CO7201 MSc Individual Project
Computer Science Department

University *of* Leicester

Department of Computer Science

University of Leicester

Learning Program Models by Observing Their Behavior

Dissertation

By: Sunil Nair Kolaserry Mohan

Email: snk13@le.ac.uk

Supervisor: Neil Walkinshaw (nw91)

Second Marker: Stanley Fung

University ID: 099024895

Date of Submission: 20/05/2011

Abstract of the thesis titled

"Learning Program Models by Observing Their Behavior"

Submitted by

Sunil Nair Kolaserry Mohan

For the degree of Masters in Advanced Software Engineering at University of Leicester, UK in May 2011

Testing large complex system is expensive and highly resource consuming. Studies estimate that about 30 % to 50 % of the resources in an average software project are typically allotted to testing alone. To add more fuel to the fire, many of these bugs remain undetected due to poor testing strategies that lead to disastrous consequences. In spite of such huge resource allocation, the unreliability is due to the fact that Software testing is typically a Manual process. Documentation is an effective way to identify the behaviour of the system, but it is hardly up to date or written by people who do not understand the system. Without a guide to effectively test the system, the human tester is forced to use intuition to select test cases that exercises the full system.

The project will aim to develop a technique to automate testing of unfamiliar systems. There are already tools that can do this, but these are not particularly accurate, and can often end up producing rules that are misleading. The technique introduced – ILUSTRATOR (**I**nductive **L**earning **US**ing **T**ests by **RA**ndom genera**TOR**) is based on novel machine learning techniques to infer models from unfamiliar systems. Inference by induction is the key concept dealt in this project. The outcome of the project will be a rigorous, automated test case generator for unfamiliar black-box systems, supported by inductive testing and novel machine learning techniques that can be applied to a larger and realistic software system.

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specially acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do these amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Sunil Nair Kolaserry Mohan

Signed: Sunil Nair Kolaserry Mohan

Date: 20/05/2011

Acknowledgments

It is probably needless to say there have been too many people to thank for their assistance in the course of my postgraduate work and the presentation of this thesis. Nevertheless I will try to acknowledge all of them.

First of all I would like to thank my supervisor Dr. Neil Walkinshaw for his unrelentless support, guidance and encouragement and in introducing me to this topic. He has often come up with mind blowing ideas and suggestions that led the way when it seemed there was no further progress. He has always directed me towards the right path when I was lost.

I would also like to thank Dr. Stanley Fung, who spent his valuable time in providing feedbacks and improving the overall understanding of the thesis. With his help I was able to make this thesis understandable to everyone.

I would like to thank the Department of Computer Science, the University Of Leicester, UK, for the financial support in the form of postgraduate scholarship. The support from the department all throughout the course has been immense and has to be mentioned.

Finally I must extend my deepest respect to my parents, whose support in every step of my life although being invisible is of utmost importance. I must also thank Sree Priya for her never ending ocean of love which kept me swimming.

Contents

List of Figure

Chapter 1

1. Introduction

1.1. The Problem

Programs could be large, complex and hard to understand. They could be thousands of lines long and the structure could be complex with many functions, methods or modules. The different methods will have individual functionality which contributes to the overall behavior of the system. Testing such large and complex system can be complex. Understanding their functionality to test them effectively is a tedious process. While trying to understand the programs behavior, the developer can easily lose track of the individual functionalities. This often results in assuming certain functionality of the modules or even worse ignoring them.

1.2. Motivation

Testing large complex system is expensive and highly resource consuming. It is estimated that about 30 % to 50 % of the resources in an average software project are typically allotted to testing alone [1]. Three quarters of organizations (77%) claim that testing is an essential investment, but more than half (55%) admit they do not have a consistent approach to software testing. In spite of such huge resource allocation, certain bugs remain undetected and lead to disastrous consequences. This unreliability is due to the fact that testing is primarily a Manual task.

Trying to read the code manually will result in massive time consumption. Documentation usually gives a fair amount of understanding about the overall systems behavior, but programs are hardly well documented. They are either not clear or not up to date or documented by people who don't understand the system. In Worse case, documentation is ignored. Lack of coding standards adds more fuel to the fire.

"A consistent set of coding standards will result in - 30% fewer bugs - 30% faster development - 80% lower maintenance cost. Above all 100% of us coders will be far happier with this change." – Unknown

Without a guide to effectively test the system and to compare against, the human tester is forced to use intuition to select test cases that exercises the full system. The motivation behind the project was this same fact that testing

unfamiliar systems is forced to explore them in terms of its externally observed behavior. Generating a model of the system based on its behavior and testing them rigorously with expected inputs and corresponding outputs would build test cases that exercise the entire functionality of the system. By probing the systems behavior we build up the test cases along with the model which guides the developer to improve understandability of the unfamiliar System.

1.3. Aim & Objectives

The aim of the project will be to develop a technique called ILUSTRATOR (**I**nductive **L**earning **US**ing **T**ests by **RA**ndom genera**TOR**) that would help a developer to understand a large complex program and test it using *Inductive testing*. The technique will learn rules of a program behavior, by watching it execute. This is otherwise known as *Dynamic analysis*. The original aim of the project was to use *Inductive Logic programming* for this purpose and to develop a tool which will analyze existing programs, and infer the rules using existing ILP framework (Called Progol). The task was to extract traces from the programs we want to analyze, and to recode them in a suitable manner for Progol to analyze. This however failed and details will be discussed later (Section 1.4.2) in the paper.

Inductive testing is based on the idea of inferring models inductively from a program execution. It is a combination of model inference and software testing, in which we probe the system's behavior by execution and build a hypothetical model. The key challenge of inductive testing would be to find a finite training set that hold enough information about the system's behavior and which could be used to generate a sufficiently accurate model. This can be obtained only when there is a broad range of examples.

The idea proposed is not new but none of the previous attempts were successful. Tools based on these techniques have been developed in the past but they were not accurate in inferring models or often produced misleading results. In the past, I nductive testing focused greatly on the inferred models and ignored the suitability of test cases generated in the process [2].

This project will be based on *Inductive inference* [3], which is the process of hypothesizing *a model* from a set of examples of the system. Software testing and Inductive Inference are like two sides of a coin. The former generates a finite set of test cases that attempts to fully exercise the system while the latter attempts to create a model of the system from the finite set of examples. In reality Inductive Inference is the reverse of software testing.

The hypothesized model inferred from the system will play an ideal role in testing the system. Since software testing and inductive inference are tightly coupled, it gives us the opportunity to exploit the usage of both to create a *"virtuous loop"*. The model will be generated by inferring the subject system and will create test samples which will be fed back to the model to make it a perfect representation of the system. Once the model based tester cannot find any contradictory test cases, the test set can be defined as an adequate sample. The continuous generation of test cases that try to contradict the inferred model would help in refining the model and provide a sufficiently comprehensive test set. The end product of the project will allow us to see an accurate representation of the systems rules that can be widely used for activates like testing and documentation. The model based tester can make use of the model to understand the behavior of the system instead of reading the code, hence understand the program functionality.

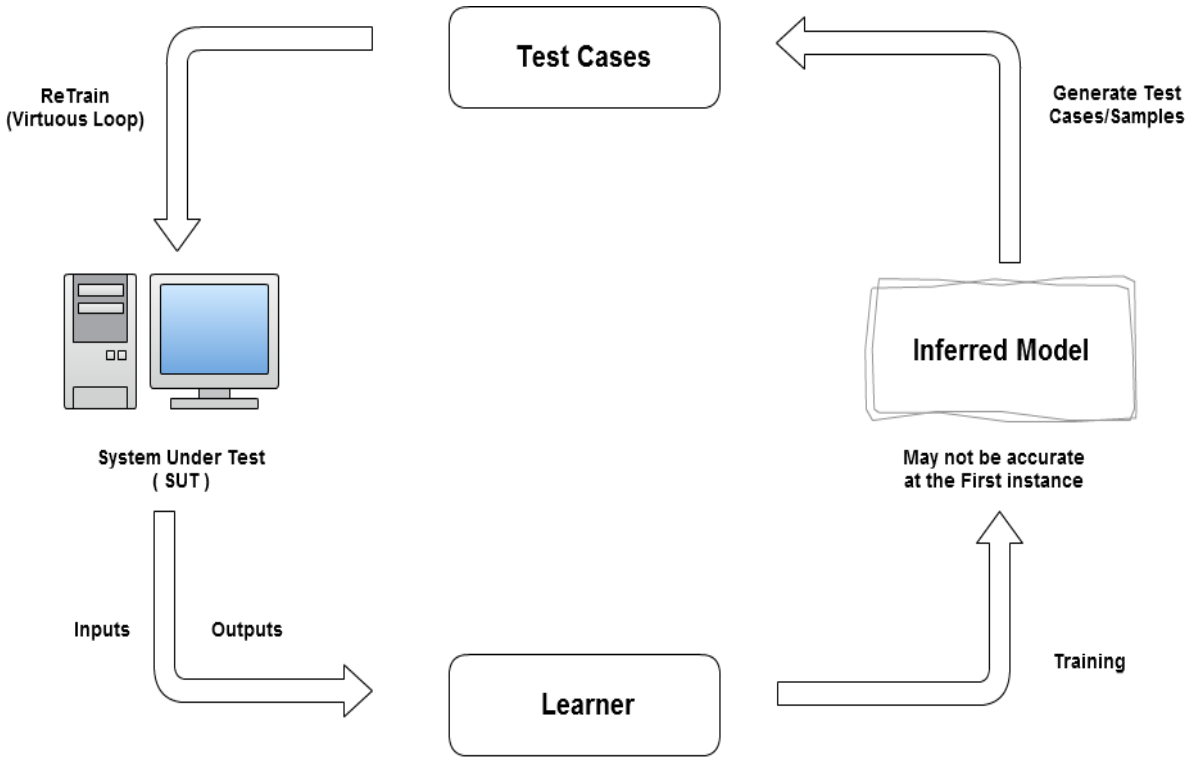

Figure 1. Learning by Inference

The grand aim of the project will be to successfully complete the virtuous loop for a given subject system which is complex and considerably large. The process begins by extracting traces of program execution from the subject system which will be converted into a learner understandable form. Ideally the system will be given random inputs and will produce the expected outputs which will be fed to the learner. The learner will infer these set of finite samples (combination of inputs and outputs) otherwise known as datasets and generate an inferred model from them. The model inferred at the first instance may not necessarily be a perfect representation of the system. Once the model is trained with a considerable amount of sample datasets, it is tested with more sample inputs that are randomly generated and corresponding outputs generated by the SUT. We expect the model to generate a series of predicted and actual values for the outputs by inference. This method of testing the model will attempt to find new program executions that contradict the model. The contradictions are fed back to the model to learn more about the system hence creating the virtuous loop. The idea is that a sufficiently comprehensive test set can be "grown" by progressively inferring increasingly accurate models of the system.

As mentioned before the model generated may not be a perfect representation of the SUT and this would result in some erroneous predictions. The key to achieve a near to perfect model lies in re-training the model with these misclassified samples, making the model learn from its own mistake. So a virtuous loop is created which would supply these incorrect predictions back into the training dataset and re-train the model until there are no erroneous predictions. The model is perfect when for any value of the actual output; the predicted output is the same.

1.4. Background

In view of the importance of inductive testing, much research has been done. Before we state the results we achieve, we give in this section a brief overview of some related previous work.

1.4.1. Inductive Inference

The basic idea of inductive inference was proposed by Weyuker [4] in 1983, in her paper "*Assessing Test Data adequacy through Program Inference",* where she builds a bridge between inductive inference and model based software testing. She proposed the idea of using machine leaning techniques to infer programs and generate a model from a finite set of examples. She realized that testing and inductive inference is tightly coupled. A perfect model can be

inferred only with a sufficient set of examples and a finite set of examples can yield a perfect model.

Weyuker suggested that this relationship can be exploited and this would form the foundation for what we refer as inductive testing. Her work was followed by a series of other papers which focused on the idea of combining inductive inference and testing [5-14].

"*Inductive Inference and Software Testing*" [6] by H. Zhu, P. Hall, and J. May held a key progress in this field of study. Their paper focused on the study of relationship between inductive inference and software testing to see if inductive inference can be used as a theoretical foundation for testing. Their theoretical idea is turned into a working technique in this project.

1.4.2. Inductive Logic Programming

Initially the project was planned to be based on Inductive Logic Programming (ILP). The basic idea was proposed by S.H.Muggleton in his paper "Inductive Logic Programming", *New Generation Computing* [15]. It was aimed to be achieved by inferring rules of a system using existing ILP framework such as Progol [16]; to extract traces from the programs we want to analyze, and to recode them in a suitable manner for Progol to infer. The plan [17] was to provide a set of positive examples and negative examples which when combined with the background knowledge and fed to Progol, it would give us a set of rules about the program execution.

Although it is achievable, the Progol program failed to capture the SUT's traces effectively. The inferred hypothesis was either wrong or misleading. Progol 4.4 was used to infer rules of a simple Multiplication program which would calculate the product of two real numbers. But it failed to capture the behavior of the program and inferred traces were wrong. The documentation in the Progol homepage for multiplication program was performed with Progol 4.1. But this version had errors in the main class originally written in C.

The major drawback was that there was not enough documentation of the Progol system. It was developed but not well documented and maintained. The available version of Progol worked well only with specific examples narrowing the scalability of this project. Frequent changes made in the source code introduce new bugs in Progol and lack of proper` documentation support makes it difficult to debug them. All this gave way to exploring a more commonly used machine learner – WEKA which will be discussed in detail in the upcoming section.

1.4.3. WEKA – A Machine learning suite

Various Machine learning tools are available which would infer a model from a given set of examples. One such widely used tool is *WEKA* (*Waikato Environment for Knowledge Analysis*). WEKA is a popular and an open source suite of Machine Learning Software written in Java [18]. WEKA contains a large repository of learning algorithms for data analysis and predictive modeling. Since written in java, WEKA can be used through Java API. To perform predictive modeling, the set of SUT examples should be provided in a WEKA understandable file. WEKA takes inputs either in a *CSV* (*Comma Separated Values*) file or an *ARFF* (*Attribute-Relation File Format*) file. This technique produces an ARFF file which is an ASCII text file that describes a list of instances sharing a set of attributes [19].

An ARFF file has two distinct sections – Header information and Data information. The Header of the ARFF file contains the name of the relation, a list of the attributes and their types.

Let us consider a simple *Body Mass index* (BMI) calculator as a subject system. It is a unique example as it is a combination of both numerical and statistical data. The System takes Height (cm) and Weight (Kg) as inputs and will calculate the BMI. It would then categorize the index into one of the following: Underweight, Normal, Overweight, Obese and Severely Obese.

An example header on the BMI dataset is shown in Fig 2:

```
% 1. Title: Body Mass Index Calculator
%
@RELATION  bmi

@ATTRIBUTE height NUMERIC
@ATTRIBUTE weight NUMERIC
@ATTRIBUTE index  NUMERIC
@ATTRIBUTE result  {Underweight, Normal, Overweight, Obese, Severely_Obese}
```

Figure 2. BMI dataset

Lines that begin with % are comments and are not considered. The @relation defines the relation name. The @attribute statement defines the name of the attribute and its data type. There are four data types currently supported by WEKA:

- Numeric
- String
- Date
- Nominal-Specification

Nominal values are defined by providing an <nominal-specification> listing the possible values: {<nominal-name1>, <nominal-name2>, <nominal-name3>…}

For example, the class value of the BMI dataset can be defined as follows:

```
@ATTRIBUTE result {Underweight, Normal, Overweight, Obese, Severely_Obese}
```

The Data of the ARFF file contains potentially infinite examples separated with commas. The data has to be supplied in the same order it is defined in the Relation section. An example of the Data is shown in Fig 3.

```
@DATA
9,21,182.0,Severely_Obese
6,43,841.0,Severely_Obese
81,102,10.0,Underweight
35,82,47.0,Severely_Obese
29,91,76.0,Severely_Obese
83,24,2.0,Underweight
89,182,16.0,Underweight
58,121,25.0,Overweight
13,9,37.0,Obese
```

Figure 3. Data section – BMI dataset

As mentioned before, the order of the attributes declared indicates the column position in the data section of the file. So the first column is the Height, followed by the weight, index and result.

1.4.4. The Learning Algorithm – C4.5

For WEKA to infer a model out of the ARFF file, we would have to choose an appropriate leaner. Since WEKA has a large collection of machine learners, the task of choosing one is not simple. None of the learner algorithm is inherently better than the other and which one is most appropriate tends to be context dependent. Some of the learners focus greatly on the classification of data served, which is apparently the problem in this project. We want to learn the relationship between inputs and corresponding outputs which would yield the behavior of the system.

One such widely used classification rule generator which is easily amenable to interpretation is *C4.5 decision tree algorithm*. The tree like graph represents

decisions and their consequences allowing us to model the basic behavior of the SUT. Developed by Quinlan [20] it is an extension of his earlier ID3 algorithm [21]. C4.5 algorithm generates pruned and un-pruned decision tree which is interpretable of the systems behavior: certain conditions imply certain output equivalence class. And in our BMI example this equivalence class is the category of the BMI calculated. C4.5 builds trees from a set of training data by entropy i.e. using each attribute of data which can be used to make decisions by splitting them into smaller subsets [22].

At each node of the tree, C4.5 chooses one attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. The criterion is *the normalized information gain* (difference in entropy). *Entropy* is the measure of unpredictability. In simple terms whenever C4.5 encounters a set of items in the dataset provided, it identifies the attribute that discriminates the various instances most clearly. The feature that gives us most information about the instance so that we can classify them the best is said to have the highest information gain. The attribute with the highest normalized information gain is chosen to make the decision. The Pseudo code [23] is as follows:

1. Check for base cases
2. For each attribute a, Find the normalized information gain from splitting on a
3. Let a_best be the attribute with the highest normalized information gain
4. Create a decision node that splits on a_best
5. Recur on the sub lists obtained by splitting on a_best, and add those nodes as children of node

Figure 4. C4.5 Pseudo code

For the BMI example, and based on a sufficient sample dataset, a tree generated by C4.5 algorithm in the context of WEKA tool is shown in Fig 5.

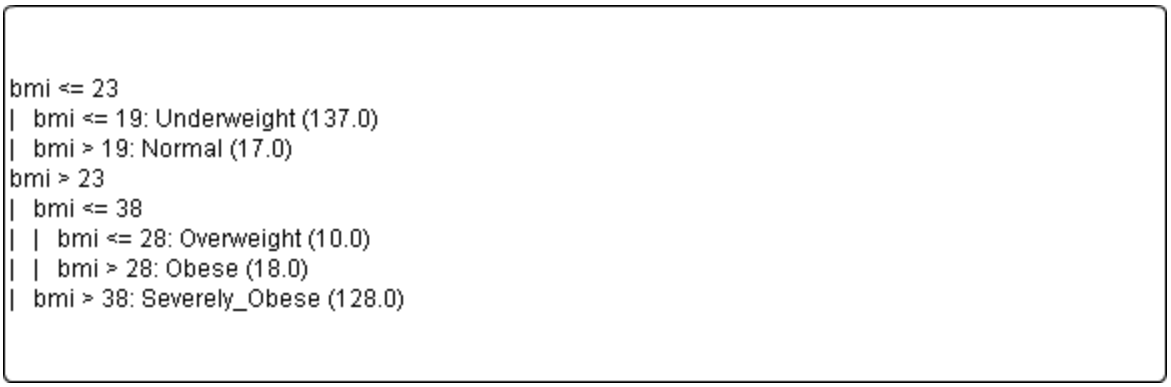

Figure 5. BMI DT

This should be read as follows: if the *bmi* value is <=23 we split the tree into two branches. If *bmi* <=19 then the category is *Underweight* or bmi >19 it is *Normal.* On the other branch, if *bmi* is <=38 we split the nodes further: bmi <= 28 as *Overweight*, *bmi* >28 as *Obese*. If *bmi* is >38 the category is *Severly_Obese.* The numbers in the brackets next to each category is the number of instances that were classified in that particular category.

C4.5 handles both continuous and discrete attributes and predicts missing values and attributes. The user can specify parameters like whether to use binary splits on nominal attributes, the minimum number of instances per leaf, whether pruning is performed, the confidence factor used for pruning (smaller values incur more pruning), amount of data used for reduced-error pruning (one fold is used for pruning, the rest for growing the tree), etc. It is widely known for its high accuracy compared to Naïve Bayes or Support Vector Machine (SVM).

1.5. Organization of the Thesis

The rest of the thesis is organized in the following manner. In Chapter 2, we discuss the overview of the steps involved in the ILUSTRATOR methodology. In Chapter 3, we discuss the tool developed from the technique which is used to infer models from subject system and test them to generate test cases. We will also discuss about how WEKA along with C4.5 is used to achieve this. In chapter 4 we would discuss the evaluation of the technique with two typically large and complex subject systems such as the Tax calculator and Credit Rating System. In Chapter 5 we discuss other related methodologies and works that follow a different technique to generate test cases. We also evaluate the technique introduced in this thesis and draw out some results. The final chapter concludes this thesis.

Chapter 2

2. Specification and Design

2.1. ILUSTRATOR - Overview

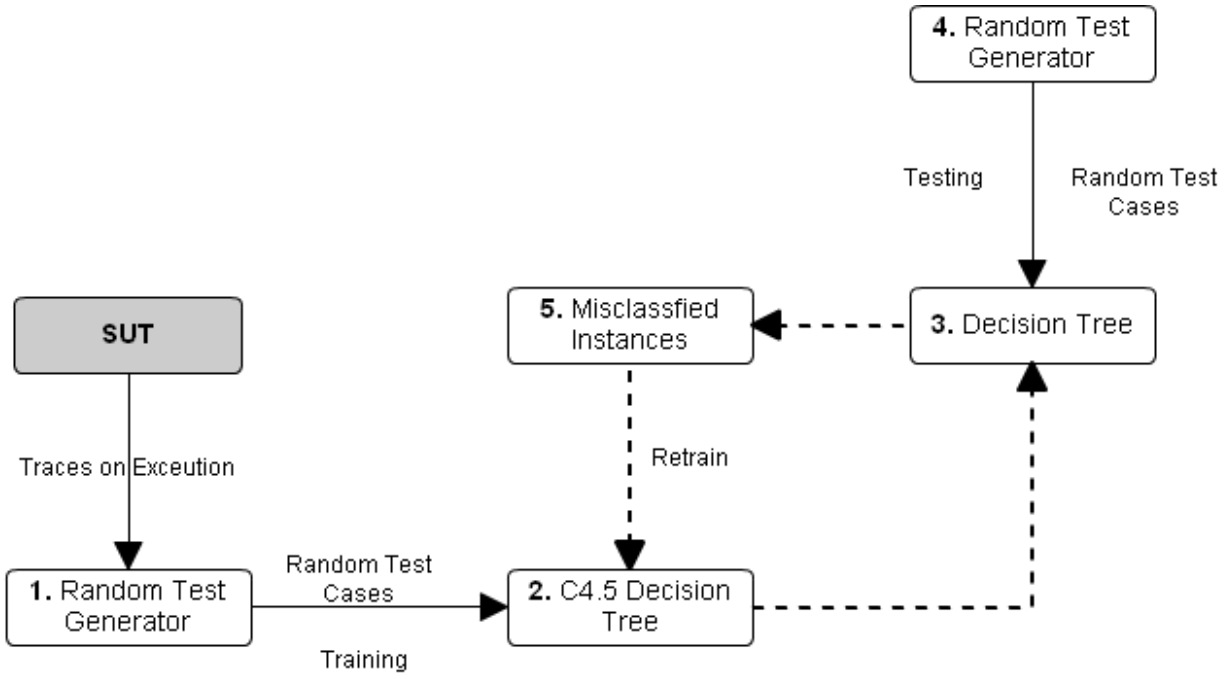

Figure 6. The ILUSTRATOR Methodology

Figure 6 gives an overview of the steps involved in ILUSTRATOR (**I**nductive **L**earning **US**ing **T**ests by **RA**ndom genera**TOR**) methodology. We will describe this in detail in the following section.

2.2. Iterative Learning

The technique relies on the fact that continuous learning in a loop will increase the knowledge of the model paving way to new branches and paths. Moreover a model which learns from its own mistakes is considered to be more knowledgeable. ILUSTRATOR uses this technique as it retrains the model generated with misclassified instances creating a loop which tests the model till

there are no misclassifications. The entire process is automated without any human intervention.

The model generated by C4.5 is represented in the form of decision tree (DT) where the paths from the root node of the tree to any leaf can be considered as a rule. The set of examples provided to the learner are traces of the SUT execution. The learner would infer a tree based on these traces. The tree may not be the accurate representation of the subject system and many branches of the tree may be missing. This is because the initial training set provided does not contain enough information about the system for the learner to represent them accurately. To obtain a perfect tree which captures the behavior of the system for all possible inputs, the leaner has to be fed with more knowledge about the systems reaction to these possible inputs. Generating a tree from a small training sample would not mean anything unless we test the tree obtained. The idea is to test the tree obtained with more program traces generated from the SUT and see how the tree predicts the output. For any misclassified instances that the tree predicts, the particular sample from the testing dataset is fed back to the training dataset for the learner to learn from its mistake. This process of testing and training the model continues until there are no misclassified instances predicted by the model.

The improvement process is an iterative process (in Fig. 6) represented in dotted lines. New test cases added to the training set will lead to identification of new problems and misclassifications. The size of the training set at each iteration increases as new instances are added to it from testing the model. The entire process will be repeated in a loop with the training set size increasing in iterations until there are no more contradictions found in the rules learnt by the machine learning algorithm.

The process of extracting program traces from the SUT is done with a random test generator (Activity 1). It is an automated process and it generates a collection of SUT's input and corresponding output. This is generated by using a random number generator which creates random inputs and their corresponding outputs. This is written in an ARFF file separated by commas (Section 1.4.3). The random test generator is written in Java using the *Class Random* from the *java.util.Random* Package. The dataset generated randomly is fed to the C4.5 learner algorithm (Activity 2) which treats this dataset as a training set and builds a decision tree (Activity 3). During the initial loop of learning we generate only a small fraction of random inputs and corresponding outputs. This is due to the reason that we want to see how the learner learns from small increments. The size of the initial training set is dependent to the

SUT as we may not achieve a significant tree to test for a large complex system with a few random inputs.

Once we have an initial model from the training set, we need to test them to see what behavior of the SUT the learner has inferred. WEKA allows us to save the model generated from a dataset and then later test them. The code sample of this process will be dealt with in the oncoming sections. Testing the model generated typically follows the same steps as training. We generate more traces of the SUT using another Random test generator (Activity 4) and test the model using these. When tested using WEKA, the learner predicts the particular attribute mentioned during testing and gives us a table of predicted and actual values. Any misclassification of attributes predicted is displayed in this table with the corresponding instance number. The process of making the model more perfect relies greatly on these misclassifications and how we retrain the model with these misclassifications. Activity 5 does exactly this by adding the misclassified instances back into the training dataset. This creates the loop of iterative learning: a continuous and regress testing and training of an inferred model until we achieve a potentially perfect model.

The learner algorithm is repeatedly executed with new training samples so as to infer a better model. This is generally a quick process as C4.5 Decision tree algorithm takes only a few seconds to infer a model from a few thousand test cases. The final model obtained when there are no more misclassifications, represents the accurate behavior of the SUT for a wide range of inputs. This model can be used by the developer to improve the understandability of the subject system without having to look into the source code. The Decision Tree gives a clear picture of the subject systems behavior for varied inputs and is easy to interpret. The more important result obtained from this iterative process is the training dataset, which acts as a test suite for the subject system. This collection of random inputs and corresponding outputs gives the subtle and complex relationship between the data variables of the system and the impact of these relationships on its subsequent behavior.

The next chapter will speak more in detail about how to generate and train models suing WEKA command line options. We will also see how testing of inferred model takes place and how to obtain the continuous iterative loop of learning from them. We will be using the Body Mass Index (BMI called henceforth) as a running example for this purpose.

Chapter 3

3. Theory to Tool

3.1. Model Generation and Training

WEKA can be used with ease through a graphical user interface called *Explorer*. An ARFF file can be read using a series of menu selection and form filling and a model can be inferred from it. Sensible default values ensure that you can get results with minimum effort [24]. There are two other graphical user interfaces to WEKA namely – *The Experimenter* and *The Knowledge Flow*. The former is designed to help provide answers to choose methods and parameter value that best suites the given problem when applying classification techniques. The latter allows designing configurations for streamed data processing [24].

Along with the Graphical User interfaces, WEKA also provides the simple and basic command line interface. The command line interface provides access to all the features of WEKA. Alternatively, WEKA can also be accessed by setting the CLASSPATH environment variable through the operating systems command line to run classes in *weka.jar*. We will be using the WEKA command line interface for generating model for a BMI dataset and test it rigorously.

As mentioned before the BMI dataset for inferring an initial model is generated using a Random Test Generator. The BMI calculator program is written in Java which generates random heights and weights. The BMI is calculated from these values and corresponding category is chosen. The random height and weight along with the BMI calculated and the category is written into an ARFF file *(BMI.arff)* as mentioned in the section 1.4.3.

A small code snippet of generating random inputs is shown in Fig. 7.

```
height= randomGenerator.nextInt(100);
weight= randomGenerator.nextInt(200);
BMI = 704.5 * weight/(height*height);
BMI = Math.round(BMI*100)/100;
if (BMI >= 39)
{
 cat = "Severely";
}
else if (BMI >= 29)
{
 cat = "Obese";
}
else if (BMI >= 24)
{
 cat = "Overweight";
}
else if (BMI <= 19)
{
 cat = "Underweight";
}
else
{
 cat = "Normal";
}
```

Figure 7. Code Snippet for BMI Calculator

We generate only a few sample instances for the first iteration, say 10 instances and we try to infer a model from them.

To generate model from an ARFF file using a specific classifier, we can call the Java Virtual Machine and instruct it to execute the classifier. Since we are using the C4.5 DT classifier, we call the J48 classifier, which is the java implementation of the Quinlan's C4.5 Algorithm. WEKA is organized in packages that correspond to directory hierarchy. J48 resides in the package *trees,* which is a sub-package of *classifiers*, which is in turn a sub-package of the overall *weka* package. The *classifiers* package contains a number of classifiers and numeric predictors. We are keenly interested in the *tree* sub-package as it is easy to interpret.

```
java weka.classifiers.trees.J48 -C 0.25 -M 2 –t C:\Users\Sunil\Desktop\BMI.arff -d
C:\Users\Sunil\Desktop\BMI_Model.model
```

Figure 8. Model Generation

In the above code, *-t* option is used to instruct the learning algorithm J48 about the location of the training dataset to infer model from. The *–d* allows us to save the model generated from the training data to a particular location which can be later used to test the model inferred on an independent test set. There are many other options that can be used with any learning scheme. Some of these values are defaulted if not mentioned. For example the –C specifies the confidence threshold for building the tree and pruning it and the –M specifies the number of instances in any leaf. We set the confidence to the default value of 0.25 and the number of instances per leaf as 2.

WEKA infers a model out of a given training sample using a technique called *Cross-Validation* [25]. Cross-validation is a statistical method of evaluating learning algorithms and guarding testing hypotheses suggested by a set of data, when there is no independent testing data available or it is costly to obtain one. It splits the data, once or several times into two segments: one used to train the model and the other to evaluate the inferred model. The most common form of Cross-validation is the *k-fold Cross-Validation*. Other forms of Cross-Validation are typically special cases of *k-fold* cross-validation with repeated rounds over a number of times.

In *k-fold* Cross-Validation the original data is randomly partitioned into *k* subsamples or folds. Out of these *k* subsamples, one subsamples is retained for testing or evaluating and the remaining *k-1* subsamples are used for training the model. Then the cross-validation is repeated *k* times (folds), with each *k* subsample used at least once for validation [26]. Fig. 9 gives an example of 3-fold validation with k = 3. In the figure the lighter sections are treated as validation data and the darker sections are treated as training data.

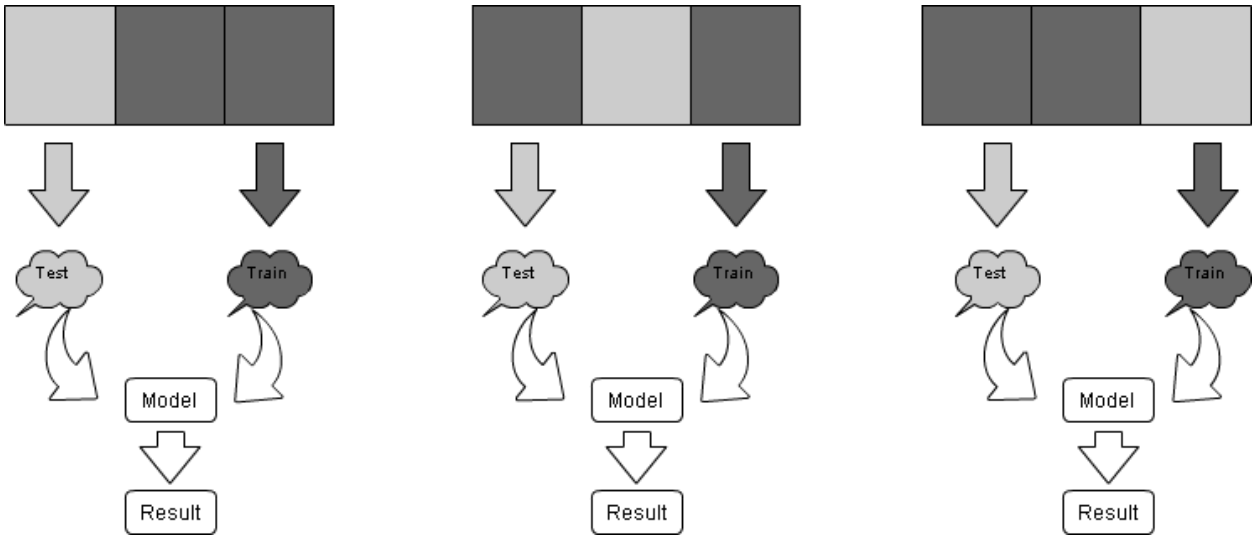

Figure 9. 3-fold Cross Validation

The most extensively used k value is *10* and *10-fold* cross-validation is regarded default in the machine learning and data mining community. Extensive tests and theoretical proofs on a large number of different data's have proved *10-fold* cross-validation to be a right number of folds for error estimation. Though *10* is no magic number, for larger datasets even *3-fold* cross-validation is accurate. The choice of folds depends greatly on the size of the dataset. In our BMI example we use the default *10-fold* validation to perform the initial inference. Moreover we test the inferred model separately with a specially dedicated and independent test dataset, so the number of folds does not greatly influence our results for this particular example.

With 10 random samples generated by the random test generator for the BMI example, the initial tree inferred from them is shown in Fig. 10. The next section will discuss how to test this model and generate a potentially perfect model by iteration.

```
J48 pruned tree
------------------

bmi <= 22: Underweight
bmi > 22: Severly

Number of Leaves  :  2

Size of the tree :  3


Time taken to build model: 0.03 seconds
Time taken to test model on training data: 0 seconds
```

Figure 10. DT for BMI example during 1st iteration

3.2. Model Testing

After the first iteration of training and model inference, we need to test it with more program traces to see what behavior of the SUT has been captured effectively. This is done by generating more random test samples with a test generator and feeding the same to the model to obtain predicted results. This is typically activity 3 and 4 in the *ILUSTRATOR* methodology. The idea behind testing the model is to find the missing behavior that the learner failed to infer due to small training data it was fed. Expecting the model to be perfect at the very first instance is too much of an asking, given a small training set. The model gets refined and closer to the SUT only if it has been trained with a finite set of samples which capture the behavior of the SUT fully.

A similar ARFF file to the *bmi.arff* is generated for testing the model saved during generation. This is can be done by instructing the Java Virtual Machine to test the model as follows:

```
java weka.classifiers.trees.J48 -p 4 -l C:\Users\Sunil\Desktop\BMI_Model.model -T
C:\Users\Sunil\Desktop\test.arff
```

Figure 11. Model Testing

Since we are reusing the generated model, the *–l* tag allows us to reuse the saved model during generation. The *–T* specifies the Test file for evaluating the model with an independent test set. If not mentioned, WEKA uses the cross-validation technique to validate the model. The test file is usually similar to the training set. Attributes which are to be predicted by WEKA can be specified with "?" and are called Missing Values. If we want to compare actual values and predicted values through inference then we need to supply the test set without missing values. By default the class is the last attribute in an ARFF file, but we can declare another one to be the class using the *–c* followed by the position of the desired attribute.

The *–p* followed by the number of the attribute specifies which attribute is to be predicted. We use *4* as *result* is the fourth attribute which needs to be predicted from the inferred model. When tested with an independent test set on an inferred model, we get a series of predicted and actual values which can be used to identify the misclassified instances.

3.3. Misclassified Instances

The result of running a test on a predicted model contains a table of predicted and actual values. The table contains the instance's number followed by the index of its class value and the actual value, the index of the predicted class value and the predicted value. A "+" is displayed next to the predicted class value if the class was misclassified. These misclassifications are the key in refining the model in the upcoming iterations.

With respect to the BMI example the table obtained on testing the model obtained during the first iteration, we see a number of misclassified instances. For example the actual class value for instance 1 was *Overwieght* but the predicted class value is *Severely_Obese*. This is because of the reason that the model did not have sufficient training sample to differentiate *Severly_Obese* and *Overweight.* During the first iteration (Fig. 4.) the model inferred any BMI value lesser than or equal to 22 is *Underweight* and any BMI value above 22 is *Severely_Obese.* The tree inferred from the training sample supplied did not have enough samples to capture the BMI limit for *Normal* and hence when tested for a *Normal* BMI value, the predicted class value was misclassified as *Severely_Obese*.

The following table is part of the output obtained by testing the model inferred with 10 test cases generated by random generator.

```
=== Predictions on test data ===

inst#     actual      predicted    error    prediction ()
   1       3:Overweig  5:Severly      +          1
   2       1:Underwei  1:Underwei                0.75
   3       5:Severly   5:Severly                 1
   4       3:Overweig  5:Severly      +          1
   5       1:Underwei  1:Underwei                0.75
   6       5:Severly   5:Severly                 1
   7       1:Underwei  1:Underwei                0.75
   8       1:Underwei  1:Underwei                0.75
   9       1:Underwei  1:Underwei                0.75
  10       2:Normal    5:Severly      +          1
```

Figure 12. Error Log for BMI Model

3.4. Retrain

The Key in refining the model lies in retraining the learner with these misclassified instances. The model has to be fed with the correct knowledge of the various assumptions it made during the inference process in the first iteration. To do this the model has to be retrained with the instances that were misclassified while testing it with an independent test set. Retraining the model with just the misclassifications would ignore the correctly classified instances during testing. The model has to be fed with the positive knowledge about inference as well. So the idea is to retrain the model with both the correctly classified and misclassified instances during the testing process hence creating the virtuous loop of testing. This would increase the training set's size considerably therefore allowing the learner to infer the behavior of the SUT in depth.

At the end of the first iteration after retraining the model inferred, the process continues by generating more random test samples and testing the model with the new training set which is training set during iteration 1 + Testing set during iteration 1. This iterative process continues till there are no misclassifications in the inferred model. The model obtained at this stage would be closest representation of the Subject system. The training set obtained along with the inferred model will be a finite set of test cases that the system was tested with. From a testing stand point, these test cases help understanding the systems behavior to various random inputs and how the system handles to these inputs hence producing the corresponding output.

Following our BMI example, the model inferred during the first iteration is fed with the misclassifications obtained during the testing process and retrained. The iterative improvement of the inferred model is shown in Fig 13. With respect to the tree generated it is obvious in the Figure that the size of the tree increases as the iteration increases. The number in brackets at the end of each leaf is the number of instances that were classified in that particular class. At iteration 2 there is a significant change in the tree inferred from that obtained during iteration 1. The learner had information in the training sample to differentiate between *Normal* and *Underweight* category. This was obtained from the testing sample which was used to retrain the model.

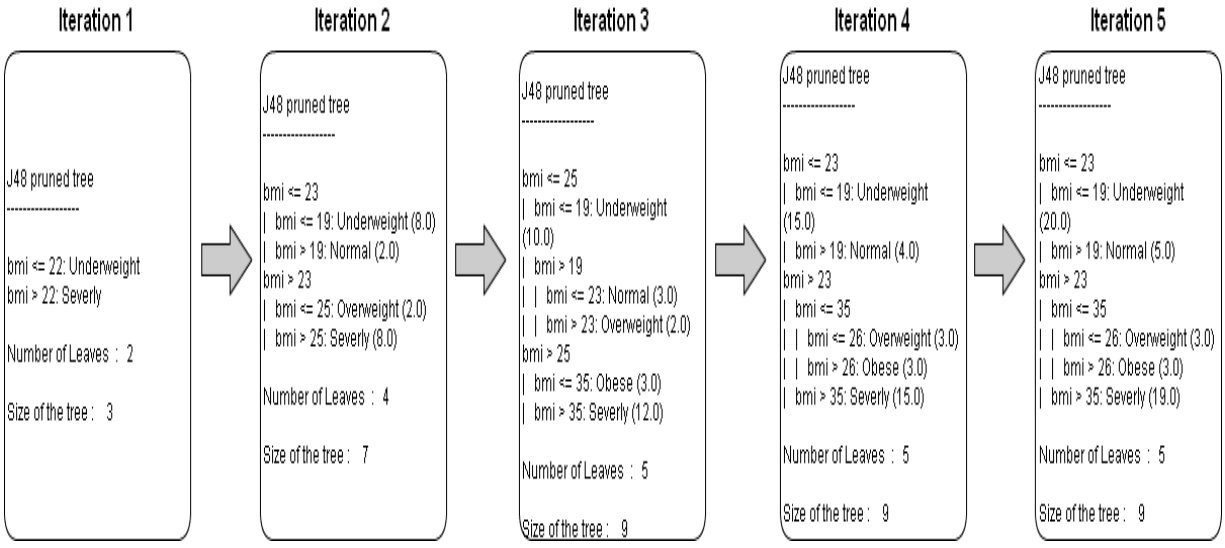

Figure 13. Iterative Improvement of BMI example

During iteration 4, there were no contradictions predicted when testing the model with independent test samples and that is the point when the model was close to being perfect to the subject system. The tree structure did not change during iteration 5 because of the reason that the model did not infer anything new from the training sample. The BMI example was considerably a small system and hence the number of iterations taken to infer a perfect model was less. The case studies (Chapter 5) deals with more complex and larger systems and we will see how the inferred model depends on the number iterations taken.

The ILUSTRATOR technique proved to be a good learning process for the model to learn from its own mistakes. A Learning curve graph (Fig. 14.) was plotted for the BMI example to see how the inference process took place. The graph was plotted with the Accuracy Vs No. of test samples used.

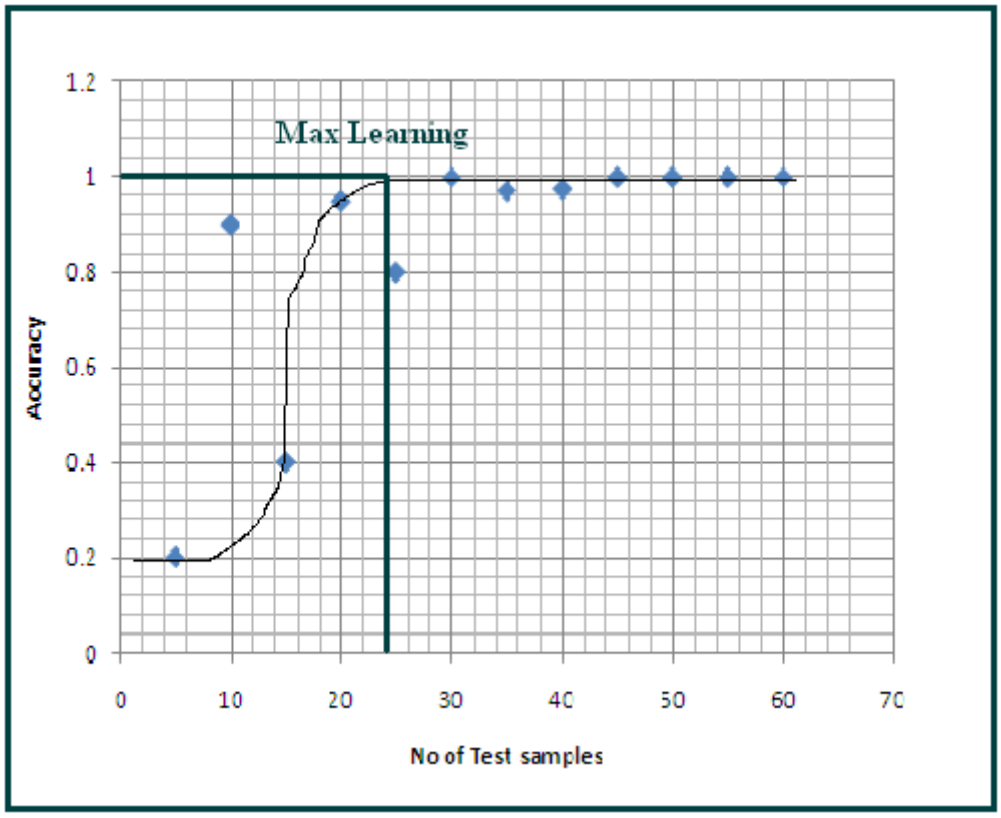

Figure 14. Learning Curve for BMI example

The graph is a classic S-Curve which is highly considered in the machine learning community. It is very evident that the learning process was consistently improving as the number of test sample increased. The graph was plotted without using any particular statistical curve-fitting algorithm.

$$Accuracy = \frac{\text{No. Of Correctly Classified Instances}}{\text{No. Of Test Samples}}$$

At a certain limit in the graph, the learning curve attained its peak threshold (Maximum Learning) and it was consistently stable after irrespective of the number of test samples supplied or tested with.

There are number of other information obtained while generating and testing the model using WEKA. These will be discussed in detail in the case study in Chapter 5. In the next chapter we will see how other similar methodologies namely MELBA and Category partitioning compares to the ILUSTRATOR technique.

Chapter 4

4. Evaluation

We will see the performance and evaluation of ILUSTRATOR technique on 2 large and complex subject systems in this section. We will explain the systems design. This section will also provide you a clear idea of the scalability and flexibility of the technique along with the results obtained.

4.1. Case Study 1: Tax Calculator

The first subject system that ILUSTRATOR was applied to was to a simple tax calculation program [27]. The program computes tax codes and amount of tax payable that includes allowances for a United Kingdom citizen in the tax year April 1998 to 1999. The allowance depends upon the status of the applicant, represented by *Boolean* variables *blind*, *married* and *widowed* and the integer variable *age*. This allowance is no taxed. Tax is charges at the rate of 10%, 23% and 40% for the three tax bands. Except for the 10% tax band which depends on the status of the person, the other two bands 23% and 40% are fixed irrespective of the status.

The source code was an attempt to capture taxation rules that constitutes a governmental business system. The original source code of the system is available in the appendix but for our case study its output was manipulated so as to capture only the band of taxation instead of the actual tax amount. Though the actual tax amount was calculated the models inferred from the training sample focused greatly on the tax band as they can act as class values and hence generate a tree with these as final classes.

The random test generator and the subject systems source code was implemented in Java. The random generator generates random real numbers for age and income. The Boolean variables were randomly chosen using the *nextBoolean( )* method of class *Random* in Java. The program calculated the tax amount and the corresponding tax band. The inputs namely the integer variables (age and income) along with the Boolean variables (blind, married, widow) and the outputs; personal allowance, tax amount and the tax band were written into an ARFF file. A snippet of the ARFF file for the tax calculator is shown in Fig. 15.

This ARFF file was then fed to the learner algorithm C4.5 to infer an initial model. The size of the initial training sample for iteration 1 was chosen as 5

and a model was inferred from them. A random test generator which generates random test sample was also written which produced a test ARFF file similar to the training sample. The size of the test sample was incremented by intervals of 5 at each iteration. So the initial test sample had 5 test cases, followed by 10 for iteration 2, 15 for iteration 3 and so on.

```
@RELATION Tax_Cal

@ATTRIBUTE age      NUMERIC
@ATTRIBUTE blind    {true,false}
@ATTRIBUTE married  {true,false}
@ATTRIBUTE widow    {true,false}
@ATTRIBUTE income   NUMERIC
@ATTRIBUTE personal NUMERIC
@ATTRIBUTE t        NUMERIC
@ATTRIBUTE pc10     NUMERIC
@ATTRIBUTE tax      NUMERIC
@ATTRIBUTE code     {L,H,P,V,T}

@DATA
9,false,false,true,8914,4335,0,3470,6173,L
50,false,false,true,1881,4335,0,3470,188,L
62,false,false,false,2713,4335,0,1500,3674,L
49,true,false,false,3535,5715,0,1500,0,T
33,false,true,true,27917,4335,0,3470,6500,H
```

Figure 15. ARFF training sample for Tax calculator

A 10 fold cross validation was set up for training and testing the model. Two different tests were performed (Test 1 and Test 2) with two random sets of training and testing data. Both the tests produced perfect models which accurately captured the systems behavior. The number of iterations performed to obtain a perfect model was 15 and 13 during test 1 and test 2 respectively.

The entire test result obtained during one of the tests is shown in Fig. 16. The beginning line states the settings used for the learning algorithm; the confidence level and the number of leaves per node. This was set to the default values. Then comes the pruned tree in textual form. The first split is on the *personal* amount calculated and then the second split is on *age* and so on. In the inferred tree, the colon introduces the class label to which the leaf has been assigned to, followed by the number of instances that reach the leaf. This value is expressed in decimals as the algorithm uses fractional instances to handle missing values. The tree size inferred was 17 with 9 leaves. The model was inferred in less than half a minute.

```
Options: -C 0.25 -M 2

J48 pruned tree
------------------

personal <= 5416
|   age <= 64
|   |   married = true: H (77.0)
|   |   married = false: L (119.0)
|   age > 64
|   |   age <= 74
|   |   |   married = true: V (10.0)
|   |   |   married = false: P (14.0)
|   |   age > 74: T (53.0)
personal > 5416
|   blind = true: T (290.0)
|   blind = false
|   |   age <= 74
|   |   |   married = true: V (5.0)
|   |   |   married = false: P (5.0)
|   |   age > 74: T (32.0)

Number of Leaves  :  9

Size of the tree :  17


Time taken to build model: 0.29 seconds
Time taken to test model on training data: 0.13 seconds

=== Error on training data ===

Correctly Classified Instances         605               100      %
Incorrectly Classified Instances         0                 0      %
Kappa statistic                          1
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Coverage of cases (0.95 level)         100      %
Mean rel. region size (0.95 level)      20      %
Total Number of Instances              605


=== Confusion Matrix ===

   a   b   c   d   e   <-- classified as
 119   0   0   0   0 |   a = L
   0  77   0   0   0 |   b = H
   0   0  19   0   0 |   c = P
   0   0   0  15   0 |   d = V
   0   0   0   0 375 |   e = T
```

Figure 16. Output for Tax Calculator

More detailed performance of the inference is provided after the tree size and number of nodes. Since the subject system we use has nominal class variables as final output, we are most concerned about the first two lines of the final section. The first line shows the number and percentage of instances that were correctly predicted. The nest line shows the misclassifications. Since the Fig is the output of the final iteration and since the model inferred was perfect, the number of correctly classified instances is 100% (all 605 out of 605). This was achieved as a result of teaching the model from its own mistakes through a series on continuous testing and retraining. During the first couple of iterations the number of misclassified instances varied from 20% to 70%. The number of misclassifications reduced at a steady rate as the number of training samples increased. The time taken to infer the model from the training sample provided also increased with increasing sample size. But the overall inference was quicker and not more than a minute. A Graph was plotted to show the time line of the inference and is shown in Fig. 17.

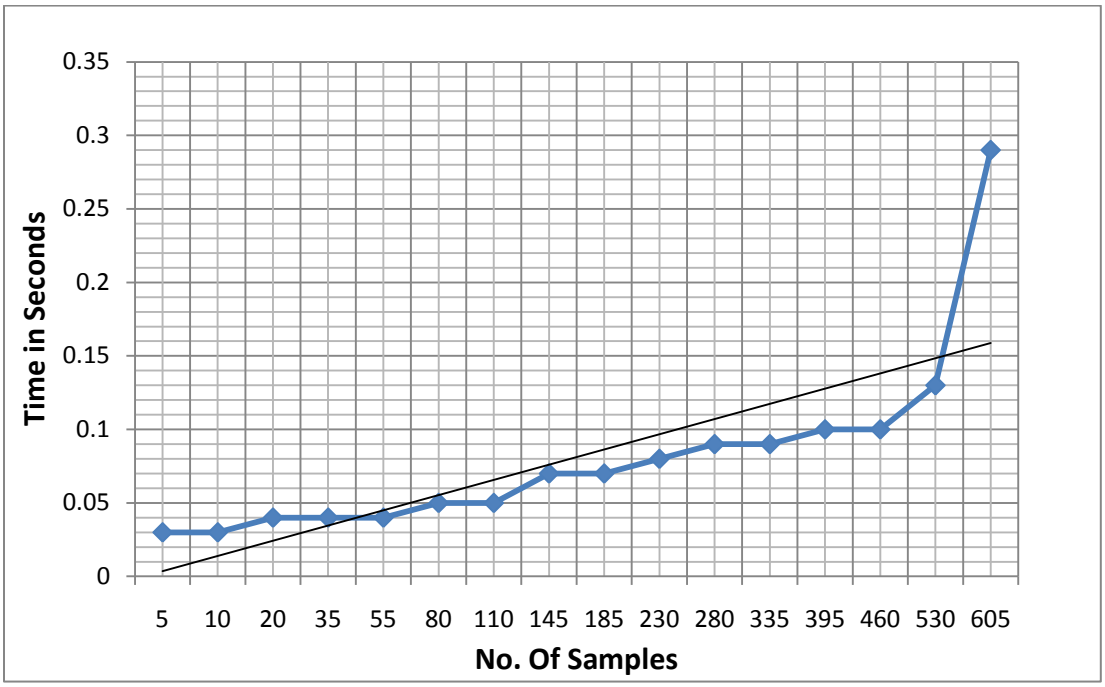

Figure 17. Time Graph for Tax System

The third line in the final section shows the Kappa statics, which measures the agreement of predictions with the actual class. This is not very informative as the static can have low values even when there are high levels of agreement. They are best appropriate in cases of testing whether the agreement exceeds the chance levels, i.e. the predicted and actual classes are correlated. We also obtained 100% coverage of all the cases. We also obtained the mean absolute error, and the root mean-squared error of the class probability estimates

assigned by the tree. The root mean squared error is the square root of the average quadratic loss [28]. The mean absolute error is calculated in a similar way using the absolute instead of the squared difference [29]. The relative error is calculated based on the prior probabilities i.e., if the inference is performed with some other learning algorithm. It gives the relative error comparing both the results obtained.

The final part of the output shows a confusion matrix which describes the result of the experiment in an easy way. The columns represent the predictions, and the rows represent the actual class. It shows that 605 instances were correctly predicted. The correct predictions are always the diagonal running from left to right of the table. These cases are called the *"True Positives"*. In the table we can see 375 instances were classified in the Class value T, 19 instances in the class value P and so on. Since we do not have any misclassifications in our output there were no values in the rows. These values are also known as *"False Positives"*. When there are misclassification the confusion matrix is represented with them so has to give us a number for instances that were misclassified. One should in general adjust misclassification costs and threshold levels so that sufficient accuracy and sensitivity in the desired class is obtained [29].

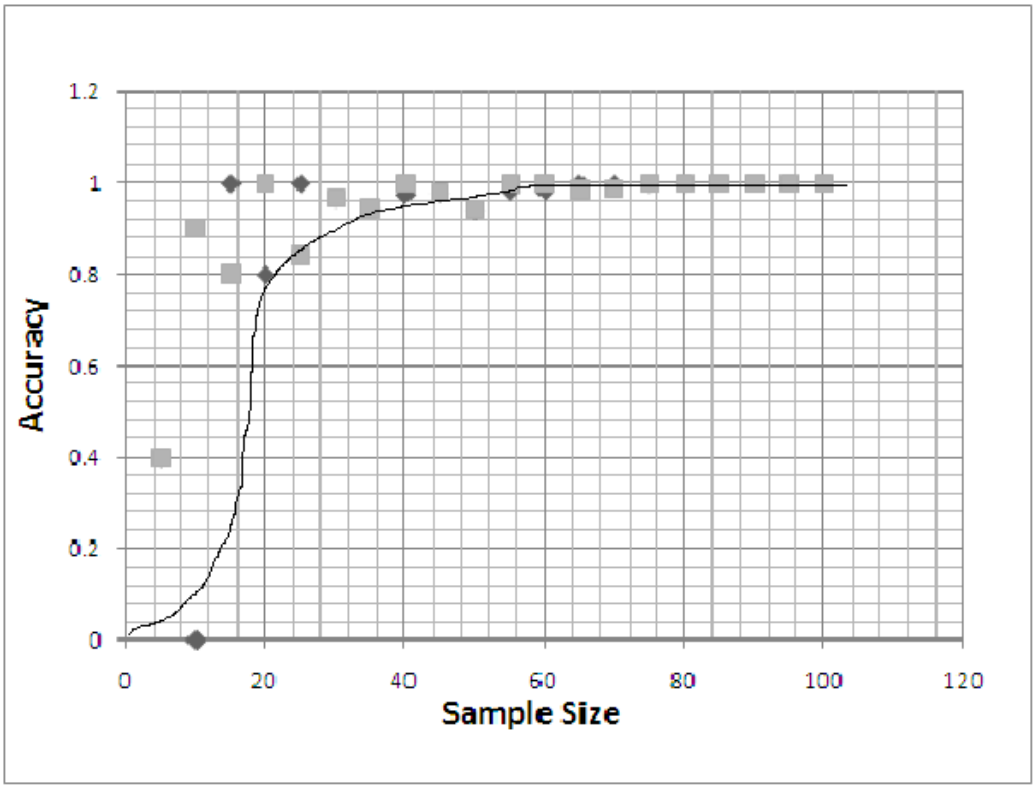

Figure 18. Learning Curve for Tax Calculator

The ILUSTRATOR technique provided valuable results when applied on the tax system. The technique of random testing exercised the system to the fullest and captured the behaviour of the system. The learning Graph Fig. 18 shows the learning curve of the technique on the tax system.

The technique was quick and less error prone as the model was re-trained from its own mistakes enabling it to identify its own weakness. More over the entire process was automated without any human intervention. We will see how the technique scales for a more complex and larger system in the following section.

4.2. Case Study 2: Credit Evaluation System

The second case study deals with a more complex and a larger system – Credit Evaluation system (Source code Snippet is available in the Appendix). It involves more decision making and is critical in a financial environment such as a bank. The system is not an exact mortgage lending decision maker but rather an adaptation of few of the commonly considered criteria's in such process. The system was built based on the *German Credit Data.* The Data set was originally built by Prof. Hofmann [30]. It was then amended by Strathclyde University that produced the file *"german.data-numeric"* which included several indicator variables so as to make it suitable for algorithms which cannot cope with categorical variables.

A java implementation of the German credit rating system was developed as a case study for this project. ILUSTRATOR technique was applied on this subject system to see the scalability of the technique. The system basically decides if a lending for a particular applicant is *Good* or *Bad*. The system takes 20 inputs which are randomly generator with our random test generator. Some of the inputs taken by the system are Purpose of the mortgage, Savings amount, Employment status, Property type, No. of applicants, marital status and so on. The system then has a series of if and else conditions through which the inputs are compared and a final decision on the lending is given. It the decision is Good then the lending can proceed as the applicant meets required criteria. If the decision is Bad then the lending does not happen. Simple criteria's like if the savings amount is *"500<=Savings<1000"* then the decision is Good. Similar conditions were implemented in the Java Source code to capture the financial decision making process. Such systems are critical and needs to be extensively tested as the final decisions made by them are concerned with financially important. Such systems are widely used in many financial organisations which depend entirely on the decision made by the system. These systems have

to be exercised for all possible inputs and the testing has to be regress so as to ensure the creditability of the decision made.

```
J48 pruned tree
------------------

checking_status = <0
|  own_telephone = none
|  |  duration <= 11: Good (5043.0)
|  |  duration > 11: Bad (17660.0)
|  own_telephone = yes: Good (22557.0)
checking_status = 0<=X<200
|  credit_amount <= 9800
|  |  credit_history = no credits/all paid
|  |  |  duration <= 42
|  |  |  |  other_parties = none
|  |  |  |  |  purpose = new car: Good (212.0/26.0)
|  |  |  |  |  purpose = used car: Good (224.0/21.0)
|  |  |  |  |  purpose = furniture/equipment
|  |  |  |  |  |  duration <= 21
|  |  |  |  |  |  |  duration <= 10: Bad (60.0)
|  |  |  |  |  |  |  duration > 10: Good (61.0)
|  |  |  |  |  |  duration > 21: Bad (120.0)
|  |  |  |  |  purpose = radio/tv
|  |  |  |  |  |  own_telephone = none
|  |  |  |  |  |  |  installment_commitment <= 2
|  |  |  |  |  |  |  |  duration <= 9
|  |  |  |  |  |  |  |  |  duration <= 3: Good (4.0/1.0)
|  |  |  |  |  |  |  |  |  duration > 3: Bad (5.0)
|  |  |  |  |  |  |  |  duration > 9: Good (22.0/4.0)
|  |  |  |  |  |  |  installment_commitment > 2
|  |  |  |  |  |  |  |  age <= 54
|  |  |  |  |  |  |  |  |  credit_amount <= 7893
|  |  |  |  |  |  |  |  |  |  age <= 50: Good (27.0/5.0)
|  |  |  |  |  |  |  |  |  |  age > 50: Bad (2.0)
|  |  |  |  |  |  |  |  |  credit_amount > 7893: Bad (3.0)
|  |  |  |  |  |  |  |  age > 54: Good (32.0/1.0)
|  |  |  |  |  |  own_telephone = yes
|  |  |  |  |  |  |  credit_amount <= 2538
|  |  |  |  |  |  |  |  credit_amount <= 1692
|  |  |  |  |  |  |  |  |  duration <= 6: Bad (3.0/1.0)
|  |  |  |  |  |  |  |  |  duration > 6: Good (21.0)
|  |  |  |  |  |  |  |  credit_amount > 1692
|  |  |  |  |  |  |  |  |  duration <= 16: Good (4.0/1.0)
|  |  |  |  |  |  |  |  |  duration > 16: Bad (6.0/1.0)
|  |  |  |  |  |  |  credit_amount > 2538: Good (68.0/2.0)
|  |  |  |  |  purpose = domestic appliance: Good (221.0/28.0)
|  |  |  |  |  purpose = repairs: Good (218.0/29.0)
|  |  |  |  |  purpose = education: Good (222.0/24.0)
|  |  |  |  |  purpose = vacation: Good (239.0/27.0)
|  |  |  |  |  purpose = retraining: Good (198.0/21.0)
|  |  |  |  |  purpose = business
|  |  |  |  |  |  residence_since <= 2: Good (75.0)
|  |  |  |  |  |  residence_since > 2: Bad (151.0)
|  |  |  |  |  purpose = other: Good (223.0/32.0)
|  |  |  |  other_parties = co applicant: Good (2453.0)
|  |  |  |  other_parties = guarantor
|  |  |  |  |  purpose = new car: Good (217.0/26.0)
|  |  |  |  |  purpose = used car: Good (235.0/27.0)
|  |  |  |  |  purpose = furniture/equipment
|  |  |  |  |  |  duration <= 21
|  |  |  |  |  |  |  |  duration <= 10: Bad (52.0)
|  |  |  |  |  |  |  |  duration > 10: Good (65.0)
|  |  |  |  |  |  |  duration > 21: Bad (97.0)
|  |  |  |  |  |  purpose = radio/tv: Good (212.0/28.0)
|  |  |  |  |  |  purpose = domestic appliance: Good (192.0/19.0)
|  |  |  |  |  |  purpose = repairs
|  |  |  |  |  |  |  own_telephone = none
|  |  |  |  |  |  |  |  residence_since <= 3: Good (92.0/3.0)
|  |  |  |  |  |  |  |  residence_since > 3
|  |  |  |  |  |  |  |  |  existing_credits <= 2: Good (21.0/2.0)
|  |  |  |  |  |  |  |  |  existing_credits > 2
|  |  |  |  |  |  |  |  |  |  credit_amount <= 2391: Good (5.0)
|  |  |  |  |  |  |  |  |  |  credit_amount > 2391
|  |  |  |  |  |  |  |  |  |  |  credit_amount <= 5800: Bad (6.0)
|  |  |  |  |  |  |  |  |  |  |  credit_amount > 5800
|  |  |  |  |  |  |  |  |  |  |  |  credit_amount <= 9219: Good
(8.0/1.0)
|  |  |  |  |  |  |  |  |  |  |  credit_amount > 9219: Bad (2.0)
|  |  |  |  |  |  own_telephone = yes: Good (109.0/22.0)
|  |  |  |  |  purpose = education: Good (223.0/22.0)
|  |  |  |  |  purpose = vacation: Good (243.0/35.0)
|  |  |  |  |  purpose = retraining: Good (202.0/25.0)
|  |  |  |  |  purpose = business
|  |  |  |  |  |  residence_since <= 2: Good (83.0)
|  |  |  |  |  |  residence_since > 2: Bad (145.0)
|  |  |  |  |  purpose = other: Good (228.0/37.0)
|  |  |  duration > 42
|  |  |  |  other_parties = none: Bad (446.0)
|  |  |  |  other_parties = co applicant: Good (492.0)
|  |  |  |  other_parties = guarantor: Bad (474.0)
|  |  credit_history = all paid: Good (8822.0/1082.0)
|  |  credit_history = existing paid: Good (8848.0)
|  |  credit_history = delayed previously: Good (8736.0)
|  |  credit_history = critical/other existing credit: Good
(8933.0)
|  credit_amount > 9800: Bad (875.0)
checking_status = >=200: Good (45177.0)
checking_status = no checking: Good (45324.0)

Number of Leaves  :  59

Size of the tree :  92
```

Figure 19. DT for Credit Rating System for 25 iterations

ILUSTRATOR technique does exactly this and requires no human intervention. The results obtained were positive. We were able to capture the entire behavior of the system for all possible inputs. The learning process was long but the final output we obtained was very useful to understand the system without

having to look into the code. Fig.19 shows the part of the tree obtained by applying the ILUSTRATOR technique. This was the closest inference of the perfect model as there were no contradictions when tested with random test samples. Two different tests (Test 1 and Test 2) were performed with random test samples. To see the techniques ability to infer models and correct misclassifications, we used different sized training and testing sets. During the test 1, we used training and testing sets with size 20 and during test 2, the size was increased to 50. Both the test took more than 20 iterations to infer a perfect model. Due to technical reasons when performing the tests and shortage of memory to store the training sample, the test had to be stopped after 25 iterations. But provided larger memory and a faster and technically advanced system, ILUSTRATOR will infer a perfect model.

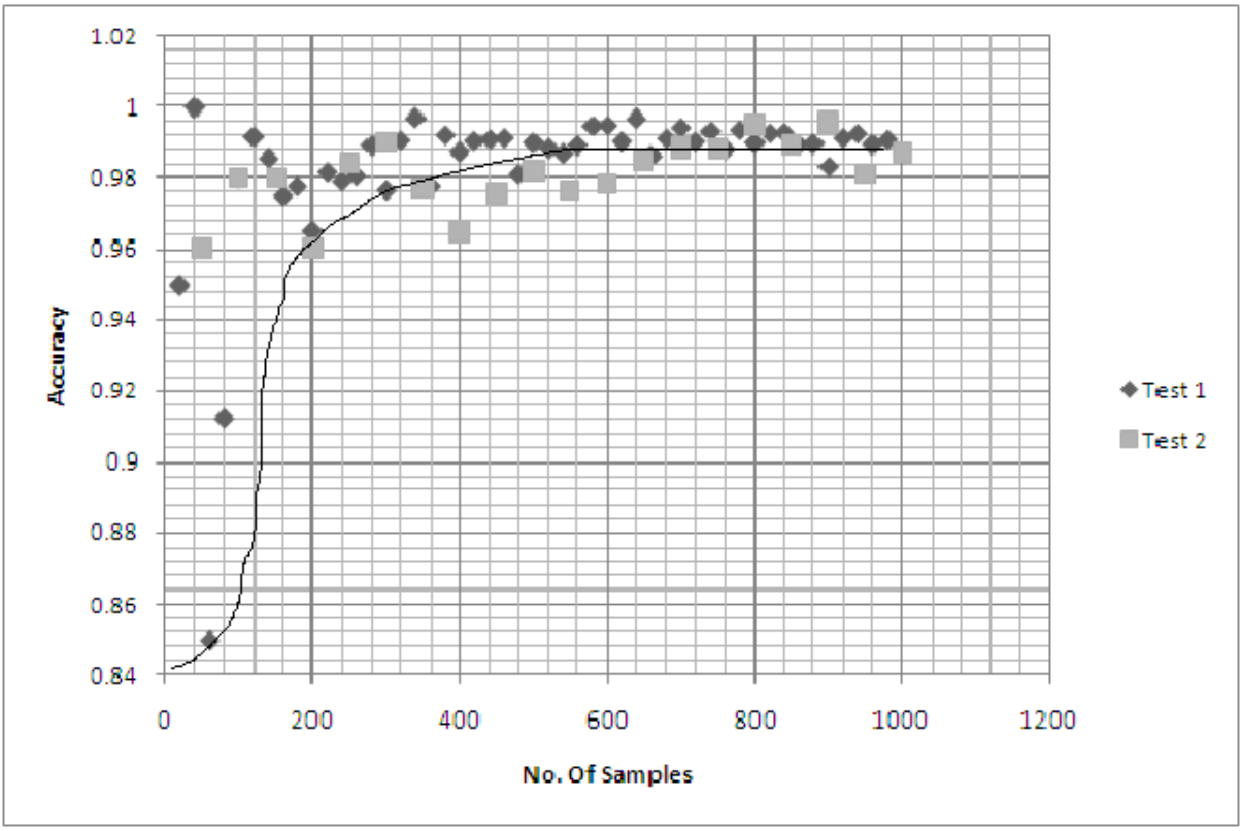

Figure 20. Learning Curve Credit Rating System

The accuracy of the tree obtained after 4 iterations in both the tests were more than 97% and was increasing on a steady phase with increasing size of the training sample. The graph shows a learning curve that is obtained by iterations. The initial models inferred from the small training samples had minimum accuracy and the model was refined more after 200 odd training

samples i.e., Iterations 5 or 6. The model learned from its own mistakes and had a steeper learning process from there on. Fig. 21 shows a graph for the time taken to infer models from the training samples. Since this subject system was more complex and larger, the time taken to infer models was longer compared to Case study 1. ILUSTRATOR still managed to infer perfect models from training samples in less than a minute.

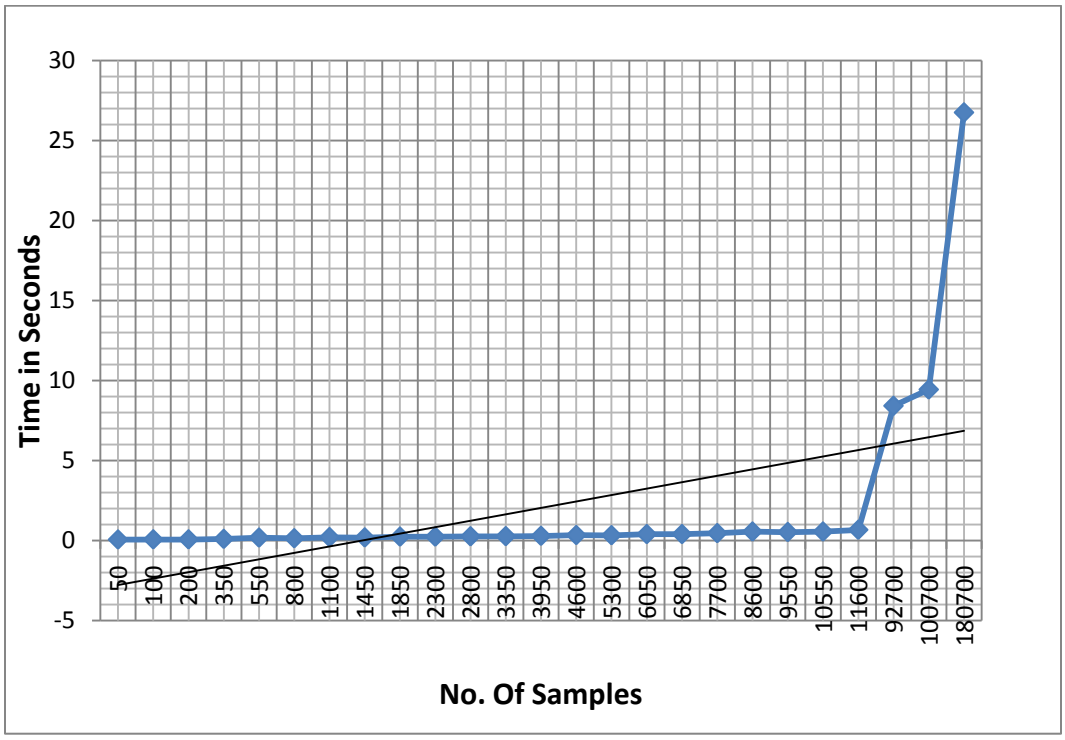

Figure 21. Time Graph for Credit Rating System

In both the case studies, ILUSTRATOR technique inferred perfect model that captured the behavior of the subject system. The training sample obtained at the end of inference was a finite set of test cases that exercised the system to the fullest. The scalability of the technique was under scrutiny but it performed well and the results obtained proved the same. We achieved 99.14% accuracy in the credit rating system and 100% accuracy in the Tax calculator system. The average time to infer a perfect model by retraining was less than 5 minutes.

The next chapter discusses other related works that helped in refining the ILUSTRATOR further and compare key results obtained.

Chapter 5

5. Related Work

We will discuss two areas of work related to the ILUSTRATOR technique in this section, namely the MELBA methodology and Category Partitioning. We will also discuss results obtained from our technique and how our work differs from the others.

5.1. Category Partitioning

Proposed by *Thomas J. Ostrand* and *Marc J. Balcer* in their paper "*The category-partition method for specifying and generating functional tests"* [31], it specifies rules to generate test suites from choices and categories. There are a number of research activities that have taken place using Category-Partition (CP) method. CP requires us to identify properties of a system that will affect the execution behavior of a system and hence its output. The basic idea is to identify the categories of a given system and their corresponding choices. For example in our BMI example, the properties may correspond to how is BMI value different for the different classes like *Underweight* and *Overweight*. This is called the "*Category"* and each category has a distinct set of *"choices".* For example, taking how BMI value differs between Underweight and Overweight, the choices could be *BMI >= 24 is Overweight* and *BMI <=19 is Underweight*. In addition to categories and choices, CP also uses *"Properties"* and *"Selectors"* to be identified for identifying the various interdependencies between the choices and hence can be used to find impossible combinations of choices in all the categories [32].

Apart from categories and choices that are used to describe the inputs for the subject system, CP also requires the environment conditions that may affect the programs behavior like load on the network, contents of the Database etc. CP can be used to capture both functional and non-functional behavior.

Following are the steps [31] involved in CP method:

A. **Analyze the specification** – Identify parameters of individual functional units and their characteristics, objects in the environment that might affect the system and their characteristics. This is collectively known as the *categories*.
B. **Partition the categories into choices** – Identify different cases that might occur in the above categories.

C. **Determine constraints among the choices** – Identify the relationship between each choice and the constraints they hold on each other.
D. **Write and process test specification** – A formal test specification is written which contains the category, choices and the constraints. This is then fed to a generator that produces a set of test frames.
E. **Evaluate generator output** – Manual intervention required to see if changes are to be made to the generator output.
F. **Transform into test scripts** – Once satisfied with the test specification, the tester changes them into test cases and organizes them into test suites.

During step E, reasons for changing the test specification could be because of impossible combinations of tests that are not achievable: Redundant test cases, Absence of some important test situation or repeated test samples. If the specification is changed by the human tester then Step D has to be repeated again. Step C and Step D are repeated over a number of times to achieve a desired functional test level. The Test *Specification Language (TSL)* is used to implement the above mentioned steps of CP.

Category Partition technique can be applied to the ILUSTRATOR technique at activity 3. Once we have inferred a decision tree from a random training sample, we can generate test cases from the tree by applying CP. We can identify the various categories and choices to each node of the tree and write down all possible test samples. But this requires human intervention: a human tester has to analyze the Decision tree and extract possible categories and choices from them including the environment conditions into a specification and then feed it to a generator to generate formal test frames. The whole idea of the project was to automate the black box testing and generate test cases automatically without human involvement in it. Hence it was not feasible to use CP in our technique. A similar attempt of using CP to generate test cases from model was already implemented in a methodology named MELBA, which is discussed in the next section.

5.2. MELBA Methodology

Fig. 22 provides an overview of the steps involved in the MELBA (MachinE Learning based refinement of BlAck-box test specification)

MELBA methodology [32] proposed in the paper "*Using machine learning to refine Category-Partition test specifications and test suites"* by *Lional C.Braind, Yvan Labiche, Zaheer Bawar and Nadia Traldi Spido* , is a technique for black box technique which exploits the Category partition technique.

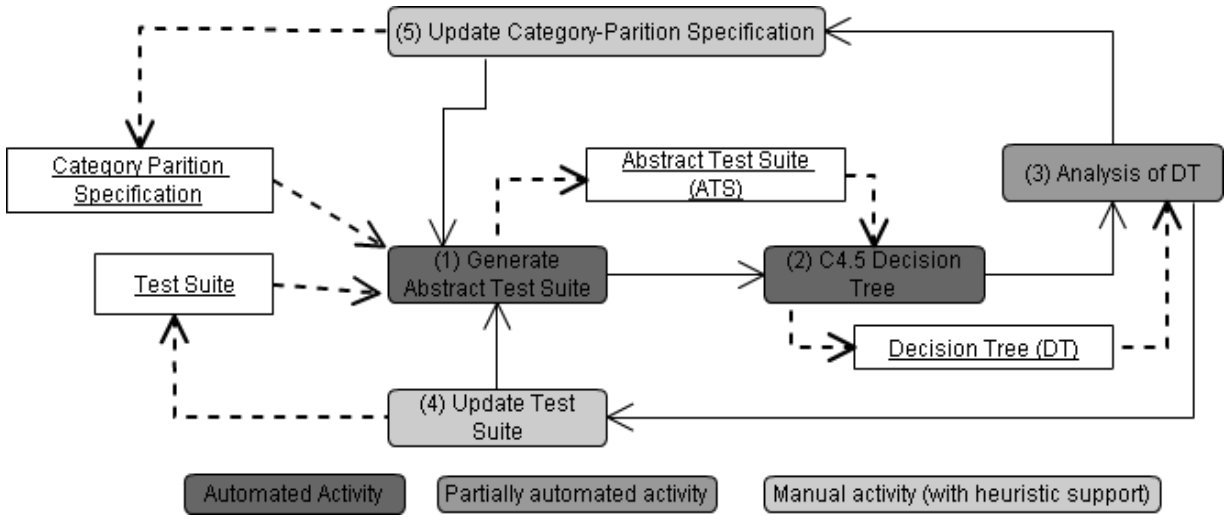

Figure 22. The MELBA methodology

The methodology is based on abstracting test suite information and then changing the test suite. It uses the most common black box test specification technique: Category Partitioning which was discussed in the above section. The methodology abstracts test cases into categories and choices as per CP technique and then applies a Machine Learning algorithm on them to identify the relationship between specific inputs and corresponding outputs. This exercises the system to identify the behavior for particular inputs. The technique also provides certain heuristics to facilitate improvements of the test specifications and the test cases. Before the technique can be applied to a subject system, the test suites of the system have to be transformed into abstract test suites. The technique uses automated tools to transform test cases into abstract test cases. The abstract test cases are a combination of output equivalence class and pairs (category and choices) that characterize the input and the environment parameter. When fed to a machine learner algorithm which is C4.5, it yields rules of the system behavior that relates the categories and output equivalence class. This is a partially automated technique and still need human tester intervention in analyzing the rules and updating the Category partition specifications with the missing information.

The technique is similar to ILUSTRATOR in terms of improving the test cases iteratively by identifying contradictions and updating the training sample accordingly. The learner algorithm is executed repeatedly until there are no

more contradictions identified in the rules learnt by the machine learning algorithm.

The drawback of technique is that it assumes that the initial test suite or test specification used at the beginning of the technique is already executed and failures have been corrected. In short the initial test specification wouldn't have any contradictions when tested. But as the training sample is added in the next iterations, failures can arise and faults can be detected. Though this is similar to ILUSTRATOR technique, another major pitfall of MELBA is that to define the categories and choices, the human tester has to have a certain degree of understanding about the subject system domain. The technique argues that this is however not a pitfall as the tester would anyway know these information when trying to re engineer the test suites. The paper also argues that there is no way one can reuse test suites without understanding the relationships between inputs and outputs.

Though MELBA provides tool support to convert test cases into abstract test cases and provides heuristics to identify and improve potential contradictions, the methodology still requires a certain level of human intervention: the tester has to understand the system domain first and discover categories and choices so as to create an abstract test suite which is not the case in our technique. Some attempt to cover the possible pitfalls of MELBA and category partition method is done using our technique and are discussed in the next sections with some results obtained.

5.3. Our Results

As discussed in the previous section, ILUSTRATOR tries to overcome some of the drawbacks of the other existing automated black-box testing techniques. We have applied the technique on a number of small and large scale complex systems and noted that the technique is flexible and the results are positive. The biggest advantage of the technique is that it is fully automated and does not require any human intervention. The model generation, training, testing and retraining misclassification are achieved by executing a series of batch files. Comparing to the MELBA methodology which requires a certain degree of human involvement, ILUSTRATOR proves to be far effective though the number of iterations are more. An iterative learning model that learns from its own mistakes using random test samples is more precise as the system is exercised to its full limits. This is because the test cases generated are random and are not based on the systems domain. When applying the MELBA methodology, extracting test cases from the model would only pave way to certain test

situations, arguably the most expected test set that the system is designed for. Since the test cases generated are random when applying our technique, it exercises the model inferred to a wide range of input data that the system may or may not be designed to handle. The contradictions that rise when testing the inferred model will give us the answer to what data could the system not handle. Narrowing the test samples to test the model will only result in exercising the system to limited possible inputs which is not a good testing technique.

To summarize, the following are the key aspects of ILUSTRATOR that makes it different from the above related work:

1. It generates model from non-specific training samples which are generated randomly and improved iteratively.

2. Test cases are generated randomly which exercises the subject system to the fullest rather than testing it to specific test samples.

3. The model generated learns from its own mistakes and adds the misclassified samples to its training set and retrains itself. In short more the mistakes the model makes, more it learns.

4. The final training sample obtained once there are no more contradictions, acts as a finite set of test cases with a broader range of inputs that test the system to its full limits.

5. The entire technique is automated and does not require any human intervention hence removing the human component from the testing cycle.

Chapter 6

6. Conclusion

This paper introduced ILUSTRATOR technique, a fully automated iterative methodology based on novel machine learning techniques that helps developers understand the program functionalities and test unfamiliar black-box systems effectively. The technique was based on iterative learning using random test generators. The technique infers models from initial training samples that are generated randomly and then tested with more random test cases. The model is perfected by retraining it with misclassifications that was predicted and testing it again, hence creating a virtuous loop of training and testing. The process comes to a halt when there are no more misclassifications and the model obtained is deemed as a perfect model- the closest representation of the subject system. We used C4.5 learning algorithm to infer models from training samples which are represented in the form of Decision Trees (DT).

We have shown how ILUSTRATOR technique is applied on real scale systems (BMI calculator, Tax Calculator and the Credit Evaluation System) and evaluated the effectiveness of the test cases generated by them. The case study showed us how iterative learning improves the inference process and how the model corrects itself by learning from its own mistakes. We evaluated the results of each case study with learning graphs that were obtained by plotting the accuracy of inference with the number of training samples. We also evaluated the time taken for inference of a perfect model. The end product of each study was a finite set of test cases that the system was exercised on. These test cases along with the model inferred can be used to improve the understandability of the subject system without having to look into the code.

We also discussed other similar black-box specifications like category partitioning and methodologies like MELBA based on them for generating test suites for unfamiliar systems. We compared results obtained from both the technique and pointed out some pitfalls that ILUSTRATOR overcame.

The main advantage of ILUSTRATOR was that the entire process was automated and did not require any human tester component. Also since the test cases are generated using a random test generator the range of test cases produced is wider and the system is exercised to all possible values rather than specific test suites. This enables us to infer the systems behavior in detail and hence understand the systems functionality. However using random test case

might not be feasible for all example system and might hit the wall at some point. Generating random test cases might infer only one part of the systems behavior and hence ignoring important functionality of the subject system. Though this might not be the case always, there is a fine probability of such situations arising when applying the technique to more complex systems.

Automating black-box testing of unfamiliar systems and removing the human tester from the testing cycle was a challenging task. Overall the project was a great learning process and introduced some new and exciting areas of research in black-box testing framework. Future work can investigate on other possible black-box specifications other than random testing and Category Partitioning. Applying ILUSTRATOR to more complex and larger systems and evaluating the results, investigating on other possible model representation other than decision trees and developing new machine learning algorithms like C4.5 can be some of the other possible areas of research. We hope that this piece of work contributes to this area of research and future works provide better results.

References

[1] T.H. Tse, Francis C.M. Lau, W.K. Chan, Peter C.K. Liu, and Colin K.F. Luk, Testing Object-Oriented Industrial Software Without Precise Oracles or Results

[2] Neil Walkinshaw, Kirill Bogdanov, John Derrick and Javier Paris. Increasing Functional Coverage by Inductive Testing: A Case Study

[3] Neil Walkinshaw, The Practical Assessment of Test Sets with Inductive Inference Techniques, TAIC PART'10 Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques.

[4] Elaine J. Weyuker. Assessing Test Data adequacy through Program Inference, Courant institute of Mathematical Science, 1983.

[5] Bergadano, F., Gunetti, D.: Testing by means of inductive program learning. ACM Transactions on Software Engineering and Methodology, 1996.

[6] Zhu, H., Hall, P., May, J.: Inductive inference and software testing. Software Testing, Veri_cation, and Reliability, 1992.

[7] Zhu, H.: A formal interpretation of software testing as inductive inference. Software Testing, Verification and Reliability, 1996.

[8] Harder, M., Mellen, J., Ernst, and M.: Improving test suites via operational abstraction. In: Proceedings of the International Conference on Software Engineering, 2003.

[9] Xie, T., Notkin, D.: Mutually enhancing test generation and specification inference. In: Proceedings of FATES 2003.

[10] Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Fundamental Approaches to Software Engineering, 2005.Rffelt, H., Steffen, and B.: Learnlib: A library for automata learning and experimentation. In Baresi, L., Heckel, R., eds.: FASE, 2006.

[11] Bollig, B., Katoen, J., Kern, C., Leucker, and M.: Smyle: A tool for synthesizing distributed models from scenarios by learning. In: Proceedings of Concurrency Theory, CONCUR, 2008

[12] Shahbaz, M., Groz, R.: Inferring mealy machines. In: Proceedings of Formal Methods FM, 2009

[13] Raffelt, H., Merten, M., Steffen, B., Margaria, and T.: Dynamic testing via automata learning. STTT, 2009

[14] Walkinshaw, N., Derrick, J., Guo, Q.: Iterative refinement of reverse-engineered models by model-based testing. In: Proceedings of Formal Methods FM, 2009.

[15] S.H.Muggleton, Inductive Logic Programming, New Generation Computing, 1991.

[16] S. Muggleton, Inverse Entailment and Progol, New Generation Computing Journal 13, pp. 245-286, 1995.

[17] S. Muggleton, Learning from positive data, Proceedings of the Sixth International Workshop on Inductive Logic programming, 1997.
[18] Ian H. Witten; Eibe Frank, Data Mining: Practical machine learning tools and techniques, 2nd Edition. Morgan Kaufmann, San Francisco, 2005.
[19] Attribute Relation File Format, April 1st, 2002, [online] http://www.cs.waikato.ac.nz/~ml/weka/arff.html [Accessed 15 March, 2011]
[20] Quinlan, J. R. C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers, 1993.
[21] Quinlan, J. R, Induction of Decision Trees, Machine Learning 1, 1986.
[22] Ross J. Quinlan: Learning with Continuous Classes. In: 5th Australian Joint Conference on Artificial Intelligence, Singapore, 343-348, 1992.
[23] S.B. Kotsiantis, Supervised Machine Learning: A Review of Classification Techniques, Informatics 31(2007) 249-268, 2007.
[24] Ian H. Witten, Eibe Frank, Mark A.Hall, Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)
[25] Payam Refaeilzadeh, Lei Tang, Huan Liu, Arizona state University, Cross validation.
[26] Kohavi, Ron. "A study of cross-validation and bootstrap for accuracy estimation and model selection". Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence.
[27] Sebastian Danicic, Chris Fox, Mark Karman, Rob Hierons, ConSIT: A Conditional Program Slicer.
[28] Nikulin, M.S. (2001), "Loss function", in Hazewinkel, Michiel, *Encyclopaedia of Mathematics*, Springer, ISBN 978-1556080104, http://eom.springer.de/L/l060900.htm
[29] The Explorer Interface - Classification, 1999, [Online] http://wekadocs.com/node/13 [Accessed 4, April 2011]
[30] UCI, Machine Learning Repository, Statlog (German Credit Data) Data Set, 1994, [Online] http://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data) [Accessed 28 April, 2011]
[31] Thomas J. Ostrand and Marc J. Balcer. “The Category-Partition Method for Specifying and Generating Functional Tests”, *Communications of the ACM*, 31(6), June 1988.
[32] L.C. Briand et al., Using machine learning to refine Category-Partition test specifications and test suites, Inform. Softw.Technol. (2009), doi:10.1016/j.infsof.2009.06.006

Appendix

1. UK Income Taxation Calculator – Source Code

```
main() {
int age, blind, widow, married, income

scanf("%d",&age);
scanf("%d",&blind);
scanf("%d",&married);
scanf("%d",&widow);
scanf("%d",&income);

if (age>=75) personal = 5980;
else if (age>=65) personal = 5720;
else personal = 4335;

if ((age>=65) && income >16800)
{ t = personal - ((income-16800)/2) ;
if (t>4335) personal = t;
else personal = 4335; }

if (blind) personal = personal + 1380 ;

if (married && age >=75) pc10 = 6692;
else if (married && (age >= 65)) pc10 = 6625;
else if (married || widow) pc10 = 3470;
else pc10 = 1500;

if (married && age >= 65 && income > 16800)
{ t = pc10-((income-16800)/2);
if (t>3470) pc10 = t;
else pc10 = 3470; }

if (income <= personal) tax = 0;
else { income = income - personal ;
if (income <= pc10) tax = income / 10;
else { tax = pc10 / 10;
income = income - pc10;
if (income <= 28000) tax = ((tax + income) * 23) / 100 ;
else { tax = ((tax + 28000) * 23) / 100 ;
income = income - 28000;
tax = ((tax + income) * 40) / 100; }
}
}

if (!blind && !married && age < 65) code = 'L' ;
else if (!blind && age < 65 && married) code = 'H';
else if (age >= 65 && age < 75 && !married && !blind) code = 'P';
else if (age >= 65 && age < 75 && married && !blind) code = 'V';
else code = 'T';
}
```

.

2. German Credit Rating System – Source Code

```
if (Status == "'>=200'")
Lending = "'Good'";
else if (Status == "'no checking'")
Lending = "'Good'";
else if (Status == "'0<=X<200'") {
if (LoanAmount > 9800)
Lending = "'Bad'";
else {if (Savings == "'>=1000'")
Lending = "'Good'";
else if (Savings == "'no known savings'")
Lending = "'Good'";
else if (Savings == "'500<=X<1000'")
Lending = "'Good'";
else if (Savings == "''100<=X<500'") {
if (Purpose != "'business'")
Lending = "'Good'";
else {
if (HousingType == "'own'")
Lending = "'Good'";
else if (HousingType == "'for free'")
Lending = "'Good'";
else {
if (ExistingCredit <= 1)
Lending = "'Good'";
else
Lending = "'Bad'";
}
}
} else if (Savings == "'<100'") {
if (Parties == "'co applicant'")
Lending = "'Good'";
else if (Parties == "'guarantor'") {
if (Purpose == "'new car'")
Lending = "'Bad'";
else
Lending = "'Good'";
} else if (Parties == "'none'") {
if (Duration > 42)
Lending = "'Bad'";
else {
if (MartStatus == "'female single'")
Lending = "'Good'";
else if (MartStatus == "'male single'")
Lending = "'Good'";
else if (MartStatus == "'male mar/wid'") {
if (Duration > 10)
Lending = "'Bad'";
else
Lending = "'Good'";
} else if (MartStatus == "'male div/sep'")
Lending = "'Bad'";
else if (MartStatus == "'female div/dep/mar'") {
if (Purpose == "'business'") {
if (ResidenceSince > 2)
```

```
Lending = "'Bad'";
else
Lending = "'Good'";
} else if (Purpose == "'furniture/equipment'") {
if (Duration <= 10)
Lending = "'Bad'";
else if (Duration > 10) {
if (Duration <= 21)
Lending = "'Good'";
else
Lending = "'Bad'";
}
}
}
}
}
}
}
} else if (Status == "'<0'") {
if (ForeignWorker == "'No'")
Lending = "'Good'";
else {
if (Duration <= 11) {
if (ExistingCredit > 1)
Lending = "'Good'";
else {
if (Property == "'real estate'")
Lending = "'Good'";
else if (Property == "'car'")
Lending = "'Good'";
else if (Property == "'no known property'")
Lending = "'Bad'";
else if (Property == "'life insurance'") {
if (OwnPhone == "'none'")
Lending = "'Bad'";
else
Lending = "'Good'";
}
}
} else {
if (JobType == "'high qualif/self emp/mgmt'")
Lending = "'Good'";
else if (JobType == "'unemp/unskilled non res'")
Lending = "'Bad'";
else if (JobType == "'unskilled resident'") {
if (Purpose == "'retraining'")
Lending = "'Good'";
else if (Purpose == "'business'")
Lending = "'Good'";
else if (Purpose == "'other'")
Lending = "'Good'";
else if (Purpose == "'used car'")
Lending = "'Bad'";
else if (Purpose == "'domestic appliance'")
Lending = "'Bad'";
else if (Purpose == "'repairs'")
Lending = "'Bad'";
```

```
else if (Purpose == "'education'")
Lending = "'Bad'";
else if (Purpose == "'vacation'")
Lending = "'Bad'";
else if (Purpose == "'radio/tv'") {
if (ExistingCredit > 1)
Lending = "'Good'";
else
Lending = "'Bad'";
} else if (Purpose == "'new car'") {
if (OwnPhone == "'none'")
Lending = "'Bad'";
else
Lending = "'Good'";
} else if (Purpose == "'furniture/equipment'") {
if (Employment == "'unemployed'")
Lending = "'Good'";
else if (Employment == "'1<=X<4'")
Lending = "'Good'";
else if (Employment == "'4<=X<7'")
Lending = "'Good'";
else if (Employment == "'>=7'")
Lending = "'Good'";
else
Lending = "'Bad'";
}
} else if (JobType == "'skilled'") {
if (Parties == "'guarantor'")
Lending = "'Good'";
else if (Parties == "'co applicant'")
Lending = "'Bad'";
else if (Parties == "'none'") {
if (Duration > 30)
Lending = "'Bad'";
else {
if (Savings == "'>=1000'")
Lending = "'Good'";
if (Savings == "'500<=X<1000'")
Lending = "'Good'";
else if (Savings == "'100<=X<500'") {
if (History == "'no credits/all paid'")
Lending = "'Good'";
else if (History == "'all paid'")
Lending = "'Good'";
else if (History == "'delayed previously'")
Lending = "'Good'";
else if (History == "'critical/other existing credit'")
Lending = "'Good'";
else
Lending = "'Bad'";
} else if (Savings == "'no known savings'") {
if (ExistingCredit > 1)
Lending = "'Good'";
else {
if (OwnPhone == "'yes'")
Lending = "'Good'";
else
```

```
Lending = "'Bad'";
}
} else if (Savings == "'<100'") {
if (History == "'no credits/all paid'")
Lending = "'Bad'";
else if (History == "'all paid'")
Lending = "'Bad'";
else if (History == "'delayed previously'")
Lending = "'Bad'";
else if (History == "'critical/other existing credit'")
Lending = "'Good'";
else if (History == "'existing paid'") {
if (OwnPhone == "'Yes'")
Lending = "'Bad'";
else {
if (ExistingCredit > 1)
Lending = "'Bad'";
else {
if (Property == "'no known property'")
Lending = "'Good'";
else if (Property == "'life insurance'")
Lending = "'Bad'";
else if (Property == "'car'") {
if (LoanAmount > 1400)
Lending = "'Good'";
else
Lending = "'Bad'";
} else if (Property == "'real estate'") {
if (Age > 26)
Lending = "'Good'";
else
Lending = "'Bad'";
}
}
}
}
}
}
}
}
}
```