance of the error is very large and it is quite likely that it can have an error rate exceding the maximum. Cohen decided to create a new stopping condition that involves the entire current rule set instead of just the rule currently being created. After each rule is created, a *description length* of the rule set and the covered examples is computed. IREP* will stop adding rules with this description length is d bits larger than the smallest overall description length. Cohen's implementation of IREP* uses $d = 64\ bits$ [6].

RIPPER$_k$

RIPPER$_k$ is a process that works in concert with IREP* to create optimized rules [6]. RIPPER$_k$ will take as input the rule set generated by IREP* and create two new rules for each rule in the set. The first rule RIPPER$_k$ creates a rule that is formed by growing and pruning a rule where the rule is pruned in order to minimize the entire rule set, replacing the rule that is currently being inspected by this new rule. The second rule that is created is formed by greedily adding conditions to the current rule being inspected. After this is done, for each original rule in the rule set, it is determined which of the three rules will be in the final set. This is done by determining which rule induces the best performance of the overall set. After this process is complete, RIPPER$_k$ generates

rules to cover the remaining positive examples in the set.

RIPPER_k has the subscript k because it is an incremental process. This process of taking the rule set and creating competing rules can be done multiple times. The version that the results in [6] were produced with were RIPPER_2.

2.3.3 Naïve Bayes

Naïve Bayes is a classifier that uses a probalistic model to classify its instances. Unlike the other models described in this chapter, there is no process of creating trees or rules. In fact, the Naïve Bayes classifier only requires one pass over the entire data set to construct a look−up table of frequencies. From that point on, Naïve Bayes will use that look−up table, coupled with a modified Bayesian probability equation to predict which class an instance of data is. The Naïve Bayes classifier was originally meant to be a *straw−man* classifier in that it was supposed to be a baseline learner to compare other learners against [8, 45]. One of the reasons behind this was because the look−up table that Naïve Bayes generates over the data makes no assumptions about any correlations between attributes that exist. In data sets where the attributes are highly correlated, this nuance of the classifier should be seen and performance should be poor. However, Domingos and Pizzani show that this performance problem is actually only a problem in a vanishingly small number of cases [8]. The Naïve Bayes classifier is implemented in the wEka [17] under *weka.classifiers.bayes.NaiveBayes*.

Attribute	Frequencies
Play	$\left(\frac{5}{14}, \frac{9}{14}\right)$
Outlook	$\left(\frac{3}{5}, \frac{2}{5}\right)$, $\left(\frac{0}{4}, \frac{4}{4}\right)$, $\left(\frac{2}{5}, \frac{3}{5}\right)$
Temperature	$\left(\frac{2}{4}, \frac{2}{4}\right)$, $\left(\frac{2}{5}, \frac{3}{5}\right)$, $\left(\frac{1}{3}, \frac{2}{3}\right)$
Humidity	$\left(\frac{4}{7}, \frac{3}{7}\right)$, $\left(\frac{1}{7}, \frac{6}{7}\right)$
Windy	$\left(\frac{2}{8}, \frac{6}{8}\right)$, $\left(\frac{3}{6}, \frac{3}{6}\right)$

Table 2.3: Look−up Table Generated by Naïve Bayes using the data given from Figure 2.1.

Using the data set illustrated in Figure 2.1, Naïve Bayes will construct a look−up table that can be seen in Table 2.3. Each tuple under the frequencies column in the data stands for $(freqency\ of\ no, frequency\ of\ yes)$ and there is one tuple for each attribute−value. It can be seen here that this frequency look−up table approach will not work on continuous values because there would be no real way to compute these frequencies. As can be seen from Table 2.3, the $outlook = overcast => play = no$ frequency is zero. This can create a problem because if the instance we are trying to classify contains $outlook = overcast$, the probability no will be zero. To counter this effect, a small decimal number is added to each of the probabilities. Exactly what this small number is is discussed later in this section.

$$p(H|E) = \frac{p(H)}{p(E)} \prod_i P(E_i|H) \tag{2.9}$$

The modified Bayesian probability equation can be seen in Equation 2.9. This equation can be put into simpler terms as $next = old * new$. We can calculate the current instance of data to be classified by using information about the probabilities of each attribute appearing in instances we have already seen, coupled with the new information we have just gained.

Using the example data set illustrated in Figure 2.1, we shall attempt to classify the instance $i = (rainy, cool, high, TRUE)$. In order to do this, we must compute the above equation to get the probability the instance is yes and the probability the instance is no. Comparing these two probabilities, we can classify this new instance. Since $p(i|yes) > p(i|no)$, Naïve Bayes will classify this new instance as yes. Note that $p(i|yes) + p(i|no) < 1$, this is because Equation 2.9 has been simplified to allow for easier computation. There are transformations that can be done to aquire the actual probabilities for each of the classes, but they are beyond the scope of this thesis.

$p(i|yes) = p(yes) * p(rainy|yes) * p(cool|yes) * p(high|yes) * p(TRUE|yes)$
$= \frac{9}{14} * \frac{3}{5} * \frac{2}{3} * \frac{6}{7} * \frac{6}{8}$
$= 0.165$
$p(i|no) = p(no) * p(rainy|no) * p(cool|no) * p(high|no) * p(TRUE|no)$
$= \frac{5}{14} * \frac{2}{5} * \frac{1}{3} * \frac{1}{7} * \frac{2}{8}$
$= 0.0017$

As was stated earlier, in the data set of Figure 2.1, the frequency for $outlook = overcast =>$ $play = no$ is zero. There a few techniques for countering this [42,43]. The first is the m−estimate, seen in Equation 2.10. In Equation 2.10, I is the total number of training instances in the set, C is the total number of classes in the set I, $N(H)$ is the frequency of the hypothesis, H, within I, and m is a small, non−zero constant, often $m = 2$.

$$P(H) = \frac{N(H) + m}{I + m * C} \tag{2.10}$$

There are three special cases of Equation 2.10. These are:

- When it is a high frequency hypothesis in large training sets. $N(H)$ and I are the dominant terms in the Equation 2.10 and much largerthan the other terms. As such, the equation can be simplified to $P(H) = \frac{N(H)}{I}$.
- When the frequency of the class is very rare in a very large training set, $N(H)$ is small and I is large. In this case, the Equation 2.10 can be simplified to $P(H) = \frac{1}{I}$. If this were not the case, rare classes would never appear in prediction.
- When the data set is very small Equation 2.10 approaches the inverse of the number of classes, or $P(H) = \frac{1}{C}$. This is very useful in situations discribed above where an attribute−class combination has not been seen yet.

The probabilities calculated using Equation 2.10 is a very useful lower bound for $P(E_i|H)$. If some value v is seen $N(f = v|H)$ times in feature f's observations for hypothesis H, then the lower bound can be calulated using Equation 2.11.

$$P(E_i|H) = \frac{N(f = v|H) + l * P(H)}{N(H) + l} \tag{2.11}$$

Here, l is the *Laplace−estimate* and is set to a small constant [42, 43]. As with Equation 2.10, there are some special cases of Equation 2.11.

- When there are many examples of a hypothesis H and a number of observations have been made for a value v, $N(H)$ and $N(f = v|H)$ are large and Equation 2.11 approaches $\frac{N(f=v|H)}{N(H)}$.
- When there is very little evidence for a rare hypothesis, $N(f = v|H)$ and $N(H)$ are small, Equation 2.11 approaches $\frac{l*P(H)}{l}$, or that the default observation of an hypothesis is a fraction of the probability of that hypothesis. This estimation is appropriate when very little data is available.

2.3.4 Random Forests

Random forests are a classifier that uses decision tree classifiers internally. Random Forests were first proposed by Leo Breiman in [4]. The random forest will, when given an instance of data to classify, poll the decision trees in its forest and take the majority vote given by the forest as the classfication of the instance. In order to create this forest of decision trees, each tree is given a column−pruned subset of the data. This is, each tree is given a different look at the data in that certain attribute are removed from the data set before being given to the tree. This causes the trees to build completely different classification models about the data. Also, the trees in the random forest are grown *greedily*, or without the minimum support stopping criteria described in

Section 2.3.1 as well as without the pruning process described Section 2.3.1. The idea behind this is that if any tree is too over−fit on the data it was given, the resulting vote of the other trees should be too great for this to effect the overall classification of the instance. Also, since the random forest is growing an exponential number of trees, removing the process of pruning altogether would incur a significant speed improvement. Random forests are implemented in the wEka [17] under *weka.classifiers.trees.RandomForest*.

The method of selecting which attribute are removed from each file is, as the name of the classifier says, random. Essentially there are 2^n possible combinations of attributes for each data set, where n is the number of attributes in the set. In our example in Figure 2.1, there are 4 attributes, so a total of $2^4 = 16$ possible trees could be created. Generally, there is some minimum number of attributes required for a tree to learn on. The word used to describe the set of attributes a given tree in the learner is meant to use to grow the tree is called a *vector*. A vector is essentially a boolean list that contains only zeros and ones. An example vector for the data set described in Figure 2.1 could be $a = [1\ 0\ 1\ 0]$. This would mean the tree is only given the attributes of outlook and humidity to learn on, and should ignore the rest or that the rest are to be removed prior to the tree learning the data set.

Random forests were not used in any of our experiments, however, they were used in an experiment by Lan Guo et al. [18] in the field of defect detection. Section 4.6, Section 4.7, and Section 4.8 are all experiments on defect detection, so we felt it was necessary, since their experimental results are reported in Section 2.8, to briefly discuss the random forest classification model.

2.3.5 TAR3

TAR3 is the third machine learner in a line of machine learners that are fundamentally different than the classic learners of j48, Ripper, and Naïve Bayes . First of all, TAR3 is not a classification learner, it is a lift learner. Lift learning, or treatment learning as it is sometimes called, is a process of learning that involves creating a rule to *lift* the overall value of a data set up. A lift learner

typically will output several rules that are not meant to be used like the rule−sets described in Section 2.3.2. Instead of being taken as a collection of rules meant to classify an instance of data, lift rules are a collection of rules that are meant to isolate one specific class of data. The TAR3 learner will report several rules that are isolated form each other. They all are meant to be able to classify one specific class, but that class is the same class. In instances where a data set only consists of two classes, like Figure 2.1, treatments can act as classifiers by simply adding an else clause. For instance, if a treatment for the data of Figure 2.1 was $outlook = sunny$, then we could create the rule set $outlook = sunny \rightarrow yes, else \rightarrow no$. This idea only works for two class data sets however, and further discussion on using treatment learning for in two class systems can be seen in Chapter 4.

To better explain, we will use the data set illustrated with Figure 2.1. There are two classes values in this data set, yes and no. Let us suppose that we want to know what attributes best predict what days we will play. It is entirely possible to take the output of a decision tree learner and parse the branches that end in play to come up with what attributes best predict play, but this process on large trees would be very cumbersome. Lift learning is meant to isolate those attributes that best predict one class, or higher valued classes. What TAR3 does is rank these class values from worst to best. It does this by assigning a number to each class value. In our above example, no would be given the value $2^1 = 2$ and yes would be given the value $2^2 = 4$. This creates a seperation of the classes. The method of calculating the value of a class value is illustrated with Equation 2.12. In this equation, n is the number of classes, in ranked order, in the data set, $f(i)$ is the number of times that class appears, or the frequency of that class, S_{Rx} is the subset this treatement rule covers, and $lift_{all}$ is the lift of the normal data set. What lift learners then attempt to do is create rules that maximize the overall value of the data set. In Figure 2.1, the overall value of the unlifted data set is $\frac{5*2^1+9*2^2}{14} \approx 3.286$. If we were to create the treatment $outlook = overcast$ would we have a new data set to compute the lift of, by removing the subset that is covered by this treatment. This new data set has a lift of $\frac{4*2^4}{4*3.286} \approx 1.217$. This number must be greater than one for it to be an

acceptable rule. This is because if it is less than one than it actually produces a new subset that contains less of the best class values than the original.

$$lift_{R_x} = \frac{\sum_{i=1}^{n} f(i) * 2^i}{|S_{Rx}| * lift_{all}} \quad (2.12)$$

TAR3 has several parameters that can be changed alter the behavior of the learner. This parameters will be used throughout the description of TAR3 and can be found in Table 2.4.

Parameter	Description
granularity	Number of bins used for equal frequency discretizations.
maxNumber	Number of treatments to keep at each iteration.
maxSize	Largest number of attribute−value pairs a treatment can have.
randomTrails	Number of iterations of treatment building.
futileTrails	Number of iterations allowed to happen without new treatments being created.
bestClass	Support measurement. Treatment must select this percentage of the best class to be acceptable.

Table 2.4: Configurable Parameters for TAR3.

TAR3 will attempt to build treatments almost entirely at random. It will first generate a number N from one to maxSize. This is the number of conjunctions this treatment R_x is going to have. The method of selecting the attributes to be used in R_x is done by scoring each attribute−value pair in isolation. Essentially, TAR3 computes the *lift* of each attribute−value pair. It then randomly selects one of these pairs as the first conjunction, weighting the chances to pick a high−scoring pair. It randomly selects the N conjunctions before testing the treatment. After all N pairs have been picked the rule's lift is gotten via Equation 2.12. Rules that have a lift less than one are promptly forgotten. This process of creating rules is done randomTrails number of times. After a randomTrails number of treatments are created, maxNumber of the best treatments are selected and the process is repeated. This process is repeated indefinitely or until futileTrails number of

trails occur in a row where no new treatment is generated.

2.4 Isometric Space Evaluation of Performance Heuristics

Fürnkranz et al. proposed a novel method of comparing the evaluation heuristics in machine learners in [15]. In order to understand their method of comparison, a few evaluation properties must first be understood.

2.4.1 Probabilities of Detection and False Alarm

Probability of Detection, or PD, and Probability of False Alarm, or PF, are two statistics that are commonly taken about the performance of detectors created by machine learners. Both of these methods are meant for the two class case, but can be extended to multiple classes fairly easily. The PD of a detector is the probability that a detector will correctly detect a member of the positive class. Equation 2.14 is the equation that is used to calculate PD. p in this equation is the number of positive classes that were detected by a detector and P is the total number of positive classes in the data set. The PF of a detector is the probability that a detector will misclassify an instance of the negative class as an instance of the positive class. Equation 2.14 is the equation used to calculate PF. n in this equaition is the number of negative examples classified as positive by the detector and N is the total number of negative examples in the data set.

$$PD = \frac{p}{P} \tag{2.13}$$

$$PD = \frac{n}{N} \tag{2.14}$$

The most common represention of a learner's performance is the *confusion matrix*. A confusion matrix is a table of classes that represents how the class values in the data set were classified by the

detector. Figure 2.7 represents a confusion matrix of five classes, A, B, C, D, and E. The diagonal from (A,A) to (E,E) represents to number of times the detector classified the class correctly. PD and PF cannot be calculated directly from this matrix. As was stated, PD and PF can only be calculated for a two class data set. In order to do this the confusion matrix must be transformed to one that consists of $+$ and $-$ classes. The $+$ class consists of all of the positive instances of a chosen class. The $-$ class consists of all other instances in the data set.

	A	B	C	D	E
A	9	1	0	2	2
B	4	8	2	3	1
C	2	0	9	0	0
D	0	0	0	9	6
E	0	1	2	3	5

Figure 2.7: Example Confusion Matrix

As an example, let us choose $+ = A$ and construct a confusion matrix from this that we can calculate the PD and PF for A from. Figure 2.8 represents this constructed confusion matrix for just A as the $+$ class. Adding together the rows first row will result in the total number of positive class in the data set, or P in Equation 2.14, adding together the second row will result in the total number of the negative class in the data set, or N in Equation 2.14. p and n can be retrieved by taking the square marked $(+,+) = p$ and $(-,+4) = n$. We can see from this example that $pd = \frac{9}{15} = 0.6$ and $pf = \frac{4}{47} = 0.09$. It can be seen from this example that PD and PF are not complementary of each other.

	$+$	$-$
$+$	9	4
$-$	6	47

Figure 2.8: Confusion Matrix for A from Confusion Matrix in Figure 2.7.

These values of PD and PF can be calculated for each class in the confusion matrix to get the overall performance of the detector.

2.4.2 Receiver Operating Characteristic Curves

Receiver operating characteristic, or ROC, curves, are a method of visualizing the probability of detection and probability of false alarm. A ROC curve is a two dimensional plot with a domain of $(0,1)$ and a range of $(0,1)$. A ROC curve has several interesting areas about it that are illustrated with Figure 2.9.

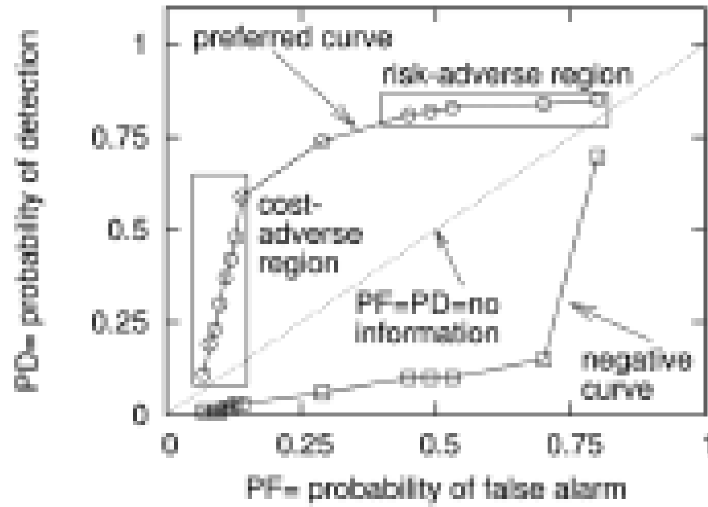

Figure 2.9: Illustration of Different Areas of the ROC Curve.

There are several interesting properties of these curves. The ideal place for a detector to be is at the point $(0,1)$, this means that the detector has a 100% detection rate and a 0% probability of false alarm rate. In practice, this type of performance cannot be met, so different sacrifices can be made to better accommodate the task. The area marked as *risk adverse* in Figure 2.9 explains an area of the curve that has a very high PD but also a very high PF. This could be equated to a mission critical situation. Airport security is a good example of detectors that need to be in the risk adverse region. An airport employee may check thousands of people a day and generally none of them are of any threat, however, letting the threats through security for the convenience of the non−threatening passengers is not an option. The opposite end of the spectrum is the cost−adverse region. An example of this might be the filling of cereal boxes in a factory. The amount of money saved from

not checking the weights of every box far outweighs the several under and overweight boxes that slip through the system. The diagonal line in Figure 2.9 represents the line of no information. A detector that falls on this line has no better performance than randomly guessing at every instance. The prefered curve is a curve that is above the line and the prefered position on that curve is the one closest to the optimum point.

2.4.3 Isometric Space

Isometric Space was developed by Fürnkranz et al. as a visual means to compare the performance heuristics of machine learners [15]. A detailed explanation of ROC spaces and Isometric spaces can be found at [14].

Isometric space is similar to the ROC space that was described in Section 2.4.2 in that it takes into account number of positve and negative examples found in the data set. Unlike ROC space, isometric space is not normalized, that is the domain is the number of negative examples in the data set and the range is the number of positive examples in the data set. Also, unlike ROC curves, isometric curves do not illustrate the performance of a detector, instead they illustrate where a detector would be in this space. That is, each point in isometric space represents a detector generated by a machine learner and each curve in isometric space represents a connection between all detectors that a learner would give the same score. Equation 2.16 through Equation 2.18 give different scoring heuristics for several different machine learners. By visualizing these in isometric space it is easy to see how different heuristics behave as the probability of detection and probability of false alarm change. Figure 2.10 illustrates some of these heuristics.

$$H_{j48} = -log_2\left(\frac{p}{p+n}\right) \tag{2.15}$$

$$H_{precision} = \frac{p}{p+n} \tag{2.16}$$

$$H_{ripper} = \frac{p-n}{p+n} = 2 * H_{precision} - 1 \tag{2.17}$$

$$H_{balance} = \frac{\sqrt{pd^2 * \alpha + (1-pf)^2 * \beta}}{\alpha + \beta} \tag{2.18}$$

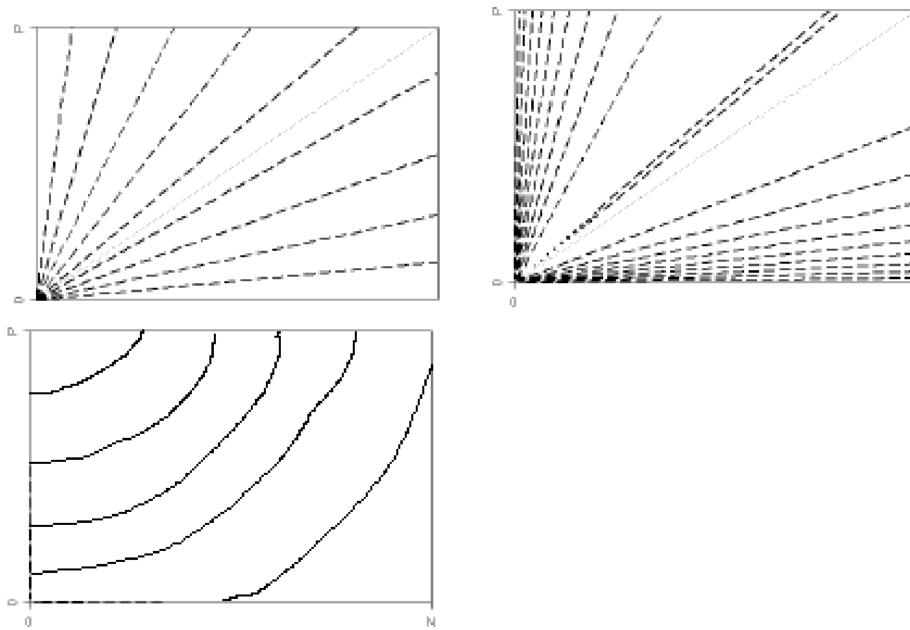

Figure 2.10: Isometrics of Precision, J48, and Balance.

2.5 Cross Validation

The experiments of Section 4.3, Section 4.4, Section 4.6, Section 4.7, and Section 4.9 all use the ARFF file described in Section 2.1. The files were taken from two different repositories, UCI [3] and PROMISE [29]. There is a problem in testing the perfomance of a machine learner

on files in this form. If the file were to be left alone and used as both the train and the test set for performance, an over−estimation of the performance would exist because the learners are given the unreal advantage of using the same information they will be tested on to learn their detection sets. In order to prevent this from happening in our tests, we use a method of dividing the data up into a set of train data and a set of test data called cross validation [39].

Cross Validation is a method of testing a machine learner when only one, flat data set exists [39]. The purpose of cross validation is to allow a learner to be tested on data it has not trained on when two distinct sets do not exists. Cross validation is a purely test based approach to evaluating machine learners. Cross Validation is a two step process that allows for a strong evaluation of a learner.

Cross Validation has an inner and an outter phase that each consist of a variable number of steps. The inner phase involves breaking up the data set into two sets that are completely disjoint. A typical approach to this is to put 90% of the data into the *train* set and the remaining 10% in the *test* set. This process is repeated 10 times so once it is complete there will be 10 train and 10 test sets where each test set has completely unique data instances. This number 10 is typically refered to as N. The outter phase randomizes the data a set number of times, usually 10, and repeats the inner phase for each randomization. This outter phase count is typically refered to M. The reason for randomizing the data every pass is to remove the potential for there to be patterns in the data file itself. Overall, this method runs the learner MxN times. This large number of runs allows different evaluation methods to determine how well the learner works overall.

For the purposes of the MDP and Turkey data sets refered to in Section 4.6 and Section 4.7 respectively, a M value of 10 and an N value of 3 was used. The reason 3 was used for N was because some of the data files have very few defects in them and breaking up the data into 10 train−test sets causes some test sets to have no defects in them. This caused erroneous reporting of the probability of detection of the learners in some sets. Using this setup, each learner was ran and tested 30 times for each data file.

2.6 Evaluation Processes

2.6.1 Sum of Differences

The sum of differences evaluation process is used in Section 4.3 with the experiments that compare the performance of the TAR3 learner to the performance of the Which learner described in Section 2.3.5 and Chapter 3 respectively. It is a simple process of comparing the score of the detectors generated by each of the learners for each data set they are given. Essentially, each learner will report one rule with one score for each instance of the data set after cross validation. These numbers are compared and the winner gets a point. In the event where the learners tie, they are given a tie point. The tables that are used in Section 4.3 give the results in the form of Which compared to Tar3. Figure 2.11 will be used as a reference for the following example.

Win	**Loss**	**Tie**	**Sum**
100	30	100	70

Figure 2.11: Example Row in a Sum of Differences Chart.

In Figure 2.11, there are four different columns. These columns refer to: the number of times Which $>$ Tar4, the number of times Which $<$ Tar3, the number of times Which $=$ Tar3, and $Win - Loss$. If the final column is negative, it means that the Which learner did worse than Tar3 overall. It is important to note that in the example of Figure 2.11, there were a total of $win + loss + tie = 100 + 30 + 100 = 230$ total trails and that Which won $\frac{100}{230} \approx 43.4\%$ of them, lost $\frac{30}{230} \approx 13\%$ of them, and tied $\frac{100}{230} \approx 43.4\%$ of them. In our evaluation, we consider any positive number in the final column to be a win, but a large number of ties implies that generally, both learners performed the same.

2.6.2 Quartile Charts

Quartile Charts have been used in several different experiments as a means of measuring and comparing the performance of different machine learners [30, 34]. The quartile chart is a method of summarizing a large amount of results into a simple graph. Also, it is very easy to see standout performance results amongst the different learners being compared.

Figure 2.12: Example Quartile Chart.

Figure 2.12 is an example of the quartile charts used to report many of the experimental results in Chapter 4. They consist of five defining characteristics. The first is the circle, this stands for the position of the median of the distribution of numbers. The white space in between the left of the circle the left−most horizontal line is the second quartile. The white space between the right of the circle and the right−most line is the third quartile, the left−most horizontal line represents the first quartile, and the right−most horizontal line represents the fourth quartile. The vertical line in the middle stands for the 50% mark. This type of performance method is meant to be used for normalized numbers in the range of $(0,100)$. The process of determining the the different quartile−ranges is done by sorting the numbers in ascending order and placing the numbers into four groups, each group represents a different quartile. The quartile chart example of Figure 2.12 used the following distribution of scores for generation.

$$(4,7,15) \;\; (20,30) \quad 40 \quad (52,64) \;\; (70,81,90)$$

We use quartile charts to report the results in a variety of our experiments. They are nice because in many of our experiments we have thousands of resulting scores to compare and a simple table of quartile charts, one for each learner, is able to effectively display the overall performance of each learner in a very succinct way. Also, quartile charts make no assumptions about the underlying distribtion of the results, like t−tests do [40].

2.6.3 MannWhitney U−Tests

MannWhitney U−Tests are non−parametric method of ranking different distributions [27]. In many of our experiments, we use the MannWhitney U−Tests in conjunction with the quartile charts of Section 2.6.2 to report the results. U−Tests are similar to t−tests in that one can supply a desired confidence level to establish whether one learner performs better than another learner at that confidence level.

The main process that was used in our experiments with the MannWhitney U−Test is one of ranking the performance of our learners when compared to each other. This process of ranking is done by combining the two distributions to be compared into one distribution and sorting these distributions in ascending order. From there, each value is given an integer rank based on its position in the sorted list. The rank is the positions. After this, all values in the distribution that are the same are given a new rank that is the average of all of their ranks. Then, these ranks are appplied and the sum of these rankings and their median rank is calculated. From there, their $U-$statistics are calculated via Equation 2.19. In all of the equations in this section, N_i is the number of values in the ith distribution

$$U_x = sum_x - N_1(N_1+1)/2 \tag{2.19}$$

$$\mu = \frac{N_1 N_2}{2} \tag{2.20}$$

$$\sigma = \sqrt{\frac{N_1 N_2 (N_1 + N_2 + 1)}{12}} \tag{2.21}$$

$$Z_x = \frac{(U_x - \mu)}{\sigma} \tag{2.22}$$

Using the $U-$statistic, a $Z-$curve is calculated via Equation 2.20, Equation 2.21, and Equation 2.22. At the 95% confidence level, $|Z_x| > 1.96$ in order for the two distributions to be consid-

ered different. Using this method of comparing two distributions, $Z_1 = -Z_2$. Figure 2.13 illustrates an example of this process in detail for two distributions, A and B.

The distributions and their combinations are:

$$\begin{aligned} A &= 4\ 6\ 8\ 9\ 40\ 100 \\ B &= 1\ 3\ 5\ 7\ 9 \\ A \cap B &= 1\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 9\ 40\ 100 \\ A \cap B &= 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8.5\ 8.5\ 10\ 11 \\ A_{rank} &= 3\ 5\ 7\ 8.5\ 10\ 11 \\ B_{rank} &= 1\ 2\ 4\ 6\ 8.5 \end{aligned}$$

The sum and median statistics are:

$$\begin{aligned} sum_A &= 3+5+7+8.5+10+11 = 44.5 \\ sum_B &= 1+2+4+6+8.5 = 21.5 \\ median_A &= 7.75 \\ median_B &= 4 \end{aligned}$$

The U−Statistic can then be calculated:

$$\begin{aligned} U_A &= sum_A - \frac{6*7}{2} = 23.5 \\ U_B &= sum_B - \frac{5*6}{2} = 6.5 \end{aligned}$$

Frome these variables, the Z−Curve values can be found:

$$\begin{aligned} \mu &= 15 \\ \sigma &\approx 5.477 \\ Z_A &\approx 1.55 \\ Z_B &\approx -1.55 \end{aligned}$$

(Note that $Z_A = -Z_B$. This was discussed earlier and will always be the case.) As was stated, we compare $abs(Z) > 1.96$. Since this is false, both A and B get a tie. In other words, they cannot be distinguished from each other.

Figure 2.13: Example of Using MannWhitney U−Tests on Two Distributions.

The MannWhitney U−Test was designed as a non−parametric test that ignores the underlying distribution of the values entirely. This test also does not require the distributions be the same size either, which can be seen in Figure 2.13. This test was also designed to determine of two distributions were different from each other. This is not enough for our evaluation purposes because saying the scores of one learner are different from the scores of the other learner does not tell us

which one is better. In order to do that, Menzies et al. devised a method of using the MannWhitney results coupled with an additional test afterwards [31]. Once the MannWhitney test has been run, if the two distributions are the same, both of them get a tie point. However, if they are not the same a comparison is made between the medians of the ranks of the two distributions. For the purposes of our experiments, we are using distributions of scores, the higher the score the better. For this case, we can say:

- If $median_A > median_B$, then $wins_A$ and $losses_B$ get points; else,
- If $median_A < median_B$, then $wins_B$ and $losses_A$ get points; else,
- $ties_A$ and $ties_B$ get points.

2.7 The Pareto Distribution

The Pareto distribution was originally developed by Vilfredo Pareto to describe the distribution of wealth in society [26]. This distribution occurs very profoundly when looking at the distribution of faulty modules or files in software projects. Several other researchers in the field of defect detection have reported this same type of distribution appearing in the software projects they had researched on [20,23,36,44,46]. An example plot of the Pareto distribution is seen in Figure 2.14.

The Pareto distribution has a more common name of the "$80-20$" rule, or in terms of the economic portion of society, 80% of the wealth is controlled by 20% of the population. Our Koru diagrams of Section 4.1 are specifically layed out the way they are to exploit this same phenomenon in the software projects that we explore. We have recognized this distribution occuring in our experiments of Section 4.6, Section 4.7 and Section 4.8. For these sections, we define what we call the *oracle*, the oracle is the plot of all of the defective modules that exist in the project sorted by their lines of code. In all of these plots, the oracle follows the same curve as the Pareto distribution curve in Figure 2.14.

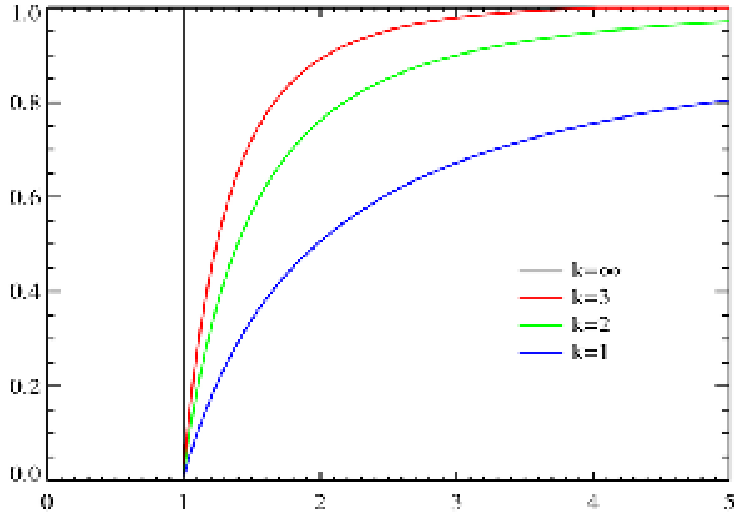

Figure 2.14: Example Pareto Distribution.

2.8 Previous Experiments

Many of our experiments in Chapter 4 deal with the problem of trying to select which modules or files in a software project are likely to be defective. There have been several different approaches in the past to attempt to use the more classic learners to solve this problem [18, 30, 32, 33] and also the use of some statistical models instead of classic learners [1, 2, 35, 37].

2.8.1 Classic Machine Learners

[18] use the random forest machine learner described in Section 2.3.4 to predict defects on some of the MDP data sets that were used in Section 4.6. Their results were very promising in that the reported scores of $(PD, PF) = (0.87, 0.25)$ They also applied the random forest approach as a method of selecting the most important attributes in these data sets and selected five attributes from each. Their results in this area showed that none of the attribtues were equally as important per data set. For a description of the attributes see Table 4.6. While their results were promising in terms of PD and PF, the paper did not study the effects of adding in the amount of code needed to

be searched, or *effort*, into their evaluation. Effort can be defined by Equation 2.23. In other words, given a detector, effort is the total lines of code instances that the detector classifies as defective contain over the total number of lines of code all instances in the set contain.

$$effort = \frac{\sum_{i=1}^{D} LOC_i}{\sum_{j=1}^{N} LOC_j} \tag{2.23}$$

[30] discusses the performance of the classic machine learners of Naïve Bayes , j48, and oneR on the MDP data sets discussed in Section 4.6. Naïve Bayes and j48 were discussed in detail Section 2.3.3 and Section 2.3.1 respectively and oneR was discussed breifly Section 2.3.2. They also compared the learners with each other and with themselves using no preprocessing and a preprocessor that takes the logarithm of the attributes. Their conclusions were that Naïve Bayes with the preprocessor performed outstandingly better than the other five preprocessor−learner configurations. They report average performance results of $(PD, PF) = (0.71, 0.25)$ for Naïve Bayes . Like the previously discussed experiment, these performance results to not take into account the amount of code that needs to be searched in their evaluations.

[32,33] both use a learner called ROCKY to report their results on the MDP data sets described in Section 4.6. ROCKY is a machine learner that has a very simple rule learning policy. ROCKY will create rules of length one that are tuned for not only the detection of defective modules but also to minimize the amount of effort these modules use. Essentially, attempt to detect as many of the fault modules as possible under some effort constraints.

[32] creates a commercial scenario as well as a mission critical scenario and give different results for each. With a commercial scenario they arrive at $(PD, PF, effort) = (0.37, 0.37, 0.43)$ and in the mission critical scenario they arrive at $(PD, PF, effort) = (1.0, 0.75, -)$, where effort is not a factor that matters. This project was meant to be a comprehensive study of the effects the different Halstead [19] and McCabe [28] attributes have on the performance measures of PD, PF,

and effort.

[33] creates an experiment to further the testing of ROCKY on the MDP data to discuss the different performances of ROCKY on sets of the attributes. They divide these sets up into three parts. These are the same divisions that are described Table 4.6. These are thh Halstead, McCabe, and LinesOf attributes. These experiments conclude with three different behavioral facts about the relationship of PD, PF effort, and accuracy. These are that PD rises with effort, but rarely exceeds it, PD and PF rise together and that PD, PF, and effort can vary significantly while accuracy is uneffected.

2.8.2 Statistical Models

[2, 37] discuss the use of a statistic model called negative binomial regression to provide a sort order to inspect the modules. They do not explicitly discuss the line of code style effort in their work, but instead use a percentage files to evaluate the performance of their tool. While this is not an exact correlation to lines of code, it still gives some estimate of the effort their model requires to produce the detection rates. The two experiments of these authors use a the same tool on two different projects and in both projects report finding 80% of the defects in 20% of the files. These experiments use the AT&T data described in Table 4.10 to evaluate their model.

[35] uses a type of linear model called discriminant analysis on static code attributes to order them much in the way the previous two experiments did. They report results of being able to classify between 62% and 72% of the modules that they they used this model on. This experiment does not take any form of effort into account during the evaluation.

[1] created statistical models for the use of a Java legacy system. As a result their model attempting to detect fault prone Java classes as opposed to the module and file based approaches described previously. Much like the work in [30], the authors saw that the attribute's distribution was heavily skewed to the right and applied a logarithm transformation to the data before using the model. They use a statistical model of logistic regression to achieve similar effects in detecting the

fault prone classes. While they do not give any explicit numbers for PD, they do give a PF rate of less than 20% and a false negative rate of less than 20%.

2.9 Summary

Chapter 2 has been a broad overview of the different aspects of this very diverse field. The remaining portion of this thesis will cover these specific aspects of this field:

- The introduction of a stochastic best−first rule learner, Which;
- The performance of Which when compared to the treatment learner, TAR3;
- The effects of treatment learning when applied to differing train and test sets;
- The performance of Which in the realm of classification accuracy;
- A novel form of evaluating the performance of defect detectors, the Koru Diagram;
- Two manual searching methods of defect data;
- The effects of treatment learning when applied to heurisitics other than lift;
- The performance of the classic learners in the realm of defect detection;
- The performance of the manual methods;
- The performance of Which in the realm of defect detection; and,
- Creating external validity for the subsampling process of micro−sampling;

Chapter 3

Which

The basis for the Which machine learner is that it was meant to be an improvement over TAR3. Both TAR3 and Which are treatment learners. As was discussed in Section 2.3.5, treatment learners can be used to classify instances in a two class system, which is what several of the experiments in Chapter 4 exploit. TAR3 and Which both create a list of potential treatments that when applied to the data set will increase the frequency of the higher−valued classes in the subset. There are two very key differences in the processes that they both take.

The first key difference is in how Which and TAR3 build their rules. TAR3, as discussed in Section 2.3.5, builds its rules by randomly selecting N pairs of attributes and their values and then testing the rule. Which attempts to build rules slowly by attaching existing rules to new attributes, this process is discussed in more detail in Section 3.2.2 and Section 3.2.3. This difference means that Which spends far less time creating and destroying futile rules, because it has more intelligence in its selection policies.

The second key difference is in the reporting of TAR3 and Which. TAR3 was constructed to report as many promising rules as it can find, given the trail constraints. Which was constructed to find the best possible rule that it can. While it has the option to report the top N rules that it found, it is very often the case that those rules are either subsets or supersets of the best rule. Which

makes no attempts to create multiple rules, any other rules that are promising that are created are simply a side effect of the processes Which uses to create its best rule.

3.1 Idea Behind Which

Which is a stochastic best−first search. The idea behind this is a method of searching that relies on previous knowledge that has already been discovered. Essentially, Which uses what it has previously discovered via expansion to attempt to take great leaps through the search space to approach a solution quicker. The following chapter will discuss the different properties of Which. The beginning of this chapter discusses the intial idea and creation of Which, Section 3.1.1 dicusses the best first search algoritm that Which uses in its search for treatments, Section 3.2 discusses the different data structures and code structures used in the Which implementation with Section 3.2.2 and Section 3.2.3 discussing how Which builds its rules and Section 3.2.4 discussing how Which scores its rules.

3.1.1 Best−First Search Implementation

The basic idea of Which is to use a stochastic best first search [38]. As described in [38] a best first search is a method of searching a tree by exploring the nodes that score well based on a heuristic function first. This expansion continues down the tree until a desired solution is found. This function may depend on the description of the node, the description of the goal, the description of the path from the root to the current node, and any heuristic knowledge of the domain. Which itself mainly uses the path as the description of the node as well as the description of the goal to score its nodes. However, as is explained in Section 3.2.4, this scoring heuristic is completely arbitrary to the core Which functionality.

The idea behind using the best first search here is a little different than the definition above. Which has no direct root, but instead one root for each attribute in isolation. Which will imme-

diately score each attribute's ranges independently of each other. After this phase Which consists of n attributes each with m nodes expanded. It is here that Which decides which attribute range to expand next. Its method of doing this is to choose two paths from two trees, combine them, and then score them to see if they score well. If this is a very promising path, there is a good chance that Which will attempt to combine this new node with a different node. This process of combining is further explained in Section 3.2.2 below. The two attribute-ranges that are picked can be from the same tree. In this case the rule produced has a disjunction, instead of a conjunction. That is, if two ranges of the same attribute are picked, Which will join them together as a disjunction. The aforementioned good chance above relates to the method in which Which will pick the next to paths to combine. This is explained below in Section 3.2.3. Which will continue to pick two paths, combine them, and score them until one of the possible stopping conditions listed in section Section 3.2.5 occurs.

This differs from the basic definition of the best−first in that a best−first search will only add singletons to the current best path. That is, once the current paths have been evaluated and the next path has been picked, a best−first search will only expand the best path with one more node. Which, on the other hand, will happily attach an entire other search path to another path to create a new, possibly significantly more complex, search path. More on this method is discussed in Section 3.2.2. An example of this sub−tree combination can be seen in Figure 3.1. In this example, our differences to a normal best−first search are illustrated. In Figure 3.1, the black nodes represent two disjoint promising paths and the white nodes represent the other expansions that had to occur to find those paths. A normal best−first search would only expand the node with the best score by one more attribute. In our modified search, however, we graft the entire other well−performing subtree to the other tree. This allows us to quickly assess deeper paths in the tree.

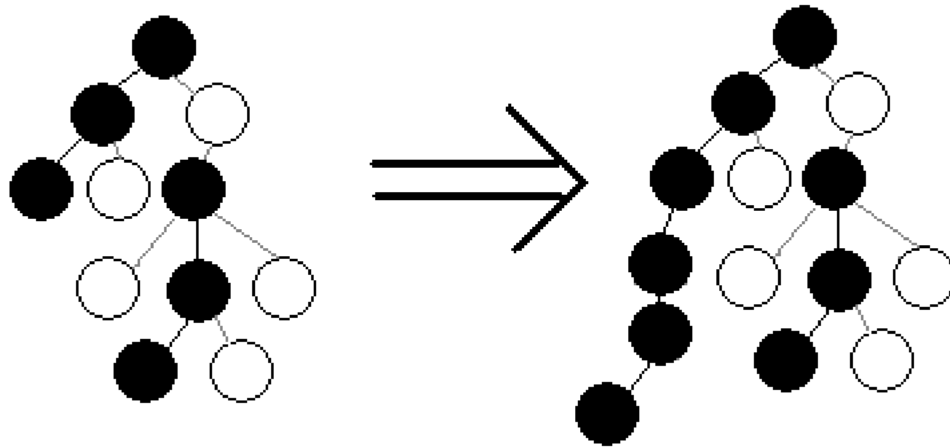

Figure 3.1: Example of the Modified Best−First Search.

3.2 Implementation

The following sections will describe the actual implementation of Which, instead of just the idea behind it. Which does not actually have a collection of best first trees in it. It instead uses a simple list to handle its storage.

3.2.1 Sorted Linked List

Instead of keeping track of the paths that have been expanded by keeping all of the different trees in memory, Which instead uses a linked list that is sorted by the score of each path. In its basic form, all paths that have ever been seen exist in this list and therefore have a chance of being combined to other paths to create new paths. Every time Which creates a new path in the search space, it scores via an arbitrary heuristic method and then inserts it into the linked list at its position. Since this list is sorted, a newly created path will be inserted not at the beginning or the end, but at a meaningful location. This sorting is very important to the probabilistic selection that is described Section 3.2.3.

This method emulates the tree in terms of the selection algorithm and the expansion explained

above in Section 3.1.1. In fact, it could easily be shown that this forest of best first trees could be created from this stack in linear time. The reason a linked list was chosen over the use of the actual tree was its ease of implementation, memory efficiency, and it is faster to traverse a list than it is to traverse a forest. This last point is paramount in the choice because Which will spend most of its time traversing this list and combining different paths from it. This process is the most common process in the entire algorithm.

There is actually a second criteria that is used in determining where a newly created traversal is placed in the stack. The major component of the sorting is the score of the path, however the number of nodes in the path is also considered in the case of a tie. If two paths have an equal score but one of them has less nodes than the other, the smaller path is considered better by Which and will be placed ahead of the longer path. This is due to the nature of the selection described in /sectselection. The reason Which values shorter paths more is because it is a goal of Which to have shorter, more compact rules than longer.

3.2.2 Rule Combination

A very important concept in the implementation of Which is its method of jumping about the search space. This allows Which to attempt to find the solution faster by growing its rules faster. This idea can be explained by considering a best first forest situation where several of the trees have been expanded to a height greater than one and each of these expansions is proving to be a promising path to continue on. This is where the idea of jumping about the search space comes in. Instead of simply picking one path and growing it by one more attribute and scoring all of the new nodes, Which may take two paths that have multiple attributes already expanded in them and combine them. The logic here is that if two long paths are both promising, there is reason to believe that combining these two paths together, instead of slowly adding attribute ranges one at a time, could lead to a promising path in itself.

The implementation of the combination is essentially a union of sets. It is very possible that two

paths in two different trees in the forest have shared attribute ranges, especially if these attribute ranges are very good at predicting the goal. The idea of a union of two sets is that all items in the new set are unique. Also, as mentioned in Section 3.1.1, if two ranges of the same attribute are in this new set, they are simply a disjunction. In other words, it can be envisioned that two paths are being simultaneously traversed in the tree and both are equally promising in finding the goal. If there is ever the case that all possible attribute values are being traversed at the same time, that attribute is removed from the current path entirely. The logic behind this is that ignoring the attribute all together is the same as exploring all of the paths of that attribute.

A point of interest here is to note the method in which Which will traverse the paths that exist in the best first forest. If we consider each rule in the linked list to be an isolated path in the currently explored best first forest, we can make the point that Which does not actually traverse the paths at all. Instead, Which keeps in its list not paths, but random states of different traversals in the best first forest. What this means that instead of traversing paths in a best first manner, it simply jumps from different states in a best first search through the act of combining different promising states it has seen so far. This allows Which to see much further into the future with one combination than a best first search because it does not have to move through the states by adding one new attribute at a time. Also, the linked list construct combined with the probabilistic selection described in Section 3.2.3 allows Which to easily select which new state to jump to next. For ease of explanation, this conjecture will not be used in other description of Which, instead the standard relations of Which to a best first search will be used.

3.2.3 Probabilistic Selection

As stated at the beginning of Section 3.1, Which is a stochastic best first search. This means that it does not exhaustively explore the search space, or to put it in terms of the best first search algorithm, not all paths are explored. Like a best first search, however, Which will rank each of the currently explored paths by some arbitrary heuristic and attempt to further explore the most

promising paths first. The key difference here is that it is not required for Which to only traverse the most promising path by one more attribute value. In fact it is not required for Which to keep traversing on the current most promising path. This is where the probabilistic selection comes in.

Which will choose its next traversal by pseudo-randomly selection two already explored paths and combining them. This pseudo-randomness is a core element to the Which algorithm. Which keeps a sorted linked list to hold its rules because of the probabilistic selection. This selection works by the pseudo-code in Figure 3.2 below.

```
for r in rules
        sum = sum + r.score
        scores[r] = scores[r-1] + r.score
end for
for r in rules
        r.score = r.score/sum
end for
x = random(0,1)
for r in rules
        if x < scores[r] return r
        else x = x    scores[r]
        end if
end for
```

Figure 3.2: Probabilistic Selection Pseudo Code

Figure 3.2 above illustrates the randomness of the algorithm. If the two most promising paths are equal in their score and some order of magnitude better than the third most promising path, it is very likely that they will be picked and combined, however, it is not a guarantee. This allows Which to sometimes jump off of the current search path and attempt to create a new one that could lead to a different optimal solution. Figure 3.3 further illustrates this point. It consists of a lists of items and shows how they can be combined.

In the Figure 3.3 there is a 91% chance that one of the two most promising rules will be picked for combination. This method of probabilistic selection allows for a high chance of paths that are very promising to continue to be traversed while adding some chance that a less promising path can also be further traversed, or a promising path and a less promising path can be combined to determine where that branch would lead.

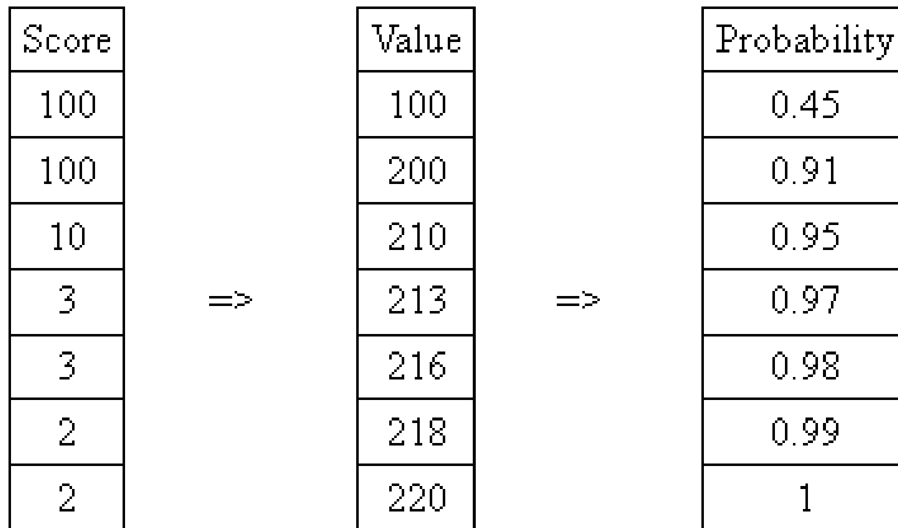

Figure 3.3: Illustration of the Pseudo-Code in Figure 3.2

3.2.4 Scoring Functionality

One of the very important features of Which is that greatly differentiates it from other machine learners is that its scoring method is arbitrary. In all of the experiments described below in Chapter 4, Which is being used as a contrast set learner. However, it is not necessary for Which to be used as such. In all of the algorithms that Which uses to search the space, none of them ever make use of the scoring heuristic. In fact, Which only uses the value of the scoring heuristic to evaluate if the moves that it is making are promising or not. The method of growing and evaluating the paths is determined by a combination of weighted random samples of the space of paths that have been explored so far. Which has no notion of what it is that makes a path promising, just that a path is promising and should be explored further.

This allows Which to be used in applications of search that vary widely. The implementation of this arbitrary scoring is function pointer that is passed into Which before it goes through its initial phase of creating the height one forest. Which can then use the method to score any new paths that get created.

For instance, the ROC n' Rule paper [15] discusses several methods of scoring heuristics and

compares them to show how some sophisticated methods are really much simpler than they seem. For instance, Fürnkranz et al. make the claim that RIPPER's evaluation metric is actually just a complex version of precision. The authors prove this mathematically but with Which it can easily be shown that they are equivalent empirically by running Which several times on the same data using the two scoring methods and comparing the rules that result from it. The actual end result for their proof can be shown in Equation 3.1 below.

$$h_{ripper} = 2 * h_{precision} + 1 \tag{3.1}$$

3.2.5 Stopping Conditions

This concept of Which being a stochastic best first search has been shown in the sections above to be an interesting approach to traversing the search space. However, currently there has been no explanation of how Which stops searching. If Which was an exhaustive search this method of jumping about the space would be unnecessary because Which would eventually have to see all paths before quitting, so enumerating all possible paths would work just as well. To solve this problem, Which has two simple stopping criteria.

The first stopping criteria is a ceiling on the total number of times Which can combine things. The larger this number, the more combinations that can occur. It has been shown empirically that machine learners eventually stop learning new information after a certain point [34]. This upper limit on the number of combinations that Which can do is user defined so it allows this to be tuned to the current application Which is being used for.

The second stopping criteria is a check that happens every so many combinations. While Which does have a maximum number of combinations it is allowed to make, there is a chance that this number may be higher than what is really necessary. The second stopping criteria does

a check to see if the current best path is a certain amount greater than the best path was so many combinations ago. The improvement amount and the number of combinations to allow before another check occurs is completely user defined. This stopping criteria is more of an efficiency check than a true reason to stop the search.

3.3 Finite List

Throughout all of the above descriptions of the list structure used in Which, we have assumed that the size of that list is infinite. That is, every single search path that has been explored so far remains in that list and each of these paths has some likelihood to be selected for combination in the future. It is possible through the parameters given to Which to change this list length so that if so many paths are found to be more promising than a different path, that obsolete path is removed from further consideration entirely. Results of these experiments can be found later in this thesis in Section 4.2.3.

3.4 Advantages

There are several things that give Which an advantage in not only the traversing of the possible search space but also its flexibility in the goal. As stated under Section 3.1, a node in a best first tree can be scored based on the description of the node and the description of the goal. In Which's case, the description of the goal could be very different from application to application. This allows Which to be used in a variety of different ways and allows a user of the application to define a different goal function for their specific application.

Another advantage is the method of search that Which uses. Since Which does not slowly grow a path one attribute value at a time, it has the potential to find an optimum path much faster than a standard best first search approach. Also, the stochastic method in which Which combines its path

allows it to potentially jump to a path that was not the most promising but when combined with a promising path could lead to a promising path.

Chapter 4

Experiments

This chapter will discuss the different performancs experiments that were conducted for this thesis. There are a total of six different experiments that cover 4 different categories of learning that we tested the Which learner against. Category I is a comparison to a learner that has the same overall goal of Which, but the processes used to get the result are different. We compare the abilities of TAR3, discussed in Section 2.3.5, with the abilities of Which on some class UCI [3] data. Category II is one experiment to illustrate the limitations of our approach currently. This experiment goes into detail about the properties of Which's rule creations and the limitations of Which in more−than−two class data sets. Category III is the largest and coveres three seperate experiments. This category compares Which to the classic learners of j48, Ripper, and Naïve Bayes in the field of defect detection in software projects. These three experiments use very diverse data sets that come from completely different areas of software programming. Category IV is one experiment that uses the results of Category III coupled with a sampling technique described in Section 2.2.2, micro−sampling. This experiment uses the data from the first two experiments of Category III and attempts to create Which variants that use micro−sampling and compares them to the Which2 that was created in Category III.

Section 4.1 discusses a novel evaluation metric of software defect data that was created during

the experimental process of this thesis. Section 4.2 is not categorized in any of the categories in the above paragraph, but consists of a few experiments that were done with the Which paremeters to justify our choices in the other experiments. Section 4.3 belongs to Category I. Section 4.4 belongs to Category II. Section 4.6, Section 4.7, and Section 4.8 all belong to Category III. Section 4.9 is the final experiment and it belongs to Category IV.

4.1 Evaluation Metric: "The Koru Diagram"

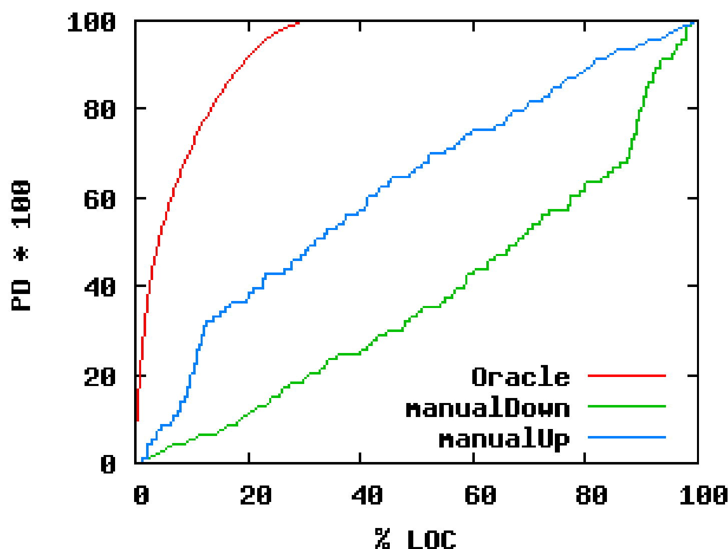

Figure 4.1: Illustration of Main Components of Koru Diagram.

Figure 4.1 is an example of the basic stucture of the Koru Diagram. The Koru Diagram is a method of evaluating the performance of different detectors generated by machine learners or other means, some of which are discussed below in Section 4.1.1 [24]. The three lines in Figure 4.1 correspond to the three detectors described in Section 4.1.1. The way a detector is plotted using this diagram is the key to understanding its evaulation. The x−axis in the diagram is a normalized version of effort. Effort here can explained as the current total number of lines of code explored by this detector over the total number of lines of code in the entire project. This will be better

described in Section 4.5.1. The y−axis is the probability of detection of the detector being plotted. Koru et al. came up with the idea of these diagrams [24]. They stated that an effective measure of the performance of a detector would be to take all of the instances that the detector classifies as defective and sort them in ascending order based on their line of code measure. A plot of PD vs. Effort could then be created that shows the growth of the effort with relation to the grown of PD.

The curves of each detector on the Koru Diagram are a measure of how much effort is gained as defective modules are identified. There are a few things one can gain from this graph that are not explicitly defined in the plots. For instance, if a curve in the Koru Diagram has a lot of plateaus in which effort is growing but the probability of detection is not, one can see infer that the detector is reporting non−faulty modules as faulty. Another implicit observation that can be made will be explained in Section 4.1.1. A third observation that can easily be infered from the diagram and is illustrated in Section 4.5 is that most detectors do not find all defective modules. This is due to the fact that machine learners are not perfect and have to insist on some flaws to avoid over-fitting. The way this is reflected in evaluation of the detector can be found in Section 4.1.2. A final observation of the Koru Diagram are the points 0,100 and 100,100. No detector will ever pass through or stop at the point 0,100. This is because that point implies that all of the defective modules are found and no code has been searched. This is an impossibility because a module must have some source code associated with it. The position 100,100 is special because any learner that finishes at this point will have looked at every single module in the project. The reason the manual and launam curves on Figure 4.1 end there is explained in Section 4.1.1 and Section 4.1.1.

In order to generate the information to plot a curve on the Koru Diagram, one must take all of the modules that the detector classifies as defective and sort them based on their lines of code in ascending order. All curves start at point 0,0 on the graph. From there the effort and pd points are plotted such that for each defective module in the list, if that module is actually defective, increment the probability of detection by the fraction that one module is worth. For every module that was classified as defective, increment the effort by the portion of the effort that module adds

to the current total. So a point on the Koru Diagram can be interpreted as the amount of effort this detector has incited to detect the number of modules it has detected. A curve on the Koru Diagram is the path through the space the detector takes to reach its goal.

The reason the modules classified as defective are sorted in ascending order is because faults are distributed disproportionately in smaller modules [13]. This obversation can be used as a method for an expert using the detector for their system to find a reasonable stopping point. Essentially, if the distribution of the faults in a system follow the Pareto Distribtion described in Section 2.7 as was observed in [20, 23, 36, 44, 46] and repeated in the data sets that were used in Section 4.5, than sorting the modules classified as defective in ascending order by the lines of code should produce a curve most similiar to the oracle curve illustrated in Figure 4.1. If the opposite is true, that is that the largest modules contain the most faults and smaller modules have a far lower fault−density, than the oracle curve will have a shape similiar to the launam. The curve will still stop at the same location in the diagram, but the shape will grow very slowly at first and then grow very quickly towards the final position. This is the exact opposite of what is being illustrated in Figure 4.1.

4.1.1 Special Detectors

Oracle

The oracle curve described in this section can be seen illustrated in Figure 4.1. The oracle is the perfect detector. It is a purly hypothetic creation and exists to illustrate the best possible detector that could be generated by a learner. The oracle has one rule for detecting a defective module and that rule is illustrated with Figure 4.2. The main purpose of the oracle is to show the minimum

```
if [ defect = true] then
        report defective module
endif
```

Figure 4.2: Oracle Detection Rule

possible effort required to detect all of the defective modules in a project. As was observed in most of the project data that was used in the experiments of Section 4.5, the oracle grows very rapidly at first and then slows down after most of the defects have been detected. There will be no horizontal line in the oracle's curve because the detector that generated it has a probability of false alarm of zero. The reason for the slower growth towards the end is because the last remaining defects are located in very large modules with respect to the median of the module size. Statistics taken about some of the projects that were used in Section 4.5 can be found in Section 4.5.1.

Manual−Up

The manual−up detector in this section can be seen illustrated as the line titled manual in Figure 4.1. A manual−up detector is a detector that does not use a machine learner to generate a rule for each individual modules. Instead, manual−up works by sorting all of the modules by lines of code in descending order and exploring every one of the modules in that order. This, along with the manual−down detector described in Section 4.1.1 are meant to be baselines of the Koru Diagram. If a detector cannot achieve a higher area than one of these two methods, it is not worth using. An intersting thing to note about the manual−up detector is that in most of projects used in the experiments of Section 4.5, it loses out to manual−down. This is due to the distribution of faults in the modules that is described in Section 2.7 and Section 4.5.1. One article talks about the manual inspection method and its limits [20]. This article discusses a type of inspection that is very similiar to our manual up inspection.

Manua−Down

The manual−down detector in this section can be seen illustrated as the line titled launam in Figure 4.1. A manual−down detector works very similar to a manual−up detector with one slight difference that makes a lot of difference in how well it performs. The modules in the manual−down detector are sorted in ascending order, like all of the other detectors in the experiments. The causes

the growth to of the curve to be more similar to the growth of the oracle.

4.1.2 Area Under the Curve

The area under the curve for any curve on the Koru Diagram described in Section 4.1 is a metric of evaulation for a given defect detector. As was stated in a previous section, not all detectors will end at the point of 100,100. In fact, a detector ending here is an indicater of a poor detector. This is because of the reasons described in Section 4.1. The maximum area for any possible detector is the area of the oracle described in Section 4.1.1. Upon inspection of the Koru Diagram, it is easy to see that the oracle curve has very little area under it if it is taken to end at the point it actually does end. To ensure that the oracle does in fact have the most area, the area under the line from the point the oracle ends, a point that will always be x,100, to 100,100 is added to the area of the oracle. The same goes for all curves in this space. If a curve ends at x,y, than the area under the line that begins at x,y and ends at 100,y is added to the total area of the detector. If a detector does not detect all defective modules in the project, than this will be reflected in the area metric. The reason to use a straight line is because this assumes the worst possible case of how a detector will work in the space it does not fire on. That is it will miss all defective modules that exist in this space. In order to effectively rank the areas under each detector's curve in this space, each area is divided by the oracle. This implies that the perfect detector will have a score of one.

4.2 Experiments with Which's Parameters

Which has some configurable parameters that it uses during its run that are mostly meant to change the run time of the application and not the way the application reports and evaluates its rules. These parameters can vary widely depending on the data Which is meant to explore. The following sections will describe different parameters and how Which uses these parameters to potentially change its run time. These parameters were discusses under the implemention section of Section 3.2.5 ear-

lier in Chapter 3.

4.2.1 Changing the Maximium Selection Count

One of the main controls that determine the running time of Which and the potential quality of the rules that result from Which is the maximum number of combinations that can be made before Which completes execution. Which in its current state is an offline machine learner. That is, it takes as input all of the data that it will be using to create a final rule. Since a result is required in some reasonable time frame, a simple solution is to only allow Which to combine so many times before stopping and reporting the rule. This value should be set high enough to allow a reasonable search of the search space. Section 4.2.2 provides a solution with some experimental results in its favor that show that there is no real reason to worry about finding a value high enough to reasonably search the space yet low enough to not require too much time.

4.2.2 Changing the Check Parameters

Which employs a method of allowing to remove the burden of finding the optimal size for the maximum selection count described in Section 4.2.1 by allowing two parameters to be configured for every run of the application. These parameters are known as "check every" and "improvement." Section 4.2.2 and Section 4.2.2 below further describe what these two parameters do and how they can be used to gain a faster running time without sacrificing any of the quality of the rules.

Check Every

Check every is a parameter of the application that allows the user to avoid a common ceiling effect that is shown in Figure 4.3. This effect can be attributed to what Menzies et. al. said in [34] about the ceiling effects of machine learners on static data. This parameter is used to control how often Which checks for a potential ceiling effect occuring in its list. A ceiling effect can be defined as the

combonations that Which is creating to search the solution space either never having a better score or always having very similar scores. This results in Which never finding a better rule regardless of the number of combonations it makes. This ceiling effect can occur very rapidly and could result in a lot of the application's run time being wasted due to Which never finding a better rule than one just before the ceiling happened. A method of turning off this early stopping condition is to set the check every parameter to something that is greater than or equal to the number of picks Which will do.

Improvement

Improvement is a parameter of the application that allows the user to set how flat a ceiling has to be for it to be a ceiling. In other words improvement is what allows Which to determine if two rules are in fact the same in terms of their scores. This number ranges from 0 to 1 and stands for how much better the most recently generated rule is than one generated some time ago. If this number is close to 0, Which is much stricter in what it considers flat. In fact, a value of 0 for improvement will disable this early stopping condition all together. Improvement is tied to check every in that the improvement of the best rule in the stack is queried every check every times a new rule is created via combination.

4.2.3 Changing the List Size

Motivation for Experiments

As stated in Section 3.3 in Chapter 3, the basic Which algorithm uses an infinite list. What this means is that every rule, or detector, that was ever generated exists in the sorted linked list. This poses two problems. The first problem is that if the number of picks is to be very large, the number of rules generated will be very large. This could cause the placement of each new rule to be a severe bottleneck in some situations.

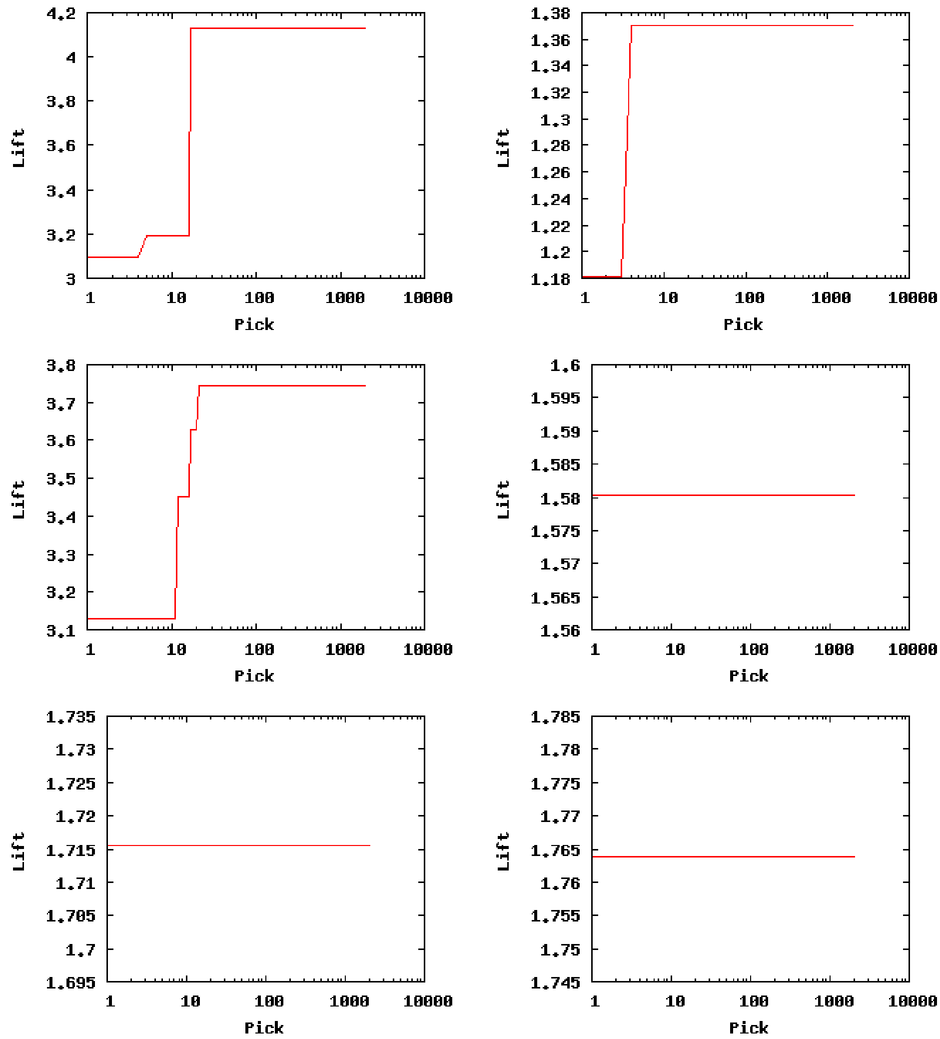

Figure 4.3: Graphs Displaying the Early Maxima Phenomenon

The second problem is in the random probability of the selection algorithm. The algorithm was meant to be so that a low scoring rule would have less of a chance of being selected compared to a high scoring rule. This uses the assumption that the combonation of high scoring rules will lead to a desired optimal solution faster than combining any rules at random at every combonation. This does, however, also still give a chance that a low scoring rule will be selected in combonation. This concept works under the assumption that a low scoring rule in conjunction with another, probably high scoring rule, will cause a desirable result that could be overlooked. The problem with having

the list contain all possible rules to select from is that the chances that any one of a number of low scoring rules being selected is higher after several combonations have already been done. This leads to Which possibly missing some better combonations because it becomes bogged down with selecting bad combonations towards the end of the run.

A solution to both problems would be to limit the size of the sorted linked list that Which uses to keep track of all of the different combonations generated so far. The idea would be to only keep the top N best scoring rules. This would make it so a worst case analysis of the Which algorithm could be calculated. Depending on the list size and number of overall combonations to be made, this could greatly enhance the run time of Which, without losing the performance in terms of proper rule generation. The last statement would only hold true if our assumption about combonations of two high scoring rules will lead to better scores than combonations of low scoring rules with either high or low scoring rules. If it is the case that this does not hold, than we should see a significant performance decrease in the Which algorithm. Section 4.2.3 explains the details of the experiments ran on this topic.

Experiments

This experiment was done using six different sorted list sizes, descriptions of which can be seen in Table 4.1. The method of training and testing on the data is explained in Section 2.5. The type of evalation that was used for this experiment was the Koru Diagram described in Section 4.1. In this experiment, only Which was used as compared to itself in the native form of having an infinite list size. This is because this experiment was only meant to judge the performance, in terms of rule score, change that may or not occur due to changing the infinite list to a finite one. In Section 4.2.3 below the results of this experiment are shown.

Learner	Description
oracle	The perfect detector as described in Section 4.1.1.
which2	The standard infinite list version of Which.
which10	Which running with a maximum list size of 10.
which20	Which running with a maximum list size of 20.
which50	Which running with a maximum list size of 50.
which100	Which running with a maximum list size of 100.
which1000	Which running with a maximum list size of 1000.

Table 4.1: Description of Different Learners Used in Experiment of Section 4.2.3.

Results

Figure 4.4 shows the results of some of the different finite list experiments. These plots represent four different data sets that the list was changed to be finite on. The oracle is changing in these plots because the data being plotted is from different data sets. These results show us that the infinite stack might actually not be necessary. In most of these plots, it can be seen that the growth of Which is almost identical to that of the growth of Which without the infinite stack. The effects of forcing Which to have a finite stack on the run time could be considerable in data sets where the attributes have a wide range of possible values. In these types of data sets the time it takes to find the position of new rule in the data set could be significatly altered if the stack were constrained to a small size independant of the number of attribute values.

It cannot be seen in these graphs, but the standard infinite stack version of Which, Which2, always runs exactly the same as Which1000. This can be explained by the fact that Which uses the parameters of CheckEvery and Improvement to shorten its run time. This causes the stack to naturally never really exceed 1000. While more experiments would need to be ran on the effects of making the stack finite, from these tests, it can be seen that the stack size being limited to a size of 10 to 100 actually improves the performance of Which in some cases and does not really hurt the performance in the other cases. We would recommend a stack size of 100 for most cases. However, in situations where the number of possible attribute values is very high, this number may too low.

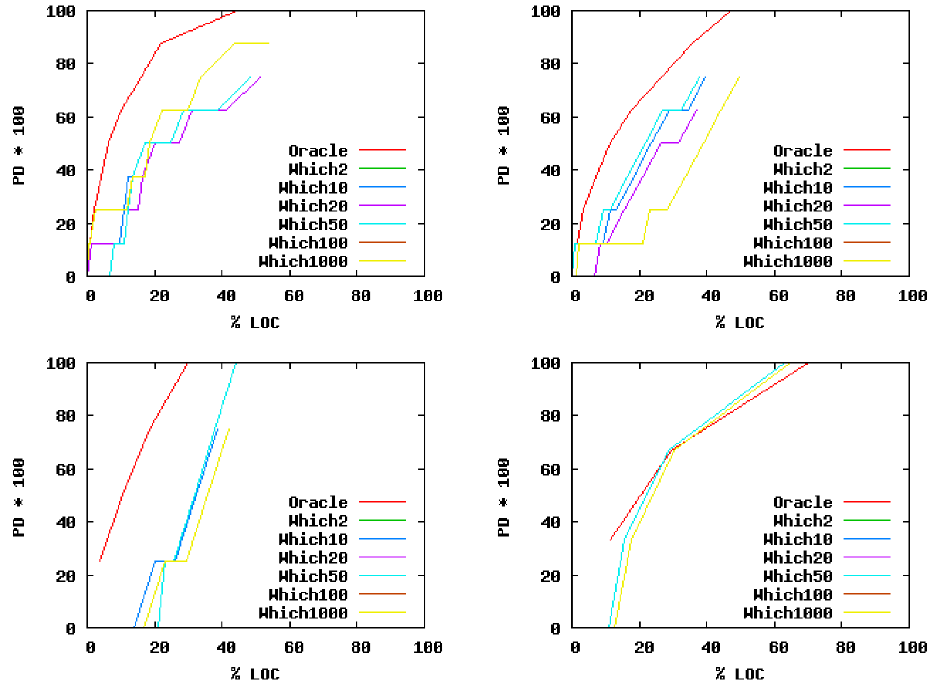

Figure 4.4: Graphs Displaying the Effects of Changing the Sorted List Maximum Size. Which2 is always under Which1000 in these graphs.

It is important to consider the data that Which is being tuned to when making these decisions.

The size of the stack contributes to the run−time of Which in two specific ways. First, when attempting to place a newly created rule in the stack it must compare the scores of each rule to the new rule until it finds the correct position. In the worst case, over all C rules that are created, this process takes $\sum_{i=1}^{C} i = \frac{C(C+1)}{2}$ comparisons. If the stack size is limited as it is shown here, this number of comparisons is at worst the maximum size of the stack every time a new rule is created. The second contribution to the run−time is when two rules are being selected from the stack. As was explained in Section 3.2.3, Which will randomly select a rule based on its score. If the stack size is very large, this process can hinder the run−time because every rule that is created requires two searches through the stack. Table 4.2 shows the differences in run−times for three different data sets of varying instance sizes.

Stack Size	**Data**	**Instances**	**Time**(**seconds**)
∞	ar3	63	1.1
50	ar3	63	0.9
1000	ar3	63	0.6
∞	pc1	1109	9.2
50	pc1	1109	9.8
1000	pc1	1109	9.4
∞	mc2mod	403	3.5
50	mc2mod	403	3.3
1000	mc2mod	403	3.8

Table 4.2: Comparison of Run−Times for Different Stack Sizes.

4.3 Category I: TAR3

As was discussed in Chapter 3, Which creates a rule that best decides one class in a data set. This was shown to be the same type of rule generation that TAR3 uses as well. This experiment was designed to show if Which's simpler rule growing process is just as effective, or more effective than TAR3's more complicated rule growing process.

4.3.1 Data

Name	# Classes	"Best Class" %	# Instances
colic	2	36.9565	726
kr-vs-kp	2	47.7337	3351
breast−cancer	2	29.4118	394
vowel	11	9.09091	1201
anneal	6	4.43951	1058
autos	7	12.9808	336
audiology	24	0.884956	298
vote	2	38.3562	651
primary−tumor	22	7.07965	596
segment	7	14.2857	2406
weather	2	35.7143	23
weather.nominal	2	35.7143	23
heart−h	2	63.95	294
heart−c	2	54.4554	303

Table 4.3: Description of the UCI [3] data sets used in the experiment under Section 4.3.

Table 4.3 descibes the different UCI [3] data sets that were chosen for this experiment. The data selected here was chosen for this experiment because it covers a wide array of class counts and class distributions.

4.3.2 Experiments

As was discussed in Section 2.3.5, TAR3 is a lift learner that will attempt to create a collection of rules that all attempt to predict the same class. For the purposes of this experiment, the ordering of the classes was chosen arbitrarily. Essentially, the order the classes were listed in the header of the file was the order of importance.

It was discussed in Chapter 3 that the evaluation heuristic is completely independant of the rule growing processes in Which. As such, we chose to use the same evaluation heuristic for Which tha TAR3 uses, lift. The equation for lift is seen in Equation 2.12.

We decided that in order to compare the learners fairly, we would compare them using different discretization preprocessors. We chose to use equal frequency discretization with two bins, four bins, eight bins, sixteen bins, and thirty−two bins. Part of our reasoning to go with the two higher bin numbers was to see the possibility of Which naturally combining fine−grained bins into larger bins. There were no conclusive results in that area, however.

[21] was the original thesis that created TAR3. In that thesis, the author never discusses the performance of TAR3 under cross−validation. As a result, part of this experiment was also to test how lift learning performs under cross−validation. Cross−validation is discussed in detail in Section 2.5. We decided to use the standard $10x10$ cross−validation for our experiment.

We report our results in the form of *win−loss−tie tables* described in Section 2.6.1 from the perspective of Which. That is, whenever Which produces a rule that is better than TAR3's on the test data, Which receives a win. In the opposite Which receives a loss.

4.3.3 Results

We report the results of the 12 different data sets over the different discretization methods in Figure 4.5. Figure 4.6 has the overall totals for each of the discretization policies.

Figure 4.5 shows the results of the experiments using the different bins in equal frequency

File	Win	Loss	Tie	Sum
colic	9	0	1	9
kr-vs-kp	9	1	0	8
breast-can	8	0	2	8
vowel	9	1	0	8
anneal	7	0	3	7
autos	3	2	5	1
audiology	0	0	10	0
vote	0	0	10	0
primary-tu	2	2	6	0
segment	0	0	10	0
weather	0	0	10	0
weather.no	0	1	9	-1
heart-h	1	4	5	-3
heart-c	1	6	3	-5
Totals	49	17	73	42

File	Win	Loss	Tie	Sum
colic	7	0	3	7
breast-can	7	1	2	6
kr-vs-kp	7	1	2	6
vowel	5	4	1	1
audiology	0	0	10	0
vote	0	0	10	0
segment	0	0	10	0
anneal	3	3	4	0
weather	0	0	10	0
autos	2	3	5	-1
heart-c	1	2	7	-1
weather.no	0	1	9	-1
heart-h	1	4	5	-3
primary-tu	1	5	4	-4
Totals	34	24	82	10

File	Win	Loss	Tie	Sum
kr-vs-kp	10	0	0	10
breast-can	7	0	3	7
vowel	7	2	1	5
colic	4	1	5	3
segment	3	0	7	3
vote	2	1	7	1
heart-h	2	1	7	1
audiology	0	0	10	0
anneal	3	3	4	0
primary-tu	2	3	5	-1
autos	0	1	9	-1
weather	0	1	9	-1
weather.no	0	1	9	-1
heart-c	2	5	3	-3
Totals	42	19	79	23

File	Win	Loss	Tie	Sum
kr-vs-kp	10	0	0	10
colic	9	0	1	9
breast-ca	8	0	2	8
segment	7	0	3	7
anneal	3	0	7	3
heart-c	0	0	10	0
vowel	0	0	10	0
weather	0	0	10	0
primary-t	2	3	5	-1
audiology	0	1	9	-1
weather.n	0	1	9	-1
heart-h	0	2	8	-2
autos	0	2	8	-2
vote	0	2	8	-2
Totals	39	11	90	28

File	Win	Loss	Tie	Sum
colic	10	0	0	10
kr-vs-kp	10	0	0	10
anneal	8	0	2	8
breast-ca	7	1	2	6
segment	5	0	5	5
vote	1	0	9	1
audiology	0	0	10	0
vowel	0	0	10	0
autos	1	2	7	-1
heart-c	0	1	9	-1
weather	0	1	9	-1
primary-t	1	4	5	-3
Totals	43	9	68	34

Figure 4.5: Sum of Differences Results for 2bins, 4bins, 8bins, 16bins, and 32bins comparison of Which and TAR3.

discretizations. In these experiments it appears that Which performed the best under a very coarse−grained discretization policy. That is it performed better when less bins were present

for each attribute. This could be caused by the overall search space being smaller. A data file with attributes ranging in only two or four values takes far less time to search exaustively than one with attributes ranging with thriry−two values. This means that the random nature of Which has a greater chance to not find an optimum solution.

It is important to note that Which and TAR3 tied far more than Which won and still more than when they tied. This means that the simpler approach to rule construction that Which has is still effective in terms of lift learning. This goes with the idea that simpler learners perform just as well as more complicated ones that Dr. Menzies et al. say in [30].

Figure 4.6 provides the total results from the five differenent discretization policies in this experiment. While Which has nearly two times as many wins at it does losses, the majority of time Which tied with TAR3. In $\frac{207}{679} \approx 30.5\%$ of the tests ran, Which created a better rule that TAR3, in $\frac{80}{679} \approx 11.8\%$ of the test ran, Which created a poorer rule than TAR3, and in $\frac{392}{679} \approx 57.7\%$ of the test ran, Which and TAR3 created a rule with the same score. These overall results do show that while Which does not create better rules over TAR3 all the time, 88.2% of the rules it creates are the same or better than TAR3. This agrees with [30] perfectly in that simpler learners can and do perform very well when compared to more complicated ones.

Bins	**Win**	**Loss**	**Tie**	**Sum**
2	84	32	84	52
4	74	33	93	41
8	55	24	81	31
16	39	11	90	28
32	43	12	85	31
Totals	207	80	392	137

Figure 4.6: Overall Sum of Differences for Which vs TAR3.

4.4 Category II: Multi Class Classification

The purpose of this experiment was to show how Which performed in a situation where more than two classes were present in the data sets. The data sets used in this experiment come from the UCI [3] data sets listed in Table 4.3. The data sets that were selected were selected solely on the criteria that they had more than two classes. This was because a data set that has only two classes can produce a high accuracy by interpreting the rule that Which generates as having an added else clause. For instance, Which may generate a rule like Figure 4.7. In the case of only having two classes, this could be understood as, "if that rule, then class X, else class Y." However, in the case of having more than two classes, the rule could only be interpreted as, "if that rule, then class X, else no answer." This leads to extreme problems in the prediction quality of the rule. As described in Chapter 3, Which produces rules that best predict the "best class" in a data set. Best class is meaningly when the goal of test is to create something that will predict the class that new instances in a data set fall into. However, in the case of most of these UCI data sets, there is no best class, that is the classes are all equally valued. So selecting one of these classes as the best is completely artificial.

```
if [ outlook = overcast ]
AND temp = [ high OR low ]
```

Figure 4.7: Oracle Detection Rule

This experiment was created to show an area that the current version of Which fails in. It was meant to illustrate a counter to the experiments of Section 4.6 and Section 4.7 discussed in this chapter. Essentially, Which fits into a niche portion of the data mining world. A possible solution for this problem in Which is discussed in Chapter 5.3.

4.4.1 Which's Heuristic

A common metric used for evaluating a decision making process that is created by a machine learner is *accuracy*. The equation for accuracy can be seen in Equation 4.1. In this equation, TP_i stands for the number of true positives of the ith class. This is the number of the ith class that was accuratly classified as the ith class. The n in the summation is the number of class. Finally, the $class_i$ in the denomenator of the equation is the sum of all the classes in this data set. Accuracy effectively depicts the total number of classes that were classified correctly.

$$h_{accuracy} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} class_i} \tag{4.1}$$

There are some problems associated with accuracy that are described in [33]. For example, if there is a data file that has one class that is very infrequent in the data set and a learner never correctly classifies that class, the accuracy may still be very high. The following is an example of a two class data file that contains 100 instances. One of the classes is 99 of the instances whereas the other is only one instance in the set. In this two class example, one class consists of 99% of the total distribution. If a machine learner produces the rule that classifies everything as class X, it will have an accuracy of 99%. The major problem of this is if the class that represents 1% of the population is the class we are trying to predict for, that information is entirely lost even though the detector is 99% accurate. This type of detection, isolating one class and developing a rule for that class, is called lift learning and described in Section 4.3. This is also the type of learning that Which uses in its algorithm, as said in Chapter 3, only one rule is produced and that is the rule that best predicts a "best class." Which should and does fail in this accuracy over a multiple class data set test.

key	ties	win	loss	win-loss-at-99%
nBayes	2	1	0	1
jRip	2	1	0	1
j48	2	1	0	1
which	0	0	3	-3

Table 4.4: Overal MannWhitney U−Test on Selected UCI Data Sets.

4.4.2 Design of Experiments

In order to do this experiment, only a few of the data sets from Table 4.3 were chosen. This data sets all had one thing in common and that was that there were more than two classes in the data set.

Once the data was chosen a discretization method known as FayyadIrani [12], discussed in Chapter 2, that was implemented in the wEka [17] was used as a preprocessor to allow each of the learners to be on the same playing field. What is meant by this is that each of the learners have their on discretization methods in them. Using FayyadIrani as a preprocessor was a way of ensuring that each of the learners were actually learning from the same data.

After discretization was complete, the data was broken in into three train and three test sets with a process that is described in Section 2.5. The learners where ran using train n and test n and the results were complied into MannWhitney U−Tests. There was one test generated for each of the individual files and an overall test on all of the scores.

4.4.3 Results

Table 4.4 contains the overall MannWhitney U test results for these learners on the data sets. It can be seen from these results that j48, Naïve Bayes , and ripper all perfrom the same on all of the data, they all tie with each other. It can be shown also that Which is completely outperformed in this environment as well. Figure 4.8 illustrates further the results of this experiment. Quartile charts are discussed in detail and used to report results in [30]. Quartile charts have certain properties

about them that make them valuable for reporting results of cross validation experiments. First of all, so called standout results are clearly visable in these charts. Looking at the results in the MannWhitney U test shows that Which did lose to all three other learners. However, the magnitude of how poorly it performed compared to the other learners is not captured in this table. Figure 4.8 illustrates this very well in that the maximum accuracy score for Which over all the tests is 57.1. Another visual cue to the performance using the quartile chart is the median value, the black circle in the figure. It can be seen clearly in this chart that Which's median is zero, since no circle exists on the chart. The other thing about the medians that can be gained from looking at this is that the other three learners have similar medians, and also that all of these medians are higher than the maximum performance of Which. Another propertiy of the quartile chart is that it makes no parametric assumption about the underlying distribution of the data like t−tests [40]. Further explaination of the u−test, t−test, and quartile charts can be found in Chapter 2.

Learner	**Mean**	**Equal**	**Description**
nBayes	82.9		
j48	79.7	=	
jRip	79.7	=	
which	0.0	<	

Figure 4.8: Quartile Chart Illustrating the Results of Table 4.4.

These results were expected before the experiment and, in actuality, the experiment was designed to illustrate this point of the Which design. The reason Which does so poorly with a multi class data file is because by design, it is unable to generate rules for each of the classes. It can only generate rules for one class. While it may be possible to create a tool around Which that runs the Which application once for each class in the data file, that is beyond the scope of this thesis. Further improvements to Which can be found in Section 5.3.

4.5 Defect Detection

One of the most impressive applications of the Which learner is that of defect detection in large software projects. Previous studies have shown that Naïve Bayes , described in Section 2.3.3. has been superior at detection compared to the other classical learners [30]. As was stated previously, there were three different experiments on the performance of the machine learners using defect detection data. These experiments all used different data Each collection of data is called a family in these experiments. The MDP family contains all of the data sets used in Section 4.6, the Turkey family contains all of the data used in Section 4.7, and the AT&T family contains all of the data used in Section 4.8.

4.5.1 Data

Basic Statistics Taken About Data

Data	cm1	kc1	pc1	ar3	ar5
Family	mdp	mdp	mdp	Turkey	Turkey
Modules	498	2109	1109	63	36
Median	17	9	13	42.5	42
Mean	29.6448	20.3723	23.3761	89.2698	75.889
σ	42.7106	29.7474	35.2681	115.802	88.8691
Min	1	1	0	3	5
Max	423	288	602	670	477
Defect Dist	9.8%	15.5%	6.9%	12.7%	22.2%

Table 4.5: Statistics Taken About LOC From a Few of the Data Sets Used.

The data represented in Table 4.5 was taken from five of the twelve data sets that were used in the defect detection tests. The data sets used to create two of the three of the defect detection experiments were taken from the PROMISE website [29]. Cm1, kc1, and pc1 are part of the mdp family and come from NASA software projects. The Turkey data comes from a software company that creates control software for household appliances. The third experimental data comes from AT&T . This data was used under a non disclosure agreement and not much is allowed to be said

about it. The results of the experiment and a discription of some of the attributes used. This table is meant to give a brief overview of some of the statistics on the the project files that were used. Refer to Section 4.6 and Section 4.7 for more information about the files and types of projects these data files were created from.

4.5.2 Halstead and McCabe

Metric Type	**Metric**	**Definition**
McCabe	v(G)	Cyclomatic Complexity
	ev(G)	Essential Complexity
	iv(G)	Design Complexity
	LOC	Lines of Code
Derived Halstead	N	Length
	V	Volume
	L	Level
	D	Difficulty
	I	Intelligent Count
	E	Effort
	B	Effort Estimate
	T	Programming Time
Line Count	LOCode	Lines of Code
	LOComment	Lines of Comment
	LOBlank	Lines of Blank
	LOCCodeAnd Comment	Lines of Code and Comment
Basic Halstead	UniqOp	Unique Operators
	UniqOpnd	Unique Operands
	TotalOp	Total Operators
	TotalOpnd	Total Operands
Branch	BranchCount	Total Branch Count

Table 4.6: Halstead and McCabe Descriptions of MDP Data Sets.

[18] contains a table that describes all of the different attributes used in both the MDP and the Turkey data sets. Table 4.6 is the table that was used in that article. All of these statistics are taken on a per module basis. In this instance, a module is the simplest whole unit in the language. In C

this would be a function and in Java this could be considered a method of a class. This is not to be confused with the broader term of module that is discussed in [2]. In that article, it is said that the file is far more fine grained than a module. Their definition of module is something similiar to a collection of Java classes that are a stronly cohesive, strongly coupled system. Here, we use the term module to be something that is more fine grained than a file.

There are three types of statistics that are taken about each module, these are the McCabe [18, 28, 32], Halstead [18, 19, 32], and line of code readings.

McCabe's idea was that it was the complexity in the pathways of the source code that would lead to more faults. The three attributes that are associated with his idea can be seen in Table 4.6. Cyclomatic complexity can be described as the number of linearly independant paths that can be taken through source code. A simple explanation of this would be a simple module that contained one if statement, or conditional. This would have a cyclomatic complexity measure of 2. Essential complexity refers to the cyclomatic complexity of a module after all well structured paths have been reduced to statements. For instance, unrolling a for or while loop into its statements. Design complexity of a module is the cyclomatic complexity of its simplified form.

Halstead's idea was that if the code was hard to read, it would have more faults. Halstead took measures about the code itself, instead of the flow of the program. These attributes can be seen in Table 4.6.

The final measurements taken about the modules are basic line of code readings. These give specific line count readings about the code and its executable.

4.5.3 Design of Experiments

Logging the Attributes

A plot of the attributes in a few of the data files reveals that they fall in an exponential distribution. This has been stated before in [30]. Three of these plots can be seen in Figure 4.9. This type of distribution is difficult to learn from, especially when using an equal frequency discretization policy like Which uses in these experiments. What an exponential distribution causes is that, in a 10 bins example, most of the bins actually contain the same range of values, since these values

are the most plentiful. It is only the last one or two bins that contain any meaningful information. Because of this, the machine learner will always pick the last bin. This type of selection leads to several false alarms. To prevent this behavior, we took the natural log of each of the attributes in the data file. This gives a linear slope and allows for the equal frequency binning to better represent the values that are contained in the bins. As was discussed in Section 2.2.1, the result of equal frequency discretization on an exponentially distributed attribute is that the majority of the lower bins contained values in the range $[0,1)$ and several of the initial bins contained the same range of values. Which and the classic learners would always create detectors that selected the higher bins, which typically contained values that had a range that was ≥ 100. Taking the natural log of these attributes provided much more similiar ranges. The equation that was used to handle the transformation of the attributes was $newval = ln(max(oldval, 0.0001))$. This was to avoid the cases where the attribute value was zero.

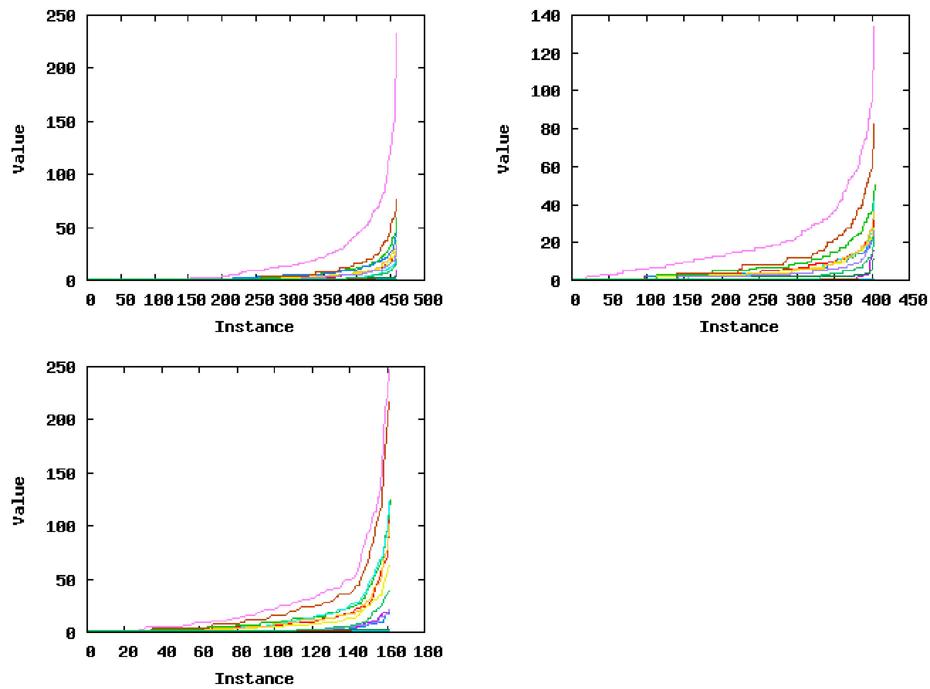

Figure 4.9: Some Example Attribute Distributions for kc3mod, mw1mod, and mc2mod. Here, we sorted the values of the attributes in each data set in ascending order. What we are measuring is the distribution of the values over the number of instances in the set.

4.5.4 Which's Heuristic

As discussed in Section 3.2.4 earlier in this writing, Which allows any scoring heuristic to determine how a generated rule scores. Several different scoring heuristics were tried in the tests, but the one discussed in this section is the heuristic that was used for the defect detection experiments.

Balance

Equation 4.2 illustrates the heuristic that Which uses to calculate the score of each rule. For our purposes, the best rule would be one that has a 100% probability of detection and a 0% probability of false alarm, this corresponds to the oracle discussed in Section 4.1.1. Because of this, the balance equation is the Euclidean distance from that point in ROC space. Scalar terms of α and β are added to the equation to allow for different levels of importance for each of the terms. By default, these results are 1 and 1000 respectively. What this does is make a very low pf 3 orders of magnitude more important than the pd term. Having $\alpha = 1$ and $\beta = 1000$ was decided upon after considering the effects of both pd and pf. We wanted to find a way to include effort into the heuristic, but effort in itself cannot be included explictly in the heuristic. This is because or reasons discussed further in this section and also because pd and effort are correlated in a very distinct way. In order to detect a module, some effort must be used. Having the α and β coefficients be set to what they are for experiment is a way of implicitly saying that while effort must be tolerated, minimizing the contribution to effort from false alarms should be a priority.

$$h_{balance} = \frac{\sqrt{pd^2 * \alpha + (1 - pf)^2 * \beta}}{\alpha + \beta} \tag{4.2}$$

As was stated earlier in this section, an equal contributor to how well a detector scores is how much effort is required to detect a certain percent of the defects in a project. This metric was added to the balance equation to make it look like Equation 4.3 and the results for Which were not ideal. As before, there are scalar terms added to the equation to allow for importance of each part to be specified.

$$h_{balancewitheffort} = \frac{\sqrt{pd^2 * \alpha + (1-pf)^2 * \beta + (1-effort)^2 * \gamma}}{\alpha+\beta+\gamma} \tag{4.3}$$

The typical rule that was selected using the heuristic of Equation 4.3 was a middle ground. What this means is that the best rule that Which could report had a pd of 0.5, a pf of 0.5 and an effort 0.5. This is because the way in which effort and pd are correlated. While it is possible for a rule to be at the 0,1 mark in ROC space, it is impossible for a rule to be at the 0,1 mark in Koru space. This is because detecting a module as defective will always add some degree of effort to the rule. This causes the phenomenon of finding the middle ground. Also, even though effort is not evaluated by the rules during the growing phase, pf is given a really high priority, thus inducing a minimal effort. As described in Section 4.1, if a detector has a high pf than the overall effort will be very high whereas if a detector has a very low pf it will more closely follow the oracle curve.

Probability of Detection Divided by Lines of Code

Equation 4.4 illustrates how a rule is scored using this heuristic. It was decided before these experiments took place that since it was easy enough to change the scoring method of Which, that we would also try a different scoring heuristic.

$$h_{\frac{PD}{LOC}} = \frac{PD}{\sum_{i=1}^{n} LOC_i} \tag{4.4}$$

Equation 4.4 was meant to create rules that have a high probability of detection and a low effort total. Instead of actually using effort here, which is a normalized amount of the lines of code totally looked at over the total lines of code, this equation uses the raw line of code measures. In Equation 4.4 PD stands for the probability of detection and the summation in denomenator stands for the number of modules that were labeled as defective by a rule. n is the total number of modules labeled as defective.

$h_{\frac{PD}{LOC}}$ was used because it was slightly different than $h_{balancewitheffort}$ and we had hoped it would yield the kind of results that we wanted. That is, results that have a high probality of detection with a low amount effort required. Another property of this equation is that it could allow for modules to be labeled as erroneous that were not actually erroneous, but if that module was too large in terms of lines of code, the score would be significantly impacted compared to a smaller module. The results of this heuristic are discussed in Section 4.6.2.

4.5.5 Evaluation Criteria

The evaluation criteria for all of the detectors used in these experiments is the evaluation criteria described in Section 4.1.2. Each detector is plotted in Koru space and then its normalized area under the curve is used for the Mann Whitney U−test. With the exception of the AT&T data used in Section 4.8, the other experiments all used a cross validation. The reasons the AT&T data did not are discussed in section discussing the AT&T experiment.

4.5.6 Probability of Detection vs. %Lines of Code

Probability of Detection vs. %Lines of Code is a general name for the Koru diagram. With each of the experiments in Section 4.6, Section 4.7, Section 4.8, and Section 4.9, this is both the method of displaying the results as well as the method of scoring the results. Several of the figures in this thesis use this type of graph.

4.6 Category III: MDP Data

The data taken from the MDP [29] refers to data used in various NASA programs. This infers that this data was taken from a company that is not profit driven and creates software via contractual agreements. This means that the different projects were actually developed by different companies from geological regions spanning the United States. Some of these projects were created from entirely new source code while other projects were created by modifying and reusing another project's code. The individual project files for this data each represent a different project and potentially a different team of developers. While the source code underlying the statistics gathered for these data files is still structured relatively the same use the standard government contract standards of producing code, the projects themselves are still diverse enough to make the claim that they are not the same in terms of programming style. This means that performing well on these project files overall creates some support for the fact that a detector or machine learner generating a detector could have some stability over projects that are diverse and not of the same type. Table 4.7 describes some statistics about the seven data sets, or project files, that were used in the the MDP experiments.

File Name	Language	Defects	Modules
cm1	C++	49	498
kc1	C++	326	2109
kc2	C++	107	522
mw1_mod	C++	31	403
pc1	C++	77	1109
kc3_mod	JAVA	43	458
mc2_mod	C++	52	161

Table 4.7: Information About the NASA data.

4.6.1 Experiment

This experiment was done on seven different MDP data sets using a $10x3$ cross$-$validation on nine different learners plus the oracle, manualUp, and manualDown detectors. The total number of

times this each data file was used is $7x10x3x12 = 2520$. The statistics taken on each of the detectors generated were the probability of detection and the total effort used after all of the modules the detector said were defective were inspected. In order to get the specific data for the Koru Diagram and for the Area Under the Curve calculations, a step by step of each detected defective module was recorded as well. This process was explained in Section 4.1.

Several variants of Which were used in these experments and some appear in the graphs while all appear in the MannWhitney and Quartile Chart reports. They were omitted from the graphs because they all performed poorly and added unnecessary clutter and confusion to them. A discription of these different variants can be seen in Table 4.8. In all of these experiments, Which2 was the majority best performing variant, so only it is plotted in the graphs of Figure 4.10, Figure 4.13, and Figure 4.14.

Variant Name	**Description**
Which2	Which using 2 bins equal frequency with Equation 4.2.
Which4	Which using 4 bins equal frequency with Equation 4.2.
Which8	Which using 8 bins equal frequency with Equation 4.2.
Which2loc	Which using 2 bins equal frequency with Equation 4.4.
Which4loc	Which using 4 bins equal frequency with Equation 4.4.
Which8loc	Which using 8 bins equal frequency with Equation 4.4.

Table 4.8: Descriptions of Which Variants used in Section 4.6.1.

4.6.2 Results

We have divided these results into three seperate classifications to illustrate different points. The three classifications are:

- Class One: Which2 $>$ manualUp $>$ classic learners;
- Class Two: Which2 $>$ everything else; and,
- Class Three: Everything $>$ Which

Each class consists of three figures that correspond to each other. The first figure is a Koru Diagram, explained in Section 4.1 for each of the data sets that are in the category. The second figure represents the quartile charts for the graphs. The charts are in the same layout as the Koru Diagram. The third and final figure is the figure representing the MannWhitney U−Test tables that are generated from the same data the quartile charts are generated from. These, like the quartile charts, are in the same layout as the Koru Diagram. By relating the three figures to each other, it is possible to see each of the different categories.

Sometimes, the quartile chart will report something different than the MannWhitney U−Test table. What we mean by this is that the two charts report their learners in descending order based on how they score. However, it can be seen by inspecting both the MannWhitney and the Quartile Chart together to determine the category each of the results is in.

Class One

The first class is the one where Which2 has the best overall score, followed by manualUp, followed by the rest of the learners. This class can be seen graphically in Figure 4.10. We feel this class is significant because it illustrates two important points. The first point is that Which2 was the overall winner and it can be seen in Figure 4.10. The second point is that the manual detector manualUp, described in Section 4.1.1, performs better than any of the classic learners. This says that the detectors generated by these learners gives us nothing better than what manually checking all of the source code in this manual up way, which is sorting the modules in ascending order by lines of code only.

Figure 4.11 and Figure 4.12 display the graphs in Figure 4.10. Figure 4.11 represents the Quartile Charts that were used in categorizing the the the different effects that were realized. Looking at the quartile charts shows three important facts. The first fact is that very few of the machine learners actually performed well. Of these learners, Which2 had the best performance, followed by manualUp. which is the definition of the category. However, the second point is that Naïve Bayes performed better than the other standard machine learners. This results is consistend with the results in [30]. The third point of interest in these results is that manualDown also performs

better than all of the standard learners besides Naïve Bayes .

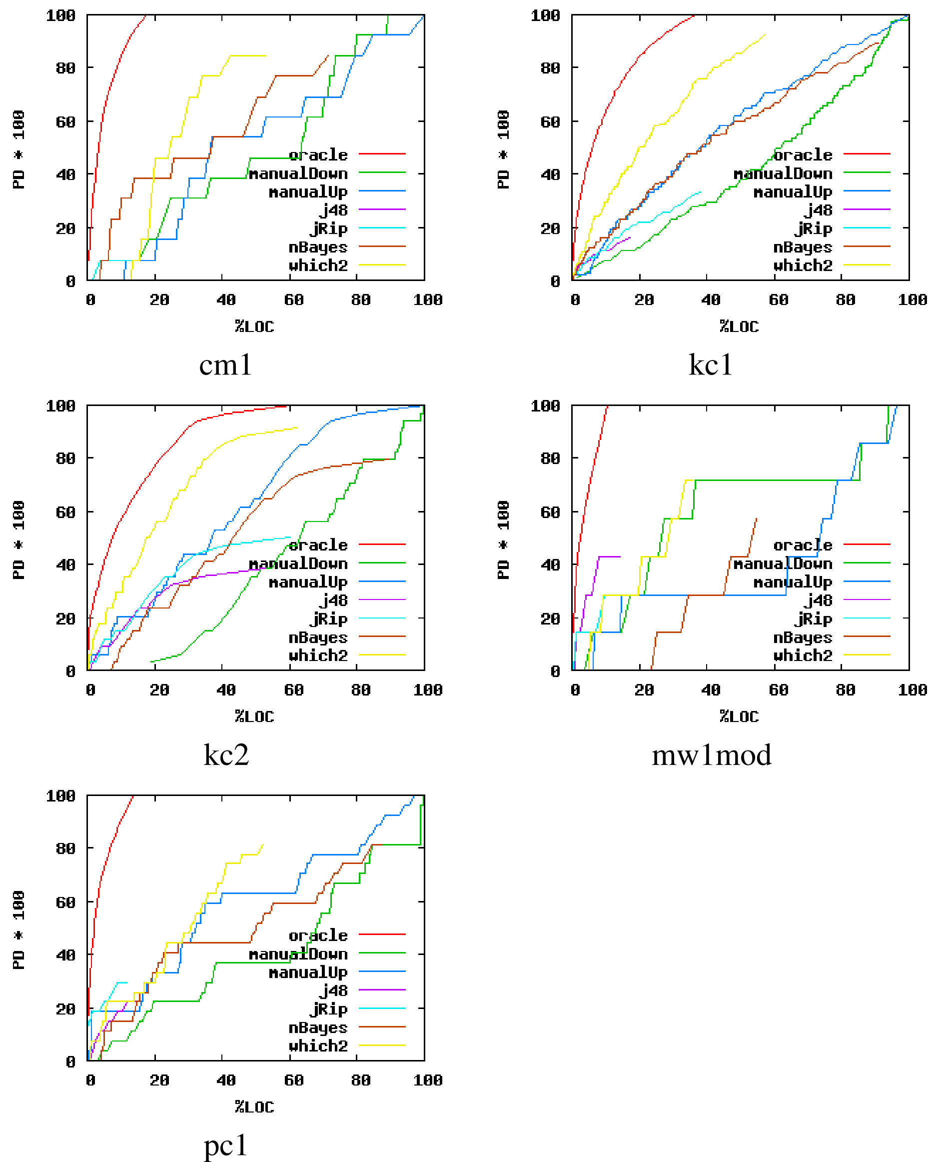

Figure 4.10: MDP Data Files where Which2 > manualUp > rest.

Class Two

The second class the one in which Which2 > everything else, but manualUp is beaten by some of the classic learners. This classification illustrates two points that we found important. The first one is that Which2 wins overall, another victory for Which2, and that some of the detectors

learner	mean	Equal	Description
which2	68.1		
manualUp	59.8	=	
nBayes	52.1	<	
manualDown	47.6	<	
which8	11.4	<	
jRip	5.8	=	
j48	0.1	=	
which8loc	0.0	<	
which4loc	0.0	=	
which4	0.0	>	
which2loc	0.0	<	

cm1

learner	mean	Equal	Description
which2	76.0		
manualUp	67.6	<	
nBayes	61.9	<	
which4	52.9	<	
manualDown	43.3	<	
j48	27.8	<	
jRip	21.3	<	
which8loc	0.0	<	
which8	0.0	<	
which4loc	0.0	<	
which2loc	0.0	=	

kc1

learner	mean	Equal	Description
which2	81.6		
manualUp	69.3	<	
which4	59.4	<	
nBayes	58.7	=	
manualDown	46.1	<	
which8	41.2	<	
j48	41.2	=	
jRip	42.2	>	
which8loc	0.0	<	
which4loc	0.0	=	
which2loc	0.0	=	

kc2

learner	mean	Equal	Description
which2	62.4		
manualDown	60.2	=	
manualUp	47.8	<	
which4	42.7	<	
nBayes	41.7	=	
which8	39.3	<	
j48	20.0	<	
jRip	15.8	=	
which8loc	0.0	<	
which4loc	0.0	=	
which2loc	0.0	=	

mw1mod

learner	mean	Equal	Description
which2	65.0		
manualUp	60.6	>	
nBayes	51.5	<	
manualDown	44.6	<	
which8	22.6	<	
j48	19.2	=	
jRip	15.1	=	
which8loc	7.4	<	
which4loc	3.8	=	
which4	0.0	>	
which2loc	0.0	<	

pc1

Figure 4.11: Quartile Charts Repesenting Graphs in Figure 4.10

generated in this class perform better than the manual inspection. This second point is illustrating that manualUp should not be the current de facto standard.

A point of interest here is that Naïve Bayes performs better than the other standard learners in this class, much like class one. This is still consistent with the results of [30].

Class Three

The third and final class is the one in which manualUp wins overall, and the classic learners perform better than Which2. This class contains only one file and it also represents the one time

key	ties	win	loss	win-loss-at-99%
which2	1	9	0	9
manualUp	1	9	0	9
nBayes	0	8	2	6
manualDown	0	7	3	4
which8	3	3	4	-1
which4	3	3	4	-1
jRip	3	3	4	-1
j48	3	3	4	-1
which8loc	2	0	8	-8
which4loc	2	0	8	-8
which2loc	2	0	8	-8

cm1

key	ties	win	loss	win-loss-at-99%
which2	0	10	0	10
manualUp	0	9	1	8
nBayes	0	8	2	6
which4	0	7	3	4
manualDown	0	6	4	2
j48	0	5	5	0
jRip	0	4	6	-2
which8loc	1	2	7	-5
which8	3	0	7	-7
which4loc	2	0	8	-8
which2loc	2	0	8	-8

kc1

key	ties	win	loss	win-loss-at-99%
which2	0	10	0	10
manualUp	0	9	1	8
which4	1	7	2	5
nBayes	1	7	2	5
manualDown	1	5	4	1
jRip	3	3	4	-1
which8	2	3	5	-2
j48	2	3	5	-2
which8loc	2	0	8	-8
which4loc	2	0	8	-8
which2loc	2	0	8	-8

kc2

key	ties	win	loss	win-loss-at-99%
which2	1	9	0	9
manualDown	1	9	0	9
manualUp	2	6	2	4
which4	3	5	2	3
nBayes	3	5	2	3
which8	2	5	3	2
jRip	1	3	6	-3
j48	1	3	6	-3
which8loc	2	0	8	-8
which4loc	2	0	8	-8
which2loc	2	0	8	-8

mw1mod

key	ties	win	loss	win-loss-at-99%
manualUp	1	9	0	9
which2	2	8	0	8
nBayes	1	8	1	7
manualDown	1	6	3	3
which8	3	3	4	-1
jRip	3	3	4	-1
j48	3	3	4	-1
which4	7	0	3	-3
which8loc	3	0	7	-7
which4loc	3	0	7	-7
which2loc	3	0	7	-7

pc1

Figure 4.12: MannWhitney U−Tests Representing Graphs in Figure 4.10

out of the eight that Which2 does not perform well.

Overall

Figure 4.19 and Figure 4.20 show the overall results of the experiment. These tables of data contain information about the experiment that give an overview over all of the NASA projects used in the MDP experiment. There are a few things that can be gathered from these figures. The first is that Which2 wins uncontested over all of the projects according to the MannWhitney U−Tests. This

Figure 4.13: MDP Data Files where Which2 $>$ rest. This is for kc3 mod.

learner	mean	Equal	Description
which2	87.3		
nBayes	64.2	$<$	
manualUp	64.2	$=$	
which4	47.8	$<$	
manualDown	47.6	$=$	
which8	46.7	$=$	
j48	23.1	$<$	
jRip	17.7	$<$	
which2loc	6.6	$<$	
which8loc	0.0	$<$	
which4loc	0.0	$<$	

Figure 4.14: Quartile Charts Representing Graphs in Figure 4.13.

key	ties	win	loss	win-loss-at-99%
which2	0	10	0	10
nBayes	1	8	1	7
manualUp	1	8	1	7
which8	2	5	3	2
which4	2	5	3	2
manualDown	2	5	3	2
j48	1	3	6	-3
jRip	2	2	6	-4
which2loc	2	1	7	-6
which4loc	2	0	8	-8
which8loc	1	0	9	-9

Figure 4.15: MannWhitney U$-$Tests Representing Graphs in Figure 4.13.

means that, overall, Which2 has the best performance over all of the other detectors. Another point of interest is that Naïve Bayes performs better than the other classic machine learners overall. This is the same assertion that was made in [30].

It can be seen that our second heuristic for Which, $h_{\frac{PD}{LOC}}$, did poor no matter the discretization policy. However, although Which4 and Which8 performed much poorer than Which2, they both

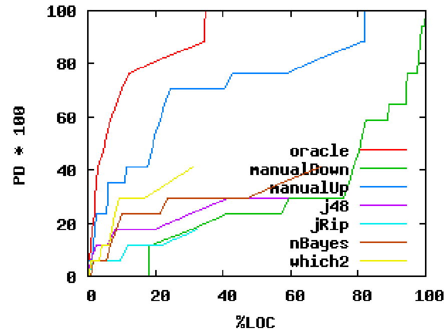

Figure 4.16: MDP Data Files where all > Which. This is the data file, mc2_mod.

learner	mean	Equal	Description
manualUp	74.3		
nBayes	55.9	<	
manualDown	42.8	<	
j48	43.7	=	
jRip	28.5	<	
which8	21.9	<	
which4	5.6	<	
which8loc	0.0	<	
which4loc	0.0	>	
which2loc	0.0	=	
which2	0.0	>	

Figure 4.17: Quartile Chart Representing Graphs in Figure 4.16.

key	ties	win	loss	win-loss-at-99%
manualUp	0	10	0	10
nBayes	0	9	1	8
manualDown	1	7	2	5
j48	1	7	2	5
jRip	1	5	4	1
which8	3	3	4	-1
which4	4	1	5	-4
which2	5	0	5	-5
which4loc	4	0	6	-6
which2loc	4	0	6	-6
which8loc	3	0	7	-7

Figure 4.18: MannWhitney U−Tests Representing Graphs in Figure 4.16.

still performed better than j48 and Ripper. The fact that Which2 performed the best overall and the best out of all of the Which variants says something about that data. As stated before, Which2 uses a two bins, equal frequency discretization policy. This means that all of the attribute values are sorted in ascending order, and the first 50% are in bin one, while the second 50% are in bin two. This kind of discretization effectively does a binary dissection of the data. The fact that Which

performs best using this kind of discretization implies that the attributes themselves, after taking the natural log of the attributes, tend to predict the class based on their value. At least the attribute values that Which2 chose in its detection policy tend to predict if a module is defective or not in a very simple way. It could be said that after or before some threshold value of a few of the attributes, a module tends to be defective.

learner	mean	Equal	Description
which2	77.6		
manualUp	63.7	<	
nBayes	61.2	=	
which4	53.7	<	
manualDown	46.4	<	
which8	35.5	<	
j48	27.9	=	
jRip	23.9	<	
which8loc	0.0	<	
which4loc	0.0	=	
which2loc	0.0	=	

Figure 4.19: Quartile Charts Representing the Overall Performance of the Detectors on the MDP Experiment.

key	ties	win	loss	win-loss-at-99%
which2	0	10	0	10
nBayes	1	8	1	7
manualUp	1	8	1	7
which4	0	7	3	4
manualDown	0	6	4	2
which8	1	4	5	-1
j48	1	4	5	-1
jRip	0	3	7	-4
which8loc	2	0	8	-8
which4loc	2	0	8	-8
which2loc	2	0	8	-8

Figure 4.20: MannWhitney U−Tests Representing the Overall Performance of the Detectors on the MDP Experiment.

4.7 Category III: Turkey Data

In order for us test the external validity of the results of Section 4.6, we chose a completely different type of software data. The data from a Turkish appliance company and was chosen just for this purpose. The Turkey data is in three seperate files that are each for a different type of appliance. Table 4.9 describes some properties about the data. An important point to gain from this table is that the number of actual modules in these projects is significantly smaller than the projects in Table 4.5. The largest number of modules in the Turkey data is 107 whereas the smallest number of modules in the NASA data is 161.

There are a few important points about the Turkey data that distinguishes it from the data taken from the NASA projects. The first point is that the Turkey data was taken from a commercial appliance company. This is must different from the NASA data in that the software is made for commercial gain and in an environment that is completely unrelated to government projects. The second point is that this software was developed by people from a country other than the United States of America. This fact creates a distinguishing divide between the two types of projects. Another point is that this software was created in an ad−hoc, informal method using groups of only two or three people. This is far less structured than the types of teams that developed the NASA data. We believe that these facts about the differences in the data, coupled with the results of Section 4.7.2, show that the Which machine learner has some support for working in a more universal way, as opposed to detectors that work on only one type of project.

File Name	Type	Defects	Modules
ar3	Washing Machine	8	63
ar4	Dishwasher	20	107
ar5	Refrigerator	8	35

Table 4.9: Information About the Turkey data.

4.7.1 Experiment

This experiment was set up in a similar fashion to the experiment discussed in Section 4.6. The same process of cross validation using a $10x3$ method. As with the MDP experiment, several different types of variants of Which were used and these are reported in Figure 4.22 and Figure 4.23, but were not reported in Figure 4.21 for the sake of clarity in the graphs. The same variants of Which that are discussed in Table 4.8 were also used in this experiment.

4.7.2 Results

Since the Turkey experiment only consisted of three seperate project files, we feel it is uncessary to catergorize in the same fashion as the MDP experiment in Section 4.6. The same layout conventions are still used, however, that is there are three figures, Figure 4.21, Figure 4.22 and Figure 4.23, that all report results on the data. An overall quartile chart and MannWhitney U−Test is also reported.

Figure 4.24 and Figure 4.25 represent the performance of each of the detectors over all three of the projects. It gives insight to how well each detector worked as a whole, instead of the individual projects that the other results show. An important fact to notice here is that overall, manual−Up was the best performer. A possible explaination of this involves the nature of the test. The single project where manual−Up had the best overall performance in the MannWhitney U−Test was ar3. ar3 has a total of 63 modules with only 8 of them being defective. Our cross−validation first randomizes the data before splitting it into 3 train−test sets. Each module has a 66% chance to be in the train set and a 33% chance to be in the test set. A total of $63(0.33) = 21$ modules are in each of the test sets. The distribution of defective modules in the set is $\frac{8}{63} = 0.126$. If only 21 of the modules are in the test set and the distribution of defects is 12.6%, then the number of defective modules, on average, in the test set will be $21(0.126) = 3$. It is quite possible for those 3 defective modules per test set to have a small line of code attribute and thus be explored first. Since we are only dealing with very small sample sizes, manual−Up has an advantage of being able to quickly identify the defective modules. The problem with manual−Up is that it has no detector, it simply sorts the modules by the line of code attribute and explores them all. As mentioned in Section 4.1,

the area under the curve measurement is the area under each of the curves plus the area of the rectangle representing the horizontal line extension of the curve to a point $(100, y)$ on the Koru Diagram. Since all of the remaining modules in the test space are not defective, manual$-$Up will travel this space in a horizontal line, in essence emulating the oracle. The oracle technically, stops at the point $(x, 100)$, however, and the straight line is artificially inserted into the space to guarentee that it will have the most area under its curve. So while the detectors generated from the machine learners perform *slightly* worse than manual$-$Up, it is important to note that they do not end at the point $(100, 100)$, like manual$-$Up does. This is only a possible explaination for this effect with the Turkey data, but one that we think holds merit.

Another intersiting point that can be seen looking at this data is that all of the machine learners performed about the same according to both the MannWhitney U$-$Tests and the Quartile charts. This does give some merit to the classic learners of j48 and Naïve Bayes , but it also gives more merit to Which. The goal of the Turkey experiment was to show that Which can be used to create detectors over a broader range of software projects. The fact that it performed well compared to the other machine learners only helps that claim. It is important to note that while the classic machine learners did perform well on this data, they did not perform well on the NASA data. This instability in their performance is countered by the fact that Which performs well on both data sets.

A final point of interest that is seen in these results is that the other variants of Which, described in Table 4.8, also performed on par with the classic learners. For the reasons illustrated in the above paragraph, however, the instability over multiple project cases is undesirable. It is interesting that our variants using the $h_{\frac{PD}{LOC}}$ heuristic performed well, however, because we had initially assumed the general correlation of pd and effort should not work, given the discussion in Section 4.5.4.

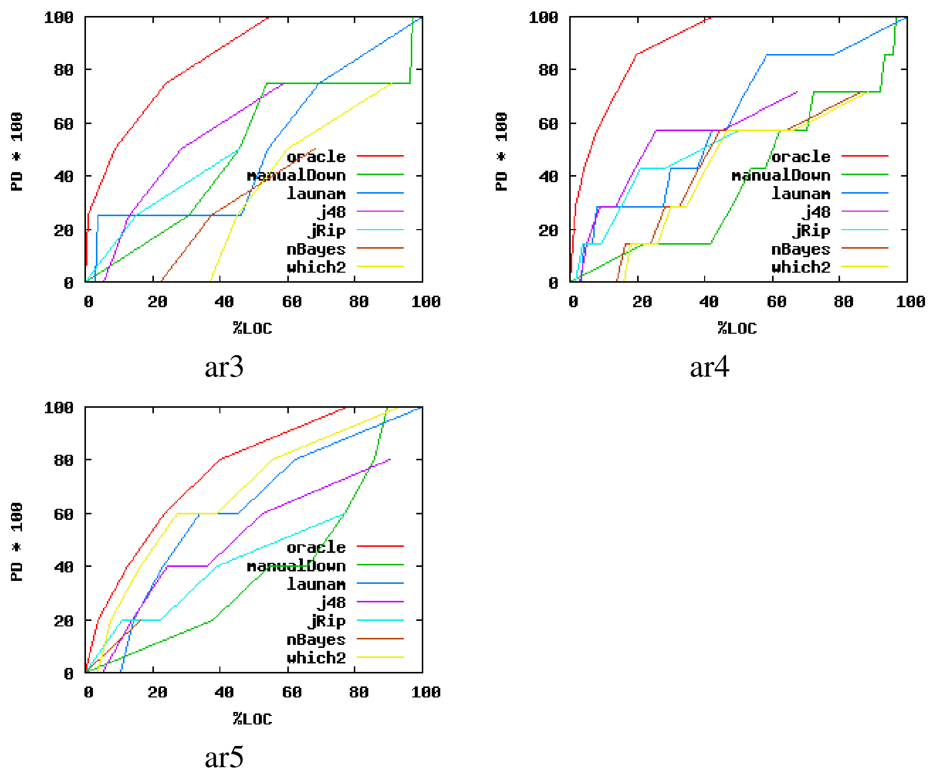

Figure 4.21: Graphs representing the results for the Turkey experiment.

learner	mean	Equal	Description
manual	54.7		
launam	52.6	<	
which8	42.7	<	
which4	41.8	<	
which2	40.4	>	
nBayes	34.4	<	
j48	47.8	>	
jRip	0.2	<	
which2loc	0.1	>	
which8loc	0.0	<	
which4loc	0.0	=	

ar3

learner	mean	Equal	Description
which2	58.6		
nBayes	56.2	<	
launam	56.2	=	
manual	56.5	=	
which2loc	55.9	=	
which4	49.3	<	
jRip	47.3	=	
which8	42.6	<	
j48	38.8	=	
which8loc	0.0	<	
which4loc	0.0	=	

ar4

learner	mean	Equal	Description
which4	71.4		
manual	69.6	=	
which2	67.6	=	
nBayes	54.1	<	
j48	56.1	=	
launam	56.5	<	
jRip	55.0	>	
which8loc	0.0	>	
which8	0.0	=	
which4loc	0.0	=	
which2loc	0.0	=	

ar5

Figure 4.22: Quartile Charts Correlating to Figure 4.21.

key	ties	win	loss	win-loss-at-99%
manual	5	13	0	13
launam	14	4	0	4
which8	15	3	0	3
which2	15	3	0	3
j48	15	3	0	3
which4	14	3	1	2
which2loc	15	2	1	1
nBayes	14	2	2	0
jRip	2	2	14	-12
which8loc	1	0	17	-17
which4loc	1	0	17	-17

ar3

key	ties	win	loss	win-loss-at-99%
which2	9	9	0	9
which2loc	14	4	0	4
nBayes	14	4	0	4
manual	14	4	0	4
launam	14	4	0	4
which4	15	2	1	1
jRip	15	2	1	1
which8	8	2	8	-6
j48	8	2	8	-6
which8loc	1	0	17	-17
which4loc	1	0	17	-17

ar4

key	ties	win	loss	win-loss-at-99%
which4	13	5	0	5
which2	13	5	0	5
manual	13	5	0	5
nBayes	14	4	0	4
jRip	14	4	0	4
j48	14	4	0	4
launam	4	4	10	-6
which8loc	3	0	15	-15
which8	3	0	15	-15
which4loc	3	0	15	-15
which2loc	3	0	15	-15

ar5

Figure 4.23: MannWhitney U−Tests Correlating to Figure 4.21.

learner	mean	Equal	Description
manual	59.9		
which2	54.5	<	
nBayes	49.6	<	
which4	51.3	=	
launam	54.6	=	
j48	49.0	<	
jRip	43.1	<	
which8	35.5	<	
which2loc	26.9	=	
which8loc	0.0	<	
which4loc	0.0	=	

Figure 4.24: Quartile Charts Representing the Overall Performance of the Detectors on the Turkey Experiment.

key	ties	win	loss	win-loss-at-99%
manual	4	14	0	14
which2	13	5	0	5
which4	12	5	1	4
nBayes	12	5	1	4
launam	12	5	1	4
j48	13	4	1	3
jRip	3	2	13	-11
which8	2	2	14	-12
which2loc	2	2	14	-12
which8loc	1	0	17	-17
which4loc	1	0	17	-17

Figure 4.25: MannWhitney U−Tests Representing the Overall Performance of the Detectors on the Turkey Experiment.

4.8 Category III: AT&T Data

The following experiment was done using data from the AT&T Laboratory in New Jersey. This data has been used in several other experiments by Dr. Thomas Ostrand and Dr. Elaine Weyuker and Dr. Robert Bell [2,36]. This data was proprietary so no statistics are reported about the data.

4.8.1 Data

The AT&T data is very different from the data used in the experiments of Section 4.6 and Section 4.7. Unlike the previous experiments, this is the only data tested on that consists not of seperate projects but instead one project broken up into 35 releases. Each release contains information relating to the previous releases. Also data does not use the Halstead [19] and McCabe [28] attributes that were present in those data sets. In fact, individual instances in this data correspond to files in the project, not modules. This is a less fine−grained approach than the modules of the previous defect detection process. A file is different from a module in that, in C++ for example, a file may contain an entire class's implementation, thus containing many modules per file. Also, if a module is flagged in being defective is would be easier to test that module for the specific defect. However, if an entire class implementation file is flagged as defective it could require more extensive testing. A very important difference with this data compared to the previous MDP and Turkey data sets is that the defect class is not a boolean. In this data, the class is numeric and contains the number of defects found in that file. This creates some problems with comparisons with our previous test because defective instances are no longer equal. That is if an instance is found to have 100 defects, it should be much more important to detect over an instances that has only 5 defects.

Table 4.10 gives a description of each of the different attributes used in this data file. This is fundamentally different from the other experiment's data for a few reasons. As mentioned previously, the Halstead and McCabe metrics are not in this data about the instances. Also, there is no information at all about the actual source code besides the language it was written in and the amount of lines of code that each instance contains. The idea behind most of these attributes is that it is the method the files were accessed and by whom they were accessed that causes the defects in the instances. It should be said that another idea about the statistical information that the creators

Attribute	Description
File Age	Describes how long this file has been in the project.
Is New?	Boolean attribute that is 1 if the file was created this project and 0 if it was created previously.
New Developers	Number representing the new developers that have worked on this file since the previous release.
New Developers -1	Number representing the new developers that have worked on this file in the previous release.
New Developers -2	Number representing the new developers that worked on this file in the previous previous release.
Cum Developers	Total number of developers that have worked on this file since its creation.
Cum Developers -1	Total number of developers that have worked on this file since its creation until the previous release.
Cum Developers -2	Total number of developers that have workeed on this file since its creation until the previous previous release.
LOC	The number of lines of code that this file contains.
Language	The programming language this file was written in.
Defects	Number of defects this file contains.

Table 4.10: Information About the Attributes Used in the AT&T Data Sets.

of this thought could cause more errors is the relation of modifications to this file that were done both one and two releases ago. It can be said that having this previous and previous previous release information creates a situation where several attributes in the data set are highly correlated. In the above table, the defects attribute was changed from an integer to a boolean where it was given true if $defects \geq 0$ and false otherwise.

As a final note, we are under a nondisclosure agreement about the data we used in these experiments and are not allowed to publish any information about the data that we haven't already. This includes the number of files, names of files, and number of defects per file, distribtion of attribtues as well as the actual data itself.

4.8.2 Experiment

As was stated in Section 4.8.1, this data consists of 35 seperate releases and this created a convenient method of training and testing the machine learners that removed the need for the $10x3$

cross−validation of the previous experiments. Instead, we ran a series of tests using releases 4 through 35as test data and as train data used the previous 5 releases. For release tests 4 and 5, all available previous releases were used, since there were less than 5 previous releases. In other words, if we wanted to test the machiene learners on release 7, we would construct a train set consisting of releases $2-6$. This is very equivalent to a real world situation since many projects are deployed or at least tested in a release format. A machine learner would have access to all of the previous releases to learn from in order to detect defective files in the current release to be tested. The reason we did not allow more than 5 previous test releases is due to time constraints, we only had 5 days to use the data and some of these experiments could take a very long time to run. However, the experiments of [2, 36], used only the previous n releases as well.

The original goal of this experiment was to compare our machine learner, Which, to the negative binomial regression used by Dr. Ostrand et al. prevously on the data that reported good results. More information on these experiments can be see in Section 2.8. However, as was discussed in Section 4.8.1, their detector did not simply attempt to single out defective files but also rank the defective files based on some predicted number of defects this file would have. The current state of our tests so far have been done using the boolean class of defective or not defective and the Koru Diagram of Section 4.1 was meant for such a boolean class system. Altering the data to have a booleana class was done and as such a fair comparison could not be made with Which and the negative binomial regression. This is because we are ranking our learners based on number of defective instances singled out whereas their system was weighting which predicted defective modules to list first. Instead of comparing Which to the negative binomial regression, we instead compared the classic learners of j48, Ripper, and Naïve Bayes with Which. The manual detectors of manual−Up and manual−Down were also included in our studies.

The distributions of these attribute values were significantly different than those of Section 4.6 and Section 4.7. So to deal with this change in distribution, we use Which10 in these experiments instead of the Which2 that we have used prior. Which10 is simply the ordinary Which process that uses 10 bin equal frequency discretization as opposed to the 2 bin equal frequency discretization that Which2 uses. Also, we have decided to include a micro−sampling varaint in these experiments

as well. This shows up in the result figures as Which10010. Which10010 uses a micro−sampling with $n = 100$ and 10 bins equal frequency discretization. The process of micro−sampling is described in detail in Section 2.2.2. The experiment of Section 4.9 is an experiment dedicated to the effects of micro−sampling as a preprocessor with Which. The reason we have included a micro−sampling variant of Which in this experiment was due to time constaints. We did not have enough time to do an entire micro−sampling experiment on the AT&T data, so we opted to only include one variant of Which that does it.

4.8.3 Results

In order to save space, only a few of the resulting graphs, quartile charts, and MannWhitney U−Tests are reported here, but the overall results reported later in this section contain result information based on all 32 total tests conducted.

Figure 4.26, Figure 4.27, and Figure 4.28 give the plot, quartile chart, and MannWhitney U−Test results for 4 different releases respectively. The results here are typical results and were chosen because these four releases had more faulty files than some of the other ones, making some of the notable points easier to see in the plots. It can be seen here that the oracle behaves in the same way it did in the experiments of Section 4.6 and Section 4.7. The oracle's line has a very high slope implying that most of the faults are found in the smaller modules. The plot of release four looks exactly like our initial illustration of the Koru Diagram, Figure 4.1, found in Section 4.1.

It is interesting to see the changes that occur with the manual−Up and manual−Down detectors as the software project matures. As was stated, release 4 looks like something we would expect, with manual−Up traversing the Koru space in a similar fashion to the oracle, even though the slope is much less, and manual−Down traversing the Koru space in the opposite way. However, in the later releases, namely releases 19 and 24 reported here, manual−Up and manual−Down traverse the space nearly parallel with each other, and in release 24 manual−Down is above manual−Up. The fact that these two detectors are parallel with each other implies that the underlying distribution of faults in the project files is evenly distributed among large and small files. However, looking at the oracle suggests that this is not the case. An explaination of this is that the files are all very

similar in size, so the effect of sorting them by their lines of code is lost here. The fact that the oracle still rises very quickly to the 100% pd line in these plots can be explained by the fact that while all of the files are close to the same size, a small percentage of those files actually contain faults. In release 19, $\frac{20}{579} \approx 3.4\%$ of the files actually contain errors. Another test on this exact data was done by Ostrand et al. and they concluded that the distribution of faults lies in the Pareto distribution described in Section 2.7 [36]. Their conclusion matches the conclusion we have stated here that explains why the oracle still maintains a very large slope even though the file sizes are all similar.

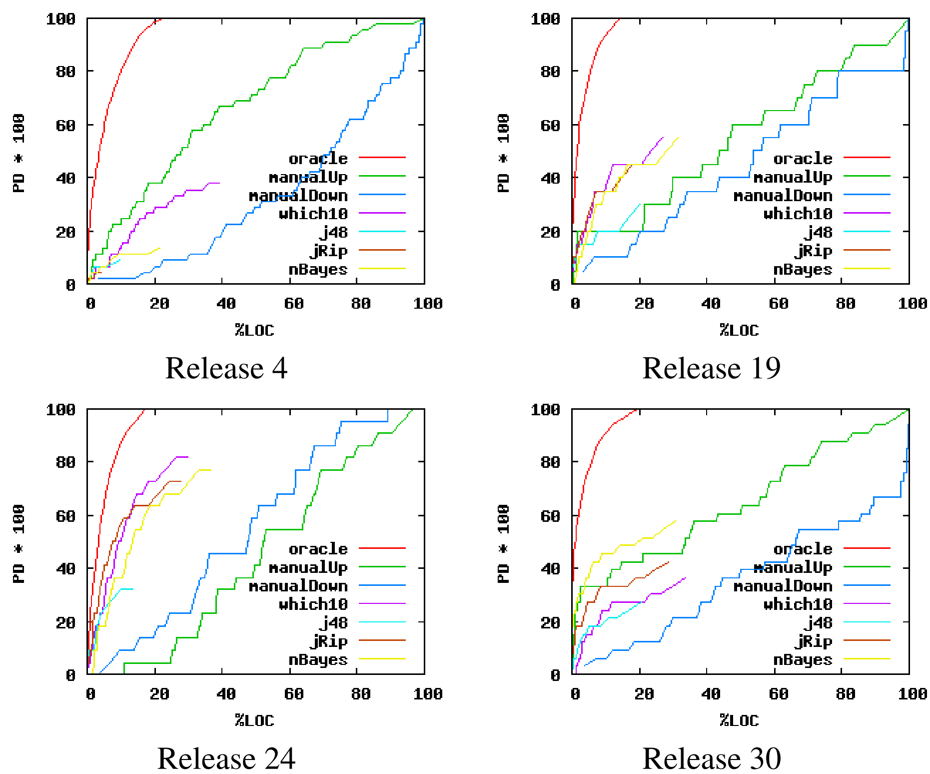

Figure 4.26: Plots of Selected Results from AT&T Experiment.

Figure 4.29 and Figure 4.30 show the overall quartile chart and Mann Whitney U−Test results for the AT&T experiment respectively. These results show that manual−Up is the overall winner over the 32 total release based experiments that were conducted. This says that manually inspecting this code would be better than taking the advise of both the classic machine learners of j48, Ripper,

learner	mean	Equal	Description
manualUp	60.4		
nBayes	48.5	<	
which10	45.9	<	
manualDown	44.7	<	
micro10010	33.3	<	
jRip	32.6	<	
j48	30.7	=	

Release 4

learner	mean	Equal	Description
manualUp	57.4		
micro10010	52.6	=	
nBayes	50.6	<	
which10	45.9	=	
manualDown	45.0	=	
jRip	27.7	=	
j48	20.5	<	

Release 19

learner	mean	Equal	Description
manualUp	63.1		
which10	51.9	<	
nBayes	48.5	=	
manualDown	48.1	=	
j48	30.7	<	
micro10010	28.1	>	
jRip	27.8	<	

Release 24

learner	mean	Equal	Description
manualUp	57.4		
nBayes	50.8	<	
which10	37.7	<	
manualDown	37.7	>	
jRip	30.5	<	
micro10010	28.1	>	
j48	26.4	<	

Release 30

Figure 4.27: 4 Result Quartile Charts Taken from AT&T Experiment.

key	ties	win	loss	win-loss-at-99%
manualUp	2	4	0	4
nBayes	3	3	0	3
which10	4	2	0	2
manualDown	3	2	1	1
micro10010	4	0	2	-2
jRip	2	0	4	-4
j48	2	0	4	-4

Release 4

key	ties	win	loss	win-loss-at-99%
micro10010	5	1	0	1
manualUp	5	1	0	1
which10	6	0	0	0
nBayes	6	0	0	0
manualDown	6	0	0	0
jRip	6	0	0	0
j48	4	0	2	-2

Release 19

key	ties	win	loss	win-loss-at-99%
manualUp	4	2	0	2
which10	6	0	0	0
nBayes	6	0	0	0
micro10010	6	0	0	0
manualDown	6	0	0	0
jRip	5	0	1	-1
j48	5	0	1	-1

Release 24

key	ties	win	loss	win-loss-at-99%
manualUp	3	3	0	3
nBayes	6	0	0	0
micro10010	6	0	0	0
manualDown	6	0	0	0
which10	5	0	1	-1
jRip	5	0	1	-1
j48	5	0	1	-1

Release 30

Figure 4.28: 4 Result MannWhitney U−Tests Taken from AT&T Experiment.

learner	mean	Equal	Description
manualUp	60.4		
nBayes	50.6	<	
manualDown	44.8	<	
which10	44.8	<	
micro10010	33.5	<	
jRip	36.1	<	
j48	30.7	<	

Figure 4.29: Overall Quartile Chart for the AT&T Experiment.

key	ties	win	loss	win-loss-at-99%
manualUp	0	6	0	6
nBayes	2	3	1	2
manualDown	3	2	1	1
which10	4	1	1	0
micro10010	4	0	2	-2
jRip	3	0	3	-3
j48	2	0	4	-4

Figure 4.30: Overall MannWhitney U−Tests Taken for the AT&T Experiment.

and Naïve Bayes as well as our new learner, Which. We have two possible explanations for these results conflicting with our results in the experiments of Section 4.6 and Section 4.7.

The first of these explanations is that the attributes that were used to describe the AT&T data are vastly different from the attribute used to describe the MDP and Turkey data. While the MDP and Turkey data use attributes to describe specific properties about the actual source code itself, the AT&T data uses attibutes to describe the methods the code was accessed and written over the course of several releases. An idea about these differences is that it is possible that the Halstead [19] and McCabe [28] statistics about the source code are easier to learn from and therefore create a performance decrease.

The second of these explanations was something we realized only after we had left the AT&T Laboratory. As was stated before, we only had one week to run our experiments on the AT&T data. Our explanation about the behavior of the oracle remaining constant throughout all three experiments but manual−Up and manual−Down differing greatly in the AT&T experiment from the MDP and Turkey experiments was that the lines of code of each file is generally the same between all files in the release but the number of files that contain faults is still in the minority. This implies that the line of code attribute naturally is not exponential in distribution. Also, looking at some of the attribute types in Table 4.10, it is highly probable that these types of attributes do not lie in an exponential distribution like the attributes described in Table 4.6. However, we had used the same rig to save time and that rig, by default, took the natural logarithm of each attribute before the machine learners trained and tested on it. The problem with exponentially distributed attribute ranges is discussed in Section 4.5.3. The same problems are seen when an attibute has a logarithmic distribution as well, except the bins with the largest range will be the first few bins in discretization,

and the remaining bins will all have very small ranges. This is problematic to learners for reasons discussed in Section 4.5.3.

4.9 Category IV: Micro−Sampling

The process of micro−sampling is discussed in detail in Section 2.2.2. For this experiment, we decided to use micro−sampling with Which2 to see if the concept of micro−sampling, which received good results in [34], would work to Which's advantage on the experiments of Section 4.6 and Section 4.7.

4.9.1 Experiment

This experiment was done after the experiments using the classical learners and the various Which variants of Table 4.8 were done. Since we concluded from the results of those experiments that Which2 was the best performer, these micro−sampling experiments will use Which2 as the variant of Which. Also, for the purposes of clarity, and because it was beyond the scope of this experiment to do so, the classic machine learners and the various Which variants were not included in this experiment. As was explained in Section 2.2.2, micro−sampling is the process of creating a $50-50$ distribution of classes, as well as a specific number of instances, in data sets that contain two classes. Since research has been done on micro−sampling and claims of this constant number being ≥ 25 have been reported, we decided to use several different sizes in this experiment to see how well Which performs with them. Table 4.11 provides an expaination of the various micro−sampling variants of Which2 used in this experiment. Smaller versions of micro−sampling were used in this experiment, namely micro5 and micro10, but their results were too unstable and heavily dependant on the 10 and 20 random instances of the data that were chosen. It is because of this fact that they were not included in the results. As with the experiments of Section 4.6 and Section 4.7, this experiment used a $10x3$ cross−validation coupled with taking the natural logarithm of each of the attributes as a preprocessor to the experiment. There were actually two experiments performed with micro−sampling, the first was on the MDP data of and the second was on the Turkey appliance data. These results were combined for an overall look of the performance of the micro−sampled variants of Which and will be discussed in Section 4.9.2.

Name	Description
Which2	A standard version of Which with 2 bins equal frequency discretization.
Micro20	Micro−sampling using 20 true and 20 false instances.
Micro30	Micro−sampling using 30 true and 30 false instances.
Micro50	Micro−sampling using 50 true and 50 false instances.
Micro75	Micro−sampling using 75 true and 75 false instances.
Micro100	Micro−sampling using 100 true and 100 false instances.

Table 4.11: Description of Different Micro−Sampling Values used in Section 4.9.1.

4.9.2 Results

Figure 4.31 and Figure 4.32 contain the quartile charts and the MannWhitney U−Tests taken from the experiment on the NASA and the Turkey data. As with the MDP experiment, there are three distinct categories of results that can be gathered from this figures.

The first category is where Which2 still outperforms its micro−sampled counterparts. These occur on the files kc3mod and ar4. These victories are gathered by looking at the MannWhitney U−Tests of Figure 4.32. By comparing the quartile charts to the MannWhitney U−Tests for kc3mod, it can be seen that while Which2 does have a better performance, the results are not nearly as standout as the results of ar4.

The second category is the category where most of the micro−sampled variants tie with Which2 and each other. This result can be seen in cm1, kc2, kc3, pc1, mw1mod, ar3, and ar5. This result is very interesting because the variant Micro20 is only dealing with a very small number of instances. cm1 contains 49 defective modules and 449 other modules. Micro20 will reduce that set to only 20 of each type of module, effectively throwing away information about 429 of the non−defective modules. This is counter−intuitive because it *should* be the case that the information about the non−defective modules would help the Which learner to better decide what not to include in its rules. This is the largest of the three categories containing $\frac{7}{10} = 70\%$ of the total data sets that fall into this category. This type of result really leads to the conclusion that [34] said about micro−sampling not hurting the performance of machine learners. More about the conclusions is discussed further down in this section.

The third category is the category where Which2 loses to some or all of the micro−sampled

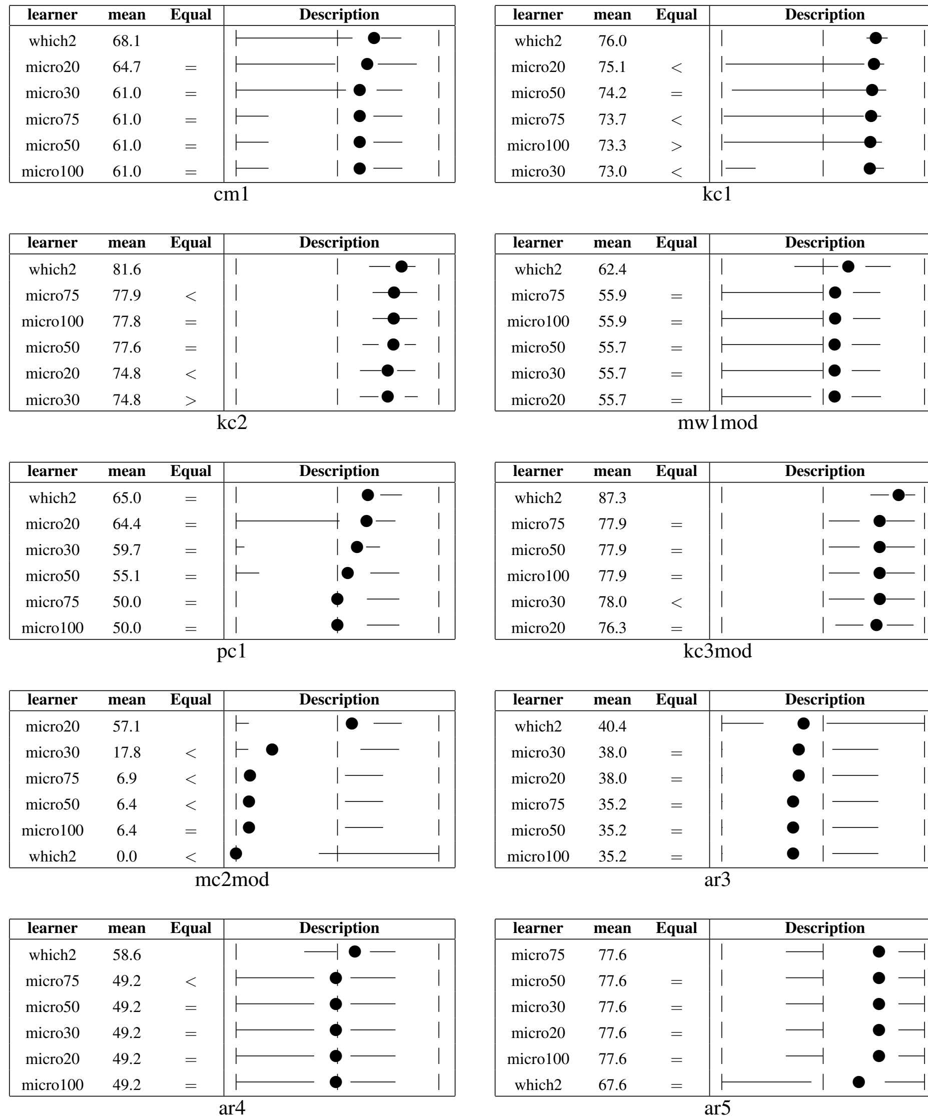

Figure 4.31: Quartile Chart Results for Micro−sampling.

variants of Which. The data set that falls into this category is mc2mod. mc2mod is the data set that Which2, and all other Which variants, lost to all of the classic machine learners in the MDP experiment of Section 4.6. The fact that Which2 also lost to its micro−sampled variants in this file is interesting by comparison. Inserting the results of Micro20 into the results of Figure 4.17 would place it below Naïve Bayes but above manual−Up. mc2mod contains 52 defective modules out of 161 total modules, according to Table 4.7. Micro20 would reduce the defective modules to $\frac{20}{52} = 38.4\%$ and the non−defective modules to $\frac{20}{109} = 18.3\%$. It is intersting to see that using

key	ties	win	loss	win-loss-at-99%
which2	5	0	0	0
micro75	5	0	0	0
micro50	5	0	0	0
micro30	5	0	0	0
micro20	5	0	0	0
micro100	5	0	0	0

cm1

key	ties	win	loss	win-loss-at-99%
which2	3	2	0	2
micro50	5	0	0	0
micro20	5	0	0	0
micro100	5	0	0	0
micro75	4	0	1	-1
micro30	4	0	1	-1

kc1

key	ties	win	loss	win-loss-at-99%
which2	4	1	0	1
micro75	5	0	0	0
micro50	5	0	0	0
micro30	5	0	0	0
micro100	5	0	0	0
micro20	4	0	1	-1

kc2

key	ties	win	loss	win-loss-at-99%
which2	5	0	0	0
micro75	5	0	0	0
micro50	5	0	0	0
micro30	5	0	0	0
micro20	5	0	0	0
micro100	5	0	0	0

mw1mod

key	ties	win	loss	win-loss-at-99%
which2	5	0	0	0
micro75	5	0	0	0
micro50	5	0	0	0
micro30	5	0	0	0
micro20	5	0	0	0
micro100	5	0	0	0

pc1

key	ties	win	loss	win-loss-at-99%
which2	0	5	0	5
micro75	4	0	1	-1
micro50	4	0	1	-1
micro30	4	0	1	-1
micro20	4	0	1	-1
micro100	4	0	1	-1

kc3mod

key	ties	win	loss	win-loss-at-99%
micro20	2	3	0	3
micro30	4	1	0	1
micro75	5	0	0	0
micro50	4	0	1	-1
micro100	4	0	1	-1
which2	3	0	2	-2

mc2mod

key	ties	win	loss	win-loss-at-99%
which2	5	0	0	0
micro75	5	0	0	0
micro50	5	0	0	0
micro30	5	0	0	0
micro20	5	0	0	0
micro100	5	0	0	0

ar3

key	ties	win	loss	win-loss-at-99%
which2	0	5	0	5
micro75	4	0	1	-1
micro50	4	0	1	-1
micro30	4	0	1	-1
micro20	4	0	1	-1
micro100	4	0	1	-1

ar4

key	ties	win	loss	win-loss-at-99%
which2	5	0	0	0
micro75	5	0	0	0
micro50	5	0	0	0
micro30	5	0	0	0
micro20	5	0	0	0
micro100	5	0	0	0

ar5

Figure 4.32: MannWhitney U−Tests for Micro−sampling.

micro−sampling, at least for mc2mod, drastically improves the performance of the Which machine learner on this data set. This is, however, only one instance where this type of standout result occured and we are not claiming it as support for micro−sampling being an effective strategy for improving the performance of machine learners.

learner	mean	Equal	Description
which2	70.9		
micro20	67.0	<	
micro100	65.0	<	
micro30	64.1	=	
micro50	64.8	=	
micro75	64.1	=	

Figure 4.33: Overall Quartile Chart for Micro−Sampling.

key	ties	win	loss	win-loss-at-99%
which2	1	4	0	4
micro20	5	0	0	0
micro75	4	0	1	-1
micro50	4	0	1	-1
micro30	4	0	1	-1
micro100	4	0	1	-1

Figure 4.34: Overall MannWhitney U−Tests for Micro−Sampling.

Figure 4.33 and Figure 4.34 show the overall performance comparison results for Which2 and the micro−sampled variants. Counter to our conclusion in the discussion above about the second category of the micro−sampling experimental results, Which2 has the best performance compared to its micro−sampled variants. The only variant that ties with Which2 is Micro20, according to the MannWhitney U−Tests. However, according to Figure 4.33, the means and quartiles of the variants are all very similiar. While it is clear from the figure that Which2 has the best mean value, it is visible here that the magnitude of difference between the means and quartile distribution between the normal Which2 and the micro−sampled variance is much smaller. The fact that Which2 tied with Micro20 is interesting because Micro20 reduces the data set to a very small size. The data set pc1 contains 77 defective modules and 1032 non−defective modules. This data set contains 1109 modules in its normal state, yet the Micro20 Which variant, using a data set consisiting of only 40 modules, performs just as well.

This experiment gives interesting insight to the effects of micro−sampling on defect detection data. It tells us that removing both defective and non−defective modules from a data set to produce a much smaller data set does not hurt nor help the performance of Which2. Since micro−sampling is a special form of undersampling, these results are not entirely unexpected. This is an important result because it tells us that with limited data for a project, it is still possible for effective defect

detection.

Chapter 5

Conclusion

This chapter consists of three different parts. Section 5.1 gives a broad overview of all we have discussed in this thesis. Section 5.2 gives our conclusions we arrived at given the results of our experiments. And Section 5.3 describes a few changes that could be done to the Which learner that might possible improve its performance.

5.1 Overview

In this thesis we have discussed several different facets of machine learning. We have discussed some of the different classical learners such as J48, Naïve Bayes , and Ripper. We have discussed different sampling and testing policies such under−sampling and cross−validations. This thesis has discussed lift learning and how it fits into the realm of classification learning. We have disussed some of the different types of discretization methods that are used in this field. We have discussed some of the evaulation methods such as sum of differences, quartile charts and MannWhitney U−Tests that are used in the field of machine learning.

We have discussed in detail a lot of the properties of software defect detection as well as the performance of other experiments in the field. We have introduced a novel evaluation concept known as the Koru Diagram that allows for quick identification of standout performance effects. We experiments with a relatively new concept known as micro−sampling.

The following are our conclusions about the results of the performance of our novel machine learner, Which, as well as the effects of fine−tuning a machine learner to the business task at hand.

5.2 Findings

Many of the results when discussing the classic machine learners appear to align with the general conclusions that of the classic machine learners, Naïve Bayes performs the best [30]. We also show that the simplification of TAR3's rule creations process into Which's does not lower the performance of Which comapred to TAR3.

It is interesting that Which performs very well in the field of defect detection, but not entirely suprising. It can be seen from the various experiments that Which with a simple equal frequency discretization policy outperforms all of the classic machine learners by a large margin. We have explained potential answers to this, but the most intuitive answer is that Which's rule creation heuristic, $h_{balance}$, is fine−tuned for the work in this field. $H_{balance}$ attempts to maximize the rules to be as close to $(0,1)$ as possible on the Koru Diagram, which is what we evaluating the performance of the machine learners on in the end. We have shown by this that being able to fine−tune a machine learner for the business task.

We have taken a fairly recent discovery in subsampling known as micro−sampling and shown its performance compared to the standard sized data set. While we did not show any improvement aside from once special case, we did show that it did not hurt the performance of the machine learner in most cases. This provides useful insight on the amount of data required to effectively learn a prediction rule.

We conclude from all of these experiments that very simply machine learners can perform very well when they are fine−tuned for the business task. It is because of this that we recommend stepping away from the classic machine learners in the are of defect detection and moving more towards specialized software.

5.3 Future Work

Thoughout this thesis there have been a few mentions of potential improvements or modifications to the Which algorithm. The following few paragraphs describe some of these potential improvements.

In the experiment of Section 4.4 a glaring weakness in the potential of lift learners was exploited to illustrate the weakness. We consider the notion that it is entirely possible to create a wrapper around the Which algorithm that could be used to create a classification rule set. This wrapper could simply order the classes in the set in a different way for each call of the Which algorithm. The rules that are created for each of these classes could then be interpreted as a rule set that could then be used for the classification processes. This idea would need extensive testing and potential tweaking to be shown to be valid, and if it is valid would be an excellent improvemnt to the weakness machine learners like Which have for accuracy.

Another future work topic that was mentioned in Chapter 3 is the possibility to use the Which algorithm with some specialized heuristics to discretize the data. The basic idea here would be to divide the data up into a very large number of bins to begin with. Using some heuristic for growing, Which could then attempt to combine the bins of attributes in isolation to create the proper bin ranges and amount of bins for each attribute. This experiment could potentially bring to question the common belief that bins of discretized data need to be contiguous bits of numbers. With the Which discretization method, it is entirely possible for $[1..5) \wedge [9...15)$ to be considered one bin. Extensive experiments could then be made that compare the performance of the classic learners with standard discretization practices to the classic leaners with this novel discretization method.

A possibility for other experiments exists in the parameters of Which. While we performed some rudimentary experiments on the effects of stack size. There are plenty of combonations of parameters that could be tested to attempt to maximize the performance of Which. An example of this is the effects of the CheckEvery and Improvement parameters on data sets with larger and smaller search spaces. It is entirely possible that this phenomenon of the plateau effect that we showed in Chapter 4 is limited to restricted search spaces.

Appendix A

Using Which

The Which application is a command−line tool that has several options that can be customized. The version of Which that we have used in the experiments uses a collection of parameters with the default values. It is entirely possible to change these values to suit the needs of the user. The Which source code and documentation can be downloaded at *http://www.unbox.org/wisp/tags/which/which.tar* and is produced under GPL 3.0.

Which was written in an attempt to mimic the command−line interface of the wEka [17] and retains some similarities with that interface. The following is a table of possible command−line flags that one can use to alter the behavior of Which. Most of these flags are self−explanitory, however a few need further discussion.

The first few flags in Table A.1 are −t, −T, and −E. These flags work together to give Which the information it requires to build and test its rules. It is required for all of the scoring methods to include the −t flag, the other two are optional however. If −T is not specified in the command−line call, Which will use the train set as the test set. −E is used only in special cases where the user wants to use a scoring method that requires the effort (IE effort, pd/loc, and pd/effort) information, but the train and test sets contain discretized data. The data file sent in with −E should be the same as the test file but without the discretized data. This is to ensure accurate scoring calculations.

The got want table mentioned under the description of the −p flag is a special table that is used in evaluating a rule's performace. This table does not give any statistics about the rule's score but

instead lists all of the instances that this rule fired on in a form like Figure A.1. This table can be interpreted as the first column being a counter, the second column being the class the rule predicted the instance to be, the third column being the instance number in the test set that this instance was, the fourth column being the class this instance actually was, and the fifth column being the number of lines of code that instance had. This flag was created when we decided to do our Koru Diagrams for evaluation in Chapter 4. This flag has uses for any of the scoring methods, however. It can be used to calculate the pd and pf for the rule.

1	true	23	true	(34)
2	true	27	true	(30)
3	true	30	false	(2)
4	true	45	true	(1)
5	true	52	true	(5)
6	true	65	false	(101)
7	true	100	true	(12)
8	true	101	true	(423)

Figure A.1: Example Got Want Table.

As a final remark, the −noeff flag is a special flag that was an artifact of the addition of the line of code based scoring models. It should always be set unless the model is an effort model.

Flag	Argument	Description	
−t	FILE	Sets the train file.	
−T	FILE	Sets the test file.	
−E	FILE	Sets the file to read in the lines of code from	
−rep	NUMBER	Sets the number of Rules to report.	
−score	TYPE	Sets the type of scoring.	
		lift	cores the Rules via lift.
		effort	Scores the Rules via effort.
		pdpf	Scores the Rules via probability of detection and probability of false alarm.
		probsupt	Scores the Rules via the equation $\frac{pd^2}{pd+pf}$.
		pd/effort	Scores the Rules via pd/effort.
		ripper	Scores the Rules via Ripper's method according to ROC n' Rule paper.
		precision	Scores the Rules via precision's method according to ROC n' Rule paper.
		pd/loc	Scores the Rules via pd over the sum of the LOC it fires on. LOCs are normalized for this.
		accuracy	Scores the rules via accuracy. This will be a low score as it is synthetic in that we cannot assume a binary class system.
−ssize	NUMBER	Sets the size of the stack(−1 for infinite size.)	
−imp	NUMBER	Sets the minimum improvement required per −check picks. Stopping criteria.	
−picks	NUMBER	Sets the max number of picks for this run of Which.	
−check	NUMBER	Sets how many picks in a row before a −imp case is checked.	
−loc	NUMBER	If scoring type is effort, this is the location in the file of the line−of−code attribute.	
−bins	NUMBER	How many bins to use to discretize the data.	
−alpha	NUMBER	Sets the weight of PD in all scoring methods that use it.	
−beta	NUMBER	Sets the weight of PF in all scoring methods that use it.	
−gamma	NUMBER	Sets the weight of effort in all scoring methods that use it.	
−micro	NUMBER	Creates a data set of even distribution of classes from the train set and removes all but the amount sent in.	
−stat		Output only the stats of each rule.	
−p		Tells Which to print the got want table.	
−noeff		Tells which to ignore effort calculation.	
−rule		Tells which to only print the Rule in a condensed format.	
−seed	NUMBER	Tells which which seed to use for the random number generator.	
−?		Prints this help file.	

Table A.1: Breif Description of the Which command−interface.

Bibliography

[1] Erik Arisholm and Lionel C. Briand. Predicting fault-prone components in a java legacy system. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17, New York, NY, USA, 2006. ACM.

[2] Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005. Member-Thomas J. Ostrand and Fellow-Elaine J. Weyuker.

[3] C. Blake and C. Merz. Uci repository of machine learning databases, 1998.

[4] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[5] W. W. Cohen. Efficient pruning methods for separate-and-conquer rule learning systems. In *Proc. of the 13th IJCAI*, pages 988–994, Chambery, France, 1993.

[6] William W. Cohen. Fast effective rule induction. In Armand Prieditis and Stuart Russell, editors, *Proc. of the 12th International Conference on Machine Learning*, pages 115–123, Tahoe City, CA, July 1995. Morgan Kaufmann.

[7] Johan de Kleer. An assumption-based tms. *Artif. Intell.*, 28(2):127–162, 1986.

[8] Pedro Domingos and Michael J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[9] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, pages 194–202, 1995.

[10] Chris Drummond and Robert Holte. C4.5, class imbalance, and cost sensitivity: Why under-sampling beats over-sampling, 2003.

[11] Marek J. Druzdzel. Some properties of joint probability distributions. pages 187–194.

[12] Fayyad and Irani. Multi-interval discretization of continuous-valued attributes for classification learning. pages 1022–1027, 1993.

[13] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, August 2000.

[14] Peter A. Flach. The geometry of roc space: Understanding machine learning metrics through roc isometrics. In *ICML*, pages 194–201, 2003.

[15] Johannes Fürnkranz and Peter A. Flach. Roc 'n' rule learning: towards a better understanding of covering algorithms. *Mach. Learn.*, 58(1):39–77, 2005.

[16] Johannes Furnkranz and Gerhard Widmer. Incremental reduced error pruning. In *International Conference on Machine Learning*, pages 70–77, 1994.

[17] S. Garner. Weka: The waikato environment for knowledge analysis, 1995.

[18] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 417–428, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Maurice H. Halstead. *Elements of Software Science, Operating, and Programming Systems Series*, volume 7. Elsevier, New York, NY, 1977.

[20] Les Hatton. Reexamining the fault density-component size connection. *IEEE Softw.*, 14(2):89–97, 1997.

[21] Ying Hu. Treatment learning: Implementation and application. Master's thesis, University of Brittish Columbia, Brittish Columbia, Canada, May 2003.

[22] Donald Joseph Boland Jr. Data discretization simplified: Random binary search trees for data preprocessing. Master's thesis, West Virginia University, Morgantown, West Virginia, USA, May 2007.

[23] J. Juran. *Juran's Quality Control Handbook*. pub-mcgraw-hill, 4th edition, 1988.

[24] A. Gunes Koru, Dongsong Zhang, and Hongfang Liu. Modeling the effect of size on defect proneness for open-source software. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 10, Washington, DC, USA, 2007. IEEE Computer Society.

[25] Huan Liu, Farhad Hussain, Chew Lim Tan, and Manoranjan Dash. Discretization: An enabling technique. *Data Min. Knowl. Discov.*, 6(4):393–423, 2002.

[26] M. O. Lorenz. Methods of measuring the concentration of wealth. *Puiblications of the American Statistical Association*, 9(70):209–219, 1905.

[27] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 18:50–60, 1947.

[28] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[29] Tim Menzies. Promise data. http://promisedata.org.

[30] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[31] Tim Menzies, Omid Jalali, Jairus Hihn, Dan Baker, and Karen Lum. Software effort estimation and conclusion stability. 2007.

[32] Tim Menzies, Justin Di Stefano, Kareem Ammar, Kenneth McGill, Pat Callis, Robert (Mike) Chapman, and John Davis. When can we test less? In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 98, Washington, DC, USA, 2003. IEEE Computer Society.

[33] Tim Menzies and Justin S. Di Stefano. How good is your blind spot sampling policy? *hase*, 00:129–138, 2004.

[34] Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang". Implications of ceiling effects in defect predictors. *ICSE PROMISE Workshop*, 1(1), 2008.

[35] John C. Munson and Taghi M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. Softw. Eng.*, 18(5):423–433, 1992.

[36] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Softw. Eng. Notes*, 27(4):55–64, 2002.

[37] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, New York, NY, USA, 2004. ACM.

[38] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*. Morgan Kaufmann, September 1988.

[39] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[40] I. H. Witten and E. Frank. *Data Mining*. Morgan Kaufmann, second edition, 2005.

[41] I. H. Witten and E. Frank. *Data Mining: Practicle Machine Learning Tools and Techniques*. Morgan Kaufman, second edition, 2005.

[42] Ying Yang and Geoffrey I. Webb. A comparative study of discretization methods for naive-bayes classifiers.

[43] Ying Yang and Geoffrey I. Webb. Proportional k-interval discretization for naive-bayes classifiers. In *EMCL '01: Proceedings of the 12th European Conference on Machine Learning*, pages 564–575, London, UK, 2001. Springer-Verlag.

[44] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Trans. Softw. Eng.*, 14(9):1261–1270, 1988.

[45] Y.Yang and G.I. Webb. On Why Discretization Works for Naive-Bayes Classifiers. In *AI 2003: Advances in Artificial Intelligence*, pages 440–452. Springer Berlin, 2003.

[46] Hongyu Zhang. On the distribution of software faults. *IEEE Trans. Soft. Eng.*, xxx(xxx):1–2, 2008.