GEOMETRIC APPROACHES TO STATISTICAL DEFECT PREDICTION AND LEARNING

NAZANIN HAZRATI

A THESIS

IN

THE DEPARTMENT

OF

CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

CONCORDIA UNIVERSITY

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

APRIL 2011

© NAZANIN HAZRATI, 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Nazanin Hazrati**

Entitled: **Geometric Approaches to Statistical Defect Prediction and Learning**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

Dr. A. Hammad

_____ Examiner

Dr. J. Bentahar

_____ Examiner

Dr. M. Chen

_____ Supervisor

Dr. A. Ben Hamza

Approved by _____

Dr. Robin A. L. Drew, Dean

Faculty of Engineering and Computer Science

Abstract

Geometric Approaches to Statistical Defect Prediction and Learning

Nazanin Hazrati

Software quality is directly correlated with the number of defects in software systems. As the complexity of software increases, manual inspection of software becomes prohibitively expensive. Thus, defect prediction is of paramount importance to project managers in allocating the limited resources effectively as well as providing many advantages such as the accurate estimation of project costs and schedules. This thesis addresses the issues of statistical fault prediction modeling, software resource allocation, and optimal software release and maintenance policy.

A software defect prediction model using operating characteristic curves is presented. The main idea behind this predictor is to use geometric insight in helping construct an efficient prediction method to reliably predict the cumulative number of defects during the software development process.The performance of the proposed approaches is validated on real data from actual SAP projects, and the experimental results demonstrate a compelling motivation for improved software quality.

Contribution in the field of software defect prediction continues by application of Genetic Programming(GP), as the youngest of Evolutionary Algorithm family member, in field of machine learning. GP is a flexible and expressive tool to build models based on the minimizing an objective function. GP does not take into account any assumptions since it is based on no bias as well as no heuristics. This method has been applied on NASA IV&V defect repository.

Acknowledgments

In the first place I would like to record my gratitude to Dr. A. Ben Hamza for his supervision, and guidance from the early stages of this research.

The second person I owe a great deal of gratitude to is Mohammad Ebne-Alian. Thank you for providing valuable input to my work, for being a constant support, and a good friend.

Words fail me to express my appreciation to my mother whose dedication, love and persistent support, has taken the load off my shoulder. My special gratitude is due to my brothers for their loving support.

Last but not least, this thesis is also dedicated in loving memory of my devoted father whose words of inspiration and encouragement in pursuit of excellence, still linger on.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Software systems are now being used in many demanding applications. Software defects cost businesses and the software industry billions of dollars in lost productivity every year. Software quality impacts development costs, delivery schedules, and user satisfaction. Software quality is directly correlated with the number of defects in software systems. In other words, increasing software quality means reducing number of defects in the software product. Errors and mistakes happened in development process as well as ambiguous, missing or incorrect requirements will lead to faults in the software product. Software industry has been seeking effective approaches on finding effective software defect prediction methods during the past years.

It is well-known that more pre-release development and testing on systems can reduce future development costs and result in higher software quality. On the other hand, the pressure to deliver an operational product quickly can frequently affect the resource allocation among development phases or within one of the phases. Unfortunately, nowadays all these decisions are made intuitionally. However, human's brain is not able to take into account all the effecting parameters at the same time. Besides, human judgements are biased. Hence, there is a high demand for a strategic, mathematically proven approach for these decisions.

Knowledge about how many defects to expect as well as detecting defect-prone modules in a software product empower software companies to gauge the expenses related to verification and validation efforts. It provides essential information for decision making in many software development activities, such as cost analysis, resource allocation, and release and maintenance time decision. It is also useful to obtain a software reliability measure. In addition, having the optimal decisions will result in software quality increase.

The major part of this thesis is devoted to methods for optimal policies in software development processes. The first problem addressed in this thesis is software defect prediction using operating characteristic curves and Laplace trend statistics. The main idea behind our proposed technique is to use geometric insight in helping construct an efficient and fast prediction method to accurately predict the cumulative number of defects at any given stage during the software development process. Real data from actual SAP projects is used to illustrate the effectiveness and the much improved performance of the proposed methods in comparison with existing approaches.

We also propose Genetic Programming (GP)-based approaches for learning defect predictors. Defect prediction will be carried out through our proposed evolutionary algorithms in a way that the global structure of the data is preserved. We tested different methods of reducing the dimensionality of data with the aim of inductive learning. We achieved the best performance by taking no heuristics into account which is the essential assumption by evolutionary algorithms. Experimental results have been assessed as significant in terms of having high detection rate while keeping misdetection rate low. Although additional research efforts might provide a more detailed analysis of the predicted defects, the results presented in this thesis provide a compelling motivation for improved software quality.

1.1 Framework and Motivation

Quality is perceived differently in various domains and different perspectives of quality have been as follows [1]:

- Quality can be recognized but not defined.

  > To many people, quality is similar to what a federal judge once said about obscenity: 'I know it when I see it.' [2]

- Quality is conformance to specification. Putting the effort to have a well-defined specification helps a lot to have a quality product. Although sometimes errors made during the requirement stage account for more than half of the defects found in a software project [3].

- Fitness for use. Many software products do not meet the user needs and expectations while they adequately meet the specifications.

- Software Quality attributes or "-ilities" of software product. Some examples are reliability, usability, availability, flexibility, maintainability, portability, installability, and adaptability.

- The monetary value that a customer is planning to spend on it. It is also dependent on the software application platform.

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures. It has been widely accepted as a practical, cost effective way to improve software quality [4]. SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance

with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits. Software development and control processes should include quality assurance approval points, where an SQA evaluation of the product may be done in relation to the applicable standards.

One of the many challenges faced when attempting to build a business case for software process improvement is the relative lack of credible measurement data. If a company does not have the data to build the business case, then it does not have the improvement project to get the data. It is the classical chicken-and-egg dilemma. The motivation behind this thesis is to implement statistical models for predicting software defects using available defect data and use this data to find the optimal strategies in software production. The practitioners collect software defect data during software development processes but the decision support power of the collected data is wasted in most of the organizations. These defect data combined with the data of other features become a well-suited repository for using Bayesian statistics, machine learning, and evolutionary algorithms based techniques to predict future defects. We have used defect dataset of a real SAP project for the contribution done in Chapter 2. A master repository of static code metrics which is created and maintained by NASA Independent Verification and Validation (IV&V) Facility has been used in Chapter 3.

1.1.1 What are software defects?

A software engineer's job is to deliver quality products for their planned costs, and on their committed schedules. Software products must also meet the user's functional needs and reliably and consistently do the user's job. While the software functions are most important to the program's users, these functions are not usable unless the software runs. To get the software to run reliably, however, engineers must remove almost all its defects. Inspection

team's objective is to locate problems and decided what issues to be recorded as errors or defects. Thus, while there are many aspects to software quality, the first quality concern must necessarily be with its defects.

Some people mistakenly refer to software defects (faults) as bugs. When programs are widely used and are applied in ways that their designers did not anticipate, seemingly trivial mistakes can have unforeseeable consequences. As widely used software systems are enhanced to meet new needs, latent problems can be exposed and a trivial-seeming defect can truly become dangerous. While the vast majority of trivial defects have trivial consequences, a small percentage of seemingly silly mistakes can cause serious problems. Since there is no way to know which of these simple mistakes will have serious consequences, we must treat them all as potentially serious defects, not as trivial-seeming "bugs".

A defect is a problem in a software system or its documentation does not meet defined requirements and is found beyond the point of origin, e.g. a requirement problem found during a code inspection [4]. A defect is thus an objective thing. It is something you can identify, describe, and count. Failure, is when a defect becomes active or in other words we face that defect.

Simple coding mistakes can produce very destructive or hard-to-find defects. Conversely, many sophisticated design defects are often easy to find. The sophistication of the design mistake and the impact of the resulting defect are thus largely independent. Even trivial implementation errors can cause serious system problems. This is particularly important since the source of most software defects is simple programmer oversights and mistakes. While design issues are always important, initially developed programs typically have few design defects compared to the number of simple oversights, typos, and goofs. To improve program quality, it is thus essential that engineers learn to manage all the defects they inject in their programs.

1.2 Software reliability growth models

Achieving highly reliable software from the customers perspective is a demanding job for all software engineers and reliability engineers [19] summarizes the following four technical areas which are applicable to achieving reliable software systems, and they can also be regarded as four fault lifecycle techniques:

1. Fault prevention: to avoid, by construction, fault occurrences.

2. Fault removal: to detect, by verification and validation, the existence of faults and eliminate them.

3. Fault tolerance: to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring.

4. Fault/failure forecasting: to estimate, by evaluation, the presence of faults and the occurrences and consequences of failures. This has been the main focus of software reliability modeling.

Fault prevention is the initial defensive mechanism against unreliability. A fault which is never created costs nothing to fix. Fault prevention is therefore the inherent objective of every software engineering methodology. Fault prevention mechanisms cannot guarantee avoidance of all software faults. When faults are injected into the software, fault removal is the next protective means. Two practical approaches for fault removal are software inspection and software testing, both of which have become standard industry practices in quality assurance.

When inherent faults remain undetected through the inspection and testing processes, they will stay with the software when it is released into the field. Fault tolerance is the last defending line in preventing faults from manifesting themselves as system failures. Fault

tolerance is the survival attribute of software systems in terms of their ability to deliver continuous service to the customers. Software fault tolerance techniques enable software systems to (1) prevent dormant software faults from becoming active, such as defensive programming to check for input and output conditions and forbid illegal operations; (2) contain the manifested software errors within a confined boundary without further propagation, such as exception handling routines to treat unsuccessful operations; (3) recover software operations from erroneous conditions, such as checkpointing and rollback mechanisms; and (4) tolerate system-level faults methodically, such as employing design diversity in the software development. Finally if software failures are destined to occur, it is critical to estimate and predict them. Fault/failure forecasting involves formulation of the fault/failure relationship, an understanding of the operational environment, the establishment of software reliability models, developing procedures and mechanisms for software reliability measurement, and analyzing and evaluating the measurement results. The ability to determine software reliability not only gives us guidance about software quality and when to stop testing, but also provides information for software maintenance needs.

Software reliability may be the most important quality attribute of software, due to the fact that it quantifies software failures during the software development process. Software reliability models usually make a number of common assumptions, as follows: (1) The operation environment where the reliability is to be measured is the same as the testing environment in which the reliability model has been parameterized. (2) Once a failure occurs, the fault which causes the failure is immediately removed. (3) The fault removal process will not introduce new faults. (4) The number of faults inherent in the software and the way these faults manifest themselves to cause failures follow, at least in a statistical sense, certain mathematical formula. There are essentially two types of software reliability models:

- those that attempt to predict software reliability from design parameters
- those that attempt to predict software reliability from test data

The first type of models are usually called "defect density" models and use code characteristics such as lines of code, nesting of loops, external references, input/outputs, and so forth to estimate the number of defects in the software. The second type of models are often called software reliability growth models (SRGMs) since the number of faults (as well as the failure rate) of the software system reduces when the testing progresses, resulting in growth of reliability. These models attempt to statistically correlate defect detection data with known functions such as an exponential function.

Each software defect encountered entails a significant cost for software companies. Hence the knowledge of the number of defects in a software product during its lifecycle is a very valuable asset. Being able to estimate the number of defects will substantially improve the decision processes in software lifecycle like time to release and maintenance time. In addition, the production process of the software can be considerably improved by employing a prediction model that reliably predicts the number of defects.

During the development process of software, many defects may be introduced and often lead to critical problems and complicated breakdowns of computer systems [5]. Thus there is a high demand for controlling the quality and reliability of software development process. As an evaluation for software reliability, number of defects can be used. In the traditional software development environment, software reliability evaluation provides useful guidance in balancing reliability, time to market and development cost [6]. Therefore, there is a greater than ever demand for prediction the quality and reliability of software.

Among all SRGMs, a large family of stochastic reliability models are based on a non homogeneous Poisson process, which are known as NHPP reliability models. These models have been widely used to track reliability improvement during software testing. These

models enable software developers to evaluate software reliability in a quantitative manner. They have also been successfully used to provide guidance in making decisions such as when to terminate testing the software or how to allocate available recourses. However, software development is a very complex process and there are still issues that have not yet been addressed.

Software fault and failure reports are available in three basic forms:

1. Sequence of ordered failure times $0 < t_1 < t_2 < \ldots < t_n$
2. Sequence of failure times τ_i where $\tau_i = t_i - t_{i-1}$, $i = 1, \ldots, n$
3. Cumulative number of faults.

The general NHPP software reliability growth model is formulated based on the following assumptions:

- The occurrence of software faults follows an NHPP with mean value function $m(t)$ and failure intensity function $\lambda(t)$.
- The software fault intensity rate at any time is proportional to the number of remaining faults in the software at that time.
- When a software fault is detected, a debugging effort takes place immediately.

Let $\{N(t), t \geq 0\}$ denote a counting process representing the cumulative number of faults detected by the time t, and $m(t) = E[N(t)]$ denote its expectation. The failure intensity $\lambda(t)$ and the mean value functions of the software at time t are related as follows

$$m(t) = \int_0^t \lambda(s)ds$$

and

$$\frac{dm(t)}{dt} = \lambda(t).$$

The cumulative number of faults detected at time t follows a Poisson distribution with time-dependent mean value function as follows

$$P\{N(t) = n\} = \frac{m(t)^n}{n!}\, e^{-m(t)}, \quad n = 0, 1, 2, \ldots, \infty$$

The software reliability, i.e., the probability that no failures occur in $(s, s+t)$ given that the last failure occurred at testing time s is

$$R(t|s) = \exp[-(m(t+s) - m(t))]$$

The mean value function $m(t)$ is nondecreasing with respect to testing time t under the bounded condition $m(\infty) = a$, where a is the expected total number of faults to be eventually detected. Knowing its value can help us to determine whether the software is ready to be released to the customers and how much more testing resources are required. It can also provide an estimate of the number of failures that will eventually be encountered by the customers. The mean value function can be expressed as follows

$$m(t) = aF(t),$$

where $F(t)$ is the cumulative distribution function. Hence,

$$\lambda(t) = aF'(t) = [a - m(t)]\frac{F'(t)}{1 - F(t)} = [a - m(t)]\rho(t),$$

where $\rho(t)$ is the failure occurrence rate per fault of the software, or the rate at which the individual faults manifest themselves as failures during testing. The quantity $[a - m(t)]$ denotes the expected number of faults remaining. The failure occurrence rate per fault (also known as *hazard* function)

$$\rho(t) = \frac{\lambda(t)}{m(\infty) - m(t)}$$

can be a constant, increasing, decreasing, or increasing/decreasing.

Table 2 and Figure 1 show examples of NHPP models with different failure intensity functions

Model name	$m(t)$	$\lambda(t)$
Log-linear	$\frac{\exp(\alpha+\beta t)}{\beta}$	$\exp(\alpha+\beta t)$
Exponential (Goel-Okumoto)	$\alpha[1-\exp(-\beta t)]$	$\alpha\beta\exp(-\beta t)$
Weibull (Generalized Goel-Okumoto)	$\alpha[1-\exp(-\beta t^{\gamma})]$	$\alpha\beta\gamma t^{\gamma-1}\exp(-\beta t^{\gamma})$
Power law	$\left(\frac{t}{\alpha}\right)^{\beta}$	$\frac{\beta}{\alpha}\left(\frac{t}{\alpha}\right)^{\beta-1}$
S-shaped	$\alpha[1-(1+\beta t)\exp(-\beta t)]$	$\alpha\beta^2 t\exp(-\beta t)$

Table 1: NHPP models.

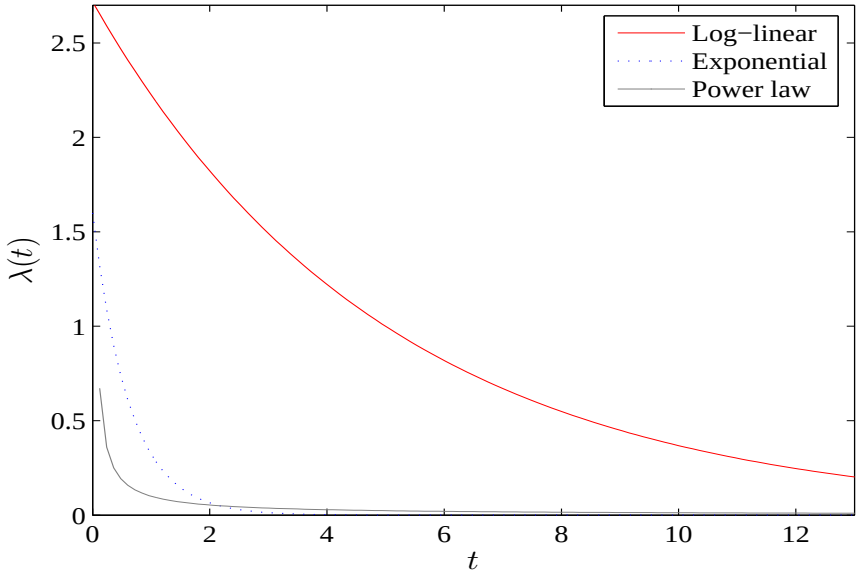

Figure 1: Illustration of failure intensity functions.

1.2.1 Operating characteristic curves

A statistical test provides a mechanism for making quantitative decisions about a process or processes [7]. The intent is to determine whether there is enough evidence to "reject" a conjecture or hypothesis about the process. The conjecture is called the null hypothesis. Not rejecting may be a good result if we want to continue to act as if we "believe" the null hypothesis is true. Or it may be a disappointing result, possibly indicating we may not yet have enough data to "prove" something by rejecting the null hypothesis. A classic use of a

statistical test occurs in process control studies, and it requires a pair of hypotheses:

$$H_0 : \text{ a null hypothesis}$$

$$H_1 : \text{ an alternative hypothesis}$$

The null hypothesis is a statement about a belief. We may doubt that the null hypothesis is true, which might be why we are "testing" it. The alternative hypothesis might, in fact, be what we believe to be true. The test procedure is constructed so that the risk of rejecting the null hypothesis, when it is in fact true, is small. This risk, α, is often referred to as the *significance level* of the test. By having a test with a small value of α, we feel that we have actually "proved" something when we reject the null hypothesis. The risk of failing to reject the null hypothesis when it is in fact false is not chosen by the user but is determined, as one might expect, by the magnitude of the real discrepancy. This risk, β, is usually referred to as the *error of the second kind*. Large discrepancies between reality and the null hypothesis are easier to detect and lead to small errors of the second kind; while small discrepancies are more difficult to detect and lead to large errors of the second kind. Also the risk β increases as the risk α decreases. The risks of errors of the second kind are usually summarized by an *operating characteristic curve* (OC) for the test [7].

1.2.2 Bayesian statistics

Bayesian inference is statistical inference in which evidence or observations are used to update or to newly infer the probability that a hypothesis may be true. The name "Bayesian" comes from the frequent use of Bayes' theorem in the inference process [8, 9]. Bayesian inference uses aspects of the scientific method, which involves collecting evidence that is meant to be consistent or inconsistent with a given hypothesis. As evidence accumulates, the degree of belief in a hypothesis changes. With enough evidence, it will often become

very high or very low. Thus, proponents of Bayesian inference say that it can be used to discriminate between conflicting hypotheses: hypotheses with a very high degree of belief should be accepted as true and those with a very low degree of belief should be rejected as false. However, detractors say that this inference method may be biased due to initial beliefs that one needs to hold before any evidence is ever collected.

Bayesian inference uses a numerical estimate of the degree of belief in a hypothesis before evidence has been observed and calculates a numerical estimate of the degree of belief in the hypothesis after evidence has been observed. Bayesian inference usually relies on degrees of belief, or subjective probabilities, in the induction process and does not necessarily claim to provide an objective method of induction. Nonetheless, some Bayesian statisticians believe probabilities can have an objective value and therefore Bayesian inference can provide an objective method of induction. Bayes' theorem adjusts probabilities given new evidence in the following way:

$$P(H_0|E) = \frac{P(E|H_0)P(H_0)}{P(E)},$$

where

- H_0 represents the null hypothesis that was inferred before new evidence, E, became available.
- $P(H_0)$ is called the prior probability of H_0.
- $P(E|H_0)$ is called the conditional probability of seeing the evidence E given that the hypothesis H_0 is true. It is also called the likelihood function when it is expressed as a function of H_0 given E.
- $P(E)$ is called the marginal probability of E: the probability of witnessing the new evidence E under all mutually exclusive hypotheses. It can be calculated as the sum

of the product of all probabilities of mutually exclusive hypotheses and corresponding conditional probabilities: $\sum P(E|H_i)P(H_i)$.

- $P(H_0|E)$ is called the posterior probability of H_0 given E.

The factor $P(E|H_0)/P(E)$ represents the impact that the evidence has on the belief in the hypothesis. If it is likely that the evidence will be observed when the hypothesis under consideration is true, then this factor will be large. Multiplying the prior probability of the hypothesis by this factor would result in a large posterior probability of the hypothesis given the evidence. Under Bayesian inference, Bayes theorem therefore measures how much new evidence should alter a belief in a hypothesis. Bayesian methods aim at assigning prior distributions to the parameters in the model in order to incorporate whatever *a priori* quantitative or qualitative knowledge we have available, and then to update these priors in the light of the data, yielding a posterior distribution via Bayes Theorem. The ability to include prior information in the model is not only an attractive pragmatic feature of the Bayesian approach, but it is also theoretically vital for guaranteeing coherent inferences.

1.3 Machine learning based software defect prediction models

In order to achieve improvement, quality must be defined and measured. In software industry, quality can be defined simply as lack of "bugs" in the final product [2]. In the process of building high quality softwares, developers and testers put their budget toward artifacts they believe most require quality assurance (QA) activities. This can make a bias to QA in terms of leaving some blind spots behind. In order to avoid having blind spots, the first approach that comes to mind is to manually inspect the code and all other aspects of software modules. It is highly time and money consuming and is considered impractical in

large projects. As Menzies [42] suggested, by practicing a ”lightweight sampling policy” we will be able to explore the rest of the software as well as raising an alert on problematic parts of the software. In other words, the primary step to address the above problem is to predict the number of defects as well as defect prone modules to distribute resources efficiently.

Data mining over static code features is the best known method for building a lightweight sampling policy. ”Data mining” can be defined as extracting patterns from data sets (matrices) having columns as features and rows as observations (examples). This is considered as supervised learning in the absence of a background theory.

As the size and complexity of software increases, manual inspection of software becomes prohibitively expensive. An effective testing method targets minimizing the number of defects while using resources efficiently [36]. In order to prevent exhaustive testing which significantly reduces relevant cost, software defect prediction models have been proposed to allocate testing resources effectively. Thus, defect prediction is of paramount importance to project managers in allocating the limited resources effectively, and it also provides many advantages such as the accurate estimation of project costs and schedules as well as improving product and process qualities. Selecting an appropriate defect predictor is a key practical issue [52] because many modeling approaches have been proposed in the literature including reliability growth models [15–18], Bayesian models [12], and artificial neural networks. Most of these models are built using historical defect data as well as software metrics and are expected to generalize the statistical patterns for unseen projects. Thus, collecting defect data from past projects to implement software metrics [1] is the key challenge for constructing such predictors.

The application of machine learning methods to find patterns in the historical data is the current century’s trend in the field of software defect prediction. Data miners can learn

predictors for software quality from code metrics. Since a major part of this thesis is devoted to application of machine learning methods toward quality, some vital concepts are described next [34]:

> *Machine learning* believes that there is a process that explains the data we observe. Though we do not know the details of the underlying processes, we are able to construct a useful approximation. Application of machine learning methods on large datasets is called *data mining*. Learning large volume of data leads to construct a model that uses small number of valuable data is its primary application. Machine learning is part of artificial intelligence too since it gives the learning ability to the changing systems.

1.3.1 K-fold Cross-validation

In order to generate a classifier, we do need to use a classification algorithm on a dataset. If we run the training once, we will get one classifier and one validation error. To average over randomness, we use the same algorithm and generate multiple classifiers [34].

In K-fold cross-validation, the dataset χ is randomly divided into K equal-sized parts, χ_i, $i = 1, ..., K$. To generate each pair, we keep one of the K parts out as the validation set, and combine the remaining $K - 1$ parts to form the training set. Doing this K times, each time leaving out another one of the K parts out, we get K pairs:

$$\nu_1 = \chi_1, \quad \tau_1 = \chi_2 \cup \chi_3 \cup ... \cup \chi_K$$

$$\nu_2 = \chi_2, \quad \tau_2 = \chi_1 \cup \chi_3 \cup ... \cup \chi_K$$

$$\vdots$$

$$\nu_K = \chi_K, \quad \tau_K = \chi_1 \cup \chi_2 \cup ... \cup \chi_{K-1}$$

As can be observed, one round of cross-validation involves partitioning a sample of data into complementary subsets, performing the analysis on $(K-1)$ *training set*, validating the analysis on 1 *testing set*. As mentioned above, in order to reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds to produce a single estimation. The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once. In other words, each folds contains roughly the same proportions of the the two types of class labels [14, 34].

1.3.2 Supervised machine learning algorithm

Supervised learning can be simply defined as learning behaviors based on empirical data to infer a function. Alpaydin [34] defined a sample as

$$\chi = \{x^t, r^t\}_{t=1}^N$$

where all the instances are drawn from the same joint distribution $p(x, r)$. The parameter t shows one of the N instances, x^t is for demonstrating the arbitrary dimensional input and r^t is the associated desired output (which is $0/1$ in a two-class learning as our upcoming case). The goal is to build a good approximation to r^t using model $g(x^t|\theta)$. Learning model g is denoted as $g(x|\theta)$ where x is the input and θ are the parameters.

The loss function is defined as

$$E(\theta|\chi) = \sum_{t} L(r^t, g(x^t|\theta)),$$

which in class learning, $L(.)$ checks for equality.

Optimization procedure is defined as

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, E(\theta|\chi)$$

where the argmin returns the argument that minimizes the approximation error. Other perspective of looking at the above problem is by estimating the reliability of the system in terms of time to failure.

1.3.3 Metrics

According to Humphrey [1,38], there are four major reasons for collecting data and implementing software metrics:

- Learning software processes, products and services.
- Evaluating as part of the decision-making process to analyze and study to see if standards, goals and criteria are being met.
- Controlling variances, control limits and standards.
- Predicting the values of attributes in the future.

Software metrics have been defined in the literature as "*standardize ways of measuring the attributes of software processes, products, and services in order to provide the information needed to improve those processes, products, and services*" [1].

There are a number of metrics that mostly support the software verification activities, like Complexity metrics (e.g. McCabe Cyclomatic Complexity Metric, Halstaed's software science metrics), defect metrics, product metrics, and process metrics [4].

We have applied our proposed approaches based on statistical and machine learning concepts on two different data repositories. In chapter 2 we are using SAP's CRM (Customer Relationship Management) system defect dataset. This defect dataset from a real SAP project contain monthly software defects that were recorded for a period of 60 months. The testing process of such a large software solution generates messages that identify potential software defects [11–13]. These messages are archived, and software companies have a wealth of historical records about them. In chapter 3, we will learn defect predictors on NASA IV&V Facility Metrics Data Program repository [44]. This public-domain defect dataset contains static software metrics and the associated error data at the module level for NASA software development projects.

The NASA Metrics Data Program Data Repository is a database that stores problem data, product data and metrics data. Menzies claimed [41] that the reason behind learning defect predictors from static code attributes is:

- Using static code attributes results in higher detection ability than currently-used industrial methods such as manual code review.
- Static code metrics (e.g. line of code, McCabe and Halstead attributes) are cheap to collect in contrast to other methods like manual code inspection which is labor-intensive.
- They are used widely by researchers and practitioners.

Static code metrics have some drawbacks too. They are hardly a complete characterization of internal procedure of a module. The value of using them to learn defect predictors has been widely debated.

1.3.4 Dimensionality Reduction

In most learning algorithms, the complexity depends on the number of input dimensions, d as well as on the size of data sample, N. Reduction of data dimensionality d also reduces memory and computation costs as well as decreasing the complexity of the inference algorithm during testing [34]. It is even claimed that certain features of the original data might even reduce the performance of the classifier [35]. The reason behind reducing the dimensionality is data analysis activities like classification that can be done in the reduced space more accurately than in the original space. Finding a subset of data which does not damage the performance of learned predictor has been studied previously [41]. The lower the number of dimensions, the easier to learn a system [47]. There has been researches which apply heuristics to reduce the dimensionality of data to gain better prediction performance [35] and [36]. Applying any heuristic however will lead to a biased search of the final features. It is possible for this bias to limit the novelty and the creativity of the found solutions.

Principal Component Analysis (PCA) - the most used linear technique - maps data to a lower dimension space such that the variance is maximized [36]. PCA technique tries to linearly transform data to a more meaningful basis. It reduces the noise by selecting more important components through diagonalizing the covariance matrix.

Nonlinear Dimensionality Reduction (NLDR) techniques have been proven practical in terms of keeping the general structure of data after transformation. Some of NLDR methods preserve the neighborhood like Maximum Variance Unfolding (MVU) while others minimize a cost function that measures differences between distances in the input and output space. A limiting issue with PCA and other dimensionality reduction methods, either linear or non-linear, is that they have a constant general model and dimension reduction procedure would be about estimation of the model's parameters.

1.3.5 Genetic Programming

Evolutionary Algorithms (EA) are population based. In other words, a whole collection of candidate solutions will be searched simultaneously in order to find the best solution. Figure 2 simply depicts how an EA algorithm works [39].

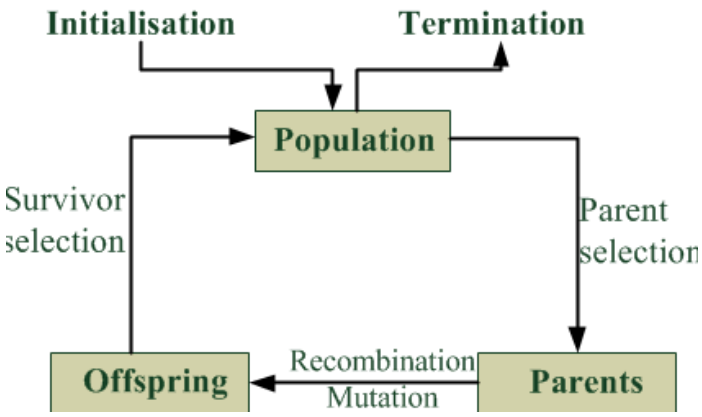

Figure 2: How evolutionary algorithms work!

Genetic Programming (GP), the youngest member of EA family, has been applied in the field of machine learning successfully [45–50]. Unlike most EA

strands which are being used in optimization problems, GP can be easily positioned in machine learning approaches [39]. By having the representation of chromosomes as tree structures, GP is usually used to seek models with maximum fit.

GP has been proven capable and flexible in building expressions based on an objective function. It combines both NLDR approaches by minimizing the cost function which has the goal of keeping the local neighborhood. GP can select linear or non-linear operators to build expressions through Parse Trees. That makes GP a wise choice to construct new lower dimensional features based on original features. As Figure 3 depicts, the penalty is getting minimized in our proposed approach.

1.4 Thesis Overview and Contributions

The organization of this thesis is as follows:

- ❑ The first Chapter contains a brief review of essential concepts and definitions which we will refer to throughout the thesis, and presents a short summary of material relevant to software defect prediction methods, Bayesian statistics, operating characteristic curves, machine learning methods, software metrics, and genetic programming.
- ❑ In Chapter 2, we present a software defect prediction model using operating characteristic curves and Laplace trend statistic [30]. The main idea behind our proposed technique is to use geometric insight in helping construct an efficient and fast prediction method to accurately predict the cumulative number of defects during the software development process.

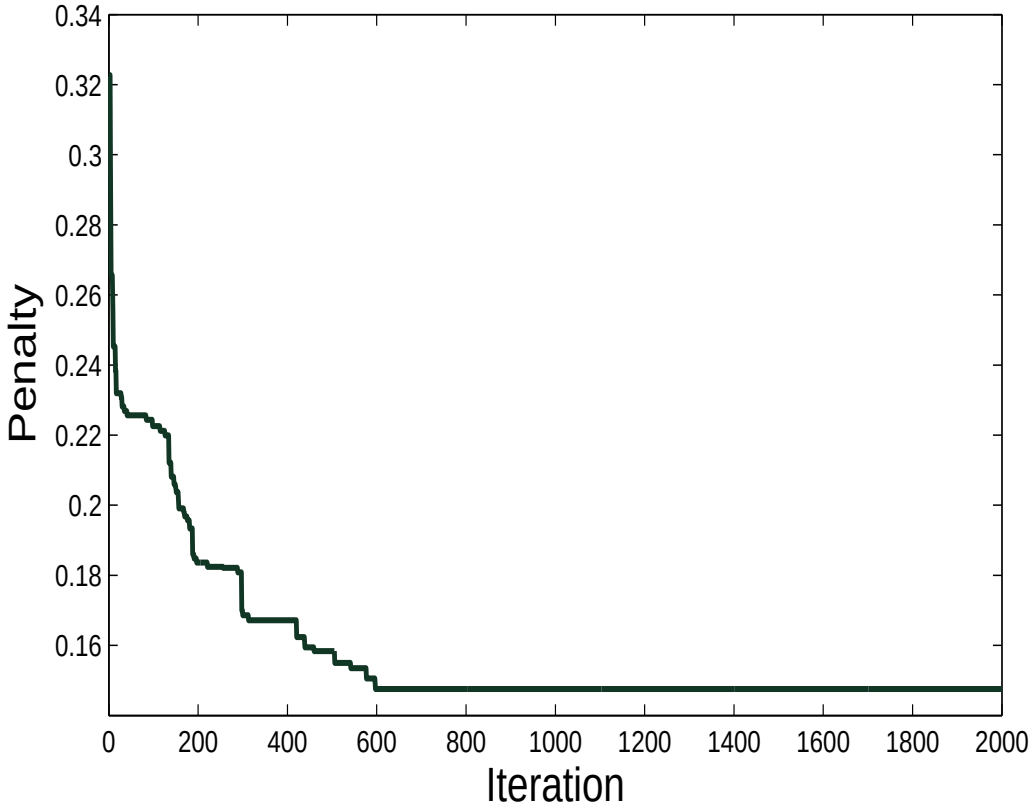

Figure 3: Penalty in one sample genetic programming run.

Experimental results illustrate the effectiveness and the much improved performance of the proposed method in comparison with the Bayesian prediction approaches.

- In Chapter 3, two defect learning methods based on Genetic Programming (GP) concepts are proposed. The first method constructs new features based primarily on the geometrical characteristics of the original data. Then, an independent classifier is applied and the performance of feature selection method is measured. The second method, on the other hand, uses a built-in classifier which automatically gets tuned for the constructed features.

- In the **Conclusions** Chapter, we summarize the contributions of this thesis, and we propose several future research directions that are directly or indirectly related to the work performed in this thesis.

CHAPTER 2

Predictive Operating Characteristic Curves

In this chapter, we introduce a software defect prediction model based on the concept of operating characteristic curve and Laplace trend statistic. The idea is to use operating characteristic curves in statistical quality control and a geometric approach to construct an efficient, fast, and accurate prediction method to estimate the cumulative number of software defects during the software development process. The experimental results demonstrate the effectiveness and the improved performance of the proposed method in comparison with the Bayesian prediction approaches.

2.1 Introduction

Knowledge about the number of expected defects in a software product at any stage provide essential information for decision making in many software development activities, such as cost analysis, resource allocation in testing and release decision time. The aim of software reliability growth modelling (SRGM)

is to explain the behavior of software testing process caused by faults. Most existing SRGMs only model fault detection processes with unrealistic assumptions such as perfect debugging. In this report, we use an improved SRGM with more accuracy and realistic assumptions.

During the development process of computer software systems, many software defects may be introduced and often lead to critical problems and complicated breakdowns of computer systems [5]. Hence, there is an increasing demand for controlling the software development process in terms of quality and reliability. Software reliability can be evaluated by the number of detected faults. A software failure is defined as an unacceptable departure of program operation caused by a software fault remaining in the software system [6, 10]. In the traditional software development environment, software reliability evaluation, which shorten development intervals and reduce development costs, provides useful guidance in balancing reliability, time-to-market and development cost [13]. Hence, there is an increasing demand for prediction the quality and reliability of software.

Several software reliability prediction models have been proposed in the literature for estimating system reliability, but all these kinds of models make unrealistic assumptions to ensure solvability [6, 15–18, 21, 27, 28]. These unreasonable assumptions have limited the applications of these models [12, 52].

Bayesian statistics provide a framework for combining observed data with prior assumptions in order to model stochastic systems. Bayesian methods aim at assigning prior distributions to the parameters in the model in order to incorporate whatever *a priori* quantitative or qualitative knowledge we have

available, and then to update these priors in the light of the data, yielding a posterior distribution via Bayes's Theorem [9]. The ability to include prior information in the model is not only an attractive pragmatic feature of the Bayesian approach, but it is also theoretically vital for guaranteeing coherent inferences.

Motivated by the widely used concept of operating characteristic (OC) curves in statistical quality control to select the sample size at the outset of an experiment [7], we propose in this chapter a software defect prediction technique using OC curves in order to predict the cumulative number of failures at any given time. The core idea behind our proposed methodology is to use geometric insight in helping construct an efficient and fast prediction method to accurately predict the cumulative number of failures at any given time.

The layout of this chapter is organized as follows. In the next Section, a problem formulation is stated. In Section 2.3, we briefly review some Bayesian prediction models that will be used for comparison with our proposed approach. In Section 2.4, we propose a new prediction algorithm based on OC curves. In Section 2.5, we present experimental results to demonstrate the much improved performance of the proposed approach in the prediction of software defects. Finally, some conclusions are included in Section 2.6.

2.2 Problem Formulation

Usually the fault reports are available in three basic forms:

1. in the form of a sequence of ordered time of occurrences

   $0 < t_1 < t_2 < \ldots < t_n$

2. in the form of a sequence of interfailure times τ_i where $\tau_i = t_i - t_{i-1}$ for $i = 1, \ldots, n$

3. in the form of cumulative number of failures detected by a time $N(t_i)$.

Failure(t_i) and interfailure ($\tau_(j)$)times are related by

$$t_i = \sum_{j=1}^{i} \tau_j,$$

The cumulative number of failures defines a non homogeneous Poisson process (NHPP) with failure intensity or rate function $\lambda(t_i)$ which is a function of time. The mean value function $m(t_i) = E(N(t_i))$ of the process is given by $m(t_i) = \int_0^{t_i} \lambda(u)du$. Moreover, the probability of having κ failures in an interval is:

$$\begin{aligned} &P(N(t_j) - N(t_i) = \kappa) \\ &\quad = \frac{(m(t_j) - m(t_i))^\kappa}{\kappa!} \exp(-(m(t_j) - m(t_i))). \end{aligned}$$

This is equal to say $N(t + s) - N(t)$ is a Poisson distributed with expected value

$$\int_{t_i}^{t_j} \lambda(u)du = m(t_j) - m(t_i).$$

where $\lambda(t)$ is the time dependant intensity. Hence, the number of failures in any interval $[t_i, t_j)$ defines a NHPP.

According to ANSI, Software Reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [33]. Although Software Reliability is defined as a probabilistic function, and comes with the notion of time, we must note that, different from traditional Hardware Reliability, Software Reliability is not a direct function

of time. Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not rust or wear-out during its life cycle. Software will not change over time unless intentionally changed or upgraded. Software reliability $R(t_j|t_i)$ is defined as the probability that no software failure is detected in the time interval (t_i, t_i+t_j), given that the last failure occurred at testing time t_i, and it is given by

$$R(t_j|t_i) = \exp\Big(-\big(m(t_i + t_j) - m(t_i)\big)\Big).$$

It is worth pointing out that if the failure intensity function is time-independent, then the cumulative number of failures $N(t_i)$ defines a homogeneous Poisson process (HPP).

Note that the interfailure times may have non-exponential distributions, and hence the cumulative number of failures $N(t_i)$ would define a general renewal process.

The problem addressed in this section may now be concisely described as follows: Given the historical failure times data $\mathcal{D} = \{t_1, \ldots, t_n\}$ and its corresponding cumulative number of failures data $\mathcal{N} = \{N(t_1), \ldots, N(t_n)\}$, find the predicted cumulative number of failures at any given time t.

2.3 Prediction using Bayesian Statistics

Scientific experimental or observational results generally consist of (possibly many) sets of data. Bayesian statistics uses both prior and sample information. Usually something is known about possible parameter values before the experiment is performed.

Model name	$m(t)$	$\lambda(t)$
Log-linear	$\frac{\exp(a+bt)}{b}$	$\exp(a+bt)$
Exponential	$a(1-\exp(-bt))$	$ab\exp(-bt)$
Power law	$\left(\frac{t}{a}\right)^b$	$\frac{b}{a}\left(\frac{t}{a}\right)^{b-1}$

Table 2: NHPP models.

We model the failure times using an NHPP with a parameterized failure intensity function $\lambda(t;\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is a vector of unknown parameters which can be obtained by historical data. Table 2 shows examples of NHPP models with different failure intensity functions $\lambda(t;\boldsymbol{\theta})$, where $\boldsymbol{\theta} = (a, b)$.

Bayesian methods aim at assigning prior distributions to the parameters $\boldsymbol{\theta}$ is the model in order to incorporate whatever *a priori* quantitative or qualitative knowledge we have available, and then to update these priors in the light of the data, yielding a posterior distribution via Bayes's Theorem. The ability to include prior information in the model is not only an attractive pragmatic feature of the Bayesian approach, but it is also theoretically vital for guaranteeing coherent inferences.

2.3.1 Predictive density

Consider the problem of making prediction for a new failure time t without any measurements on the predictors for any of the individuals so that the dataset is just given by $\mathcal{D} = \{t_1, \ldots, t_n\}$. That is, we want to determine $p(t|\mathcal{D})$, the probability density function of the new failure time conditioned on the observed failure times. The function $p(t|\mathcal{D})$ is referred to as *predictive density*

of a new failure time and may be written in integral form as

$$p(t|\mathcal{D}) = \int p(t|\mathcal{D}, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\mathcal{D}) d\boldsymbol{\theta},$$

where $p(\boldsymbol{\theta}|\mathcal{D})$ is the posterior distribution of $\boldsymbol{\theta}$ given by

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathcal{D})} = \frac{\{\prod_{i=1}^{n} p(t_i|\boldsymbol{\theta})\} p(\boldsymbol{\theta})}{\int \{\prod_{i=1}^{n} p(t_i|\boldsymbol{\theta})\} p(\boldsymbol{\theta}) d\boldsymbol{\theta}}$$

and $p(\boldsymbol{\theta})$ is the prior distribution which represents information available about the unknown parameters. The prior estimate provides a means of combining exogenous information with observed data in order to estimate parameters of a probability distribution. It is convenient to choose simple forms of prior distributions which result in computationally tractable posterior distributions. Hence, the posterior distribution is found by combining the prior distribution $p(\boldsymbol{\theta})$ with the probability $p(\mathcal{D}|\boldsymbol{\theta})$ of observing the data given the parameters. The probability $p(\mathcal{D}|\boldsymbol{\theta})$ is also called the likelihood function of the data and it is given by

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{i=1}^{n} p(t_i|\boldsymbol{\theta}),$$

where

$$p(t_i|\boldsymbol{\theta}) = \lambda(t_i; \boldsymbol{\theta}) \exp\left(- \int_0^{t_i} \lambda(u; \boldsymbol{\theta}) du \right)$$

assuming that the failure times data are independent and identically distributed (iid). The likelihood function is the probability of observing the given data as a function of $\boldsymbol{\theta}$.

Hence, the Bayesian approach consists of three main steps:

1. Assign prior distributions to all the unknown parameters.
2. Determine the likelihood of the data given the parameters.

3. Determine the posterior distribution of the parameters given the data.

Maximum Likelihood is a statistical estimator that can be used to estimate a models unknown parameters values from data. The maximum likelihood estimate (MLE) of $\boldsymbol{\theta}$ is that value of $\boldsymbol{\theta}$ that maximizes the likelihood function $p(\mathcal{D}|\boldsymbol{\theta})$ or equivalently that maximizes the log-likelihood function:

$$\log(p(\mathcal{D}|\boldsymbol{\theta}))$$

and it is the value that makes the observed data the most "probable".

2.3.2 Bayesian prediction

The Bayesian prediction approach proposed in [11] is based on the power law model shown in Table 2. The parameter b of the power law model may be estimated as follows

$$\widehat{b} = \frac{t_n}{\sum_{t=t_1}^{t_n} \log[N(t_n)/N(t)]},$$

and the predicted cumulative number of defects $N(t)$ at time t is given by

$$N(t) = N(t_n) \left(\frac{t}{t_n} F(2t, 2t_n; \gamma) \right)^{1/\hat{b}}, \tag{1}$$

where $\gamma = P\{\chi_n^2 \leq \chi_{\gamma,n}^2\}$, and $F(2t, 2t_n; \gamma)$ denotes the γ percentage point of the F-distribution with $2t$ and $2t_n$ degrees of freedom.

2.3.3 Bayesian prediction using MCMC

Markov chain Monte Carlo (MCMC) methods (which include random walk Monte Carlo methods), are a class of algorithms for sampling from probability

distributions based on constructing a Markov chain that has the desired distribution as its equilibrium distribution. The state of the chain after a large number of steps is then used as a sample from the desired distribution. The quality of the sample improves as a function of the number of steps.

If we draw samples $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(N)}$ from the posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$, then the predictive density may be approximated as follows

$$p(t|\mathcal{D}) \approx \sum_{i=1}^{N} p(t|\mathcal{D}, \boldsymbol{\theta}^{(i)}) p(\boldsymbol{\theta}^{(i)}|\mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} p(t|\mathcal{D}, \boldsymbol{\theta}^{(i)}).$$

The samples $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(N)}$ are draws from the posterior distribution of $\boldsymbol{\theta}$, and may be obtained using Markov chain Monte Carlo (MCMC) simulation algorithms [8, 29].

For the Bayesian prediction approach using MCMC, the predicted cumulative number of defects $N(t)$ at time t is also given by Eq. (1) where $\hat{b}$ is estimated using the MCMC algorithm [8].

The algorithm of MCMC estimate parameters $\widehat{b}$ consists of the following steps:

1. Using MCMC to simulate each parameter distribution.
2. Estimate the maximal likely value of parameter distribution which gives us the value of expected parameter.

2.4 Proposed Method

2.4.1 POC curve

Consider the two-sided hypothesis

$$H_0 : \quad t = t_k$$

$$H_1 : \quad t \neq t_k$$

where H_0 and H_1 are the null and the alternative hypotheses respectively.

Define $\chi^2_{\alpha,k}$ as the percentage value of the chi-square distribution with k degrees of freedom such that the probability that the chi-square distribution χ^2_n exceeds this value is α, that is

$$P\{\chi^2_k \geq \chi^2_{\alpha,k}\} = \alpha = P\{\text{reject } H_0 | H_0 \text{ is true}\},$$

where $\alpha \in (0, 1)$ is the probability of type I error (also referred to as the significance level). In other words we can be $100(1-\alpha)\%$ confident about the result.

Note that in probability theory and statistics, the chi-square distribution are k independent, normally distributed random variables.

Suppose $t = t_k + \delta$, where $\delta > 0$ (we have the same result for $\delta < 0$) then H_0 is false and H_1 is true. Hence, the distribution of the test statistic

$$Z = \frac{\chi^2_t - t_k}{\sqrt{2k}}$$

has a mean value equal to $\delta/\sqrt{2k}$, and a type II error will be made only if $-\chi^2_{\alpha/2} \leq Z \leq \chi^2_{\alpha/2}$. That is, the probability of type II error $\beta = P\{\text{accept } H_0 | H_0 \text{ is false}\}$

may be expressed as

$$\beta = \Phi\left(\chi^2_{\frac{\alpha}{2},t} - \frac{\delta}{\sqrt{2k}}\right) - \Phi\left(-\chi^2_{\frac{\alpha}{2},t} - \frac{\delta}{\sqrt{2k}}\right),$$

where Φ is the cumulative distribution function of χ^2_t.

The function $\beta(t)$ is evaluated by finding the probability that the test statistic Z falls in the acceptance region given a particular value of t.

An operating Characteristic (OC) curve is a graph used to determine the probability of accepting lots as a function of the lots or processes quality level when using various sampling plans. In other words the operating characteristic (OC) curve of a test is the plot of $\beta(t)$ against t. Note that given the OC curve parameters β, α, k, and δ, we can derive the predicted cumulative number of defects at time t as follows

$$N(t) = \left(\frac{\sqrt{2k}}{\delta}\right)^2 \left(\chi^2_{\alpha,\delta} + \chi^2_{\beta,\delta}\right)^2. \tag{2}$$

Figure 4 depicts a plot of the cumulative number of defects using OC curves.

The above method does not take into account the historical data to predict. To overcome this limitation, we propose a predictive operating characteristic (POC) curve where the predicted cumulative number of defects at time t is calculated as follows

$$N(t) = \left(\frac{\sqrt{2p}}{\delta}\right)^2 \left(\chi^2_{\alpha,\delta} + \chi^2_{\beta,\delta}\right)^2, \tag{3}$$

and the parameter p is given by (see Figure 5)

$$p = \begin{cases} N(t), & \text{if } t \leq t_n \\ N(t_n), & \text{if } t_n < t \leq T. \end{cases}$$

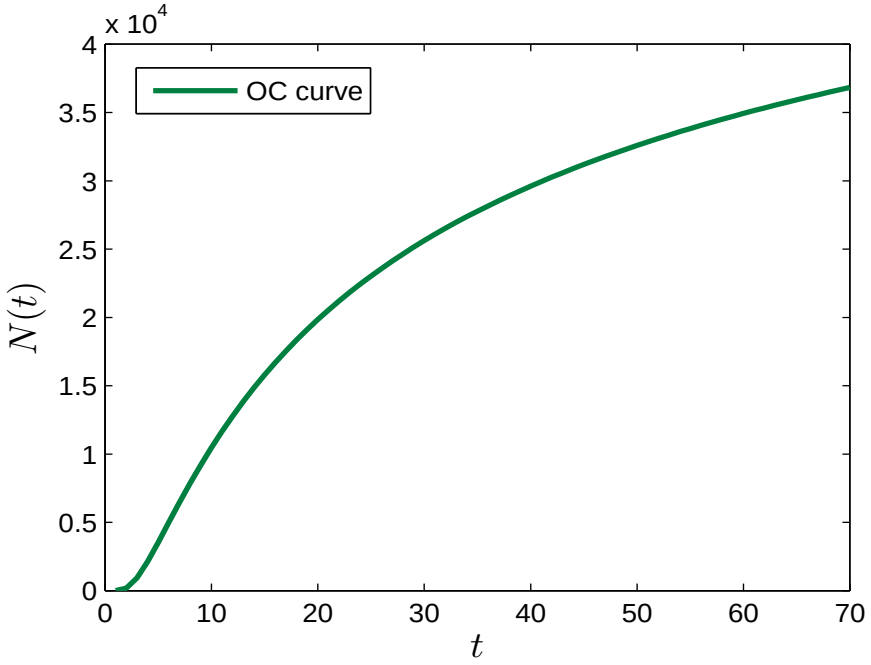

Figure 4: Illustration of cumulative number of defects using OC curves.

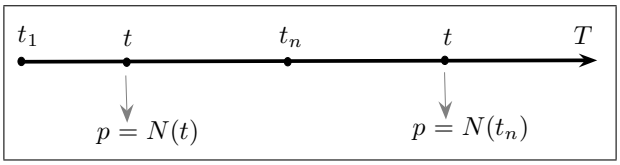

Figure 5: Illustration of the p parameter in the POC curve.

2.4.2 Laplace trend analysis

One of the drawbacks of POC prediction method is its inability to predict accurately the cumulative number of defects when the software is not stable, that is when the software does not have a reliability growth yet. To circumvent this limitation, we used a weighted Laplace trend to validate the reliability and stability of the software before using POC for defect prediction [31].

Suppose we wish to test the following hypotheses:

$$H_0 : \quad HPP$$

$$H_1 : \quad NHPP$$

where H_0 and H_1 are the null and the alternative hypotheses respectively.

Under the null hypothesis, we define the Laplace trend as:

$$U = \frac{\mathcal{L}(\theta_i)'}{E(-\mathcal{L}(\theta_i)'')}$$

where θ_i is a component of the vector θ such that its value makes the intensity function $\lambda(t;\theta)$ time independent.

With type I error probability α, we have the following interpretation of the Laplace trend value:

- $U < -z_\alpha$: reliability growth (stable system behavior).
- $U > z_\alpha$: reliability deterioration (non-stable system).
- $-z_\alpha < U < z_\alpha$: stable reliability (in control behavior).

where z_α is the upper α percentage of the standard normal distribution Z such that $P\{Z \geq z_\alpha\} = \alpha$. If H_0 is true, the distribution of the Laplace test statistic approximately follows standard normal distribution $N(0,1)$.

Note that Laplace trend analysis is used to determine whether the pattern of defects is significantly changing with respect time or not. To have a better analysis we may also use a weighted Laplace test statistic as discussed in [32]. However, for simplicity we focus on the standard Laplace test statistic.

Now we try to find a "Laplace trend stopping increase" point ($t = t_s$) as shown in Figure 6. We can start using the POC curve when Laplace trend starts to decrease ($t = t_s, ..., T$) because at this point the behavior of the system becomes stable and therefore we have reliability growth.

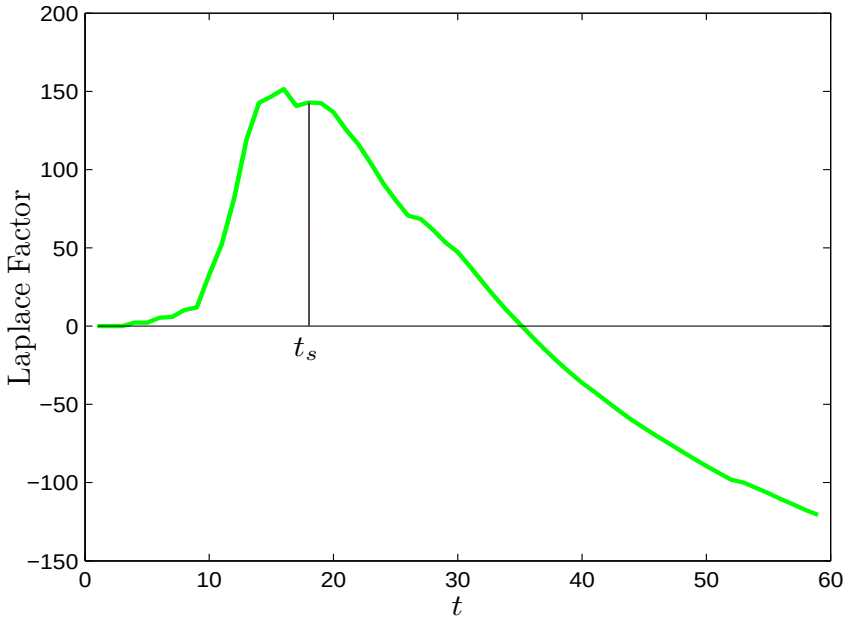

Figure 6: Laplace Factor vs. Defect Time.

2.4.3 Improved POC curve

In a real software project, removal of one defect might cause other defects in the system. In addition, the defect causing the failure cannot be removed immediately. In the improved POC curve approach, we can incorporate these assumptions to be able to predict the behavior of the software in a better way.

To overcome the problem of imperfect debugging, we assume that when a defect occurs and the correction process has been performed the defect is repaired with a probability p, in which case the defect rate is reduced by $\lambda(t)$. Otherwise the number of defects in the software and the defect rate remains the same. Therefore, the total number of expected occurrence of a defect in the system is $1/p$. Hence, the predicted cumulative number of defects in the

system at time t becomes

$$N(t) = \frac{1}{p}\left(\frac{\sqrt{2p}}{\delta}\right)^2 \left(\chi^2_{\alpha,\delta} + \chi^2_{\beta,\delta}\right)^2, \tag{4}$$

Moreover, if the information of defect detection process and defect correction process is available, we can model the defect detection process separately from the defect correction process. On the other hand, due to the fact that a defect can be removed only after its detection; it is more appropriate if the defect correction process to be delayed defect detection process. For simplicity we can assume for each detected defect takes the same amount of time Δ. Hence, given the defect rate $\lambda(t)$, the intensity of defect correction is given by

$$\lambda_c(t) = \begin{cases} 0 & t < \Delta \\ \lambda(t-\Delta) & t \geq \Delta \end{cases}$$

Hence, the predicted cumulative number of corrected defects in the system at time t is given by

$$N_c(t) = \frac{1}{p}\, N(t-\Delta) \tag{5}$$

With these improvements, we can now describe and predict the software defect behavior in its life cycle.

2.5 Experimental Results

We tested our proposed method on real software datasets (DS I and DS II) that were taken from SAP development systems. These datasets contains monthly software defects that were recorded for a period of 60 and 59 months as shown in Table 3 and Table 4 respectively.

Month (t_i)	$N(t_i)$	Month (t_i)	$N(t_i)$
1	17	31	2,217
2	39	32	2,430
3	53	33	2,586
4	87	34	3,884
5	106	35	4,099
6	140	36	4,385
7	165	37	5,104
8	286	38	8,074
9	359	39	10,120
10	412	40	12,618
11	461	41	16,715
12	555	42	21,606
13	654	43	24,592
14	747	44	27,789
15	836	45	29,739
16	926	46	30,843
17	989	47	32,011
18	1,049	48	32,599
19	1,103	49	33,010
20	1,152	50	33,707
21	1,182	51	34,103
22	1,213	52	34,426
23	1,225	53	34,736
24	1,266	54	34,903
25	1,306	55	35,110
26	1,331	56	35,261
27	1,363	57	35,440
28	1,443	58	35,614
29	1,495	59	35,763
30	1,737	60	35,876

Table 3: Software defect data (DS I).

Month (t_i)	$N(t_i)$	Month (t_i)	$N(t_i)$
1	3	31	34,909
2	5	32	35,055
3	19	33	35,129
4	30	34	35,198
5	74	35	35,269
6	115	36	35,339
7	543	37	35,421
8	1,379	38	35,556
9	3,372	39	35,617
10	7,272	40	35,664
11	11,434	41	35,707
12	14,291	42	35,789
13	17,429	43	35,852
14	18,806	44	35,922
15	21,625	45	35,951
16	24,201	46	35,974
17	26,096	47	36,004
18	27,221	48	36,032
19	28,395	49	36,047
20	29,105	50	36,292
21	29,553	51	36,374
22	30,133	52	36,448
23	30,712	53	36,469
24	32,111	54	36,510
25	32,894	55	36,521
26	33,476	56	36,574
27	34,209	57	36,606
28	34,499	58	36,617
29	34,658	59	36,631
30	34,781		

Table 4: Software defect data (DS II).

In all the experiments, we use a probability of type I error α = 0.01. The value of γ was set to 1-α. Figure 7 and Figure 8 depict the cumulative number of defects versus defect time (month) during a software life cycle.

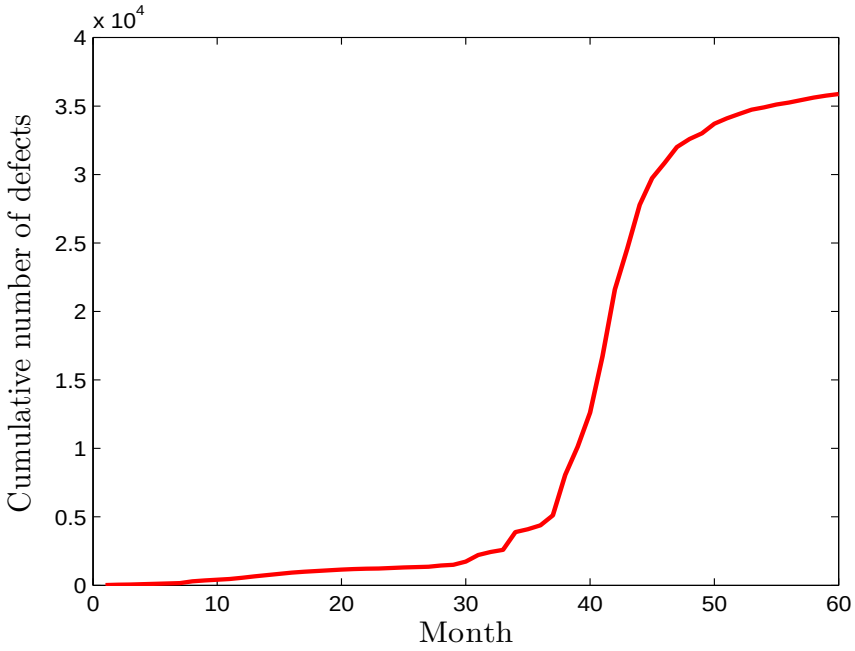

Figure 7: Cumulative Number of Defects vs. Defect Time (DS I)

Figure 9 and Figure 10 displays Laplace factor vs. Defect Time, and it clearly illustrates after the 45^{th} month for DS I and after the 15^{th} month for DS I, the Laplace trend starts to decrease.

2.5.1 Qualitative evaluation of the proposed method

In this subsection, we present simulation results where the Bayesian prediction method [11] and the POC curve algorithm are applied to the software failure dataset (DS I) and also to the truncated software failure data (DS II). Laplace trend starts to decrease, meaning that software reliability starts to grow. Based on our extensive experimentation, we decided to start applying the model from

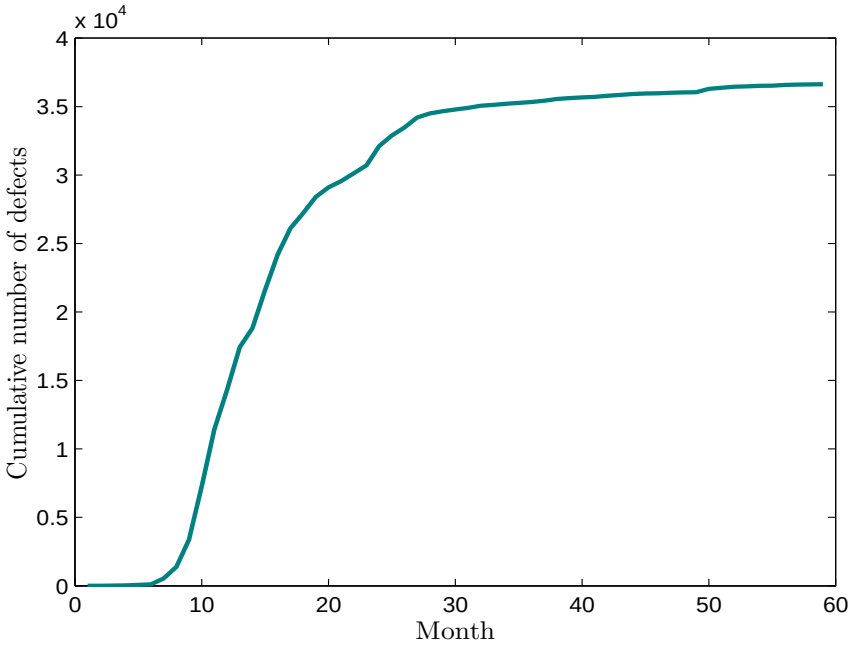

Figure 8: Cumulative Number of Defects vs. Defect Time (DS II)

this point. Figure 11 through Figure 14 show the prediction results of the proposed POC curve in comparison with the Bayesian approaches for both datasets DS I and DS II. These results indicate that our method outperforms the Bayesian techniques used for comparison. Moreover, the proposed method is simple and easy to implement.

2.5.2 Quantitative evaluation of the proposed method

Denote by $N_o(t)$ and $N_p(t)$ the observed and the predictive cumulative number of failures respectively. To quantify the better performance of the proposed predictive method in comparison with the Bayesian approaches, we computed three goodness-of-fit measures: the skill score, the Nash-Sutcliffe model efficiency coefficient, and the relative error between the observed $T_o \times 2$ data

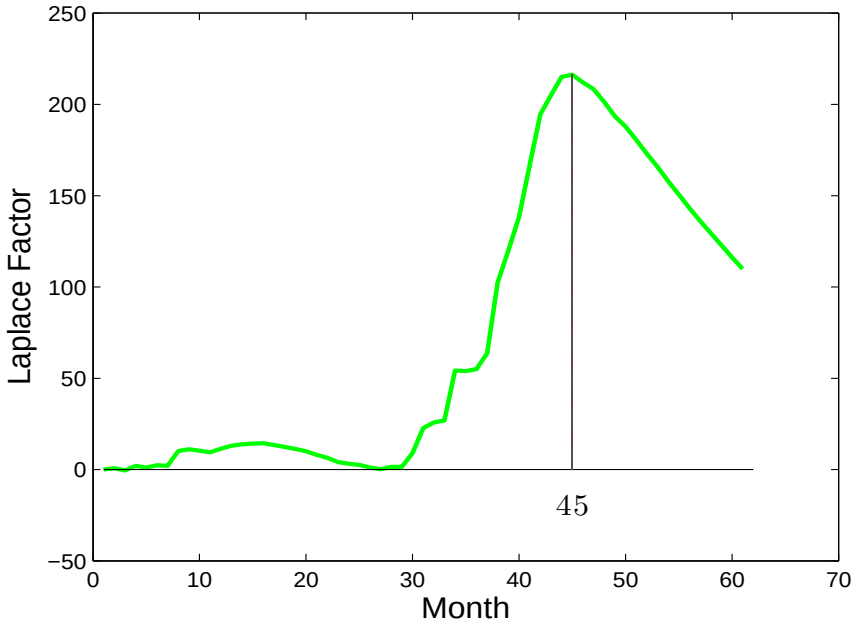

Figure 9: Laplace Factor vs. Defect Time (DS I).

matrix

$$\mathcal{D}_o = \{(t, N_o(t)) : t = 1, \ldots T_o\},$$

and the predicted $T_p \times 2$ data matrix

$$\mathcal{D}_p = \{(t, N_p(t)) : t = 1, \ldots T_p\}.$$

Note that the size of observed data matrix $\mathcal{D}_o$ may not be equal to the size of the predicted data matrix $\mathcal{D}_p$, and hence an intersection step is necessary to pair up the observed data to the predicted data. This intersection function is setup to pair up the first column in the observed data matrix and the first column in the predicted data matrix. Data values are located in the second column of both matrices. More precisely, we create a subset of matched data $\mathcal{D}_m = \{t, N_o(t), N_o(t) : t = 1, \ldots T_m\}$ that would be used to compute the following goodness-of-fit measures:

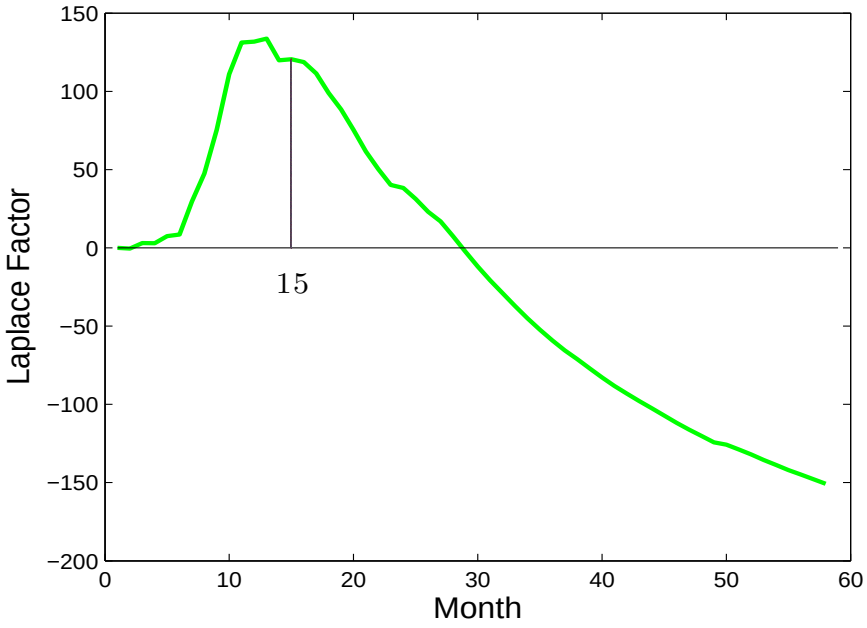

Figure 10: Laplace Factor vs. Defect Time (DS II).

1. **Skill Score**: it is a error statistic that is used to quantify the accuracy of prediction models, and it defined as follows

   and the predicted data, and σ_{N_o} is the sample standard deviation of the observed data.

$$SS = 1 - \frac{\sqrt{\frac{1}{T_m}\sum_{t=1}^{T_m}\left(N_o(t) - N_p(t)\right)^2}}{\sqrt{\frac{1}{T_m-1}\sum_{t=1}^{T_m}\left(N_o(t) - \overline{N}_o\right)^2}},$$

   The model prediction is better, when the value of the skill score SS is closer to one. When SS is less than zero, the model predictions are poor and the model errors are greater than observed data variability.

2. **Nash-Sutcliffe model efficiency coefficient**: is an indicator of the model's

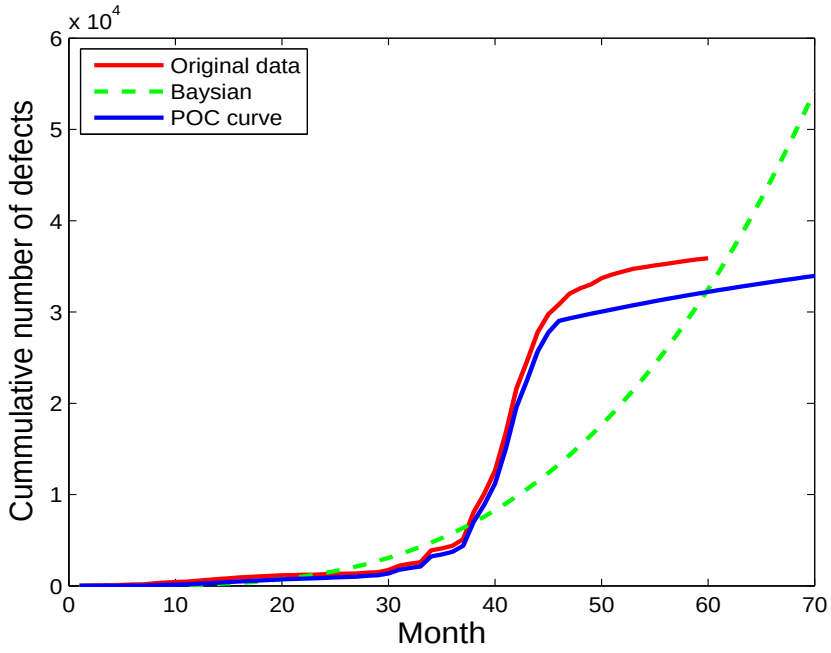

Figure 11: Comparison of the prediction results for known 46 months history DS I.

ability to predict about the 1:1 line between the observed and the predicted data, and it is defined as follows

$$E = 1 - \frac{\sum_{t=1}^{T_m} \big(N_o(t) - N_p(t)\big)^2}{\sum_{t=1}^{T_m} \big(N_o(t) - \overline{N}_o\big)^2}.$$

The Nash-Sutcliffe model efficiency coefficient is a statistic similar to the skill score in that the closer to one the better the model prediction. A value of $E = 1$ indicates that the model prediction is perfect, and if the value of E is equal to or less than zero, then the model prediction is considered poor.

3. **Relative error**: it measures how close a model is estimated with respect to the actual data. The relative error(RE) is defined as

$$RE = \frac{N_p(t) - N_o(t)}{N_o(t)}, \quad t = 1, \ldots, T_m$$

The values of the three goodness-of-fit measures for all the experiments are

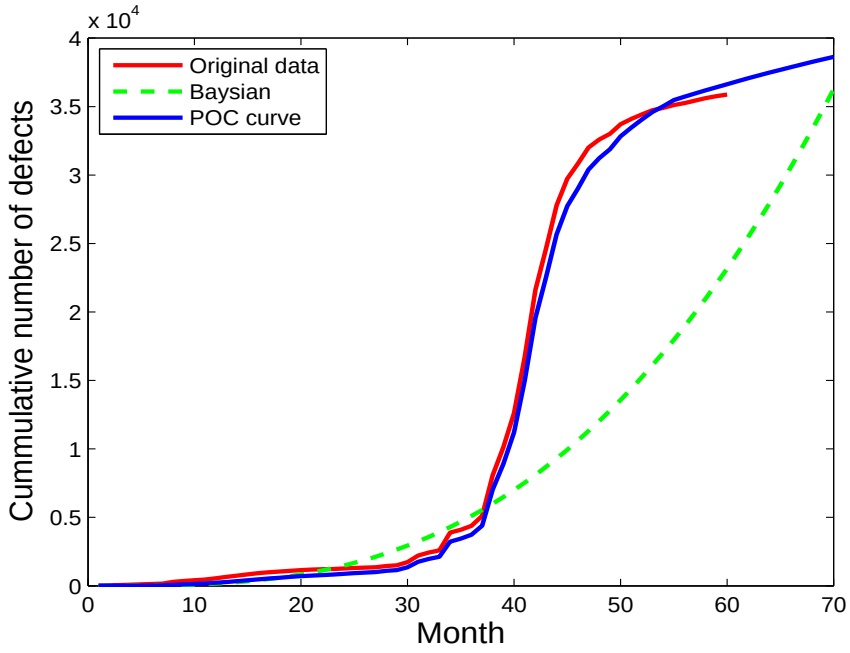

Figure 12: Comparison of the prediction results for known 55 months history DS I.

Skill Score	DS I	DS II
Bayesian	0.3964	0.4031
Bayesian MCMC	0.5426	0.628
OC curve	0.9377	0.7877

Table 5: Skill score results.

depicted in Figure 15 through Figure 20, which clearly show that the proposed method gives the best results indicating the consistency with the subjective comparison.

Nash-Sutcliffe	DS I	DS II
Bayesian	0.6295	0.6259
Bayesian MCMC	0.7872	0.8547
OC curve	0.9961	0.9527

Table 6: Nash-Sutcliffe score results.

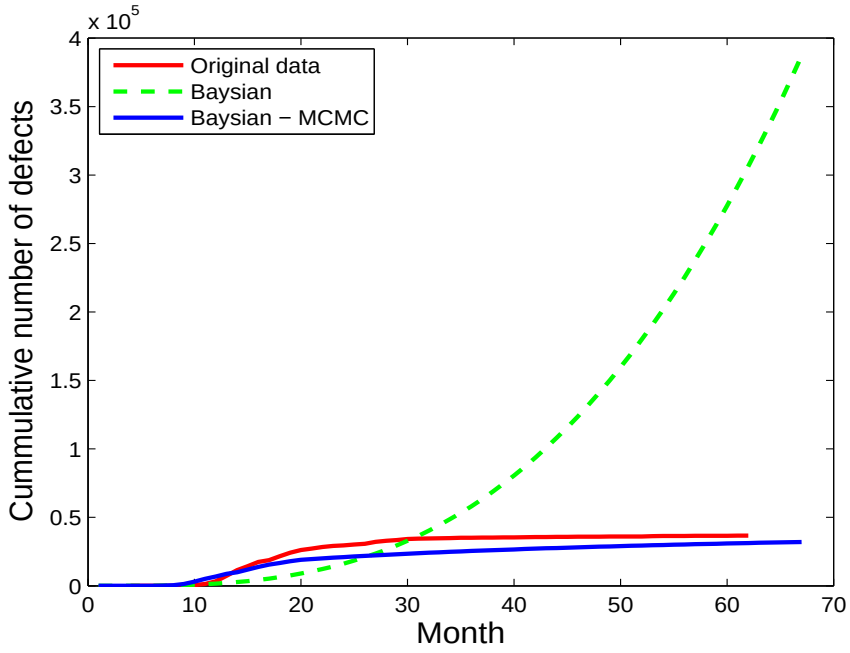

Figure 13: Comparison of the prediction results for known 20 months history DS II.

2.6 Conclusions

In this chapter, we introduced a new method for software defects prediction using operating characteristic curves and Laplace trend statistic. The core idea behind our proposed technique is to reliably predict the cumulative number of defects during the software development process. The prediction accuracy of the proposed approach is validated on a real software failure data using several goodness-of-fit measures. The experimental results clearly show a much improved performance of the proposed approach in comparison with the Bayesian prediction methods.

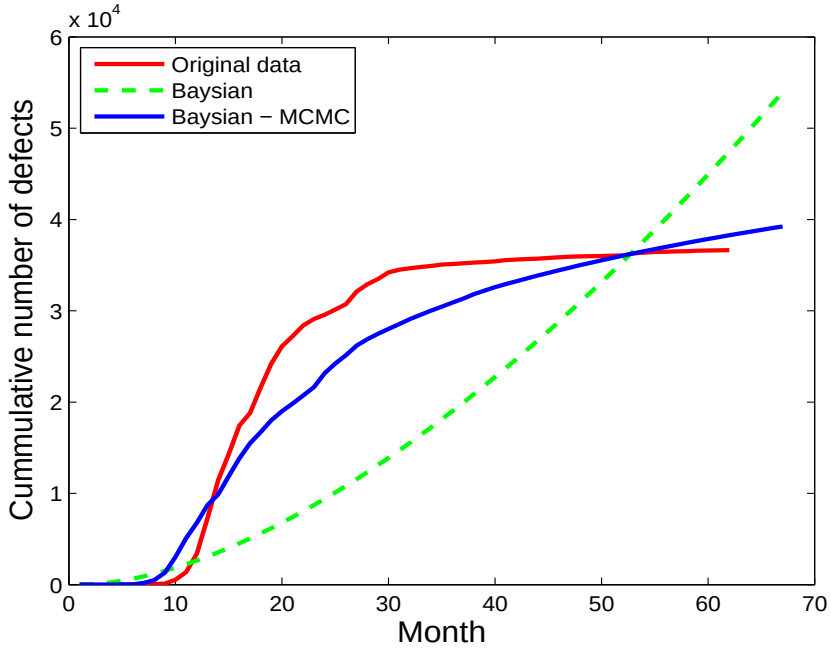

Figure 14: Comparison of the prediction results for known 40 months history DS II.

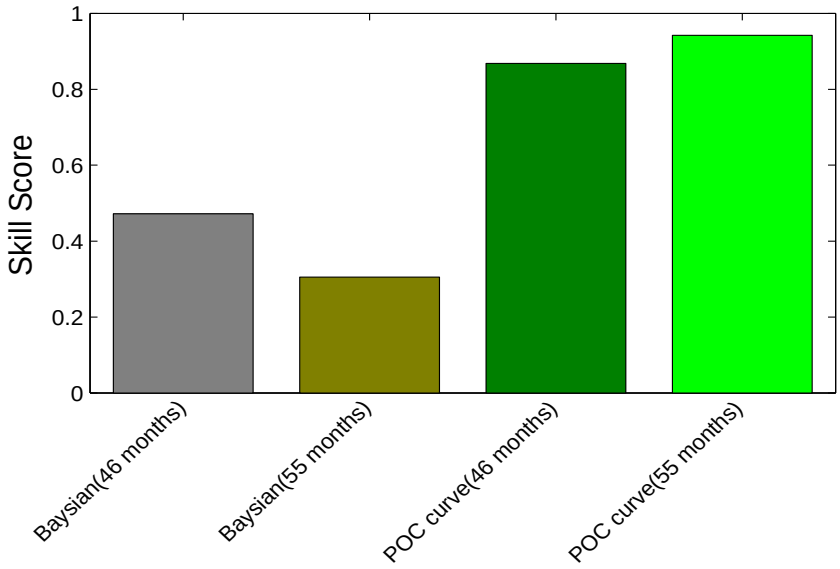

Figure 15: Skill score results for DS I.

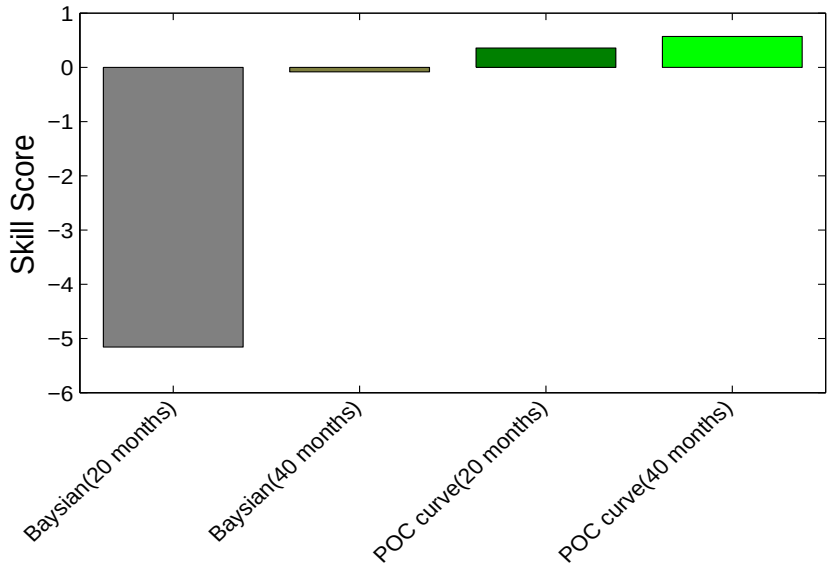

Figure 16: Skill score results for DS II.

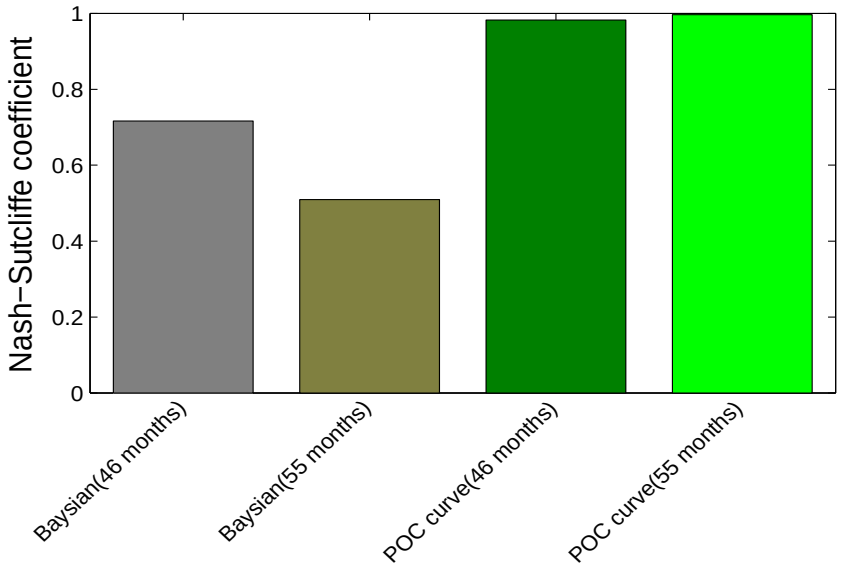

Figure 17: Nash-Sutcliffe model efficiency coefficient results for DS I.

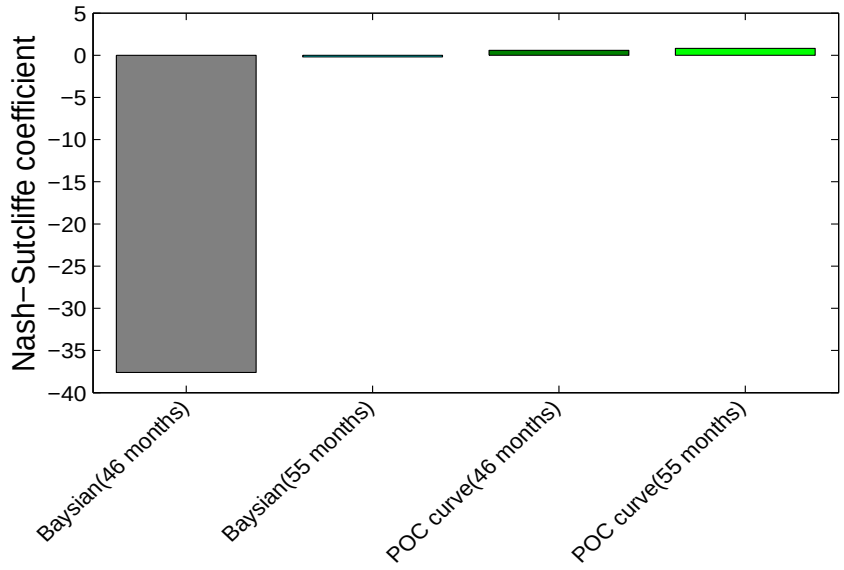

Figure 18: Nash-Sutcliffe model efficiency coefficient results for DS II.

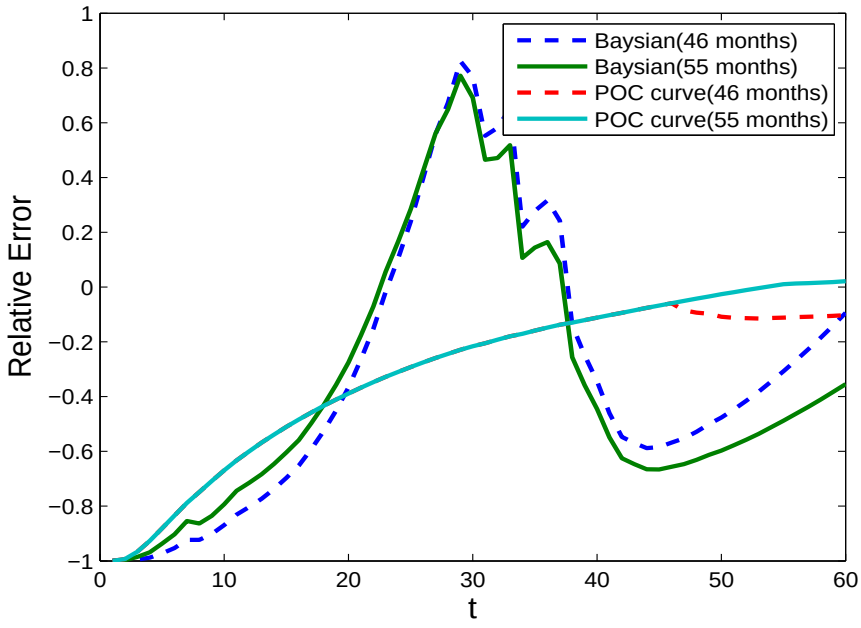

Figure 19: Relative error results for DS I.

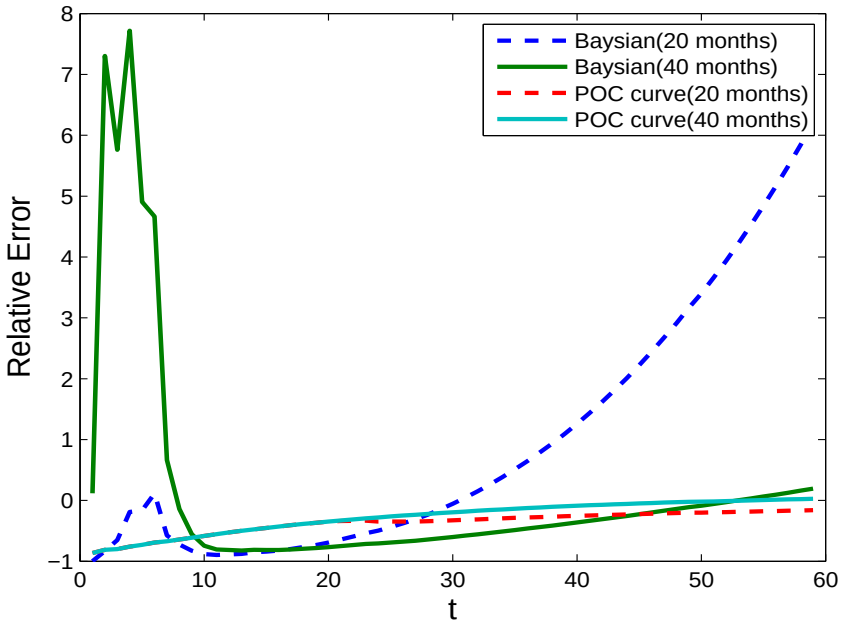

Figure 20: Relative error results for DS II.

CHAPTER 3

Defect Prediction via Genetic Programming

In this chapter, software defect prediction models using Genetic Programming (GP) are proposed. Inductive learning from examples is a widely known practice in the field of machine learning that aims at predicting the number of defects in order to allocate resources more efficiently. Feature reduction/transformation is used extensively to improve the learning capability of the learner machine [36, 41, 46]. Defect prediction will be carried out through our proposed Evolutionary Algorithm (EA) method with the aim of preserving the global structure of the data. Unlike previous approaches, our proposed approach achieves a better performance by taking no heuristics into account. We have tried to follow the baseline experiments on the NASA IV&V static metric dataset [36, 41]. Experimental studies have been assessed as significant in terms of having high detection rate while keeping mis-detection rate low.

3.1 Problem Statement

The quality of software before usage is of great concern to software companies. This issue has been extensively investigated using a variety of statistical methods over the last three decades. Most of these efforts have focused on predicting the number of defects, that is deviations from specifications or expectations which might lead to failures in operation in the software system [52]. Prediction outcome which is the number of remaining defects in a software system, is a critical controlling factor and an important informative measure for software developer and a gauge for likely delivered quality of software systems [57].

Learning defect predictors has been known as an efficient approach in the field of Software Quality Assurance (SQA). Applying them can lead to define testing priorities better in order to prevent exhausting testing, the most costly part of software development life cycle [35]. Numerous defect datasets have been collected from different projects to study various statistical and machine learning approaches. Menzies [41] introduced the baseline experiment using NASA public domain data repository [44] ,leading other researchers to use that repository in order to create repeatable, verifiable, refutable, and/or improvable predictive models in software engineering [43, 44]. Reaching that objective can be considered essential in order to accelerate toward maturity of this research discipline.

Since publishing different conclusions based on different datasets makes the comparison among used techniques almost impossible, Menzies [41] definition of the baseline experiment has been widely followed by the other researchers [35, 36, 59].

3.1.1 Static Code Attributes

Static code attributes are easy to collect and are good indicators of models that need to be reviewed and inspected. Verification and Validation (V&V) textbooks like [4] advise using static code complexity attributes to decide which modules are worthy of manual inspection. It has been stated that at the NASA IV&V facility, several large government software contractors will not review software modules unless tools like McCabe predict that some of them might be defect prone [42]. This fact clearly reveals the importance of static metrics. They have also been claimed as a trustable source for performing repeatable experiments like software quality predictors.

NASA IV&V Facility provides public domain access to NASA Metric Data Program Repository [43, 44]. The goal is to provide project non-specific data to the software community. Defect predictors can be learned from datasets containing static code features, whose class label is *defective* and whose values are *true* or *false*. Depending on the language, rows describe data from a *module*, *function*, *method*, *procedure* or *files*. Columns demonstrate one of the static code features that can be found on Table 8. The *defective* column shows the result of a whole host of QA methods that were applied to that historical data [42]. The repository contains McCabe, Halstead, Line of Code, error metrics derived from the association between errors and functions/modules, and requirement metrics of different projects as depicted in Table 7. Table 8 contains further details on these attributes.

McCabe		vg	cyclomatic_complexity
		$iv(G)$	design_complexity
		$ev(G)$	essential_complexity
locs	loc		loc_total(one line=one count)
	loc(other)		loc_blank
			loc_code_and_comment
			loc_comments
			loc_executable
			number_of_lines
			(opening to closing brackets)
Halstead	h	N_1	num_operators
		N_2	num_operands
		μ_1	num_unique_operators
		μ_2	num_unique_operands
	H	N	length: $N = N_1 + N_2$
		V	volume: $V = N * \log_2 \mu$
		L	level: $L = V^*/V$ where $V^* = (2 + \mu_2^*) \log_2(2 + \mu_2^*)$
		D	difficulty: $D = 1/L$
		I	content: $I = \hat{L} * V$ where $\hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$
		E	effort: $E = V/\hat{L}$
		B	error_est
		T	prog_time: $T = E/18$ seconds

Table 7: Static code features of NASA IV&V dataset

3.1.2 Data preprocessing

Menzies [41] has named three elements to present an experiment: Data to be processed, a processing method, and a reporting method. Therefore, we need to perform some activities in order to make the data ready for the processing phase. These preprocessing activities can be minor like initial dataset modifications, i.e. removing constant attributes and replacing NaNs (Not a Number), or can be major like transforming the data representation space. Menzies [41] and Turhan [36] made small changes to clean the data before applying the logarithmic filter on all numerical values with the hope of improving the predictor's

Table 8: Static Metrics of NASA dataset

Static Metrics	Symbol
McCabe Software Metrics	
Cyclomatic Complexity	$V(g)$
Cyclomatic Density Metric	$Vd(g)$
Decision Density Metric	$Dd(g)$
Design Density Metric	$Id(g)$
Essential Complexity	$Ev(g)$
Essential Density Metric	$Ed(g)$
Global Data Density Metric	$Gd(g)$
Global Data Complexity Metric	$Gdv(g)$
Maintenance Severity Module Design Complexity	$Iv(g)$
Normalized Cyclomatic Complexity Metric	Norm $V(g)$
Pathological Complexity Metric	$Pv(g)$
Error Count Metrics	
Number of Errors (No. associated problem records)	
Error Density (No. errors per 1000 lines of code)	
Number of Problem Records in 6 Months	
Number of Problem Records in 1 Year	
Number of Problem Records in 2 Years	
Halstead Metrics	
Halstead Length	(N)
Halstead Volume	(V)
Halstead Level	(L)
Halstead Difficulty	(D)
Halstead Intelligent Content	(I)
Halstead Programming Effort	(E)
Halstead Error Estimate	(B)
Halstead Programming Time	(T)
Line of Code Metrics	
Branch Count	
Call Pairs	
Condition Count	
Decision Count	
Edge Count	
Formal Parameter Count	
Modified Condition Count	
Multiple Condition Count	
Node Count	
Number of Lines	
Number of Operators	
Number of Operands	
Number of Unique Operators	
Number of Unique Operands	
Number of Executable of Lines of Code	
Number of Lines of Comment	
Number of Lines of Code containing both Code and Comment	
Percent of Code that is Comments	

performance. We have also implemented some minor modification steps which will be described in the next section.

Gray [37] has applied an extensive data cleaning including removing repeated and inconsistent instances, removing constant attributes, replacing missing values, balancing the data, normalization, and randomizing instance order. We strongly believe that manipulating data higher than normal modifications changes its specific characteristics such as its global structure. Data manipulation can be useful when a learner machine is weak. Having the goal of proposing a solid model, we have tried to reduce this phase to let evolutionary algorithm decides its best. We will go through our contribution in the next section.

3.1.3 Classification

Common prediction models that have been used in previous studies [36,41] are Quadratic Discriminant, Linear Discriminant, and Naive Bayes. Turhan [36] validated the assumptions of Naive Bayes in defect prediction context. These assumptions are Independence of Attributes as well as their Equal Importance. The results of relaxing independence of attributes assumption in Naive Bayes shows that it is not harmful for software defect data after Principal Component Analysis (PCA) pre-processing. Overcoming the other assumption (i.e. equal importance of attributes) may produce significantly better results than standard Naive Bayes. It has been shown that subset selection [41] can be outperformed by other dimensionality reduction techniques like PCA in conjunction with Naive Bayes classifier.

Predicted class		
Yes	No	True Class
TP: True Positive	FN: False Negative	Yes
FP: False Positive	TN: True Negative	No

Table 9: Confusion Matrix

We have implemented the same three techniques as the classifier functions in our supervised learning approach. GP does not take into account any assumptions since it is based on no bias as well as no heuristics. There will be a 10×10 cross-validation which gives us the defect prediction result based on training data.

3.1.4 Prediction Performance Assessment

Assessment of learner's performance is being done based on the Receiver Operating Curve (ROC)'s concept. Using ROC enables us to compare it with other significant results [41], [36] and [42].

Based on the ROC curve, which is mostly used in signal detection theory, the performance measures are defined as follows:

- Probability of detection

$$pd = \frac{TP}{TP + FN} \tag{6}$$

- Probability of false alarm

$$pf = \frac{FP}{(FP + TN)} \tag{7}$$

- Balance

$$bal = 1 - \frac{\sqrt{(1 - pd)^2 + (0 - pf)^2}}{\sqrt{2}} \tag{8}$$

pd which is also known as recall or True Positive Rate (TPV) shows the detection ability of predictors. In the ideal case, pd is close to one. pf is the false alarm for incorrectly detecting non-defective modules and ideally must be close to zero. With each classification algorithm, it is important to remember that increasing the number of true positives also increases the number of false alarms. Decreasing the number of false alarms also decreases the number of hits (recall) [34]. Since we need to optimize two parameters, pd and pf, a third performance measure called balance is used to choose the optimal (pd, pf) pairs. balance is equivalent to the normalized Euclidean distance from the desired point $(1, 0)$ to (pd, pf) in a ROC curve [36]. The ROC curve concept is depicted in Figure 21.

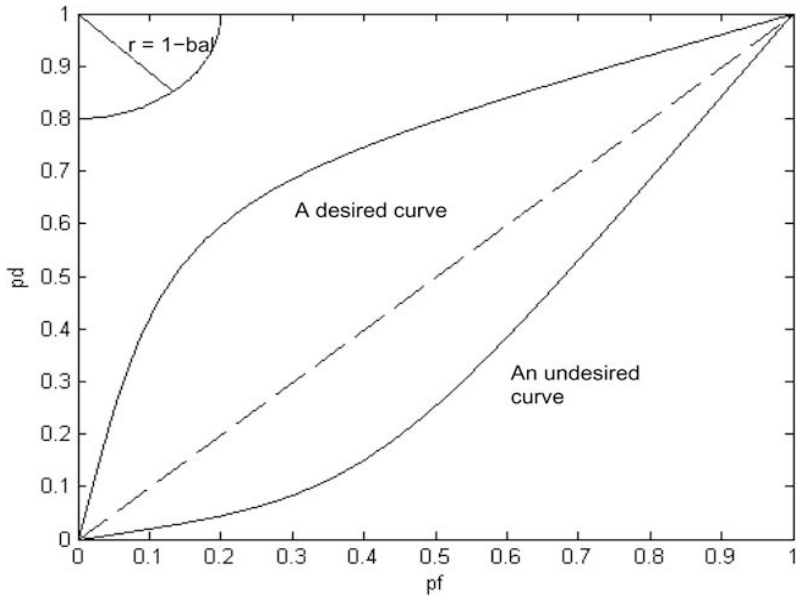

Figure 21: Balance defines a set of points in the ROC curve.

BEGIN

INITIALISE population with random candidate solutions;

EVALUATE each candidate;

REPEAT UNTIL (*TERMINATION CONDITION* is satisfied) DO

1. *SELECT* parents;
2. *RECOMBINE* pairs of parents;
3. *MUTATE* the resulting offspring;
4. *EVALUATE* new candidates;
5. *SELECT* individuals for the next generation;

OD

END

Figure 22: The general scheme of an evolutionary algorithm

3.2 Proposed Methods

In order to explain any EA-based method, we need to define Italicized items shown in Figure 22 [39]. Genetic Programming (GP), the youngest member of EA family has been selected to be applied in our proposed methods.

The significant feature that separates GP from other EA family members lies in its representation. By having the particular representation of non-linear tree structures, GP can be positioned in machine learning field easily while other EA strands are usually being used to address optimization problems. In other words, other EAs are trying to find some input realizing maximum payoff while GP is being used to seek models with maximum fit. Maximization (minimization) is the basis of using evolution for such tasks. Table 10 shows a summary of GP features.

Representation	Tree structure
Recombination	Exchange of subtrees
Mutation	Random change in trees
Parent selection	Fitness proportional
survivor selection	Generational replacement

Table 10: GP summary

3.2.1 Dimensionality reduction by genetic programming

Dimensionality reduction can be broadly divided into two general categories: Feature selection and feature extraction. Feature selection methods seek to find an optimum subset of features which suffice to solve a problem. Subset selection, Ranking, filtering, and wrapping are common approaches that can be named in this field. Since most of the feature selection techniques give rank/weights to the attributes, in other words taking heuristics into account before knowing the structure of data, we have argued that this method cannot be the best candidate as a primary dimensionality reduction method.

Feature Extraction (Construction), on the other hand, can be done by linear or non-linear projection of D-dimensional vector into d-dimensional vector ,where $d < D$.

GP is flexible enough to build mathematical models based on an objective function dynamically. The main advantage of using GP is its expressions are not bound to any predefined template; expressions can be linear or non-linear with the goal of satisfying the objective function [47]. This feature of GP makes it an excellent choice for automatic feature construction.

The whole purpose of feature reduction is transforming the data from a high dimensional space to a lower dimensional space in which the classifier can efficiently perform its tasks. We are taking the non-wrapper [47] approach toward

application of GP for feature construction in our first proposed approach. It means that it acts as a pre-processing step which does not relate to any specific classifier.

We have experimented with such an approach with the purpose of keeping the local structure of data which will be explained later.

As shown in Table 10, the following items need to be described in order to define any EA-based approach fully.

1. Representation: As mentioned earlier, Chromosomes are being shown as Parse Trees in GP. Each Parse Tree represents an expression according to some formal grammar. The main differences in representation of GP and others EA members are:

   - Chromosomes are non-linear structure while in other methods they are typically linear.
   - Chromosomes can differ in size, depending on the number of nodes in each tree while in other methods the chromosome length is usually fixed [39].

   Nodes in Parse tree can be divided into: Root, Branches and Leaves. In another word, Parents and Children. The minimum and maximum number of nodes and levels in each tree have been predefined a priori. If ℓ is the maximum level of each tree, then the number of leaves may vary from 2^{ℓ} in case of having all parent nodes as binary arithmetic functions to 1 while having all parent nodes as unary functions.

   In general, trees can be either balanced or imbalanced. By having all parent nodes as a binary function we may have balanced full binary trees

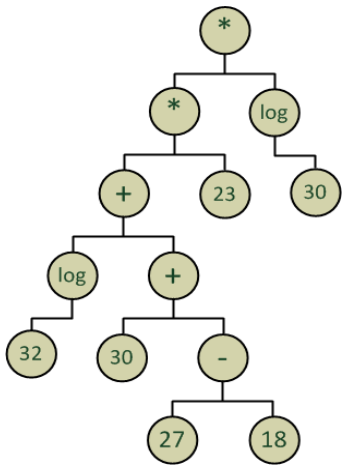

Figure 23: A sample parse tree in our approach

but in this specific implementation, we let them to be imbalanced for more generality. A sample parse tree is depicted in Figure 23.

GP selects a random number of nodes in each tree committed to the defined interval. Arithmetic operators have been used as the functions stated in Table 11.

Arity of operators is 1 or 2 (Unary and Binary operators) as mentioned earlier. Terminal set or leaves of the tree are features in the original(large) space. Each tree in an individual will take high dimensionally space dimensions as root of tree. Consequently, each tree will make a new dimension in the new lower dimension space, therefore number of trees in final solution is the same as number of dimensions in the new space.

To review the whole step, we can say: Genes are being constructed with chromosomes in the form of Parse Trees with maximum depth and maximum number of features defined. Each tree will make a new dimension in the new space. It is important to mention that we have tried to avoid having dimensionality of 2 as the resulted space. It is only being used for

Function set	$+$ $-$ $*$ $/$ $\log$ $\exp$
Terminal set	Any feature from NASA IV & V datasets + the constant values 0 and 1

Table 11: Nodes values

presentation purposes and cannot save the data characteristics [58].

2. Initialization:

   Each tree structure is created with no heuristics taken into account. The only defined parameters are depth of the tree and the maximum number of nodes. Node values are being assigned based on the node type.

   If the node type is leaf, a random feature would be placed. If node is a branch node with arity 2, a binary function from Table 11 would be selected. If arity is 1, a unary function will be selected from Table 11.

3. Evaluation:

   Calculation of some sort of fitting error would be the basis of "GP's fitness calculation". The relation among operators and operands (tree leaves) must be mapped. In general, fitness and penalty has an inverse relation given by:

$$Fitness = 1/Penalty \tag{9}$$

   Penalty or Cost function will be calculated by picking a pair of points(observations) randomly in the high dimensional space (i.e. row $\boldsymbol{a}$ and $\boldsymbol{b}$), measure the Euclidean distance between them and name it d_1.

$$\boldsymbol{a} = (a_1, a_2, ...a_N) \tag{10}$$

$$\boldsymbol{b} = (b_1, b_2, ...b_N) \tag{11}$$

$$d_1 = \sqrt{(|a_1^2 - b_1^2|) + (|a_2^2 - b_2^2|) + ...(|a_N^2 - b_N^2|)} \tag{12}$$

Then, another pair of random points in the high dimensional space(points $\boldsymbol{c}$ and $\boldsymbol{d}$) will be picked and distance would be measured like Equations 15 and 16. Therefore the d_2 would be calculated as:

$$d_2 = \sqrt{(|c_1^2 - d_1^2|) + (|c_2^2 - d_2^2|) + ...(|c_N^2 - d_N^2|)} \tag{13}$$

Ratio between two distances in the high dimensional space would be:

$$d_r = d_1/d_2 \tag{14}$$

We want to have the same ratio between the first distance and the second distance in the small space. For this, we find the corresponding four points in the small space namely $\boldsymbol{a}'$, $\boldsymbol{b}'$, $\boldsymbol{c}'$ and $\boldsymbol{d}'$. The above formulae in the new low-dimensional space become:

$$\boldsymbol{a}' = (a'_1, a'_2, ...a'_M), \qquad M < N \tag{15}$$

$$\boldsymbol{b}' = (b'_1, b'_2, ...b'_M) \tag{16}$$

$$d'_1 = \sqrt{(|{a'_1}^2 - {b'_1}^2|) + (|{a'_2}^2 - {b'_2}^2|) + ... + (|{a'_M}^2 - {b'_M}^2|)} \tag{17}$$

$$d'_r = d'_1/d'_2 \tag{18}$$

and we calculate the ratio of the two distances similar to what has been done in the original space: Since we want to have the same ratio in both spaces, the difference between these two ratios can be used as a measure for error:

$$e = d_r - d'_r \tag{19}$$

However, the error for this specific four points might not represent the change in the structure of data, as the four points are selected randomly. For minimizing the effect of random selection of these four points, we perform this calculation many times and calculate the sum of all repetitions. For practical reasons (avoiding long computation time), the number of repetitions is set to 50. In order to prevent the partial errors from cancelling the effect of each other, we use the sum square error:

$$Error = \sqrt{|d_{r1}^2 - {d'_{r1}}^2) + |d_{r2}^2 - {d'_{r2}}^2 + ... + |d_{r50}^2 - {d'_{r50}}^2|} \quad (20)$$

In addition, if we want to favor spaces with less number of dimensions we can also add a coefficient for the number of dimensions in our penalty (cost function) which needs to be minimized:

$$Penalty_{total} = Error + W \times M, \quad (21)$$

where W is a coefficient indicating how important the number of dimensions in the new space is, and M is the number of dimensions in the new space.

We can view this entire process as defining a sample large space with 3 dimensions. 4 points have been selected in Figure 24. Figure 25 shows a good transformation to a 2-dimensional space, where the approximation $\frac{ab}{cd} \simeq \frac{a'b'}{c'd'}$ is valid. Figure 26 demonstrates a bad transformation, in which $\frac{ab}{cd} \ll \frac{a'b'}{c'd'}$.

4. Parent Selection Mechanism: "Parent Selection/Mating pool creation" refers to finding better individuals in order to choose parents of the next generation. Parent Selection as well as Survivor selection - which will

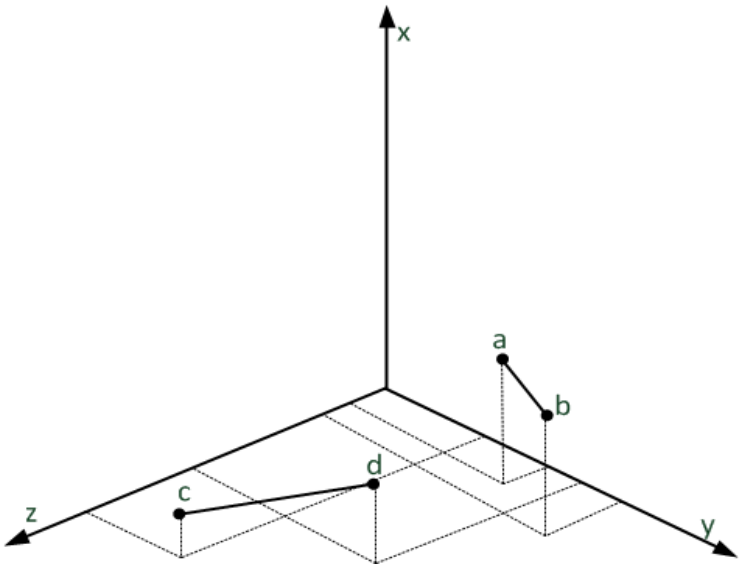

Figure 24: 4 points in original 3-dimensional space

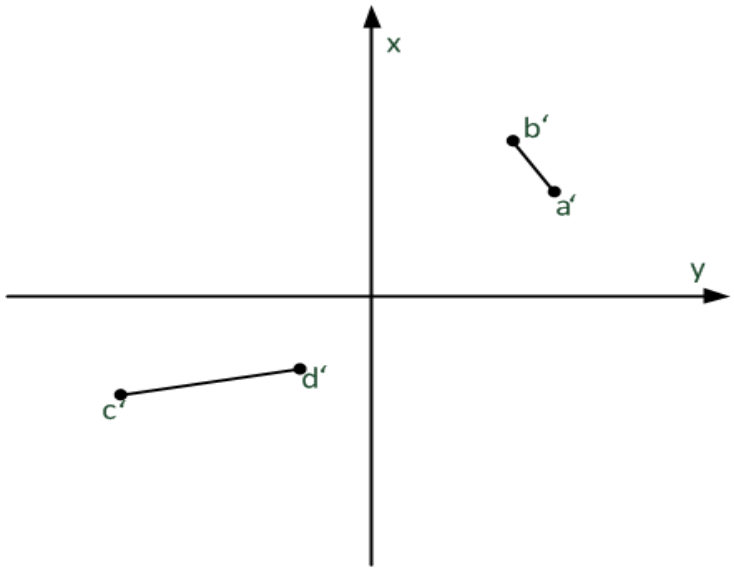

Figure 25: 4 points in 2-dimensional space - good transformation

be explained shortly - are in charge of selecting individuals with higher quality. Here, quality means having less penalty (higher fitness value).

It is important to mention that there is always a chance in our approach for new/low-quality individuals in order to avoid being stuck in a local optimum and keep the high diversity. The parent selection method of

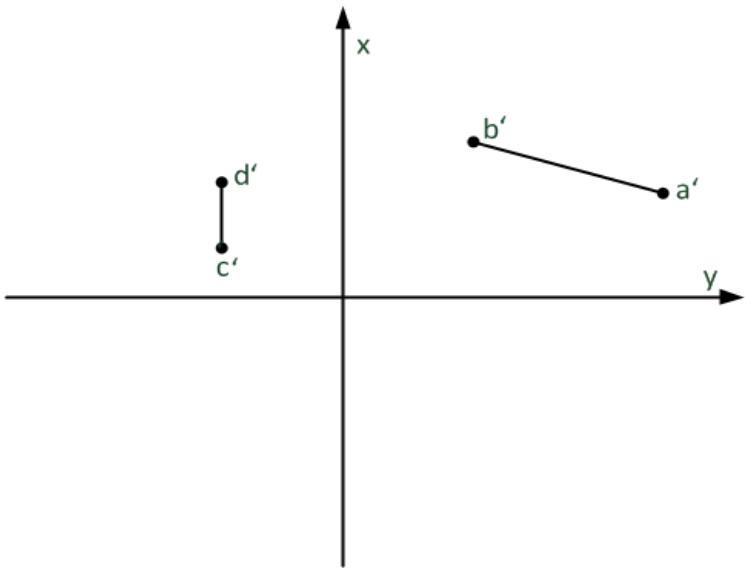

Figure 26: 4 points in 2-dimensional space - bad transformation

our proposed approach is based on ”Tournament selection” method. The Tournament size is equal to 3, and Parent pool size is usually defined as half of the population size. More details will be discussed in the experimental results section. Because Tournament selection is used, Parent selection is non-deterministic in the course of GP.

5. Genetic Operators (Variation Operators): ”Offspring creation” will be carried out through the application of two well-known genetic operators: mutation and crossover.

   (a) Mutation: is the unary variation operator. In all EA family, mutation can be defined as creating new individuals by changing the parent through slight random variation. Based on representation type of EA, this genetic operator will be specifically defined.

   (b) Crossover (Recombination): Two individual parents are being merged

to form a new individual. Crossover is done in two forms in Genetic Programming approaches: Shuffle and Fixed point. These approaches replace a whole tree of an individual with another. In the fixed point crossover, a point can be selected randomly which indicates the node that the tree must be broken from. In the shuffle crossover, each tree in a child is copied completely from either first or second parent. We will gain two children by doing crossover.

In this specific implementation, off spring functions being used are:

- Mutate - Swap nodes within a tree (does not care if you are swapping a non-leaf node with a leaf)
- Mutate - swap internal nodes
- Mutate - swap leaves
- Mutate - cut sub-tree
- Mutate - replace leaf with sub-tree
- Mutate - swap sub-trees within a tree
- Crossover - swap sub-trees between two trees:
  - fixed-point
  - shuffle

6. ”Survivor selection”: is deterministic and it is based on the lowest penalty (highest fitness). Keeping the diversity high is a task for a tournament-based parent selection. To appreciate evolution performance, two best results will be picked amongst all the others and transferred to the next generation with no change. In order to keep diversity high, three random genomes will be added to the population and the remainder of individuals will be selected deterministically based on their fitness.

The whole steps above repeat until the optimum solution is found (Penalty = 0) or we reach a maximum number of iterations. More details are provided in the experimental results subsection.

3.2.2 Prediction using classifiers and machine learning methods

In the literature, machine learning approaches have been used in order to learn the data and perform the prediction. As we have stated earlier, data preprocessing as well as dimensionality reduction techniques have a huge impact on the quality of data that will be handed to classifier. The best result gained so far belongs to Menzies [41] that other significant contributions like Turhan [36] could not outperform it yet. The discriminant analysis techniques are being widely used are: Linear Discriminant Analysis, Quadratic Discriminant Analysis and Naive Bayes.

In software defect prediction, one aims to separate classes C_0 and C_1 where samples in C_0 are non-defective and samples in C_1 are defective. In this 2-class problem, it is sufficient to find one discriminant that separates instances from the two distinct classes. Combining the multivariate normal distribution and the Bayes rule, followed by using different assumptions results in different discriminants with different complexity levels. General structure of a discriminant.

$$P'(C_i|x) = \frac{P'(x|C_i)P'(C_i)}{P'(x)} \tag{22}$$

Based on normalized evidence, we can replace Eq. (22) with:

$$g_i(x) = \log P(x|C_i) + \log P(C_i) \tag{23}$$

Linear discriminant considers the correlation of the features but assumes the variances and correlation of features are the same for both classes. This means classes can have any orientation with respect to axes but aligned to each other. The number of parameters to estimate for covariance matrix is now independent of K.

Quadratic Discriminant considers the correlation of features too. This model does this differently for each class. It's assumptions are data samples are i.i.d(independent and identically distributed)

Naive Bayes does not take into account correlation of the features. It measures the deviation from the mean in terms of standard deviations. In other words, Naive Bayes takes two assumptions into the account, Independence of attributes as well as Equal importance of attributes.

We have applied all mentioned classifiers on the resulted data from GP.

3.2.3 GP constructs features and performs classification

As mentioned previously, GP plays the role of an effective feature extractor. The number of used features, number of nodes in trees, arithmetic operators and most of the parameters that GP uses is non-heuristic and are not predefined.

In the second experiment, we use GP to not only reduce the number of features, but also to perform classification. GP will search for certain feature constructor to create a smaller number of features, on which a simple linear classifier is applied. Preserving the structure of data is not important anymore here, and the new fitness represents the performance of this classifier. Although the classifier that GP uses is linear, we need however to keep in mind that these features are

constructed non-linearly, hence the whole process will be non-linear.

A fixed number of new features are constructed form the original data. This means that a defined number of parse trees are made from the arithmetic operators listed in Table 11 as internal nodes and a collection of features as leaves. Each parse tree will input one sample in the original space, and will construct one single feature (dimension) of the corresponding point in the new small space. The collection of all parse trees will form a transform of the large space to the small space. The number of parse trees will define the number of features in the small space. This step is similar to what we used in the first experiment.

To evaluate a set of parse trees (i.e. a transform), GP simply adds all the values of all features in the new small space, assigning "defective" labels to those with a positive sum and "non-defective" to those with a negative sum. This will be up to GP to specialize the parse trees to perform well with this simple linear classifier.

The label assigned to each sample is compared to its actual defective/non-defective value for all the samples in the training set. The values for TP, TN, FP and FN are calculated 9, and pd and pf are calculated from Eq. (6) and Eq. (7). Balance is then calculated from Eq. (8).

Balance will be used as the fitness value in both parent selection and survival selection of the GP. Other than the fitness function, all the steps and parameters of the GP stays untouched from the previous section. By comparing the performance of the overall classifiers found by GP, then we can answer a very fundamental question: Is keeping the original structure of data beneficial for feature construction. The results presented in the next section will show that the answer to this question is not always positive.

3.3 Genetic Programming Selects Metrics!

McCabe developed some metrics claiming that it would provide insight into the reliability and maintainability of modules [55]. Cyclomatic Complexity ($v(G)$) measures the number of linearly independent paths through a program flow graph. $v(G)$ is calculated as:

$$v(G) = e - n + 2 \tag{24}$$

where G is a program's flowgraph, e is the number of arcs in the flowgraph, and n is the number of nodes in the flowgraph. Many software companies and even some NASA subcontractors evaluate source code to check the value of McCabe's cyclomatic complexity. In NASA IV&V, a value of over 10 is flagged as a module that will be difficult to maintain and/or debug. Those metrics are being used the most in industry to detect defect prone modules.

Menzies et al. [40]'s study on the same dataset as us named "Metrics that Matter" argued that some widely-known metrics like McCabe Complexity Metrics are not as good as we think. In fact, he claimed that cheap and easy to collect LOC metric is the one that performs exceptionally well. It has been suggested by other researchers that in fact LOC may be a better candidate for evaluating for error-prone code. Table 12 listed the most selected metrics by our Genetic Programming based approach (second approach). It endorses Menzies' idea.

Figures 27, 28, and 29 demonstrate the case when we have ran the GP -2nd approach- 2,000 times on dataset CM1, with Population size of 1000, Parent Pool Size 500, Tournament size 3. The number of trees in each individual varies from 4 to 6.

Static Metric Group	Name	Number of a
Line of Code Metrics	Call Pairs	
	Number of lines of code containing both code and comment	
	Number of lines of comment	
	Edge Count	
	Multiple Condition Count	
	Node Count	
	Number of unique operands	
	Parameter Count	
	Percent of Code that is Comment	
McCabe Software Metrics	Cyclomatic Complexity	
	Design Complexity	
	Design Density	
	Essential Complexity	
Halstead Metrics	Halstead Difficulty	

Table 12: Selected metrics by GP as shown by Figure 27, 28, 29

3.4 Experimental results

We tested the performance of our approach on CM1, a dataset from Nasa IV&V facility. It has 505 rows and 37 features.

The maximum number of trees in each individual varies from 4 to 6. It is based on Turhan's recommended size. Population size is 1000, Parent pool size is 500 and number of generation runs are 2000. We started with a very high number of generations (approximately 10000) but in practice there is not much improvement after the 2000-th generation. Tournament size was set to 3. Probability of choosing any of the genetic operators is equal. Also probability of choosing any of the mutation (or crossover) techniques is equally probable (i.e. probability of $1/2$). Moreover, any mutation technique has the probability of $1/12$ to be chosen and crossover techniques have a probability of $1/4$.

As shown in Table 13, the second wrapper application of GP in terms of linear classification outperforms other methods of data processing by having a high

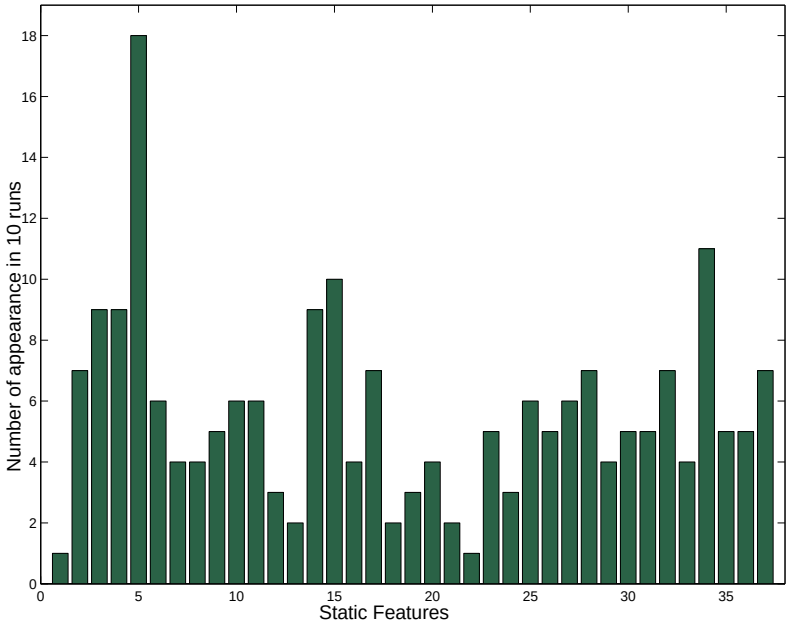

Figure 27: Number of appearance in 10 runs and 4 trees in an individual

probability of detection rate and lowering the probability of false alarms. It has been studied for the case of having 5 trees in each individual (in other words, having the dimensionality of 5 as a resulting space).

In Figures 30, 31, and 32, the minimum penalty of 10 runs of GP in the second approach while having 4, 5, and 6 trees in each individual. It clearly shows that penalty is decreasing during generations. 2000 runs is the number of generations we usually use while the picture indicates that penalty reaches its minimum point around generation 200.

3.4.1 Bloating problem

A common issue in GP runs is that chromosome sizes tend to increase. It is also known as “Survival of the fattest”. It usually happens in crossover

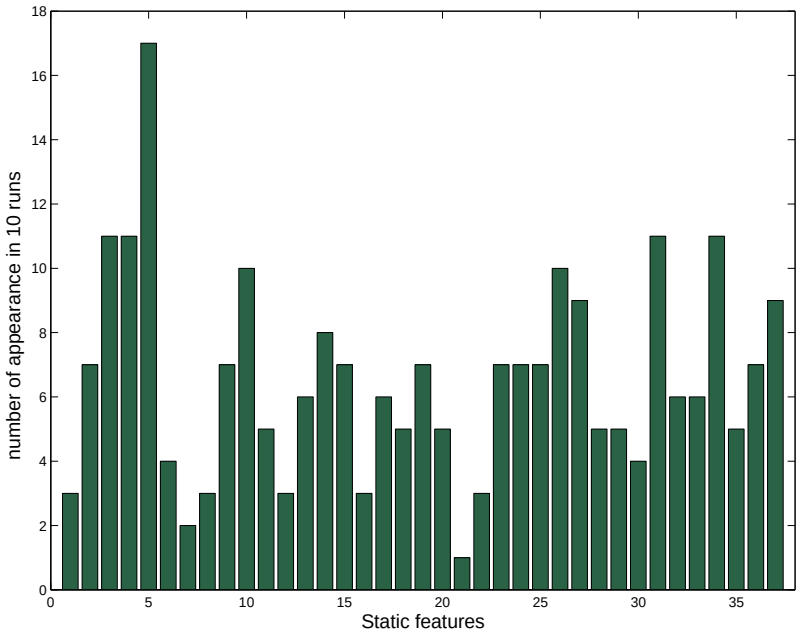

Figure 28: Number of appearance in 10 runs and 5 trees in an individual

when two large parent individuals are selected to create children. That's the place where individuals tend to have larger sizes. Population will soon become unmanageable because of memory problems and increasing time to evaluate the solutions. Based on Figures 33, 34, 35, 36, 37 and 38, this problem does not occur in our GP program. The maximum number of nodes in each tree was set to 15. It is worth pointing out that, as shown in the above Figures, GP never reaches this maximum number.

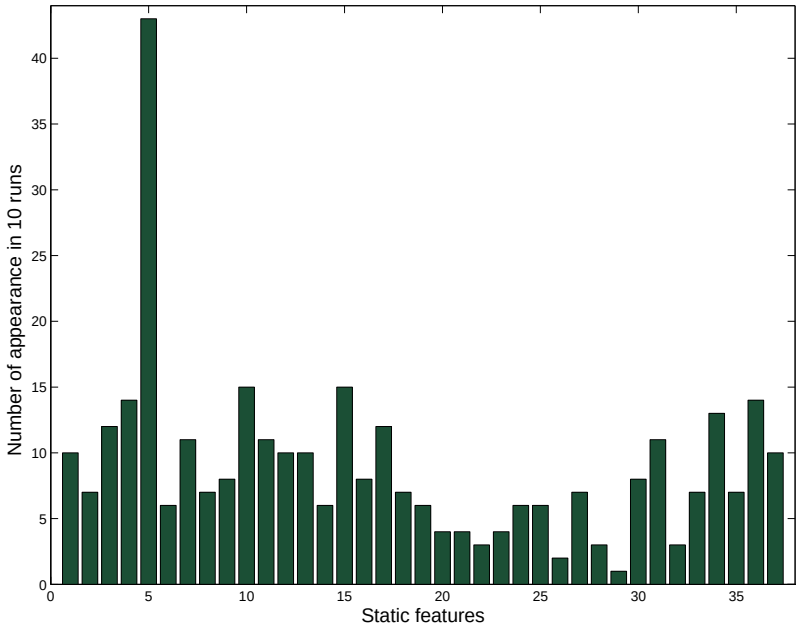

Figure 29: Number of appearance in 10 runs and 6 trees in an individual

Menzies [41]-subset selection				
		pd	**pf**	**bal**
CM1	QD	91(15)	50(8)	62(7)
	LD	89(16)	54(7)	59(7)
	NB	89(19)	31(7)	71(9)
Turhan [36]-PCA				
CM1	QD	76(22)	35(9)	70(11)
	LD	82(19)	38(8)	71(12)
	NB	82(17)	37(9)	71(10)
GP as dimensionality reduction				
CM1	QD	71(2.2)	46(3)	56(2)
	LD	82(9)	53(4)	59(2.7)
GP as classifier				
CM1	QD LD NB	83(4)	20(3.7)	81(2.5)

Table 13: Results on CM1

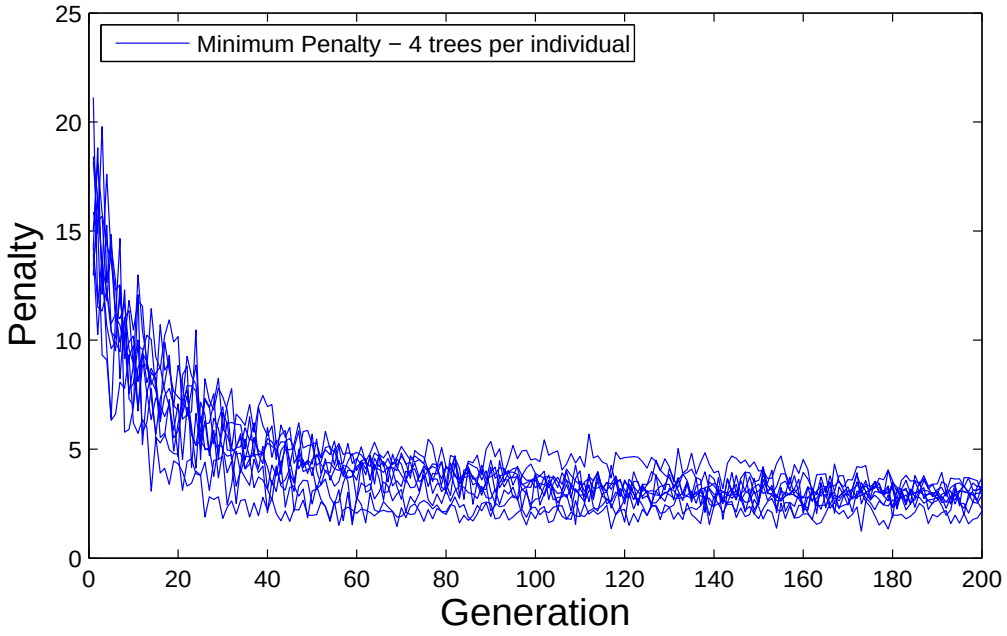

Figure 30: Minimum Penalty: 4 trees in an individual

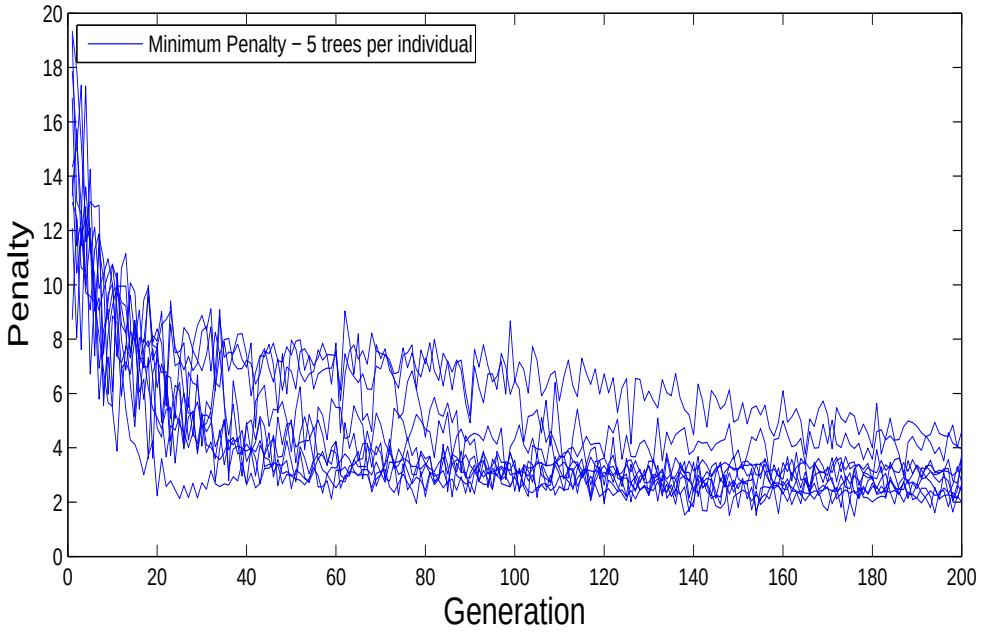

Figure 31: Minimum Penalty: 5 trees in an individual

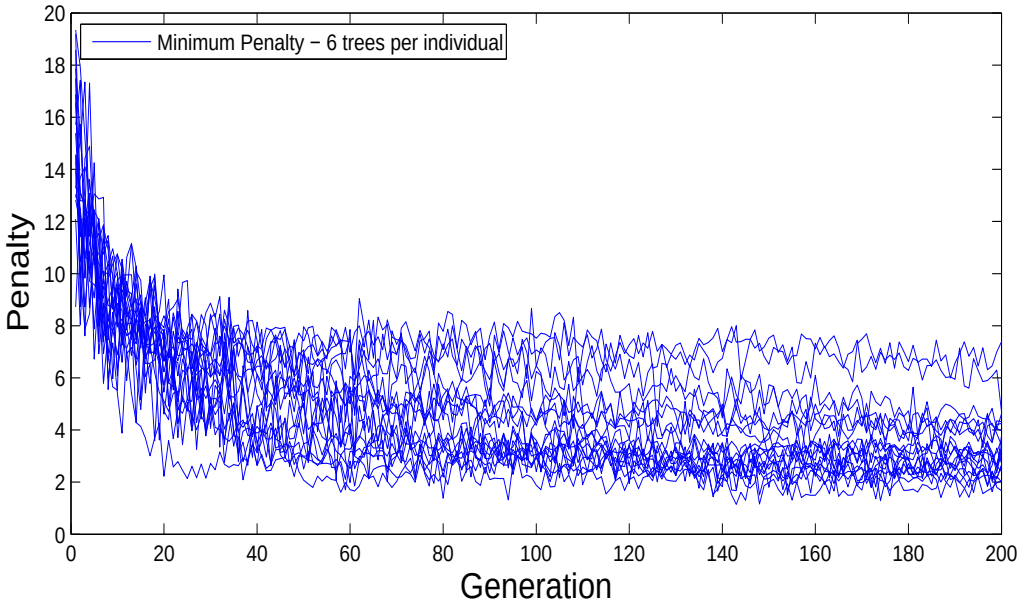

Figure 32: Minimum Penalty: 6 trees in an individual

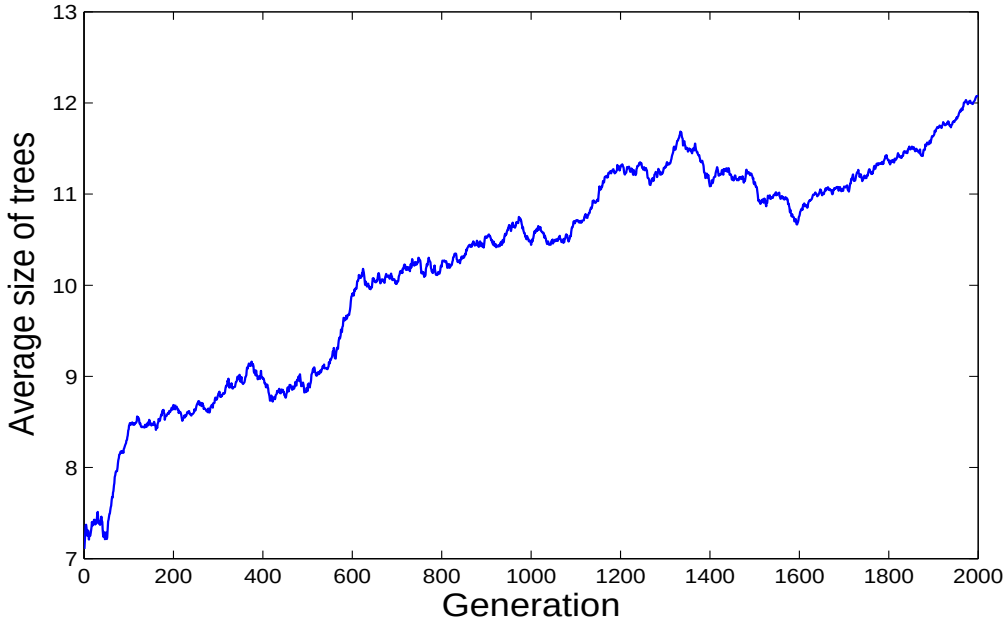

Figure 33: Average Size of a tree: 2000 Generations and 4 trees in an Individual

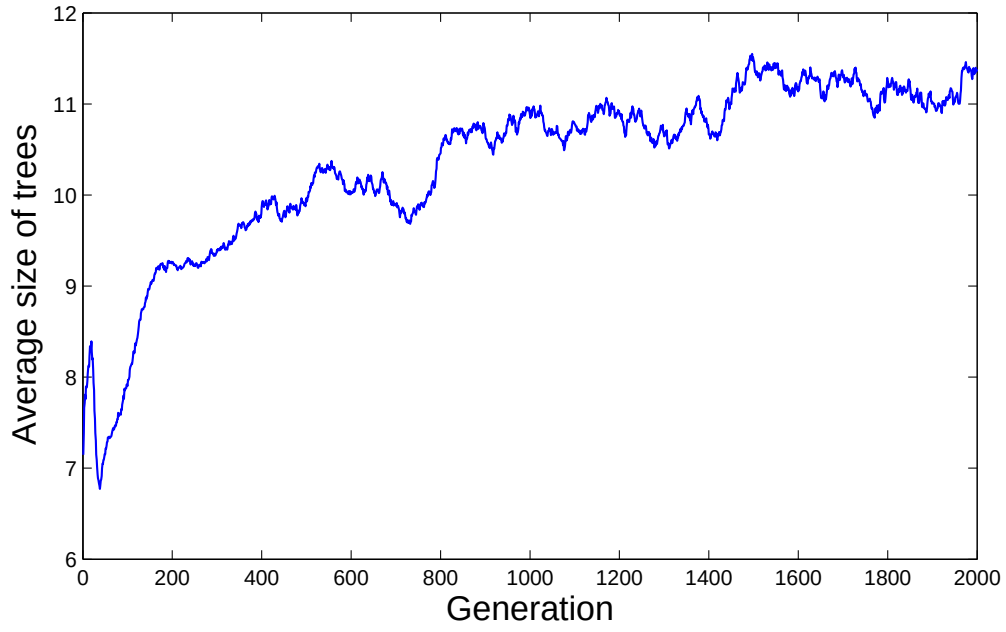

Figure 34: Average Size of a tree: 2000 Generations and 5 trees in an Individual

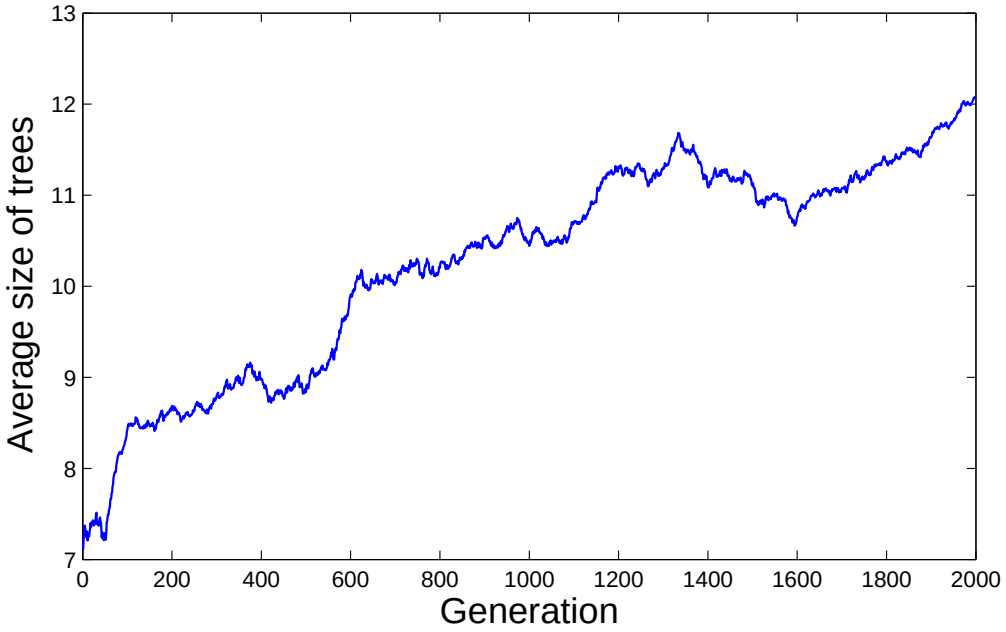

Figure 35: Average Size of a tree: 2000 Generations and 6 trees in an Individual

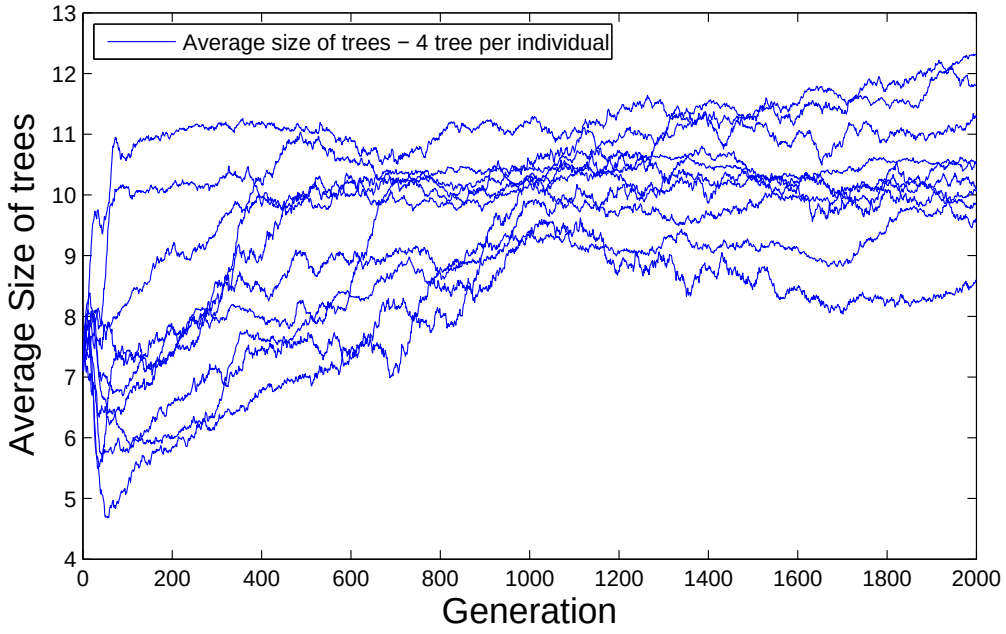

Figure 36: Average Size of trees: 2000 Generations, 10 runs and 4 trees in an Individual

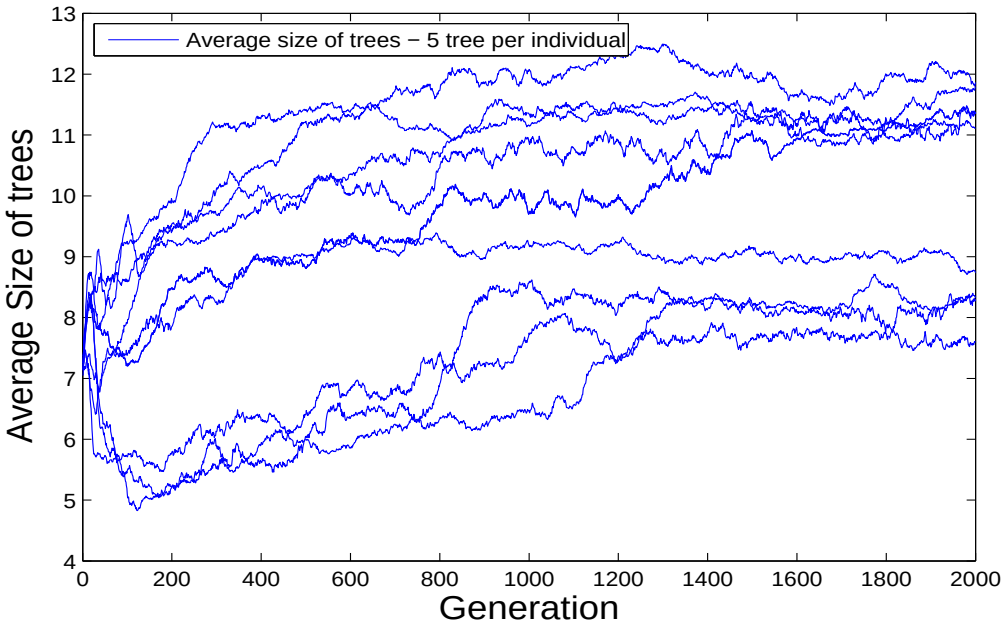

Figure 37: Average Size of a tree: 2000 Generations, 10 runs and 5 trees in an Individual

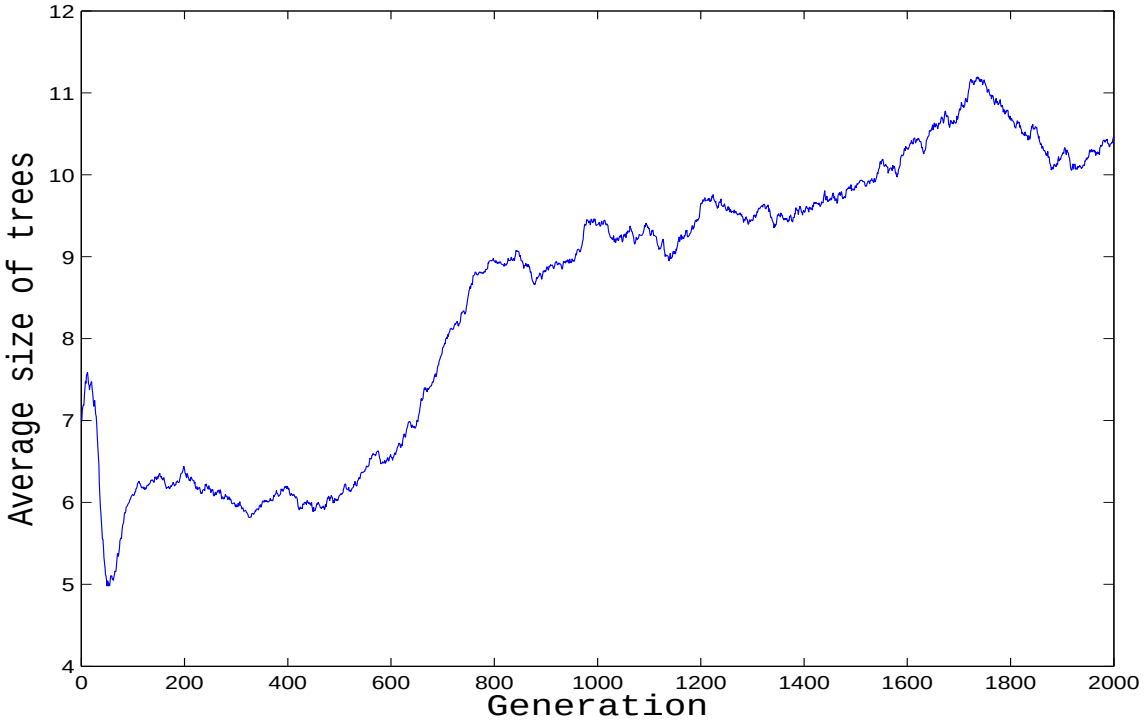

Figure 38: Average Size of a tree: 2000 Generations, 10 runs and 6 trees in an Individual

CHAPTER 4

Conclusions

A defect prediction solution provides a valuable guideline to tackle the problem of defects that might be caused due to programmers inability, failure in requirements collection or design mistakes. Thus, a defect prediction model can provide important ideas regarding the erroneous bottlenecks in the software development cycle. In particular, efficiency focused software development units can benefit from using defect cause information. They can take necessary precautions in a proactive manner. In other words, a defect focused prediction solution can also successfully lead to a major change in the development methods. Such a solution or systematic approach can affect in a positive manner to produce less defected software.

An important aspect of a defect prediction solution is that such a solution becomes necessary when there is a trade-off between to deliver earlier and to deliver with fewer defects. In today's software development industry, all companies and software development houses are in a severe competition that minimizing development time decreases the overall project cost [60, 61]. On the other hand, less development and testing time also increases the defect density

ratio in the final product. As a result, the executive management of software companies should require a quantitative indicator to find the correct point in this balance. Therefore, a defect prediction solution may provide the required quantitative metric to make a decision on the product delivery. The senior management of software development companies would be able to decide launching the product if the defect density level is below a certain threshold.

This thesis has presented statistical tools to predict the cumulative number of software defects as well as discriminant analysis through application of genetic programming that can have a significant impact on making the software quality assurance easier. We have demonstrated the performance of the proposed algorithms on a variety of software defect datasets, and we compared our proposed techniques with existing methods.

In the next Section, the contributions made in each of the previous chapters and the concluding results drawn from the associated research work are presented. Suggestions for future research directions related to this thesis are provided in Section4.1.2.

4.1 Contributions of the Thesis

4.1.1 Predictive operating characteristic curves for software defect prediction

We introduced a software defect prediction model based on the concept of operating characteristic curve. The idea is to use Operating Characteristic (OC) curves in statistical quality control and a geometric approach to construct an

efficient, fast, and accurate prediction method to estimate the number of software failures at anytime during the software development process. Our model is getting the information from past and present failure data to be more effective. In the experimental results, we demonstrate the effectiveness and the improved performance of the proposed method in comparison with the Bayesian prediction approaches.

4.1.2 Learning defect predictors via genetic programming

We proposed defect prediction learning models based on the application of genetic programming, which is the newest member of evolutionary algorithm family. Genetic programming based approaches have recently gained significant popularity in the field of machine learning due to ther specific representation in the form of parse trees. The proposed prediction models addressed the need of having a robust learner by taking no heuristics into account.

4.2 Future Research Directions

Several interesting research directions motivated by this thesis are discussed next. In addition to designing robust statistical models for software defect prediction, we intend to accomplish the following projects in the near future:

4.2.1 Metric-based applications using genetic programming

As discussed in Chapter 3, the first predictor method constructs new features based primarily on the geometrical characteristics of the original data. Then,

an independent classifier is applied and the performance of feature selection method is measured. The second predictor, on the other hand, uses a built-in classifier which automatically gets tuned for the constructed features.. Our future efforts will be focused on calculating different distance metrics to reduce the probability of false alarm that results to increasing balance measure.

4.2.2 Machine learning approaches

There are two major settings in which we wish to learn a function f: *supervised* and *unsupervised*. In supervised learning, we know the values of f for the m samples in the training set $\mathbb{S}$. We assume that if we can find a hypothesis h that closely agrees with f for the members of $\mathbb{S}$, then this hypothesis will be a good guess for f, especially if $\mathbb{S}$ is large. Curve fitting is a simple example of supervised learning of a function. In unsupervised learning, we simply have a training set of vectors without function values of them. The problem in this case, typically, is to partition the training set into subsets $\mathbb{S}_1, \ldots, \mathbb{S}_k$ in some appropriate way.

Our future efforts will be focused on evaluating various machine learning models to develop robust prediction approaches. The performance of each prediction method will be evaluated regarding their precision, recall, robustness and sensitivity using confusion matrices and simulations. A model's precision is defined as the ratio of the number of modules correctly predicted as defective, or true positive (t_p), to the total number of modules predicted as defective in the set ($t_p + f_p$). A model's recall is defined as the ratio of the number of modules predicted correctly as defective (t_p) to the total number of defective modules in the set ($t_p + f_n$). To perform well, a model must achieve both high

precision and high recall.

Bibliography

[1] L. Westfall, "The certified software quality engineer handbook," *American Society for Quality(ASQ), Quality Press*, Milwaukee, 2009.

[2] S. H. Kan, "Metrics and models in software quality engineering," *Addison-Wesley*, Boston , 2003.

[3] K. E. Wiegers, "Software Requirements," *Microsoft press*, 2003.

[4] S. Rakitin, "Software verification and validation for practitioners and manager," *Artech House*,2nd Edition, Boston , 2001.

[5] J.D. Musa, "A theory of software reliability and its application," *IEEE Transactions on Software Engineering*, vol. 1, no. 1, pp. 312-327, 1975.

[6] A.L. Goel and K. Okumoto, "Time-dependent error detection rate models for software reliability and other performance measures," *IEEE Transactions on Reliability*, vol. 28, no. 3, pp. 206-211, 1979.

[7] D.C. Montgomery, *Introduction to Statistical Quality Control*, John Wiley & Sons, 2005.

[8] C. Robert, *Bayesian Choice*, 2nd Edition, Springer Verlag, NY, 2001.

[9] W.M. Bolstad, *Introduction to Bayesian Statistics*, John Wiley, 2004.

[10] J.D. Musa, A. Iannino, and K. Okumoto, " Software Reliability: Measurement, Prediction, Application, "*McGraw-Hill Book Company*, 1987.

[11] J.W. Yu, G.L. Tian, and M.L. Tang, "Predictive analyses for nonhomogeneous Poisson processes with power law using Bayesian approach," *Computational Statistics & Data Analysis*, 2007.

[12] C.G. Bai, "Bayesian network based software reliability prediction with an operational profile," *Journal of Systems and Software*, vol. 77, no. 2, pp. 103-112, 2004.

[13] X. Zhang and H. Pham, "Software field failure rate prediction before software deployment," *Journal of Systems and Software*, vol. 79, pp. 291-300, 2006.

[14] G. J. McLachlan, K. Do, and C. Ambroise, "Analyzing Microarray Gene Expression Data," *Wiley*, 2005.

[15] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *IEEE Transactions on Reliability*, vol. 32, no. 5, pp. 475-485, 1983.

[16] A.L. Goel, "Software reliability models: assumptions, limitations and applicability," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1411-1423, 1985.

[17] M.R. Bastos Martini, K. Kanoun, and J. Moreira de Souza, "Software-reliability evaluation of the TROPICO-R switching system," *IEEE Transactions on Reliability*, vol. 39, no. 3, pp. 369-379, 1990.

[18] K. Kanoun and J.C. Laprie, "Software reliability trend analysis from theoretical to practical considerations," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 525-532, 1992.

[19] M.R. Lyu, *Handbook of Software Reliability Engineering*, IEEE Computer Society Press and McGraw-Hill, 1996.

[20] S. Yamada, T. Ichimori, and M. Nishiwaki, "Optimal allocation policies for testing-resource based on a software reliability growth model," *Mathematical and Computer Modelling*, vol. 22, pp. 295-301, 1995.

[21] J.H. Lo and C.Y. Huang, "An integration of fault detection and correction processes in software reliability analysis," *Journal of Systems and Software*, vol. 79, no. 9, pp. 1312-1323, 2006.

[22] B.W. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, *Software Cost Estimation with Cocomo II*, Prentice Hall PTR, 2000.

[23] X. Cai, M.R. Lyu, K-F. Wong, and R. Ko, "Component-based software engineering: Technologies, Development frameworks, and quality assurance schemes," *Proc. Asia-Pacific Software Engineering Conferenece*, pp.372-379, 2000.

[24] W. Kozacznski and G. Booch, "Component-based software engineering," *IEEE Software*, vol. 155, pp. 34-36, Sep./Oct. 1998.

[25] M.R. Lyu, S. Rangarajan, and A.P.A. van Moorsel, "Optimal allocation of test resources for software reliabilitygrowth modeling in software development," *IEEE Transactions on Reliability*, vol. 51, no. 2, pp. 183-192, 2002.

[26] C.Y. Huang and M.R. Lyu, "Optimal release time for software systems considering cost, testing-effort, and test efficiency," *IEEE Trans. on Reliability*, vol. 54, pp. 583-591, 2005.

[27] O. Gauodin, "Optimal properties of the Laplace trend test for software-reliability models," *IEEE Transactions on Reliability*, vol. 20, no. 9, pp. 740-747, 1992.

[28] H.E. Ascher and C.K.Hansen, "Spurious exponentiality observed when incorrectly fitting a distribution to nonstationary data," *IEEE Transactions on Reliability*, vol. 47, no. 4, pp. 451-45, 1998.

[29] W.R. Gilks, S. Richardson, and D. Spiegelhalter, "Markov chain Monte Carlo in Practice," *Chapman & Hall/CRC*, 1995.

[30] N. Hazrati, A. Ben hamza, , Y. Karabulut, and N. Mahe, "Software defect prediction via operating characteristic curves," *In Proceedings of ISSRE '09. 20th International Symposium on Software Reliability Engineering*, Mysore, India, 2009.

[31] Y. Luo, T. Bergander, and A. Ben Hamza, "Anisotropic Laplace trend to enhance software reliability growth modelling," *Proc. International Conference on Modelling and Simulation*, Montréal, Canada, May 2006.

[32] Y. Luo, T. Bergander, and A. Ben Hamza, "Software reliability growth modelling using a weighted Laplace test statistic," *Proc. IEEE International Computer Software and Applications Conference*, Beijing, China, July 2007.

[33] ANSI/IEEE, Standard Glossary of Software Engineering Terminology, STD-729-1991, *ANSI/IEEE*, 1991.

[34] E. Alpaydin, "Introduction to machine learning," *MIT Press*, Cambridge, Massachusett, 2004.

[35] B. Turhan, and A. Bener, "A Multivariate Analysis of Static Code Attributes for Defect Prediction," *Proc. Seventh International Conference on Quality Software(QSIC 2007)*, 2007.

[36] B. Turhan, and A. Bener, "Analysis of Naive Bayes' assumption on software fault data: An empirical study," *Data and Knowledge Engineering*, vol. 68, no. 2, pp. 278-290, 2009.

[37] D. Gray , D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Using the support vector machine as a classification method for software defect prediction with static code metrics, " *Engineering applications of Neural Networks*, vol. 43, pp. 223-234, 2009.

[38] W. S. Humphrey, "Managing the software process," *Addison-Wesley*, 1989.

[39] A.E. Eiben, J.E. Smith, "Introduction to Evolutionary Computing," *Springer-Verlag*, 2003.

[40] T. Menzies, J. Di Stefano, M. Chapman, and K. McGill, "Metrics That Matter," *In Proceedings of 27th Annual IEEE/NASA Software Engineering Workshop*, 2002.

[41] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors,"*IEEE Transactions on Software Engineering*, vol. 33, no. 1, 2007.

[42] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, "Defect prediction from static code features: current results, limitations, new

approaches,“*Automated Software Engineering*, vol. 17, no. 4, 2010.

[43] J. Sayyad Shirabad, and T. Menzies, “The PROMISE Repository of Software Engineering Database,” *http://promise.site.uottawa.ca/SERepository*, School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[44] “NASA IV&V Facility Metrics Data Program repository[Online],” Available: *http://mdp.ivv.nasa.gov/*.

[45] K. Neshatian, M. Zhang, and M.Johnston, “Feature Construction and Dimension reduction Using Genetic Programming,” *AI 2007: ADVANCES IN ARTIFICIAL INTELLIGENCE*,vo.4830, pp.160-170, 2007.

[46] K. Neshatian, and M. Zhang, “Genetic Programming for Performance Improvement and Dimensionality Reduction of Classification Problems, ” Evolutionary Computation, 2008. CEC 2008, pp. 2811-2818, 2008.

[47] K. Neshatian, and M.Zhang, “Genetic Programming and class-wise orthogonal transformation for dimension reduction in classification problems, ” EuroGP 2008, *Springer-Verlag Berlin*, pp.242-253, 2008.

[48] K. Neshatian, M. Zhang, and P. Andreae, “Genetic Programming for feature ranking in classification problems,” *SEAL 2008*, 2008.

[49] K. Neshatian, and M. Zhang, “Dimensionality Reduction in Face Detection: A genetic programming approach,” *4th International Conference Image and Vision Computing New Zealand (IVCZN 2009)*, 2009.

[50] K. Neshatian, and M. Zhang, “Pareto from feature selection: using genetic programming to explore feature space,” *GECCO’0*, 2009.

[51] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data Clustering: A review, "*ACM Computing Surveys*, vol. 31, no. 3, September 1999.

[52] N.E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675-689, 1999.

[53] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," *In Proceedings of the 28th international conference on Software engineering*, pp. 452-461, 2006.

[54] R. S. Pressman, "Software engineering: a practitioner's approach," *McGraw-Hill*, 1982.

[55] D. Galin, "Software Quality Assurance, from theory to implementation," *Pearson Education Limited*, 2004.

[56] G. G. Schulmeyer, "Handbook of software quality assurance," *Artech house*, Boston, 2008.

[57] Q. Song, M. Shepperd, M. Cartwright, and C. Mayer ,"Software defect association mining and defect correction effort prediction," *Software Engineering, IEEE Transactions on*, vol. 32, no. 2, pp. 69-82, 2006.

[58] L. Van Der Maaten, and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579-2605, 2008.

[59] Y. Jiang, B. Cukic, and T. Menzies, "Fault prediction using early lifecycle data," *In Proceedings of ISSRE '07. The 18th IEEE International Symposium on Software Reliability*, Trollhattan, 2007.

[60] P.C. Pendharkar, G.H. Subramanian, and J. Rodger, "A probabilistic model for predicting software development effort," *IEEE Transactions on Software Engineering*, vol. 31, no.7, pp. 615-624, 2005.

[61] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," *Proc. ACM ICSE conference*, 2005.