Information Theoretic Evaluation of Change Prediction Models for Large-Scale Software

by

Mina Askari

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

© Mina Askari 2006

I hereby declare that I am the sole author of this thesis.

This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Abstract

During software development and maintenance, as a software system evolves, changes are made and bugs are fixed in various files. In large-scale systems, file histories are stored in software repositories, such as CVS, which record modifications. By studying software repositories, we can learn about open source software development processes. Knowing where these changes will happen in advance, gives power to managers and developers to concentrate on those files. Due to the unpredictability in software development process, proposing an accurate change prediction model is hard. It is even harder to compare different models with the actual model of changes that is not available.

In this thesis, we first analyze the information generated during the development process, which can be obtained through mining the software repositories. We observe that the change data follows a Zipf distribution and exhibits self-similarity. Based on the extracted data, we then develop three probabilistic models to predict which files will have changes or bugs. One purpose of creating these models is to rank the files of the software that are most susceptible to having faults.

The first model is Maximum Likelihood Estimation (MLE), which simply counts the number of events i.e., changes or bugs that occur in to each file, and normalizes the counts to compute a probability distribution. The second model is Reflexive Exponential Decay (RED), in which we postulate that the predictive rate of modification in a file is incremented by any modification to that file and decays exponentially. The result of a new bug occurring to that file is a new exponential effect added to the first one. The third model is called RED Co-Changes (REDCC). With each modification to a given file, the REDCC model not only increments its predictive rate, but also increments the rate for other files that are related to the

given file through previous co-changes.

We then present an information-theoretic approach to evaluate the performance of different prediction models. In this approach, the closeness of model distribution to the actual unknown probability distribution of the system is measured using cross entropy. We evaluate our prediction models empirically using the proposed information-theoretic approach for six large open source systems. Based on this evaluation, we observe that of our three prediction models, the REDCC model predicts the distribution that is closest to the actual distribution for all the studied systems.

Acknowledgments

I would like to thank my supervisor Prof. Richard. C. Holt for his great support, encouragement and guidance throughout my Masters here at Waterloo. I am also grateful for his extreme patience and compassion, his knowledge and experience which have made him a valued supervisor as well as a trusted friend. He thought me many valuable lessons that I will never forget them.

In addition, I would like to thank my readers, Prof. Michael W. Godfrey and Prof. Grant Weddell, who took the time to read and critique my thesis, offering helpful suggestions for improving it. I also express my deep appreciation to all who have contributed to this work in any way. I would like to thank Ahmed Hassan who gave me the required data for using in this thesis and his help whenever I had any question regarding my research. I also can not ignore the valuable contributions of the Open Source community who made it possible for us to use their detailed source code repositories. I would also like to thank my friends and members of the Software Architecture Group at the University of Waterloo who I had many memorable time with them. Special thanks to Kristina Hildebrand for helping me to improve this thesis' writing.

Finally, it is with love and gratitude that I thank my husband Majid for helping to motivate me to continue my study, then for being supportive to me through many difficult days of study.

Dedication

I dedicate this work to Majid and our lovely son Navid.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are continually being changed to adapt to meet the needs of their users, or to correct faults appearing in systems during development or after deployment. There has been extensive research on inventing new processes and approaches for developing software systems to reduce the numbers of these changes and faults. The idea is that, during software development by following certain principles, you can decrease the probability of the occurrence of certain kinds of modifications. Despite all this progress changes and bugs during software development are inevitable. For this reason, developing new techniques to predict the future behavior of changes and bugs is important.

Regardless of the development process and the software system under development, there is always valuable information produced during that process. Software repositories contain developers' messages for each committed change. The record of changes is a useful resource that developers can use to maintain and manage the software. We believe that part of this information captured in software repositories can be used to develop models to predict what bugs and changes are likely to

happen next. After recovering these data, we need tools and techniques that can analyze them and give the developers a better view of future changes.

Developing such tools and approaches is a big challenge because of the unpredictability nature of the development process [1]. The main results of this thesis is to develop such models using software repositories and then create a novel approach for comparing them. This thesis shows that the processed historical records from the source control repositories of large software systems can be used for developing prediction models and for determining how much information is captured by these prediction models.

This chapter is organized as follows. Section 1 describes the motivating idea behind this thesis. We argue the need for these prediction models and evaluating methods. Section 2 gives an overview of the thesis in both the content and organization. And finally section 3 highlights the contributions of this thesis.

1.1 Motivating Idea: more general and realistic models and techniques

The idea for predicting which subsystems are most susceptible to having a fault in the near future is a well-known idea . There exit several prediction model [2] [3] [4] [5] [6] [7] [8] [9] and more are emerging. However, many of previous fault prediction findings have not been evaluated and are not applicable to large-scale software systems because of their complexity [10]. The general approach for evaluating these models is to run the system and collect the information during its execution and then compare it with the results which were predicted by models [11]. The problem is that these models are not general and applicable for various software

systems and in many cases there are not comparable because they are measuring different metrics [10]. Most of the techniques described in this thesis are targeted toward more general and realistic models. We are looking for the methods and models that, first, use the useful data collected during development process and, second, can be justified by determining how close they are to the distribution of data. In our information theoretic methodology for comparing different prediction model, we use the potential information stored in the historical development data in terms of entropy.

We believe that our presented approach is a novel method that can be used to validate the performance of any prediction models in form of probability distribution. We will present applying the method on such models in detail. Using available techniques and tools like C-REX [8] it is possible to extract the information for making the prediction models and evaluating them.

1.2 Overview of Thesis

Software products are embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product [12]. Considering this fact, developing new techniques to cope with recurring changes and bugs is essential. Fortunately the history of these frequent changes are often kept in software repositories. For example, software repositories contain developers' messages for each committed change. These repositories are a valuable resource that developers can use to maintain and manage the software.

This thesis deals mainly with analyzing the extracted data from software repositories to develop models to assist developers in predicting what files will contain

future bugs and changes. We also evaluate our techniques and tools using case studies of six large-scale software systems. (See Table 3.1 on page 22)

The remaining part of this section is organized as follows: in the problem definition section we describe the problem we are trying to solve. In overview of proposed solution, a brief explanation of the methods and models we have developed is given. We finish this section with organization of the paper for the rest of chapters.

1.2.1 Problem Definition

Suppose we have a record of all the events that have occurred on the set of files that make up a software system during the development process until the present time. These events are basically file changes whose purpose is to fix bugs, to add new features, or to change existing features. We have extracted these events from the history of the software stored in repositories. We might ask several questions: Are these changes random? What will be the average uncertainty of the next modification, if all past modifications are known? Depending on the properties of the observed sequence of modifications, the uncertainty of the next modification may be influenced by the knowledge of the past. For answering this question we analyze these events and present the results in chapter 3.

Based on the results we will show in next chapters, we believe that for CVS log data there should be some information from the past that can be used to predict the future. But how much information is buried in these CVS logs and how can we capture this information? How much of it can be captured? We will suppose that certain models can use the information from the past to predict the future. The question is: How good are these prediction models? By "good" we mean how close they are to the true distributions of the events. It also could mean that how well

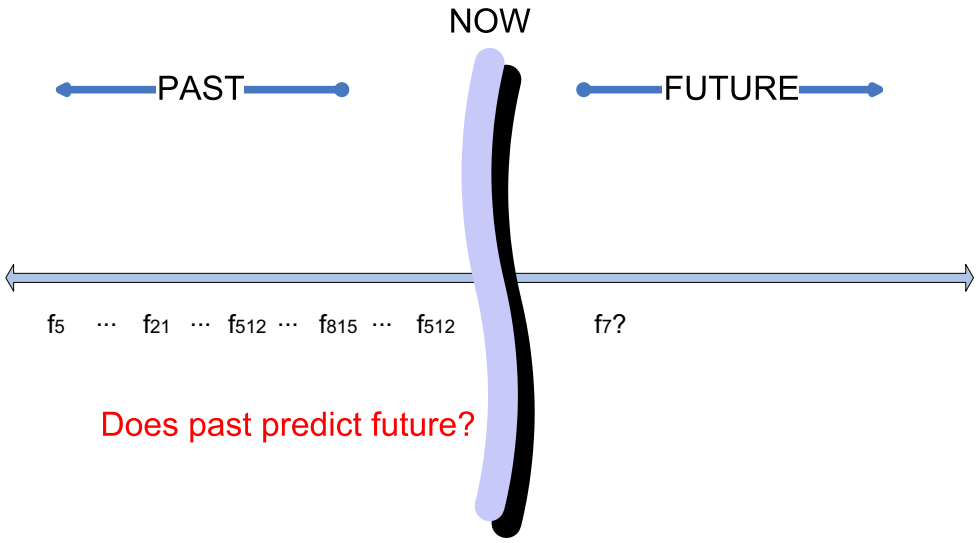

Figure 1.1: Does past history of a software development predict the future of that?

they predict the occurrence of the next event. Figure 1.1 show an abstract form of the problem which we are trying to solve. In this figure, we have a sequence of modifications which have occurred on the files to until now. The sequence shows that there have been modifications to file numbers 5, 21, 512, and 815 in the past. We would like this information to make prediction of the future. For example, we are interested to see if we are likely to have any modification on file number 7 (shown as f_7 in the figure) or any of other files.

1.2.2 Overview of Proposed Solution

One promising approach is to find the actual probability distribution of the events and then compare this actual distribution with the predicted distribution. But if finding the actual probability distribution of data were possible then we would not need to know the prediction probability distribution. So we try to use past events to create prediction models so that their distribution is close enough to the

actual distribution of changes. In fact, we suppose that whenever an event occurs, the distribution of the data is changed and our models must update themselves to make the distribution as close as possible to the observed data. To evaluate how close they are, we use the idea of Corpus Cross Entropy, which is borrowed from the field of Natural Language Processing [13]. This technique is used to measure the distribution difference between natural language models and a "corpus" which is a sequence of words in the language.

1.2.3 Organization

This thesis is organized as follows. In chapter 2 we review related and background work, including software reliability models for bug prediction, software repositories and their role in prediction models, and comparing prediction models in Natural Language Processing field. The heart of thesis is organized in three main parts presented by three separate chapters as follows:

Chapter 3 presents the techniques we used to perform several experiments to analyze the extracted data. We investigate and analyze them to assure their usefulness in our specific approaches. We found several interesting results, including Zipf's law and self similarity, in our data. Based on these results we conjecture that the extracted data are appropriate for use in making prediction models and evaluating them.

Chapter 4 presents prediction models and the way that we build them. To develop these prediction models, we speculate that whenever an event occurs, the effect of that event stays in the system, but decays as time go on. The event may not affect the future, but may be correlated only to the rest of the data. Different parameters of the models specify the speed and the shape of this decay. In the

RED (Reflexive Exponential Decay) model, the effect of each event stays in the system but decays exponentially. Whenever a new event happens on a file, the past remained effect will be added to the new exponential effect. So at any time each file has a value representing the potential of that file for having a modification in the future. This value decreases as time goes on.

In Chapter 5, we present two evaluation approaches for prediction models. First we explains the Top Ten List [3] approach developed by Hassan et al. Then we present an information theoretic method that we propose for evaluating the prediction model. For that, we first introduce some concepts of information theory used in this method. We also compare these two approach in this chapter.

Chapter 6 evaluates our prediction models using two approaches on six open source software systems. First, we use Top Ten List [3] approach to see which model predicts best. Chapter 6 applies this method for comparing our proposed prediction models and we get some interesting results, which are presented in this chapter. We show that REDCC model has statistically significant better accuracy in predicting the occurrence of changes and bugs than simply using the number of prior modifications to a file or prior faults in it as predictors of faults.

Finally, Chapter 7 concludes the thesis and discusses possible future work. This chapter points out that while the results demonstrate an order of magnitude quality improvement for some methods, other assumptions for applying these methods should be considered.

1.3 Contributions

A central contribution of this thesis is the proposal of a novel approach to measure the accuracy of a bug/change predictor. We call it an information theoretic approach because we use the concept of cross entropy from information theory for measuring the closeness of two distributions. This technique is used to see how close the prediction model distribution is to the actual data distribution. In fact, we suppose that the proposed prediction models are approximate distribution probability. The closer these distributions are to the underlying data distribution, the better they are. Since we do not have the actual distribution of observed data, this sounds impossible. Although we claim this is a new approach in this area, it is well known in natural language processing for measuring the difference between the prediction probability distribution and corpus (data) probability distribution.

The other major contribution of this thesis is to develop two new prediction models, RED and REDCC, based on the history of development which combine the previous models that only consider frequency or recency of changes.

The empirical contributions of this thesis are the application of all proposed techniques and approaches on several long lived large open source projects. This empirical case study was done for six large-scale open source systems.(Table 3.1)

Chapter 2

Background and Related Work

This chapter gives a brief overview of the existing research in the different areas related to our work. We begin with an overview of software reliability models in general, and software reliability models for bug prediction in particular. A summary of existing approaches used in Natural Language Processing (NLP) for creating prediction models and evaluating them against the actual model of the language follow, forming the ground work for this thesis. In this chapter we also introduce some research related to software repositories.

2.1 Software Reliability

According to ANSI [14], reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software reliability includes measurements such as estimation and prediction, the attributes and metrics of software development and their implications on reliability, and the application of this knowledge in guiding software development [15]. Software reliability

prediction models mostly concentrate on software failures during the execution of the system, although for developing the prediction models, different metrics from the development process must be considered. Typically, the key variables in these models are either size and complexity metrics or measures arising from testing. In this section, first we look at several different prediction models in this area, and then we examine approaches for comparing these models.

2.1.1 Software Reliability Models for Bug Prediction

To understand the reasons for software failures, and to try to quantify software reliability, many reliability models have been used, but how to quantify software reliability remains largely unsolved. A reliability model is a mathematical formula consisting of an equation (or a set of equations) that takes some software metric data as input and calculates some other values. Most software models contain the following parts: assumptions, factors, and a mathematical function that relates the reliability with the factors. The mathematical function is usually higher order exponential or logarithmic.

For example, in Musa's Basic Execution Time Model [15], one assumption is that the cumulative number of failures, by time t, follows a Poisson process with mean value function:

$\mu(t) = \beta_0[1 - exp(-\beta_1 t))$

Based on that, the failure intensity function for this model is $\lambda(t) = \beta_0\beta_1 exp(-\beta_1 t)$.

A large number of software reliability models have emerged for creating more reliable software. Software reliability models are made by measuring different aspects of software, such as product metrics (lines of code with relation to defects

and test coverage metrics), project management metrics, process metrics, and fault and failure metrics.

As many models as there are, with many more emerging, these models too often fail to capture a satisfactory understanding measure of the complexity for software failures; constraints and assumptions have to be made for the quantifying process. Therefore, there is no single model that can be used in all situations. No model is complete. One model may work well for a certain set of applications, but may be ill suited for other kinds of problems because of different characteristics of different types of software. In fact, no good quantitative models have been developed to represent software reliability without excessive limitations.

Software reliability models are usually made before deployment of the system, in software testing, verification and validation, or after deployment dealing with faults. Based on this software reliability modelling techniques can be divided into two categories: prediction modelling and estimation modelling [12]. Both kinds of modelling techniques are based on observing and accumulating failure data and analyzing with statistical inference.

The purpose of **prediction models** is to perform software reliability prediction in an early stage of software development (when software is in the design or coding stage). Gaffney and Davis Phase-Based Model [16], Musa's execution time model [17] and Rome Laboratory Model [18] are in this category. Phase-base model makes use of fault statistics obtained during a technical review of the requirements, design and implementation to predict the reliability during test and operation. The Rome Laboratory Work model predicts fault density which can be transformed into other reliability measures such as failure rates.

Alternatively, software reliability **estimation models** determine current soft-

ware reliability by applying statistical inference techniques to failure data obtained during system test or operation. For example most current software reliability models such as exponential distribution models, Weibull distribution model, Thompson and Chelson's model [15] fall into this category. Exponential models and the Weibull distribution model are usually considered to be classical fault count/fault rate estimation models, while Thompson and Chelson's model belong to Bayesian fault rate estimation models [15].

The software reliability models that are most closely related to our work are those that try to detect fault-prone modules. We are also trying to find the change-prone files in a system. In order to detect fault-prone modules, many studies have given proposed prediction models using various analysis methods based on software product metrics. In our approach we give our prediction model based on a software process metrics that is, development history of the software.

To detect fault-prone modules, some models proposed a decision tree approach to classify low-quality modules based on Huffman Coding [19]. Some models predicted fault-prone modules using a neural network model [20]. Another study identified fault-prone modules in telephone switching with using multiple regression analysis [21]. They evaluated the accuracy of several models using Alberg Diagrams. Also, researchers classified fault-prone module using discriminant analysis [22]. In another work researchers analyzed effective metrics to classify fault-prone modules using logistic regression analysis [23]. Most of these results showed that there is a close relationship between module metrics and quality, and that all of these prediction models could predict fault-prone modules to some degree of accuracy.

We can look at previous research related to predicting bugs using another approach. There are some count-based techniques, which focus on predicting the number of faults in subsystems of a software system. Another approach is classifi-

cation based techniques which focus on predicting which subsystems in a software system are fault-prone [8].

There are many open questions with respect to the realism of the underlying assumptions, accuracy, and applicability of software reliability models and our goal is to contribute towards more realistic assessment and prediction of software reliability based on theoretical and empirical studies.

2.1.2 Evaluation of Models

There has not been a general evaluation of the assumptions on which the previously developed models are based. Therefore, there has not been a general evaluation of the performance of these models on real data.

There are many studies to apply software reliability error prediction models from the literature to error data of some software projects. This error data can detail the error history of the software in a broad range, from the start of validation testing through the end of demonstration testing, and on,to include actual operational error data. These studies involved obtaining predictions from the various models used, and comparing the predicted results with the actual error data history. For example, count based models are validated by dividing the data into equal size periods and predicting the faults in one period using data from all the previous period. On the other hand, classification based models are validated by testing whether the data from one release can be used to predict if a subsystem will be fault-prone in subsequent releases. From these comparisons, analysis to determine the reasons why the predicted results did or did not agree with actual results can be performed. Some of these studies indicate that the models fit poorly, not only for shortcomings of the models but also due to problems with the error data [10]. So

not only it is important to look for new prediction models, but it is also necessary to look for new kinds of data from a new angle.

In our work, for comparing the prediction models we use this general approach, but instead of the history of errors, we use the software development history which contains the history of the bugs and changes in the system. In addition to this general approach, an information theoretic approach for evaluating of the prediction models has been used. This approach can work in different stages of the software life cycle: during development, before release, after release, between releases and during deployment.

2.2 Using Software Repositories for Prediction

Data mining is the process of extracting meaningful information from large quantities of data. It involves uncovering patterns in the data and is often tied to data warehousing, because it makes such large amounts of data usable. Data elements are grouped into distinct categories so that predictions can be made about other pieces of data. Hand defines data mining as 'the process of secondary analysis of large data aimed at finding unsuspected relationships which are of interest or value to the database owners [24]. Data mining is most appropriate when one seeks valuable bits of knowledge within large amounts of data collected for some other purpose, and when the amount of data is so large that manual analysis is not practical.

Software repositories can be considered as a specific kind of data warehouse and data mining can be done on them to retrieve useful information. In large-scale software systems, file histories are implicitly stored in software repositories such as CVS, which records modificationsto the files of system source artifacts. With CVS,

old versions can be retrieved and analyzed to find which changes caused a bug. CVS can also help when a project is being worked on by multiple people, where overwriting each others' changes is easy to do [25]. In addition to these primary goals of software repositories, they can be mined for learning about open source software development processes. Various software artifacts are available for a wide variety of open source projects such as:

- source code,
- requirements and design documentation,
- test suites,
- CVS logs which containing change information between different version releases, and
- change logs and test logs.

Many researchers

in the field of software development have realized the value of historical data and have used this data in their research, ranging from software design to software understanding, software maintenance, development process and many more areas [26] [3] [27] [5] [4] [28] [29] [30] [6] [31].

There is considerable research on developing tools to recover such historical data. Hassan et al. [8] developed C-REX, an evolutionary code extractor tool, which recovers information from source code repositories.

Zimmermann et al. [26] used version repositories to determine co-change clusters. They applied data mining to version histories in order to guide programmers

through related changes. For detecting another kind of co-changes, Gall et al. [30] used software repositories. They uncovered the dependencies and interrelations between classes and modules (logical dependencies) which can be used by developers in the maintenance phase of a system's life cycle.

Graves et al. [6] showed that there is a relationship between the number of changes a subsystem has and the future faults in that subsystem. Hassan et al. [3] presented various heuristics using historic version control data to create the Top Ten List, which highlights to managers the ten subsystems that are most susceptible having a fault. They also developed techniques to measure the performance of these heuristics.

Mockus et al. [32] studied a large legacy system to test the hypothesis that historic version control data can be used to determine the purpose of software changes and to understand and predict the state of a software project [33]. Khoshgoftaar et al. [2] [5] used process history to predict software reliability and to show that the number of prior modifications to a file is a good predictor of its future faults.

Basili and Perricone used change data to analyze the relationships between the frequency, distribution, and type of errors during software development, the maintenance of the developed software, and a variety of environmental factors [29]. In a similar work, Perry et al. used change data and survey about the software faults encountered during the testing phases in evolving a large real-time system to characterize them in terms of interface and implementation faults and to estimate the efforts associated with fixing them [34].

Eick et al. [27] presented visualization techniques to explore change data, to help engineers understand and manage the software change process. Ostrand, et al. [4] suggested a model to predict the number of faults for a large industrial inventory

system based on the history of the previous releases.

Our approach takes guidance from this previous work, but is notably different by suggesting new prediction models and by using an information theoretic approach to measure the effectiveness of such models.

2.3 Information Theory in Natural Language Processing

Natural Language Processing (NLP) involves any aspect of processing language by computer as humans speak these languages and write the words of language. The information theoretic approach we will propose, is well known in NLP, but to our knowledge has not been applied in the field of Mining Software Repositories. In NLP, a sequence of words in a language is called a corpus. One challenge in NLP is to estimate the word generation probabilities for words that occur in the corpus. There is a need to build good probability models to describe the behavior of languages (modelled as random variables producing the words, letters, etc.). Statistical inference can be used for the task of language modelling (e.g., how to predict the next word given the previous words). Statistical inference consists of taking some data — generated in accordance with some unknown probability distribution— and then making some inference about this distribution. In order to do this, a model of the language is needed. Probability theory helps for finding such a model. There are two common approaches for making these models [13]:

- Parametric:

  Assume that some phenomenon in language is acceptably modelled by one of the well-known families of distributions — such as binomial or normal —

then we have an explicit probabilistic model of the process by which the data was generated. Determining a particular probability distribution within the family requires only the specification of a few parameters using training data

- Non-parametric (Distribution free):

  Here there is no assumption about the underlying distribution of the data. For example, we could estimate P empirically by counting a large number of random events in a distribution-free method. Because there is less prior information, more training data is needed.

After making these language models, we need different ways to compare distributions to see how "close" they are. In NLP, a variant of the entropy definition — corpus cross entropy — is used to compare two probability functions. A corpus contains the training text used to construct a language model. The basic format for a corpus file is ASCII text where all words/tokens are separated by white space.

NLP also uses information theory to find the distance between prediction models and actual distribution of corpus [13]. In natural language the messages are the corpus of language, but information theory can apply to other form of messages too.

Chapter 3

Characteristics of the Extracted Data

The prediction models and the evaluation methods suggested in this thesis are based on the data that we have. Due to this important role of data in our study, investigating and analyzing it to assure its usefulness in our work seems necessary. By data, we mean the information that is generated during the development process and is stored in the software repositories such as CVS. This data can be obtained through mining the repositories of the software. Depending on the software configuration management of a system, there are different kinds of information stored during the development process, including source code, requirements and design documentation, test suites, CVS logs (which contain change information between different version releases), change logs, test logs, and defect tracking systems. For this study we have used data that has been extracted from CVS logs of several Open Source Systems (Table 3.1). The first operation on the CVS log is data mining phase. It is the process of extracting meaningful information from large

quantities of data in the CVS. We have used the provided extracted data of these CVS by Ahmed Hassan [8]. Then Data elements are grouped into categories so that we can analyze and study the specific properties of the data. For example, we categorize the changes that have occurred on the files based on the time of the change or the name of the developer.

The rest of this Chapter is organized as follows: First we will look at open source software systems and their role in our studies. We then describe a set of experiments that were performed to analyze the extracted data, which are used by our models, to see the interesting characteristics they have. We explain these experiments in the following sections: File Activities Charts, Zipf's Law, Self Similarity and Stationary-Non Stationary. We will also show the results of these experiments in more details. Our goal is to see how useful our data is, and how much information we have in the data regardless of how the data will be used in our prediction models.

3.1 Role of Open Source Systems in our Studies

Finding the historical information generated during software development is a necessary phase in our study. Using this information we have the chance to perform empirical studies for our theoretical methods. This kind of useful information is not generally available for commercial software products, since companies are usually very reluctant to provide these kinds of information to the public. On the other hand, the source code of open source software, which is programmed by different programmers across the world, is free and available to anybody who wants to use it or modify it for his or her specific purpose.

In addition to the source code of OSS , source control system information such as CVS logs and bug tracking system data are usually also available. Various software

configuration management (SCM) and version control systems are available for Open Source Software or Free Software (OSS/FS). CVS, Subversion, GNU arch, and Monotone are some of these SCM most of which are decentralized. The most popular SCM tool is CVS, which has a number of awkward limitations: changes are tracked per-file instead of per-change, commits are not atomic, renaming files and directories is awkward, and its branching limitations mean that you must tag them or you will run into difficulties later. Despite these drawbacks, CVS is still the major source code collaboration tool being used in OSS projects. CVS helps manage the files and code of a project when several people are working on the project as the same time. Bug tracking is another important aspect of OSS projects. With bug tracking you can track bugs and code changes, communicate with teammates, submit and review patches, and manage quality assurance. It includes keeping a record of all reported bugs, whether the bug has been fixed or not, the relevant version of the software to which the bug belongs, and whether the bug submitter has agreed that the bug has been fixed. Popular bug tracking systems include Bugzilla and GNATS.

With the emergence of these open source software systems researchers are able to acquire the needed information to study a variety of open source projects. As these open source systems evolve they generate a huge amount of raw information related to all kind of activities that have been performed during the development process and are recorded in the relevant repositories. Contrary to historical information for commercial systems, where access to their history is usually very limited, access to such information for open source systems is freely available and open. For our study, we used the historical information which were stored in CVS logs. We did not use the bug tracking systems, since for most of the open source systems detailed bug tracking systems do not exist. Also, in many cases, reported bugs are not an

Application Name	Start Date Date	Application Type	Programming Language	Total Files
OpenBSD	Oct 1995	OS	C	7065
FreeBSD	June 1993	OS	C	5272
KDE	April 1997	Windowing	C++	4063
Koffice	April 1998	Productivity	C++	6312
NetBSD	March 1993	OS	C	11760
Postgres	July 1996	DBMS	C	1468

Table 3.1: Information about studied systems.

indication of the occurrence of faults as the reported bugs may not be a correct bug detection.

3.2 Studied Systems

The large number of available OSS projects and the possibility of access to their history permitted us to obtain our required data in order to make and empirically verify our proposed models and approaches, and to interpret our findings. To perform our study we used several open source software systems. Table 3.1 summarizes the details of the software systems we studied.

The oldest system is over ten years old and the youngest system is five years old. We tried to choose the applications from different domains and different sizes. To build our techniques and models, we used the CVS log of these systems. We were looking for any kind of change and bug that occurs in different files of a system. The process of acquiring such specific data is challenging, since CVS logs are mainly designed for record keeping and commits are not atomic. Also, the large amount of data stored in these repositories complicates the data extraction process. For analyzing data and creating prediction models and comparing them based on the

data, our main concern was to perform our studies on the data of several CVS logs in a standard format that is easier to process. We did not focus on developing tools that automatically recover data from these repositories. So we got the CVS logs data from our colleague Ahmed Hassan. These data were extracted from these CVS logs using tools developed by him [8]. This allowed us to concentrate on analyzing the extracted data instead of spending a large amount of time creating methods and developing tools to recover the data first.

In the extracted data each change message has been classified as

- a bug fix,
- a feature introduction, or
- a general maintenance change.

For each change, the time of change is recorded, followed by the name of the file in which this change has occurred , then the type of the change – either a bug fix, feature introduction or general maintenance change. Table 3.2 shows the total number of these events and the duration of development for each studied software system. In our experiments and models, we are interested only in the bug fixing and feature introduction events. We refer to these as Bug and Change events. We discard all the information which was for maintenance changes. Table 3.2 also shows the remaining number of events after discarding this information.

3.3 File Activity Charts and Zipf's Law

By file activity we mean the events (Changes and Bugs) that happen to each file during the development. We started by counting the number of modifications which

Application Name	Duration (Month)	Total Changed Files	Total Events	Bug and Changes
OpenBSD	88	7065	80354	67149
FreeBSD	115	5272	126432	101252
KDE	70	4063	93204	77994
Koffice	58	6312	92944	73409
NetBSD	119	11760	239628	131307
Postgres	77	1468	41175	26510

Table 3.2: Number of events available for different systems.

happened for each file during the development process. Based on the history of the development, if we count how often each file is modified, and then list the files in order of the frequency of occurrence, we can explore the relationship between the frequency of a file and its position in the list, known as its rank. We computed the file activities for each file of our studied systems and ranked them. Figure 3.1 illustrates the number of modifications for each file for the different systems we studied and Table 3.3 shows the maximum frequency observed for the files. Different systems have different numbers of files. Therefore, to compare all the studied systems in a single plot, we used the percentage of files and percentage of activities for each file in Figures 3.1 to 3.3.

As it can be seen from Figure 3.1, there are few files with high frequency of changes but many files with very low number of changes. It also can be seen in the figure, these frequencies follow a similar pattern in all studied systems. This behavior indicates that the change data follows the general form the Pareto (or 80-20) law [35] and Zipf's law [36]. The Pareto law, in its generalized form, states that 80% of the objectives - or more generally, the effects - are achieved with 20% of the means. In order to show that there is 80-20 law in our data, we plotted the cumulative distributions of the file frequencies for studied systems in Figure 3.2. It can be seen from the figure that almost 20% of the files in the systems have roughly

Application Name	Duration (Month)	Total Files	Maximum Frequency
OpenBSD	88	7065	592
FreeBSD	115	5272	553
KDE	70	4063	854
Koffice	58	6312	989
NetBSD	119	11760	562
Postgres	77	1468	292

Table 3.3: Maximum frequency observed for the files of different systems during development.

80% of activities during development.

Originally, Zipf's law [36] stated that, in a corpus of natural language utterances, the frequency of any word is roughly inversely proportional to its rank in the frequency table. So, the most frequent word will occur approximately twice as often as the second most frequent word, which occurs twice as often as the fourth most frequent word, and so on. In our extracted historical data, Zipf's law is most easily observed by scatterplotting the data, with the axes being log (rank order) and log(frequency). The Zipf's law graph shows the rank on the X-axis versus frequency on the Y-axis, using logarithmic scales. To show that Zipf's law holds for the data and change frequency of any file is roughly inversely proportional to its rank in the frequency, we plotted the log-log scale of the cumulative frequency distributions, in Figure 3.3. It can be seen that the points are close to a single straight line thereby confirming that the data approximates Zipf's law [20].

3.4 Spectrographs of the Data

Hassan et al. introduce a technique to study historical data extracted from tracking the evolution of long lived processes [37]. In particular, they present a visualization

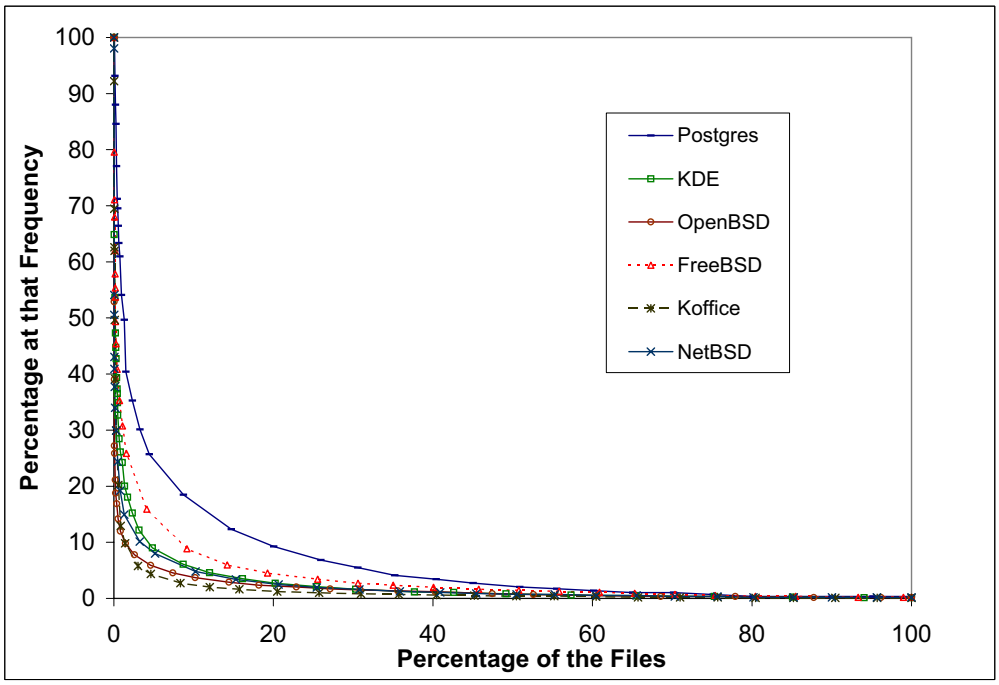

Figure 3.1: File activity charts for all studied systems

approach, called evolution spectrographs to assist in identifying interesting patterns and events during evolutionary analysis of such historical data. Spectrographs can be used to visualize how the files of a system are modified during the development period of that software system. All this information shown by a spectrograph can be useful not only for assisting developers, but can also be used by project managers to achieve an improved allocation of restricted resources [37].

We used spectrograph tool [37] to see if there is any interesting change patterns visible in our historical data, and if there is, how we can interpret these patterns regarding the effect of changes and bugs on each other. We believe that finding such patterns would be another support of our hypothesis regarding the value of the information available in such data. Figures 3.4 to 3.9 show the spectrographs of the historical data of six studied open source systems. Here the vertical axis, shows the

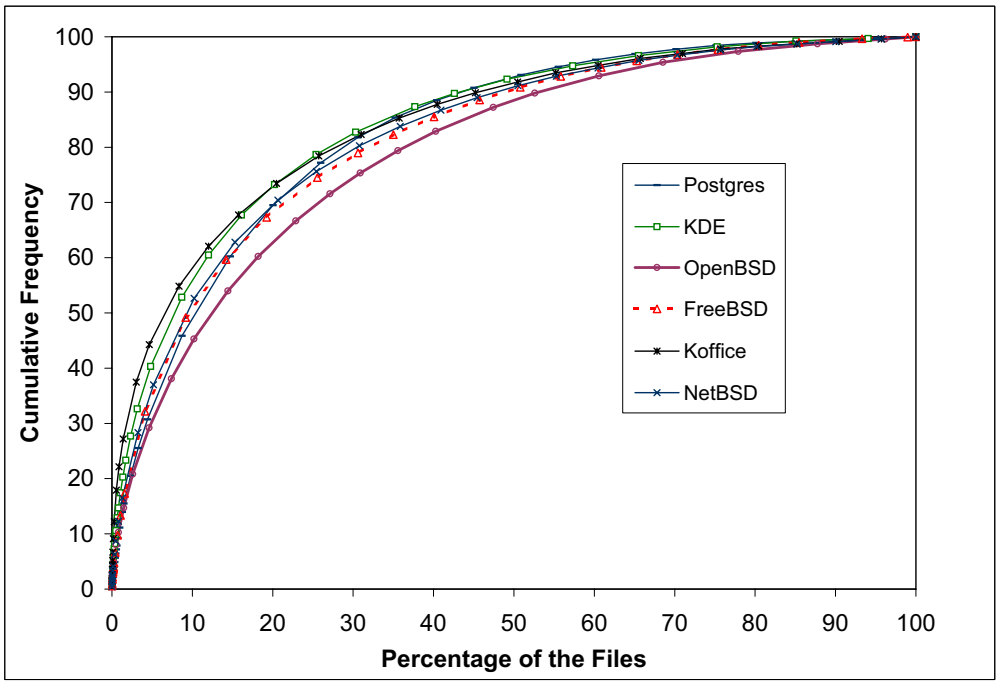

Figure 3.2: Cumulative frequency distribution for all file activity charts systems

files of a system which are different for each system, and the horizontal axis, shows the development period of that software system. The unit of this period is month. The files are sorted top to bottom based on the number of changes and bugs which they have. For the systems with a high number of files, only a portion of the files with the highest changes and bugs are shown. In the resulting display, frequent changes can be highlighted using colors. As it can be seen in these figures there are some clear patterns common in all of them. For example, almost all of these systems have an extensive changes at the end of the development process. Also, it is clear that in some periods all the files of the system are involved in some changes. It also can be seen that most of the changes are happening on some certain files supporting the our Zipf's law finding.

All in all, what is important for us is that we observe that there are some clear

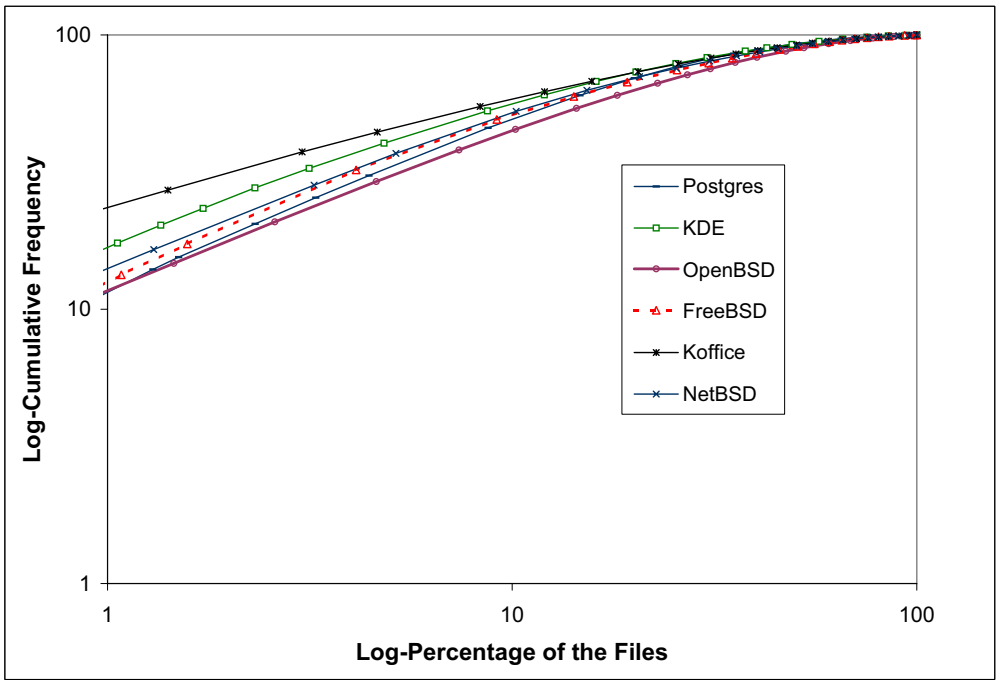

Figure 3.3: Cumulative frequency distribution for all file activity charts systems

pattern seen in all these diagrams. Having the spectograph diagrams depicting the historical changes of a system, interpreting the behavior of modifications occurred on the files of a system is possible.

3.5 Self-Similarity in Data

A self-similar object is exactly or approximately similar to a part of itself. There are some surprising observations, which we discovered through our experiments. The first observation was that for each system, when we plotted the file activity charts for different periods of software development, they followed the same pattern during these different periods. When we find that there is a relationship between these file activities during different periods, then we can use the past information

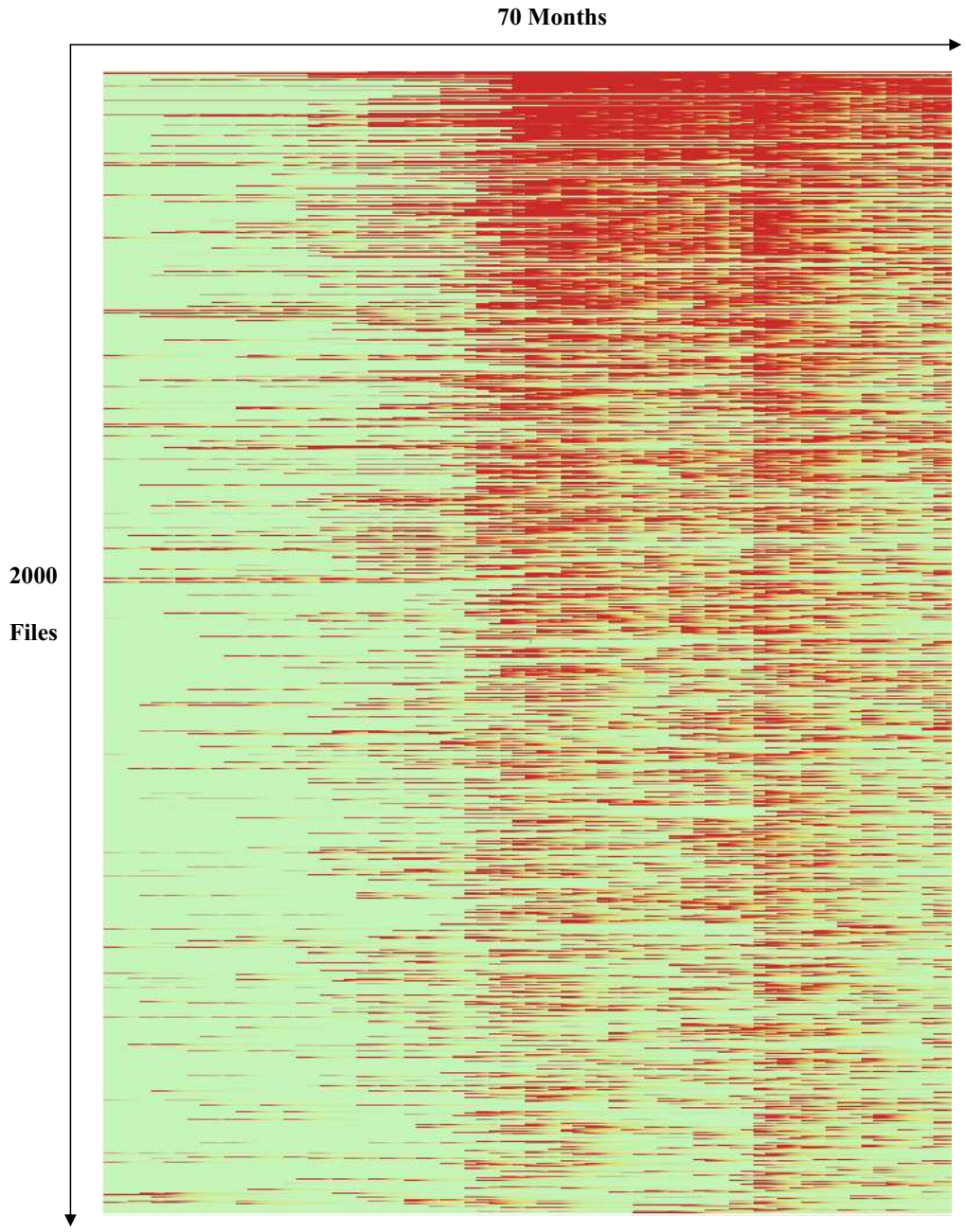

Figure 3.4: Spectrograph of KDE for top 2000 files

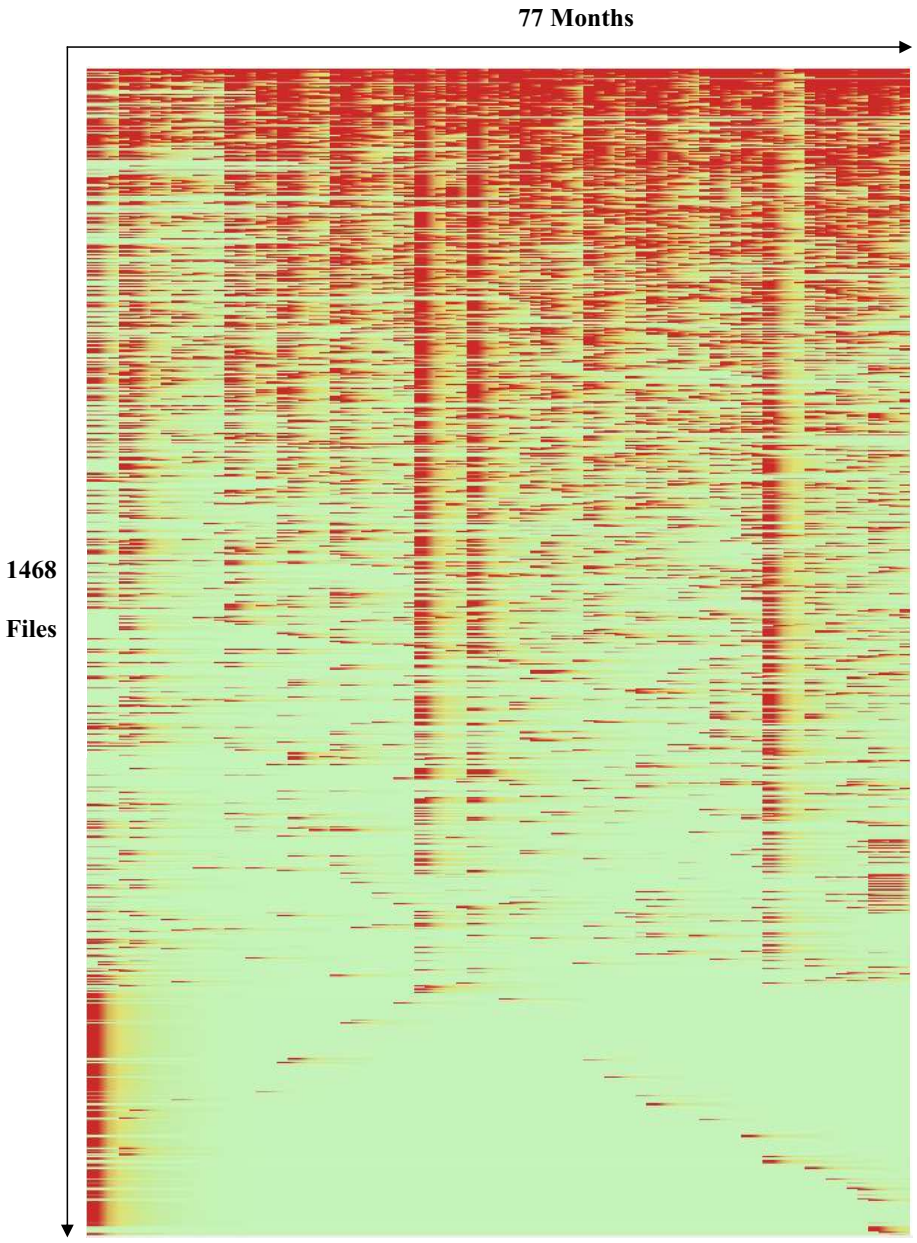

Figure 3.5: Spectrograph of Postgres for all files

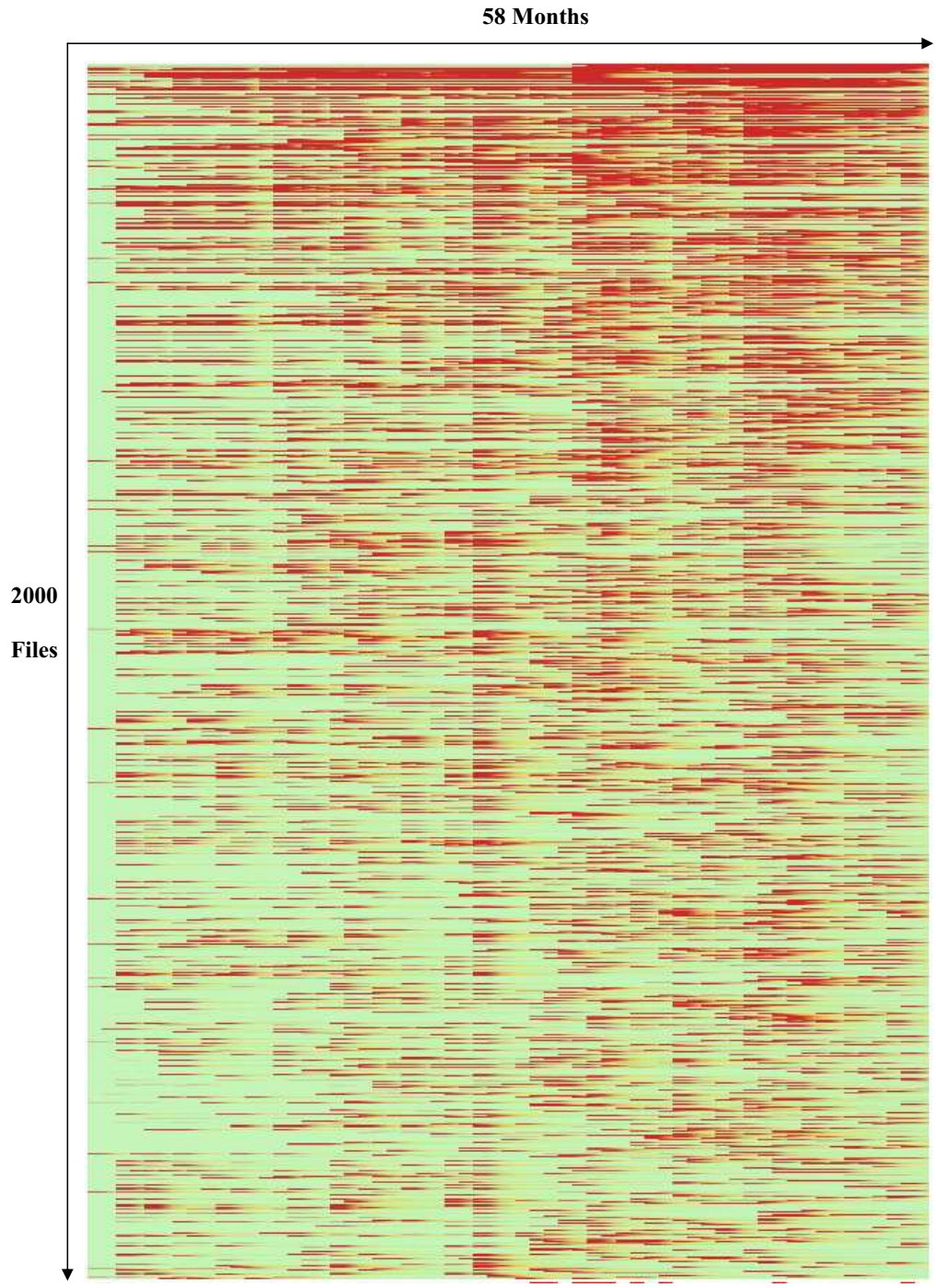

Figure 3.6: Spectrograph of Koffice for top 2000 files

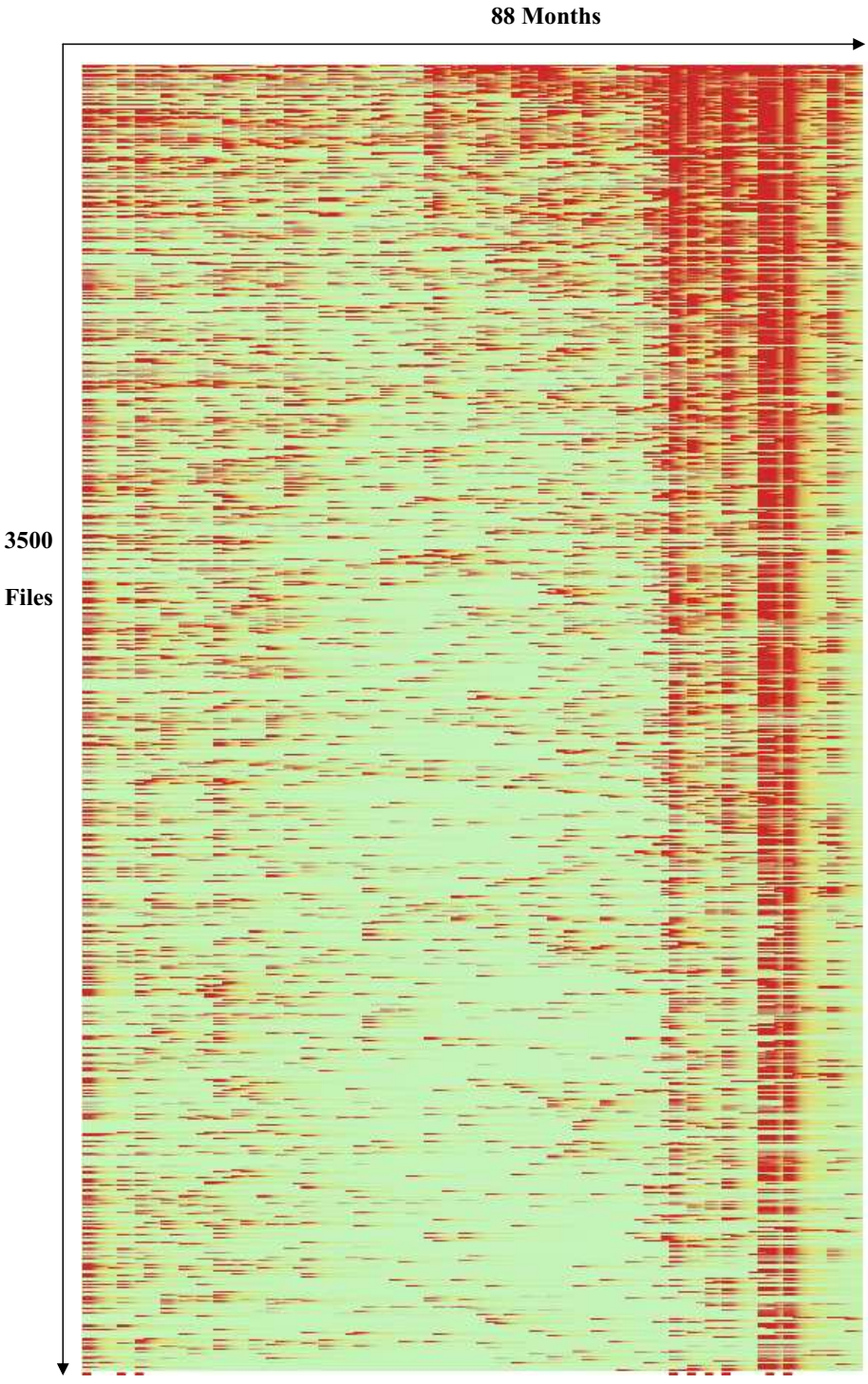

Figure 3.7: Spectrograph of OpenBSD for top 3500 files

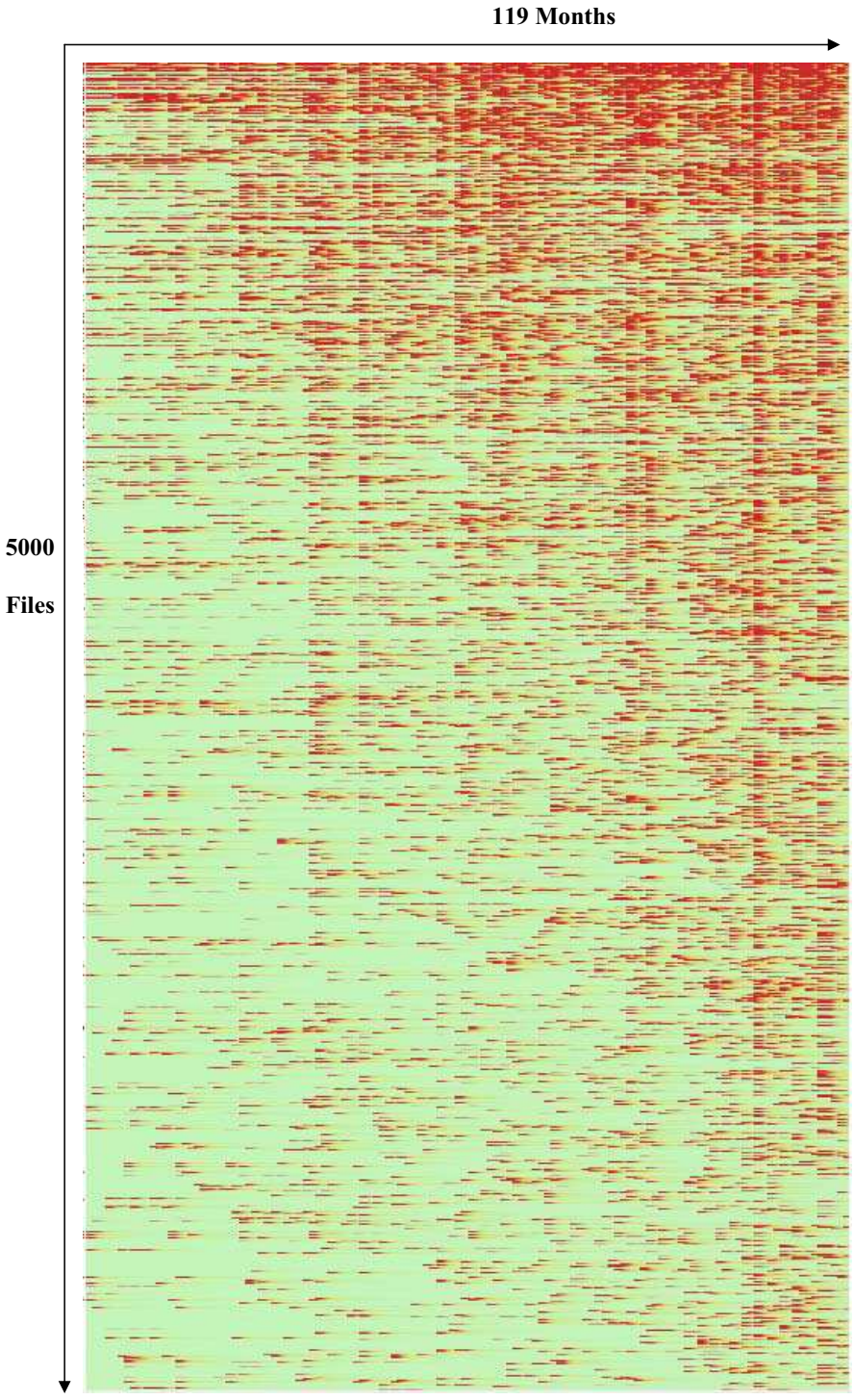

Figure 3.8: Spectrograph of NetBSD for top 5000 files

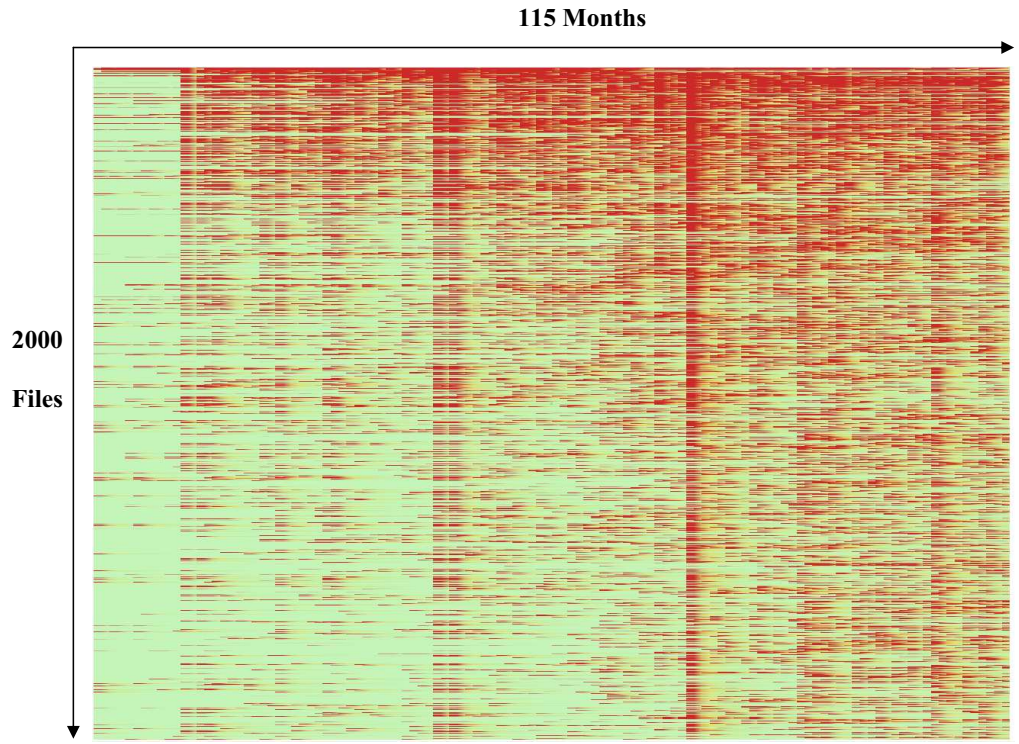

Figure 3.9: Spectrograph of FreeBSD for top 2000 files

of file activities for predicting the future. It is possible, by looking at past patterns, to learn about the future. We conjecture that this is another clue in showing the relationships in the data between past and future. In other sections, we will show that there could be relationships between these file activities and our prediction models' performance.

To investigate self-similarity in our data, we chose the OpenBSD system and plotted the file activity charts for different periods. This system has 88 months of development information, which includes 67149 events for 7065 files. Dividing these 88 months of development into four equal periods, we computed the frequency of changes for these four periods, for the total period of development, as well as for the total period excluding the first year of development. Figure 3.10 shows the frequency of changes for all these periods. The important thing to mention is that in order to compare these periods that have different number of files, we used the percentage of the number of files and percentage of the number of changes for each file. We observed that the file change frequencies are following the same pattern and they are similar to each other. The only exception is the first period which shows that the activities are distributed on more files.

In another experiment, we computed the change frequency for each year, from first through eighth. Again we observed that the frequency of changes are following the same pattern and they are similar to each other. This means that if a file has a lot of changes and bugs during a period of time, it will most probably have a lot of changes in the future too. Figure 3.11 shows these eight years activity in one plot.

Finally, in Figure 3.12, frequency of changes for all different studied periods have been plotted. As we can see here, except first period they are very close to each other.

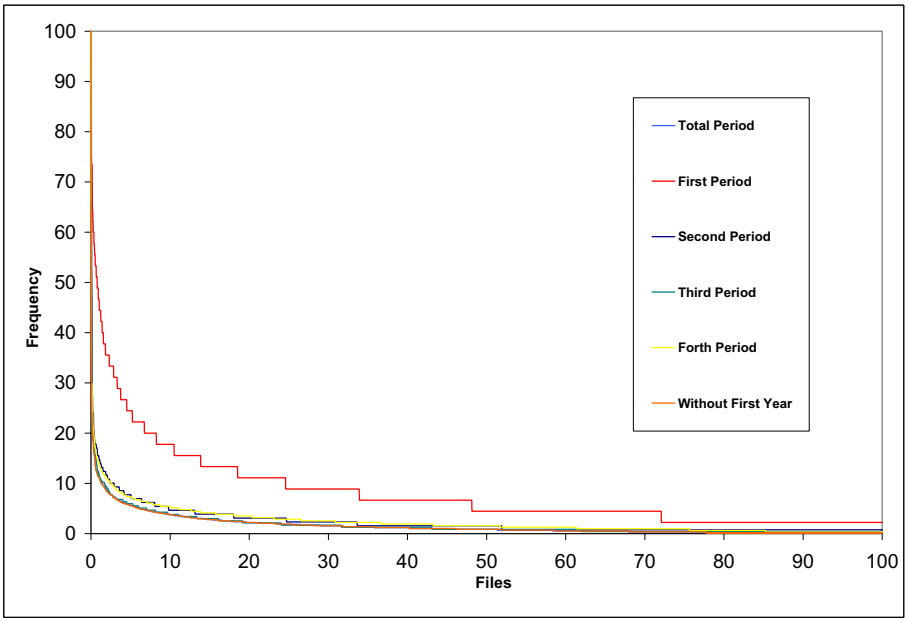

Figure 3.10: File activity chart for different development periods for OpenBSD

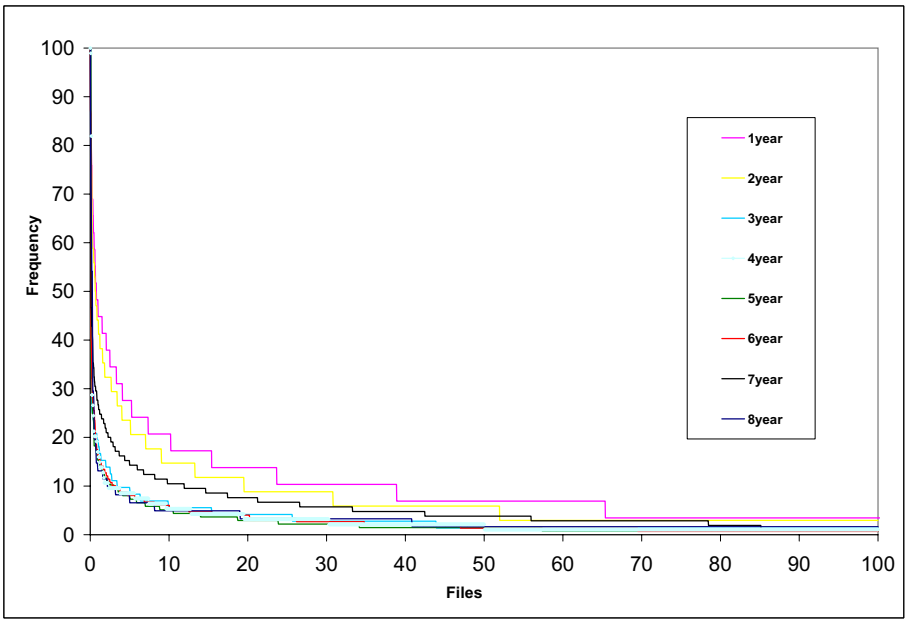

Figure 3.11: File activity chart for eight different years for OpenBSD

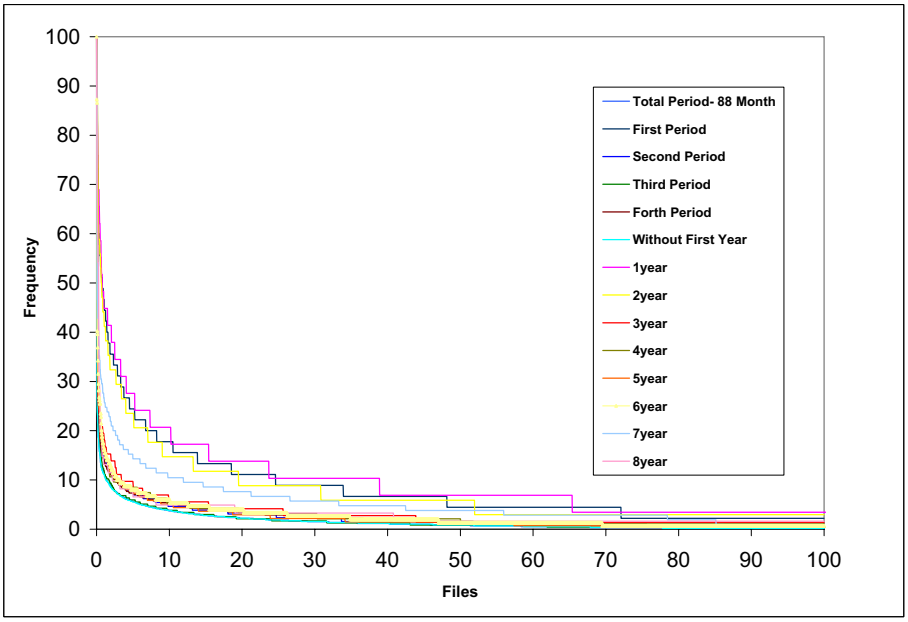

Figure 3.12: File activity chart for all periods of development for OpenBSD

We can conclude that after a while the file activities follow the same pattern. We call the described property as self-similarity. In our opinion, the property of self-similarity is very important, because it is a clear clue showing that despite all the chaos which exists during software development [1], still creating and changing the files follows a recognizable pattern.

3.6 Stationary and Non-Stationary Data Distribution

We will show in next sections in details the format of the extracted historical data from software repositories. We look at this extracted information as a sequence of events that we call it data. Stationary and non-stationary are well-known properties in statistics. A data is called stationary (distribution of the data), if the statistics

- mean, variance, ... - of any subset accurately describe the statistics of the entire data. Stationary data is so helpful that scientists assume it, even when a given data is not stationary.

For our developed techniques and models it was ideal to use a set of stationary. Our data are not stationary and the process which is producing these data is not stationary. Unfortunately, many interesting events in life are non-stationary and predictions and prejudices based on small subsets are notoriously unreliable. But even for non-stationary data, when there is a large set of information, it can be assumed that they are stationary. At any rate, this is the method regularly used. In next sections, we will talk about it in more details.

3.7 Summary

I this chapter, we looked at open source software systems and their role in our studies. We explained that how we can use their repositories such as CVS logs to extract useful information. We then described a set of experiments that were performed to analyze the extracted data. Investigating the extarcted data seemed necessary to us because they are used by our change prediction models. By investigating these data, we realized that thy have two common properties. First, the frequency of changes and bugs occurred to each file of the system during development process has a Zipf's distribution. Second, it seems that the frequency of changes and bugs occurred to each file in different periods of development are similar to each other and to the total period of development. We called this property self-similarity in data. In next chapter we will explain that how these data can be used to create change prediction models.

Chapter 4

Change Prediction Models

In previous chapters we discussed the necessity of looking for new approaches in both creating and evaluating the change prediction models. We recommended the use of software historical data that are generated during software development and can be extracted from software repositories. Since most of the reliability and change prediction models in past studies have been constructed and used for individual systems, it has not been practically investigated whether a prediction model based on one system can also predict faults and changes accurately in other systems [10]. Our expectation was that if we could build a model applicable to different types of systems based on the information that is generated during development process, e.g. CVS logs, it would be useful for software developers. In addition, it would be useful for software companies, because they would not need to invest manpower and time to gather data to construct a new model for every system, since they already have it as CVS logs. We also explained what these CVS logs are and how they can be obtained through mining the repositories of the software by existing techniques [8]. In this chapter we will introduce two new change prediction models

which can use the CVS logs to predict future bugs and changes in any arbitrary system. These models are generally in the form of probability models and at any point give the probability of the occurrence of a change or bug for all the files of a system. We intentionally have made our models in this form in order to be able to evaluate them later against our data, using the information theoretic model we will present in next chapter.

We will show how to obtain this sequence of events, which can be used to predict future comparable events. There are many files in the systems we studied, ranging from 1000 to 20000 files. We want to construct a probabilistic model of this process. In other words,we wish to define a probabilistic model that gives the result of the next element in the sequence. The first thing we need to do is to decide on the form of the model.

Suppose that we have the extracted CVS log information, consisting of changes and bugs that have occurred in different files of a system during development. Having this extracted data, we can obtain a sequence of events showing file changes to fix bugs or to add features. To convert the primary extracted data to the sequence of events, we show each entity in the CVS log as an event in the sequence. For example, an entity in CVS shows that a change has made on file f_f at time t_t. It will be shown as e_{tf}. Suppose that theses events e_{tf} are sorted according to the time of event, then we can assume that this data is generated by a stochastic process [1] with a sequence of random variables e_i that generates a sequence of events $e_1, ..., e_i, ..., e_n$. Further, suppose we also know that the possible value, i.e., the Domain D (sample space) for event e (considered as a random variable), is the set of existing files in the system. We denote the elements of this Domain as $f_1, f_2, ..., f_m$. Our goal is to

[1] A stochastic process is an indexed set of random variables, each of which is defined on the same probability space W and takes values on the same domain D.

define a model in form of probability distribution. This model gives the probability that the ith (i.e., next) event in the sequence will have a particular value (will be a particular file); in other words, it gives the probability distribution of random variable ei at any point. Ideally, this would be a conditional probability function of form $P(e_i|e_1, e_2, ..., e_{i-1})$. Computing this conditional probability is challenging for our case in that the number of events is a large number. As the size of the process grows, we would need, in general, a new probability function for each e_i, which would be a very complex model to estimate. Based on these assumptions we have proposed three probabilistic models such as Most Likely Estimation and Reflexive Exponential Decay Models which will be discussed in the remainder of this chapter. The Reflexive Exponential Decay (RED) Model relies on the fact that bugs can creep in when software is modified, and may not be detected until a long time after the modification is made. In the RED-Co-Change (REDCC) model, not only the number and time of changes-bugs that have happened to a file have an effect on the probability distribution of that file, but the changes-bugs which happen on the co-changes files also affect this file.

4.1 Most Likely Estimation

Our first change prediction model, Most Likely Estimation (MLE), simply uses the counts from the sequence to estimate the distribution. Suppose that we have a sequence of N events, $e_1, ..., e_N$, for estimating the probability of the next modification occurring on file f. We can count the number of modifications that have happened to file f in the sequence as:

$$P_{MLE}(e = f) = \frac{\text{Count}(f)}{N} \tag{4.1}$$

where $f \in D$ and N is the size of sequence.

The proportion of the number of times that a certain event occurs is called the relative frequency of the event [38]. MLE computes the relative frequency of each event to estimate the distribution of sequence. Empirically, if one performs a large number of identical trials, the relative frequency (for each file) tends to stabilize around some number (implicitly, this says that the process is stationary). That this number exists provides a basis for letting us calculate probability estimates. MLE has three convenient properties:

1. It is asymptotically consistent, which means that as the sequence size gets larger, the estimates tend to converge to the right values.

2. It is asymptotically efficient, which means that for large sequences, it produces the most precise estimates. It can be proven that the MLE assigns the highest probability to the overall sequence of any possible stationary probability distribution.

3. It is asymptotically unbiased, which means that for large samples one expects to get the right value on average.

Instead of definition 4.1, we actually use this probability distribution [39] called Lidstones estimation, for our events:

$$P_{Lindson-MLE}(e = f) = \frac{\text{Count}(f) + \lambda}{N + \lambda * |D|} \tag{4.2}$$

where $f \in D$ and N is the size of sequence and D is size of Domain and λ is Lindson parameter.

We are using this equation because equation 4.1 has two problems for which we should consider solutions if we want to use it. The first problem is that MLE assigns a zero probability to elements of domain that have not been observed in the sequence. This means it will assign a zero probability to any previously unseen element of sequence. The second problem is that it does not distinguish between different levels of certainty based on the amount of evidence we have seen.

For example, suppose we observe a sequence of events happening on different files of a system. We know there are ten different files in the system. So the sequence can contain these files only. If we observed 10 events have happened on a specific file out of 40, we have some confidence that the probability of the next modification happening on this file is 1/4. If we only observe four events, and one is happening on that file, we get the same probability estimate, but we should not be so confident. In other words, the MLE does not reflect any degree of uncertainty we might have based on the amount of evidence we have seen. One solution when we do have observations is to assign a small amount of probability λ to each possible observation at the start. This is called the Lidstones estimation [13]. We need to add the $\lambda * D$ to the denominator in order for the estimates to add to 1, making it a probability distribution. λ can have different values. As λ gets larger, the distribution becomes more uniform, and as it gets smaller, it approaches the MLE. When λ is 1 this technique is called Laplace estimation [39] and it means that every possible event has occurred at least once. If we never see an event of a type f in a corpus of size N and Domain size D, the probability estimate of an event of type f occurring will be $1/(N + |D|)$.

For the second problem, using the Laplace formula, our prior knowledge that there are D different types of events makes our estimate stay close to the uniform distribution. Consider the example above involving four events and forty events.

With ten events happening on one file out of forty events, the estimate is $(10 + 1)/(40 + 10) = 11/50 = 0.22$. So we have an estimate fairly close to the 0.25 the MLE estimate would give. With the four events, the estimate is $(1 + 1)/(4 + 10) = 2/14 = 1.4$. So although we saw a file modification 1/4 of the time in the four trials, our prior knowledge that there are ten files makes our estimate stay close to the uniform distribution value of 0.1. This is good because one observation out of four is not very strong evidence. To use MLE model for our data we used Laplace estimation which is common, putting $\lambda = 1$ in equation 4.2:

$$P_{Lindson-MLE}(e = f) = \frac{\text{Count}(f) + 1}{N + 1 * |D|} \tag{4.3}$$

where $f \in D$, N is the size of sequence and D is size of Domain.

We will will refer to Laplace-MLE as MLE in the remaining of this thesis.

4.2 Reflexive Exponential Decay Model (RED)

Our second change prediction model reflects the fact that bugs can creep in when software is modified, and may not be detected until a some time after the modification is made. We are given a sequence of events called $e_1, e_2, ..., e_n$, occurring respectively at monotonically increasing times $t_1, t_2, ..., t_n$. We assume that events probabilistically predict events- that is, bug fixes predict bug fixes. By analogy, yesterday's weather is a good predictor of today's weather.

We postulate that the predictive rate of bugs induced by any event decays exponentially. We call this model reflexive because each event in turn predicts more events. A particular event e happening at time t provokes a future probabilistic rate of more events. We assume that as a result of an event e_i occurring at time

t_i on the file j, the resulting predictive rate of $R_j(t)$ of more events is defined as follows:

$$R_j(t) = I * e^{k(t-t_i)} \qquad \text{where } k < 0 \text{ and } t > t_i \tag{4.4}$$

In equation 4.4, $e^{k(t-t_i)}$ is the Exponential function with parameter k. $t > t_i$ means that $R_j(t)$ is computed for the time after happening the event and $k < 0$ means that we choose the rate parameter (k) for our exponential model in a way that $R_j(t)$ decreases as time goes on.

Equation 4.4 can also be written as:

$$R_j(t) = I * \big(\frac{1}{2}\big)^{\frac{t-t_i}{h}} \qquad \text{where } k = -\frac{\ln 2}{h} \tag{4.5}$$

This formula means that if in the sequence of events, e_i happens at time t_i and e_i is a modification of file j, for all time $t > t_i$, the predictive effect for file j will remain with rate $R_j(t)$. In this equation, h is called half life (measured typically in months) and I is the "impact" of an event (measured in events / month). Half life is the time period in which the effect of $R_j(t)$ becomes half as time goes on and I is the initial amount for $R_j(t)$ at the time that event happens. The longer the half life, the longer it takes for the effects of a change to decay. Figure 4.1 shows the $R_j(t)$ for different half lives, with impact of 1 and $t_i = 0$.

Having the $R_j(t)$ for one event of file j, we can define the RED model as the summation of the effects of events that have happened to file j. This summation means we assume all events have an equivalent predictive effect. For making the RED probabilistic model at time t, suppose that in the sequence of events, we have l events, $e_{j1}, e_{j2}, .., e_{jl}$, which have happened to file j up this time (t). At time t,

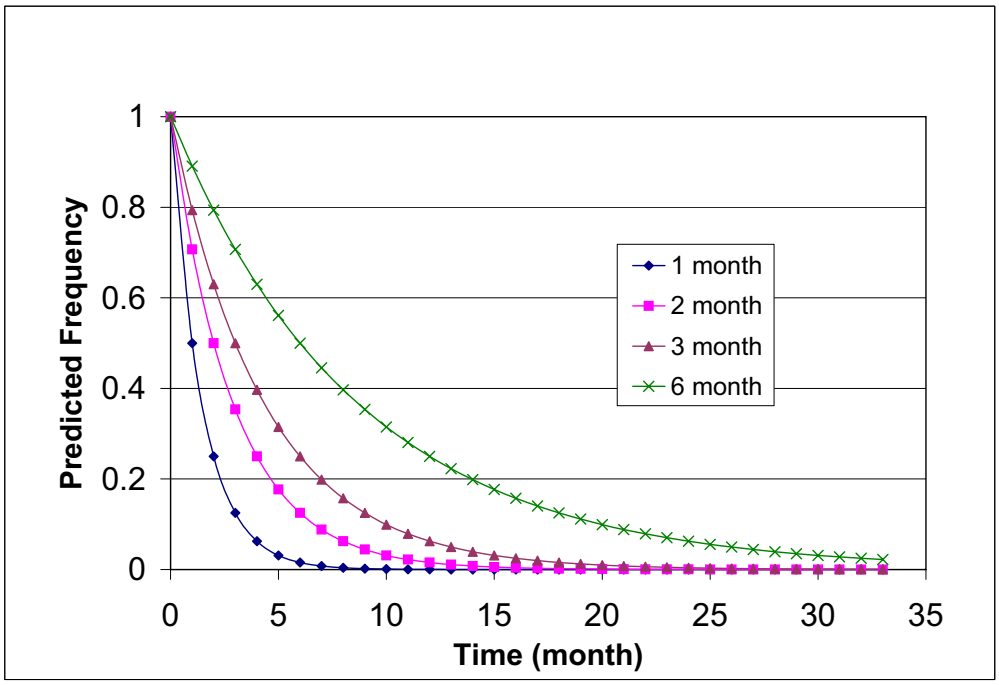

Figure 4.1: Exponential decay for different half lives.

$I * e^{k(t-t_{j1})}$ is the the remaining effect of happening e_{j1} at time t_{j1}, $I * e^{k(t-t_{j2})}$ is the the remaining effect of happening e_{j2} at time t_{j2} and so on. This means that if we want to have the rate of happening next modification on file j due to all these events, we will have:

$$RED_j(t) = I * e^{k(t-t_{j1})} + I * e^{k(t-t_{j2})} + I * e^{k(t-t_{j3})} + \cdots + I * e^{k(t-t_{jl})} \qquad t \geq t_{jl} \quad (4.6)$$

This is the final RED equation we will use to define the RED distribution model. As it can be seen, there are two parameters, h and I, in this model. We will show later in empirical study the way we estimate the best value for h which is 6 months.

Figure 4.2 shows how the effect of each event is added to the previous ones for

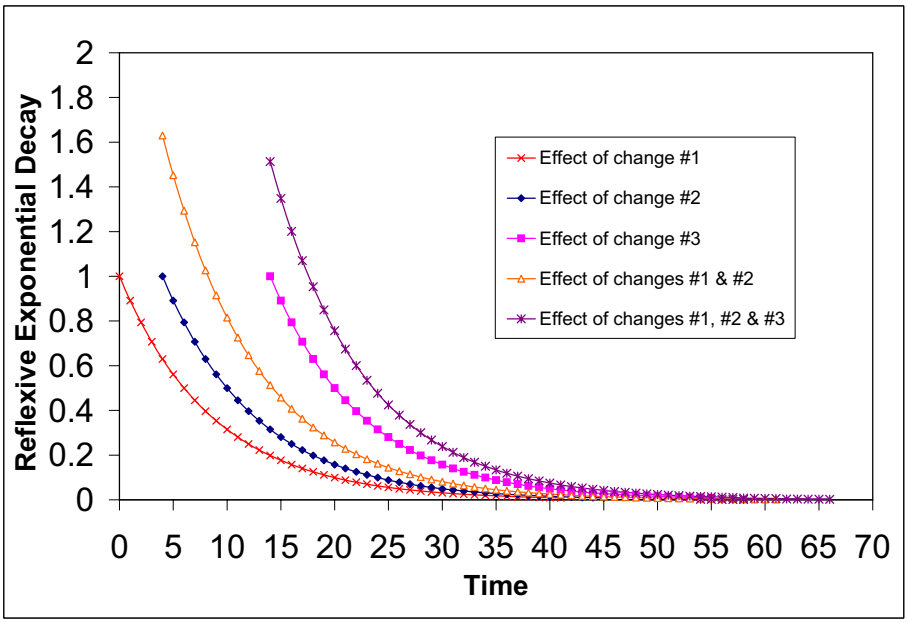

Figure 4.2: Reflexive exponential decay for a file.

a specific file f_j. In the figure, file f_j has been observed to change at times 0, 5 and 15; the individual exponentially decaying predictive effects of these three events are shown as the three lower curves. The cumulative effect of the first two of these (from times 0 and 5) is shown as another curve. Then the effect of all three of these is shown by yet another (the highest) curve.

4.2.1 RED Distribution Model

In a sequence of events: $e_1, e_2, ..., e_m$, having $RED_j(t)$ for all files $j = 1..|D|$, we can define the probabilistic distribution of RED at time t as follows:

$$RED_t(e = fj) = \frac{RED_j(t)}{\sum_{i=1..|D|} RED_i(t)} \quad for \quad t \geq t_m \tag{4.7}$$

To compute the probability of the i^{th} event occurring on file j at time t, first

the Reflexive Exponential rate for all files in the time t that event happens should be computed. The portion of Reflexive Exponential Rate for that specific file over the total Reflexive Exponential Rate for all files gives the probability model.

4.3 Reflexive Exponential Decay + Co-Change (REDCC)

In our third model, when each event occurs we update RED probability not only for the changed file but also for the co-change files. Co-change files are those which developers should modify at the same time to introduce new features or fix bugs, and they must ensure that when one of them is changed the other files in the software system are updated and consistent with these new changes. In other words, as developers change or add a file during software development and after that to introduce new features or fix bugs, they must propagate the effects of these changes to other files in the software system. This is called change propagation. Changing one of these files is a common source of introducing modifications to the system because of dependencies between different entities of software such as files, functions, variables.

There are several different approaches for clustering the files that change together during software development as well as many tools to assist a developer in determining other entities to change after a change in software [40] [41] [42] [43]. Among them we are interested in change propagation methods using historical code changes to software systems to track the effects of each change on other entities and the dependencies among them [42]. There are many approaches that use the history of changes to measure the performance of development techniques used by

developers working on large software systems. In past studies the history of source code changes has been used to study change propagation in software systems. Using historical code changes, finding these dependencies between changes is very challenging because source control systems treat source code as simple text and so record the changes at the level of lines changed, with an explicit comment by developers explaining the change [42]. After extracting the changes and bugs that occurred in different files of a system during development from the historical code changes, we have a sequence of events showing file changes to fix bugs or to add features. Having this extracted data in the form of sequence of events happening to the files of the system, we use the historical co-change definition to find the co-changed files in the system.

We will use the historical co-change definition [42] for finding the co-change files. Based on this definition, a historical co-change records that one entity changed at the same time as another entity. If file f_1 and f_2 changed together in previous change sets, then they are related via a historical co-change relation. In our approach, co-change files are those files which have changed in past at least N times within $\triangle$ days. With the occurrence of each event, the list of co-change files for each file should be updated.

For making the REDCC probabilistic model, with an event occurring in the sequence at time t on file f_x, we look at the past $N = 6$ months events including t to find the files that have changed with f_x at least $\triangle = 3$ times during this time. Suppose that in the sequence of events, we have $e_{j1}, e_{j2}, .., e_{jm}$ has occurred on file j or "on the co-change files of j up this time". The $RED - CoChange_j(t)$, which is the predictive rate of more events and means that, for a specific file j at time t, we have $REDCC_j(t)$ chance of the next modification occurring on file j, is defined as follows:

$$REDCC_j(t) = I * e^{k(t-t_{j1})} + I * e^{k(t-t_{j2})} + \cdots + I * e^{k(t-t_{jm})} \qquad t \geq t_{jm} \quad (4.8)$$

For example, if another event like e_{jm+1} happens on file j at time t_{jm+1}, and with this event happening and updating the co-change file lists we have f_p and f_q in f_j co-change list, then $I * e^{k(t-t_{jm+1})}$ should be added to $RED - CoChange_j(t)$, $RED - CoChange_p(t)$ and $RED - CoChange_q(t)$. In fact, in REDCC, we are giving higher probabilistic amount to co-change files.

4.3.1 REDCC Distribution Model

In the previous section we defined co-change files and the approach for finding and updating them. We also introduced $REDCC_j(t)$ which is the predictive rate of the next event happening on file j at time t. In this section we will compute the probability distribution of this model.

In a sequence of events: $e_0, e_1, e_2, ..., e_{i-1}$, having $RED - CoChange_j(t)$ for all files $j = 1..d$, we can compute the probabilistic distribution model $RED - CoChange(f_j)$ using $RED - CoChange_j(t)$ at time t as follows:

$$RED - CoChange(e_i = f_j) = \frac{RED - CoChange_j(t)}{\sum_{j=1..d} RED - CoChange_j(t)} \quad for \quad t \geq t_{i-1} \quad (4.9)$$

This means that to compute the probability of the ith event happening on file j at time t, first the RED-CoChange rate for all files in that time t that this event happens should be computed. The portion of that rate for that specific file over the

total Reflexive Exponential Rate for all files gives the probability model. Although we made a small modification in the RED model, as we will see in empirical study, it will make a significant improvement in the result.

4.3.2 Summary

In this chapter, we introduced three prediction models — MLE, RED and REDCC — that use a sequence of events to give the probability of next events happening on different files of a system. This sequence of events is extracted from CVS log information which includes changes and bugs that have occurred in different files of a system during development. We presented how each model is made from this sequence of events and our intuitions behind each model.

Chapter 5

Evaluation of Prediction Models

In the previous chapter we presented our approach for making prediction models using software change history. We developed three prediction models using a sequence of events. These sequences of events characterize the changes or bugs that have happened on the software during its development, where each event represents a bug or change. In this chapter we will evaluate these prediction models.

We are interested in evaluating these prediction models to measure which model perform best. As mentioned before, in previous studies there has not been a general evaluation of the assumptions on which the developed models are based, nor a general evaluation for the performance of the models on real data. Metrics and analysis methods used in these studies are various and are different case by case. Furthermore, most of results of these studies were validated on only a single software system [10].

Considering all these facts, in this chapter, we present two approaches for validating prediction models.

The first approach is a method for formulating a measure of goodness or fitness

of guessed probability m to the real probability p based on information theory concepts. We can use this approach for evaluating our prediction models because they are also in the form of probability distributions. Our approach uses entropy concepts to evaluate prediction models. Using this approach, it is possible to determine which model predicts better. Our goal is to compare these models (distributions) to see how good they are. By good we mean how close they are to the true distributions of the events. Equivalently, it means how well they predict the occurrence of the next event. One of the central problems we face in using probability models is obtaining the actual distributions of data. The true distributions are not known, and we must estimate them and then validate them against existing data. This approach is used in field of Natural Language Processing [13] where the sequence of words in language is called a corpus. They use information theory to find the distance between prediction models and the actual distribution of the corpus [44].

The second approach is Top Ten List [3]. In this approach, there is a list of Ten (or N) most likely files predicted to have changes. That list is modified over time as new changes happen on files. After each occurrence of a change or bug we look at the list (the list that our model determines) to see if the changed file is on our list. Based on the answer to this question we update the performance metrics of this approach which is called the Hit Rate. We choose this approach of evaluating our models in addition to our previously proposed approach for two reasons:

First, we want to compare our information theoretic approach with another approach like Top Ten List to see the advantages and performance that our approach has, although the metrics that they use and the numbers they give are quite different.

Second, we want to see the performance of our prediction models using another general approach in order to help validate our results.

5.1 Information Theoretic Approach

In this section, an approach will be presented by which we can evaluate the prediction models we created in the last chapter. Because this approach is based on information theory, it is called an information theoretic approach. Before introducing our information theoretic approach it is necessary to explain some related concepts. In the section on natural language processing, we will see how the language prediction models are evaluated against a sequence of words (corpus). In the methodology part, we will explain the approach and explain the steps to use this approach for evaluating the prediction models.

5.1.1 Introduction

Information theory is a branch of the mathematical theory of probability and mathematical statistics, that quantifies the concept of information. It provides a quantitative measure of how uncertain the outcome of a random experiment is. It is concerned with information entropy, communication systems, data transmission and rate distortion theory, cryptography, data compression, error correction, and related topics. It is generally believed that the modern discipline of information theory began with the publication of Shannon's article "The Mathematical Theory of Communication" in the Bell System Technical Journal [45]. Information theory techniques define the amount of information in a message. It can be viewed as a way to measure and reason about the complexity of messages. The theory measures the amount of uncertainty/entropy in a distribution. The following are some important concepts used in information theory techniques.

Entropy

Entropy is the fundamental measure of information theory. It is a very broad concept and it is used to measure the uncertainty of a random variable. It specifies the minimum number of bits required to encode the values of random variable x with probability function $p(x)$. Shannon Entropy [45], given probability $p(x)$, is defined as:

$$H(p) = -\sum p(x_i) \log p(x_i) \tag{5.1}$$

That is, the entropy of the random variable x is the sum, over all possible outcomes i of x, of the product of the probability of outcome i times the log of the inverse of the probability of i. It can also be applied to a general probability distribution, rather than a discrete-valued event. The higher the $H(p)$ is, the more bits we need for encoding the message. The logarithm in the formula is usually taken to base 2, and entropy is measured in bits. An important property of entropy is that it is maximized when all the messages in the message space are equiprobable. In this case if there are K possible outcome for the message, the entropy is computed as:

$$H(x) = \log K \tag{5.2}$$

where K is the number of possible outcomes i of x

Each of our models is simply a probability distribution. One of the simplest models is a uniform model. However, we know that not all the changes occur equally often. If a model captures more of the structure of the changes, then the entropy of the model should be lower. Though we can not use entropy as a measure of the

quality of our prediction models, it can be used to show the amount of uncertainties that exist in the probability distributions of these prediction models.

Cross Entropy

There is a related concept called Cross Entropy which allows us to compare two probability functions. The cross entropy between two probability distributions p and q over the random variable x, measures the overall difference between the two distributions and is defined as:

$$H(p, m) = -\sum p(x_i) \log m(x_i) \tag{5.3}$$

Cross entropy is always greater than or equal to the entropy: $H(p, m) >= H(p)$. For given p, the cross entropy is minimum when $p = m$. The closer the cross entropy $H(p, m)$ is to the entropy $H(p)$, the better m is an approximation of p.

Kullback-Leibler Divergence

Cross entropy is closely related to Kullback-Leibler (KL) divergence. The Kullback-Leibler divergence, or relative entropy, is a quantity which measures the difference between two probability distributions.

$$KL(p, q) = \sum p(x) \log(p(x)/q(x)) \tag{5.4}$$

It also can be written as:

$$KL(p, q) = \sum p(x)(\log p(x) - \log q(x)) = \sum p(x) \log p(x) - \sum p(x) \log q(x) \tag{5.5}$$

$$KL(p, q) = -H(p) + H(p, q) = H(p, q) - H(p) \tag{5.6}$$

As the cross-entropy is always greater than or equal to the entropy, this shows that the Kullback-Leibler divergence is nonnegative, and furthermore $KL(p, q)$ is zero if and only if $p = q$.

When comparing a distribution q against a fixed reference distribution p, cross entropy and KL divergence are essentially the same concept. In fact, they are identical up to an additive constant since p is fixed: both take on their minimal values when $p = q$, which is 0 for KL divergence, and $H(p)$ for cross entropy.

5.1.2 Natural Language Processing

Natural language processing (NLP) involves any aspect of processing language as humans speak and write it by computer, ranging from core topics in artificial intelligence such as building conversational robots, to commercial applications such as web search engines.

Shannon's measure of entropy came to be taken as a measure of the information contained in a message, as opposed to the portion of the message that is strictly determined (hence predictable) by inherent structures, such as redundancy in the structure of languages or the statistical properties of a language relating to the frequencies of occurrence of different letter or word pairs, triplets, etc.

5.1.3 Corpus Cross Entropy (CCE)

Before presenting the Corpus Cross Entropy definition, it is necessary to see Cross Entropy definition in NLP.

In NLP, Cross Entropy has been used to compare two language models presented by two probability functions: Given two probability functions p and m, we CE as:

$$H(p, m) = -\sum p(x_i) \log(m(x_i)) \tag{5.7}$$

where x_i is different values for random variable X

If one of these probability functions (say p) is the true probability distribution of language and the other one (m) is the probability distribution of the language model, then the CE will show how close the prediction model m is to p. The problem is that the true distribution (say p) is not available. Most of the time in many applications, when we are comparing two distributions, we actually don't know the target distribution (p), but have a sequence of data which the unknown target distribution is representing the true distribution of this data. So how can we tell whether one distribution is better than another given all we have is the sequence of data?

In NLP, a concept related to entropy definition which is called Corpus Cross Entropy (CCE) is used. Given a corpus of size N consisting of tokens $e_1, ..., e_N$, the corpus cross entropy of a probability function m is defined as

$$Hc(m) = -(1/N) \sum \log\ m(ei) \qquad i = 1...N \tag{5.8}$$

In this equation the summation is over the tokens in the corpora, in contrast with summing over different values of the tokens in the Domain.

It is straightforward to prove that as the size of corpus goes to infinity, the corpus cross entropy becomes the cross entropy measure for the true distribution that generated the corpus. For proving this equivalence it is necessary to assume

that the corpus has a stationary distribution. It depends on the fact that the Maximum Likelihood Estimator goes to the true probability distribution as the size of corpus goes to infinity. Thus as the corpus grows in size, the MLE starts to generate good approximations of the true distribution.

Based on [13] even though this is a wrong assumption, it is common to use CCE for Language processing. "A stationary process is one that does not change over time. This is clearly wrong for languages: And so, it is not exactly correct to use this result to allow the calculation of a value for cross entropy for language applications. Nevertheless, for a snapshot of text from a certain period (such as one year's newswire), we can assume that the language is near enough to unchanging, and so this is an acceptable approximation to truth. At any rate, this is the method regularly used."

In the next section, we will explain why CCE works for evaluating our data although our data is not stationary.

With the assumption stationary data it is easy to prove that CCE is equivalent with CE, when the size of corpus goes to infinity.

The approach is to rewrite the definition of corpus cross entropy in terms of MLE estimates.

Assume that the Domain of the corpus is $D = d_1, ..., d_t$. Suppose that each value of Domain d_j occurs n_j times in the corpus. Thus the MLE estimate for D_j occurring is n_j/N. According to CCE definition:

$$H_c(m) = -(1/N) \sum \log m(ei) \qquad i = 1...N \tag{5.9}$$

Instead of summing over all the corpus values e_i, we gather all the values of the

same type together and sum over the Domain items $(d_1, ..., d_t)$:

$$H_c(m) = -(1/N) \sum \log(n_j * m(d_j)) \qquad j = 1...t \tag{5.10}$$

Which is equal to:

$$H_c(m) = - \sum \log(n_j/N) * m(d_j) \qquad j = 1...t \tag{5.11}$$

As we mentioned before: $p_{MLE}(d_j) = n_j/N$

$$H_c(m) = - \sum \log p_{MLE}(d_j) * m(d_j) = H_c(p_{MLE}, m) \qquad j = 1...t \tag{5.12}$$

This is the cross entropy formula using p_{MLE} as the target probability function. If we have a Stationary process, the MLE estimate approaches the true distribution as the corpus size grows to infinity. So we have:

$$H_c(m) = - \sum \log p(d_j) * m(d_j) = H_c(p, m) \qquad j = 1...t \tag{5.13}$$

It means that Corpus Cross Entropy becomes the cross entropy of m with the true probability distribution p underlying the corpus.

5.1.4 Information Theoretic Approach

In this section we present the information theoretic approach to quantify the goodness or fitness of a guessed probability m (m is a prediction model) compared to the

actual probability p. Our approach uses entropy concepts to evaluate prediction models.

Suppose we have a sequence of events which have occurred on the set of files that make up a software system during development process until present time. We are given this sequence of events called $e_1, e_2, ..., e_n$, occurring respectively at monotonically increasing times $t_1, t_2, ..., t_n$. We assume that events probabilistically predict events, e.g., bug fixes predict bug fixes. In addition to this sequence of events, we are also given a prediction model which may be made using the preceding information in the sequence. Our goal is to see how close this model is to the underlying distribution of this events.

The other problem is that given this sequence and two models we can say which model is "better". We believe that the model that is closer to the underlying distribution of events is the better model. It can be concluded that with solving the first problem, the second one is also solved.

For finding the closeness of two probability distributions p and m, cross entropy introduced in last section can be used.

$$H(p, m) = -\sum p(x_i) \log m(x_i) \tag{5.14}$$

The cross entropy is minimum when p and m are identical. The closer the cross entropy is to the entropy, the better m is an approximation of p. So we can conclude that if we have two models m_1 and m_2, if $H(p, m_1) < H(p, m_2)$ then m_1 is a closer distribution to p. This unique approach would require that we know p, the actual distribution of data, which unfortunately we do not. One of the central problems we face in using probability models is obtaining the actual distribution p of data. The true distributions are not known, and we must estimate them (probabilistic

models) and then validate them against existing data. Here there is a paradox: if we had p in advance, we would not need to make any model for estimating p. This problem is very similar to finding and evaluating probabilities associated with a corpus in NLP as described in previous section.

So this problem can be also solved with using Corpus Cross Entropy. We could use the "Sequence Cross Entropy" phase instead but for avoiding confusion, we will stay with "Corpus Cross Entropy" phase.

Given a sequence with size N consisting of events $e_1, ..., e_N$, the Sequence Cross Entropy of a probability function m is defined as follows:

$$H_{CCE}(m) = -(1/N) \sum \log m(e_i) \qquad i = 1...N \tag{5.15}$$

The sequence cross entropy approaches the cross entropy of the model with true distribution as N approaches infinity.

Here we prove that even though our data and distributions are not stationary, that is valid.

5.1.5 Corpus Cross Entropy for Non-Stationary Distributions

We start by considering a stream of events $e_1, e_2, e_3, ..., e_N$. Each event is a value, which is from the set $f_1, f_2, f_3, ..., f_k$.

For example, if event e_7 has value f_{16} this means: f_{16} was modified as a result of event e_7.

Each event e_j has a probability jp_i that the value of e_j will be f_i.

If all events have the same probability, then we say the events have stationary probability p, then:

For all j, k and $i : jp_i = kp_i, \quad p$ is stationary

We have a predictive model m that predicts the probability jm_i that the value of event j will be i. If all events have the same probability distribution, then we say the model m is stationary.

With these assumptions, corpus cross entropy for corpus (CCE) of size N is defined as:

$$H_{CCE}(m) = -(1/N)\sum \log jm(e_i) \qquad j = 1...N \tag{5.16}$$

That is, $H_{CCE}(m)$ is the average value of $\log jm(e_i)$,- the average of the log of the model's predicted probability of the values of the events. This can be interpreted as follows:

$H_{CCE}(m)$ gives the average number of bits to encode a message giving the value of each event e_i, where the number of bits is determined on the assumption that the probability of that value is given by $jm(e_i)$.

If p and m are stationary then CCE can be proven to be converge to cross entropy H_c as N goes to infinity.

What is the interpretation of $H_{CCE}(m)$ when both p and m are non-stationary?

In the case of non-stationary p and m, the following interpretation remains valid. For event ej, if we use jm_i as our estimate of the probability that e_j is f_i, then to encode this message takes $\log(jm(e_i))$ bits. In the ideal case, when our prediction probability jm exactly equals the actual probability jp, we get the minimum or best number of bits.

Now we consider the case the of whole corpus of events. The formula for $H_{CCE}(m)$ can still be used to compute a result. This result will be the average of $\log(jm(e_i))$. The interpretation is that this is the average number of bits for encoding each of the events, where we base our encoding for event e_j on probability j_m. The minimum of this average occurs when for all $j, jm = jp$, i.e., when the estimated probability for each event exactly equals the true probability for that event.

It seems fair to conclude that, when p and m are non-stationary (and N is large), $H_{CCE}(m)$ provides a reasonable measure of how good m is as an approximation of P. This amount is minimum when $m = p$ and gets worse as m diverges from p.

This means that for two models m_1 and m_2 we can compute $H_{CCE}E(m_1)$ and $H_{CCE}(m_2)$ although we do not know the true distribution p. If $H_{CCE}(m_1) < H_{CCE}(m_2)$ then m_1 is a closer distribution to true distribution and hence is a better model.

We can compute their difference: $Kc(m_1, m_2) = H_{CCE}(m_1) - H_{CCE}(m_2)$ This difference has the following interpretation: It measures the number of additional bits that m_1 requires on average, compared to m_2, to encode each event's value. This neatly gives a quantitative measure with a nice interpretation for comparing predicted probabilities, namely: $Kc(m_1, m_2)$ = "wasted bits" on average due to a less accurate prediction or more succinctly: m_1 is $Kc(m_1, m_2)$ bits more predictive than m_2.

5.2 Top Ten (N) List Evaluation

Here we will introduce the Top Ten (N) List approach which measure which model predicts more accurately. In [3] Hassan et al. presented this approach by which they have two goals:

- Predicting the files that are most likely to contain a fault in the near future, can help managers to focus their limited resources and maximize their resource usage.
- Providing a tool for researchers who have developed heuristics and fault prediction to validate them.

This approach consists of three components:

- A prediction model.

  At any time during the development process, based on the historical changes and bugs extracted from repositories, the prediction model/heuristic assigns a number to the files, showing the potential of the next change or bug happening in each file. This number can be a probability as we saw in our prediction models.

- A list of files most likely to have changes.

  A list will contain the files most likely to have future changes or bugs. That list is modified over time as new files are modified within a subsystem. The size of this list can be changed based on the amount of resources which a manager has to work on them. For example when the size of list is ten, top ten list is a list of the top ten files which are most susceptible to have a fault

appear in them in the near future. This list is similar to a resource cache because a cache is used to store a limited number of resources for fast and efficient access.

- A performance metric.

  There is a performance metric which is called Hit Ratio. Hit Ratio, which is a measure of the performance of a caching system, is the number of times a referenced resource is in the cache. For the Top N list approach, this means that the number of times the files in the list had faults in them as predicted by the model used to build the list. Obviously, the higher the Hit Ratio, the better the prediction power of the prediction model.

That list is modified over time as new changes happen on files. After each occurrence of a change/bug we look at the list (the list that our model determines) to see if we have the changed file on our list or not. Based on the answer to this question, we update the Hit Ratio of that model.

5.3 Summary

In this chapter we presented two different approaches for validating prediction models which use a sequence of events to give the probability of the next events happening on different files of a system.

In the first approach, entropy concepts are used to evaluate prediction models. Using this approach, it is possible to determine which model predicts better and how much better it is.

In the second approach, Top Ten List [3], there is a list of Ten (or N) files predicted to be most likely to have changes. This list is updated based on the

prediction models and the prediction model which gives the highest Hit Ratio to this list is considered as a better model.

Chapter 6

Empirical Studies

In Chapter 5 we presented two approaches that can be used to evaluate the prediction models proposed in Chapter 4. In this Chapter we evaluate our proposed prediction models empirically by applying two approaches to the CVS repositories of six large open source systems. As we mentioned before, we use the provided extracted data of these CVS logs by Ahmed Hassan [8]. The studied systems have different sizes, numbers of developers, and project durations. Table 6.1 summarizes the details of the software systems we studied.

We applied these approaches on three proposed models, namely MLE, RED and REDCC, for all the studied systems. To evaluate the performance of our models, first we use the Top Ten (N) List [3] approach. We then apply the information theoretic approach to compare these probabilistic prediction models. We will then compare the results of each approach.

Application Name	Start Date	Application Type	Programming Language	Total Files
OpenBSD	Oct 1995	OS	C	7065
FreeBSD	June 1993	OS	C	5272
KDE	April 1997	Windowing	C++	4063
Koffice	April 1998	Productivity	C++	6312
NetBSD	March 1993	OS	C	11760
Postgres	July 1996	DBMS	C	1468

Table 6.1: Information about studied systems.

6.1 Top Ten List Approach

As we introduced this approach in the previous chapter, in this approach the model predicts a list of the 10 files (more generally, a list of N files) that are most likely to be changed next. A new list is generated for each new event. Given a predicted distribution m for the next event, we create the corresponding Top Ten List for that upcoming event by picking the ten (or N) files with the highest probability according to m. With the occurrence of each event, there is a change to a file, f_i. If f_i was in the list, the hit rate will be increased by 1. At the end of sequence, the Hit Ratio records how many times the files that were in the list had faults in them as predicted by the model. at any time Hit Ratio is the percentage of the Hit Rate to the length of the sequence. Models with higher Hit Ratios are considered to be better models.

6.1.1 Performance of TTL as Time Varies

In this section, we plot the Hit Ratio versus the number of changes and bugs over the five year period for three different sizes of the list. For some of these systems, five years covers the total development period. It will show the Hit Ratio of the

three proposed models for all the studied systems during the first five years of development. Using these plots we can also see how the Hit Ratio changes as time goes on. In the next section we will show the performance of proposed prediction models depending on Top N List size and how the increasing the size of the list would improve the performance of the models.

For plotting the Figures, with the occurrence of each event, the Hit Ratio of the top list is computed by the time of occurrence. Then it is plotted by the number of events that have occurred by that time and the amount of the Hit Ratio. Figures 6.1 to 6.6 show the performance for the three proposed models for different systems. In the figures there are the performance of TTL when the list size is equal to 10%, 30% and 60% of the files and for each size performance of three models are depicted.

For example, for FreeBSD, once there are 6000 changes and bugs, the Hit Ratio for the models are: MLE (47%), RED (50%) and REDCC (52%) when the list size is 10%. It is MLE (80%), RED (85%) and REDCC (90%) when the list size is 30% and MLE (98%), RED (99%) and REDCC (100%) when the list size is 60% of the files. Looking at the figure, we note that the RED and REDCC model that use the Exponential Decay have the best performance in all the studied systems. In contrast, the other model (MLE) which only counts the frequency of modifications in a file does not perform as well.

As we will discuss in more detail in the next section, the increase in the size of the list improves the performance of the models. But what is important for us in this section is the behavior of the performance of the models over time for different sizes. We can see that the bigger sizes have better performance but they usually follow the same pattern over time.

In all of these figures, we can see that there is a start up chaos. We have

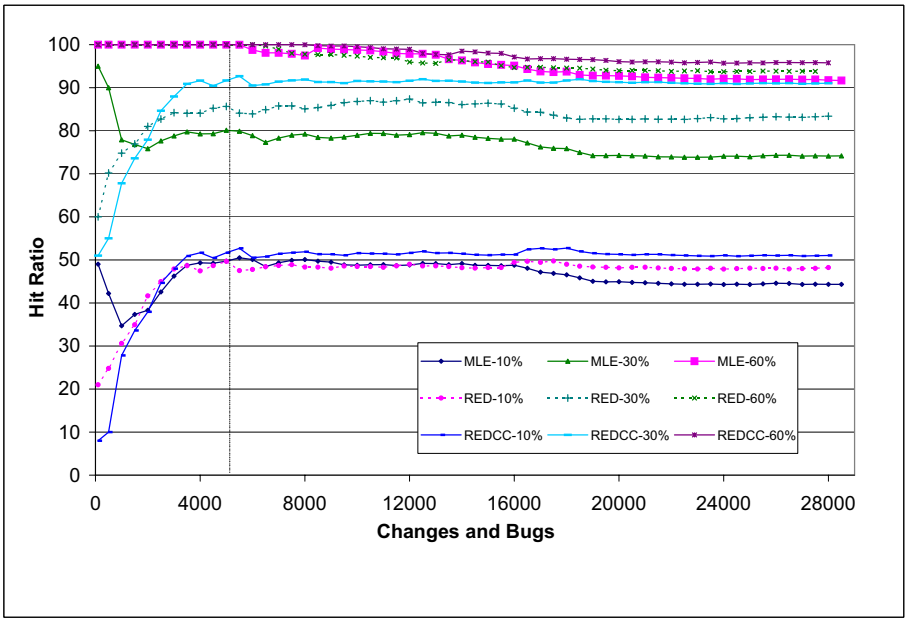

Figure 6.1: FreeBSD Hit Rate for the 3 proposed models.

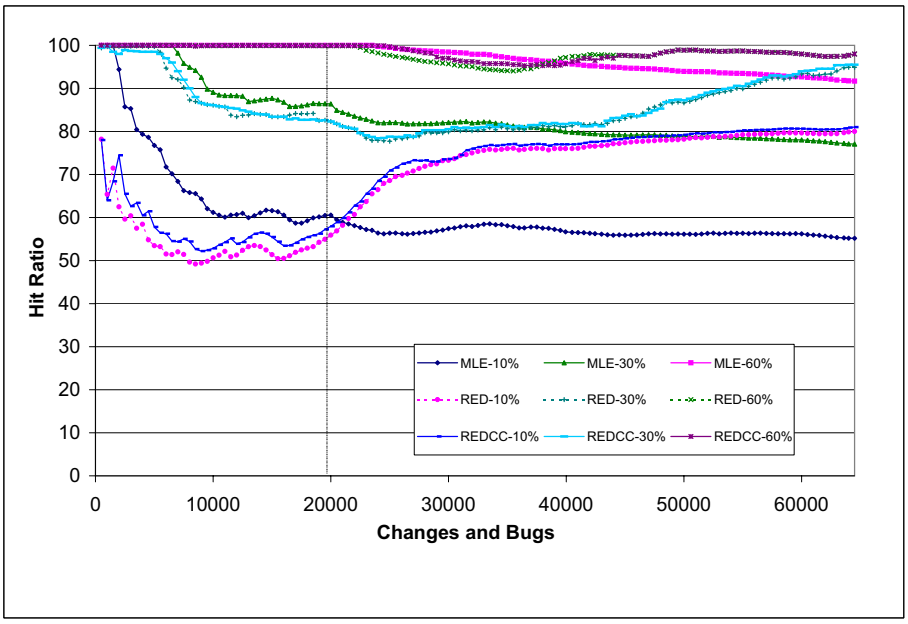

Figure 6.2: KOffice Hit Rate for the 3 proposed models.

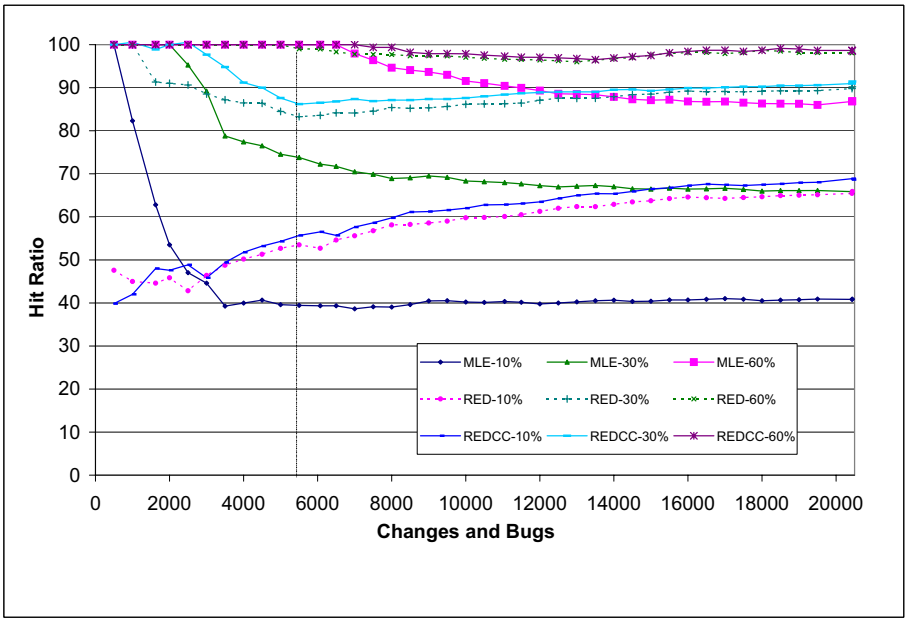

Figure 6.3: OpenBSD Hit Rate for the 3 proposed models.

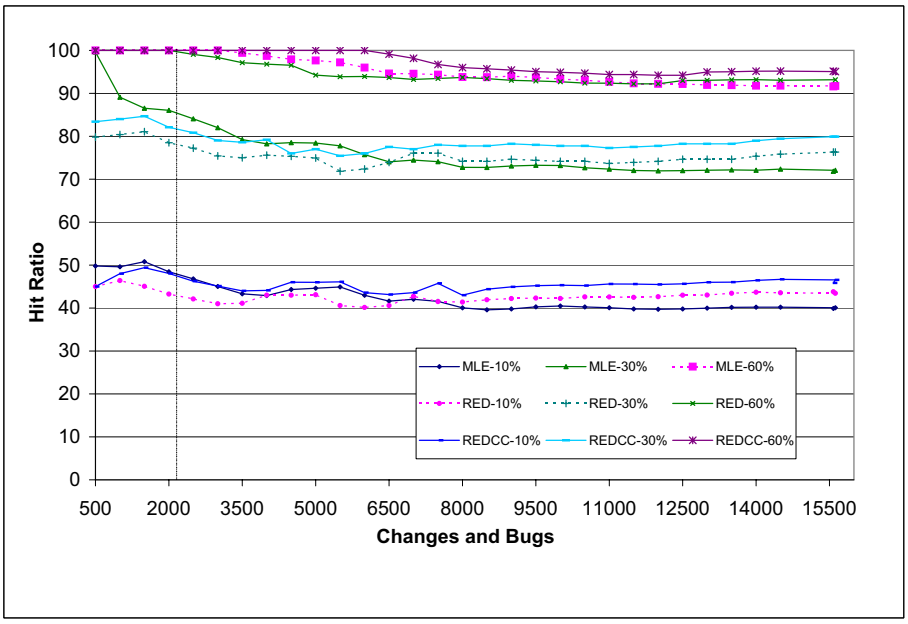

Figure 6.4: Postgres Hit Rate for the 3 proposed models.

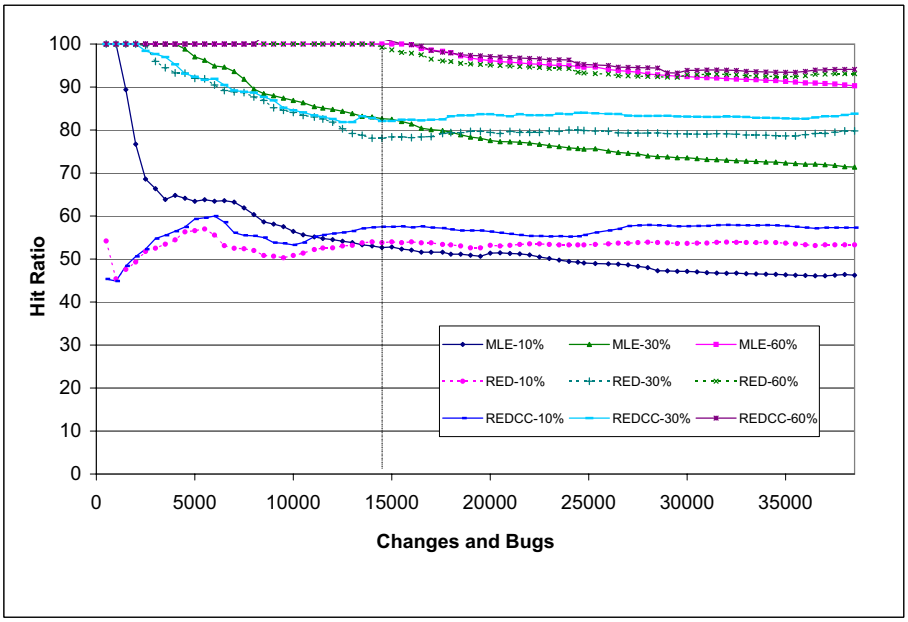

Figure 6.5: NetBSD Hit Rate for the 3 proposed models.

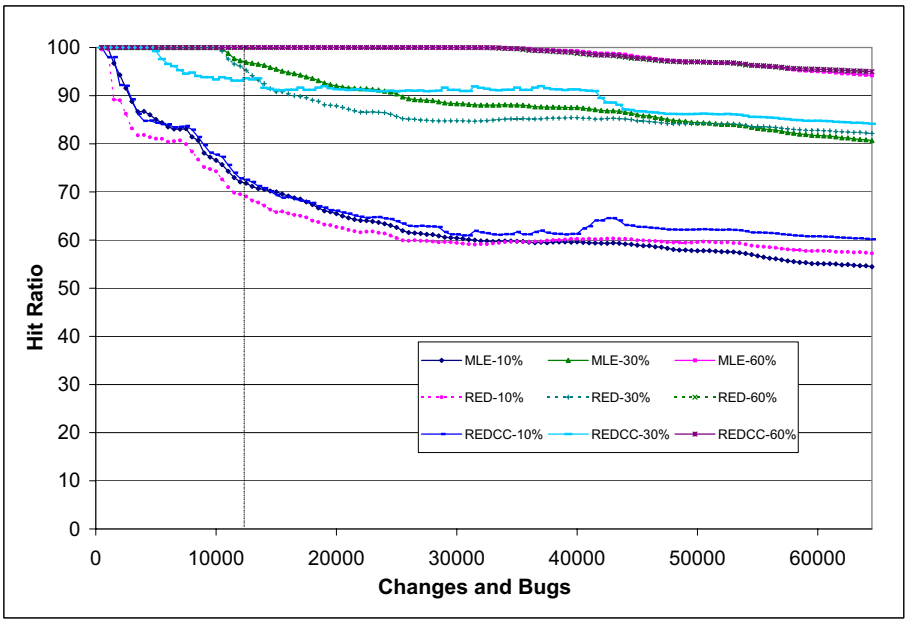

Figure 6.6: KDE Hit Rate for the 3 proposed models.

separated with a vertical line that part of the pictures where their history that is apparently not reliable because the models are calibrating themselves. In this part for all of the models the performance of the models at the beginning is higher due to the fact that at the beginning there are not as many files in the history and hence the list size is big comparing to the number of files. It is especially visible when the size of the list is bigger. The larger size of list generally improves the Hit Ratio.

The general behavior in all of these figures is that MLE has a higher performance at the beginning of the process, but it declines as time progresses. MLE also needs a large number changes and bugs to calibrate itself. In contrast, both RED and REDCC increase over time due the fact that they keep the effects of the previous changes and bugs.

RED's half life

As we have already discussed in the Prediction Models Chapter, the RED model is defined as follows:

$$R_j(t) = I * (\frac{1}{2})^{\frac{t-t_i}{h}} \qquad \text{where } k = -\frac{\ln 2}{h} \tag{6.1}$$

$R_j(t)$ means that if in the sequence of events, e_i happens at time t_i and e_i is a modification on file j, for all time $t > t_i$, this effect for file j will remain with rate $R_j(t)$. In this equation, h is called the *half life* (measured typically in months) and I is the *impact* of an event (measured in events / month).

Half life is the time period in which the effect of $R_j(t)$ becomes half as time goes on and I is the initial amount for $R_j(t)$ at the time that event happens. The longer half life is, the longer it takes fort the effects of a change disappear.

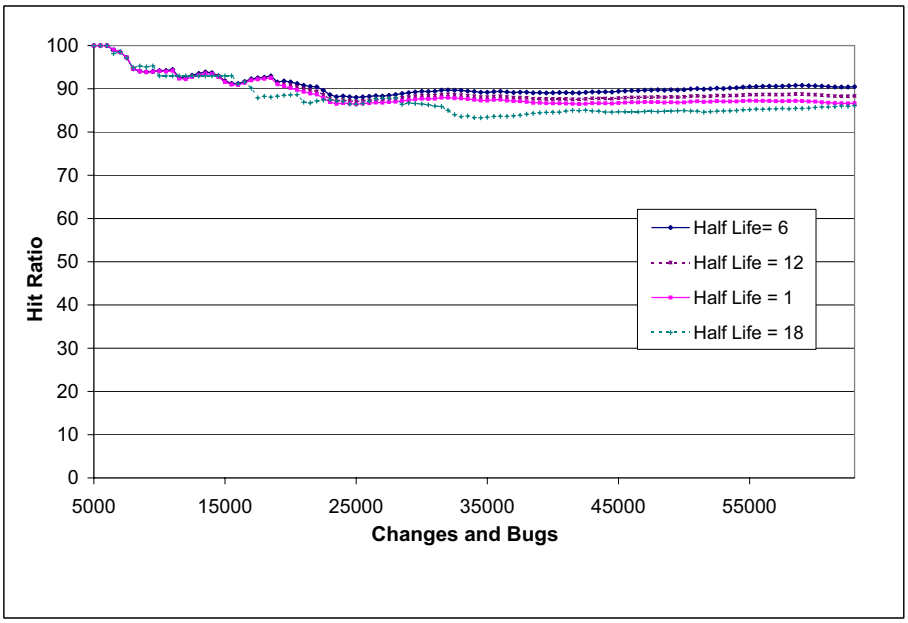

Figure 6.7: KOffice- Hit Rate for RED Model with Different Half Lives.

In our experiments for using RED model, a question is what is the best amount for half life and impact parameters. We had a separate group of experiments which ran on some of the studied systems to study the impact of these parameters on the performance of RED model.

Figure 6.7 shows the performance of TTL for KOffice for different half lives during 5 years of development. We applied it when the list size is 20% and half life is 1, 6, 12 and 18 months. It can be seen that when the half life is 6 it has a better performance. Based on these experiments we chose half life= 6 months and 1 for impact parameter for RED model in our other experiments. In fact, the initial value of impact parameter is not important in our experiments because it is a constant amount which is multiplied to the rest of the formula and in comparing the different values does not have any effect.

6.1.2 Performance of TTL as Size Varies

In the previous section, we presented the performance of the Top Ten list approach using various models for three different sizes of list. Here we want to see how the performance of proposed prediction models is dependent on Top List size and how the increasing the size of the list might improve the performance of the models.

Figures 6.8 to 6.13 show the performance of proposed prediction models depending on Top List size. As it can be seen in the figures, as the size of list increases, we have a higher hit ratio and the performance of the models increases. Using REDCC model, when we use 20 percent of total files in the system, the hit ratio is almost 80 percent.

We observed that again REDCC model has the highest Hit Ratio in all the studied system. RED and REDCC have very close Hit Ratio in all studied systems, but the difference between MLE and RED appears to be significant in many of them.

6.2 Information Theoretic Approach

In the previous chapter, we introduced some concepts of information theory and explained how an information theoretic approach can be used to quantify the goodness or fitness of a guessed probability m (m is a prediction model) compared to the actual probability p. Our approach uses entropy concepts to evaluate prediction models.

In this section we will apply this approach on the extracted information of our studied systems presented in Tables 3.1 and 3.2.

As can be seen in Table 3.2, we have a number of events that have occurred on the set of files that make up a software system during development process

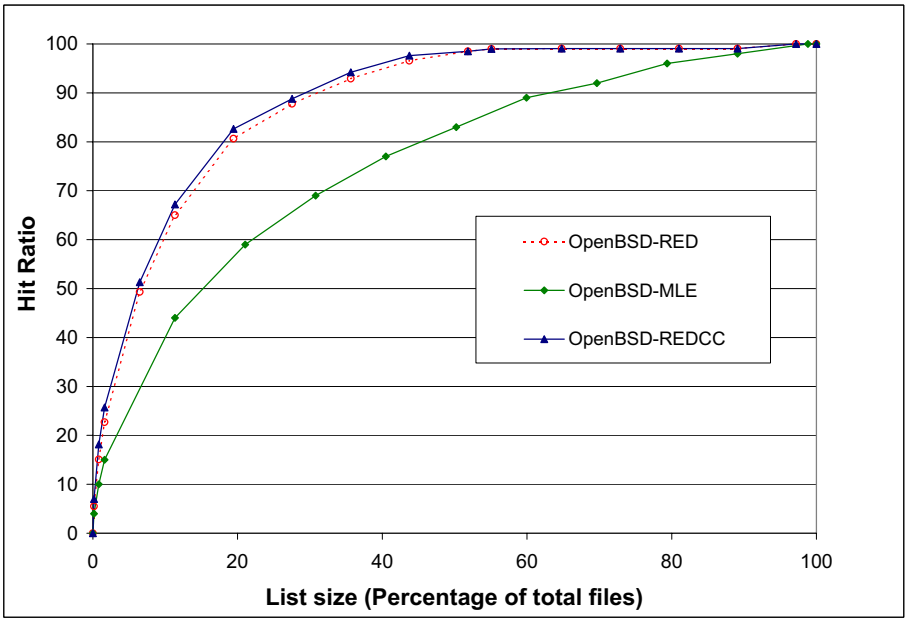

Figure 6.8: OpenBSD- Hit rate growth due to increasing list size for different models
.

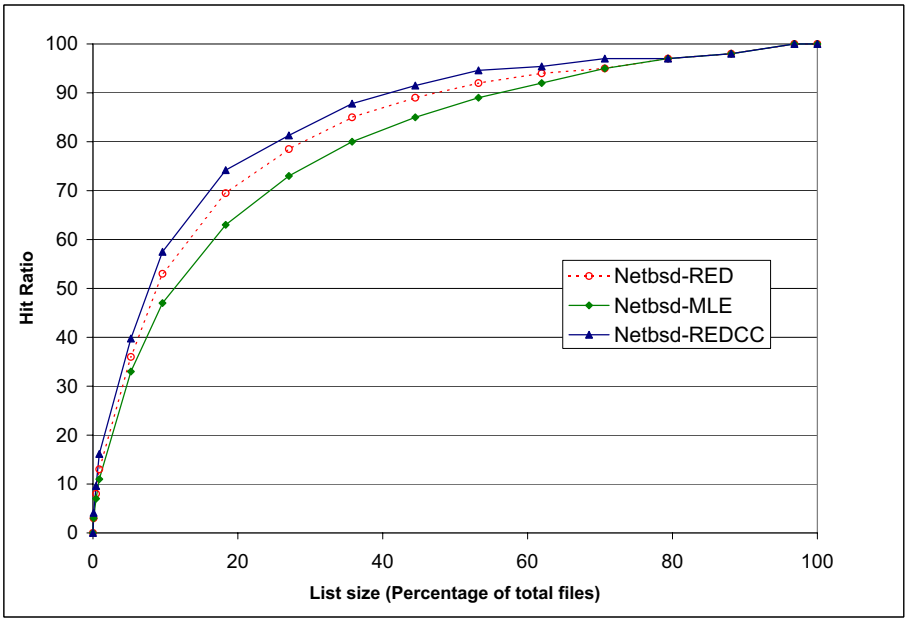

Figure 6.9: NetBSD- Hit rate growth due to increasing list size for different models.

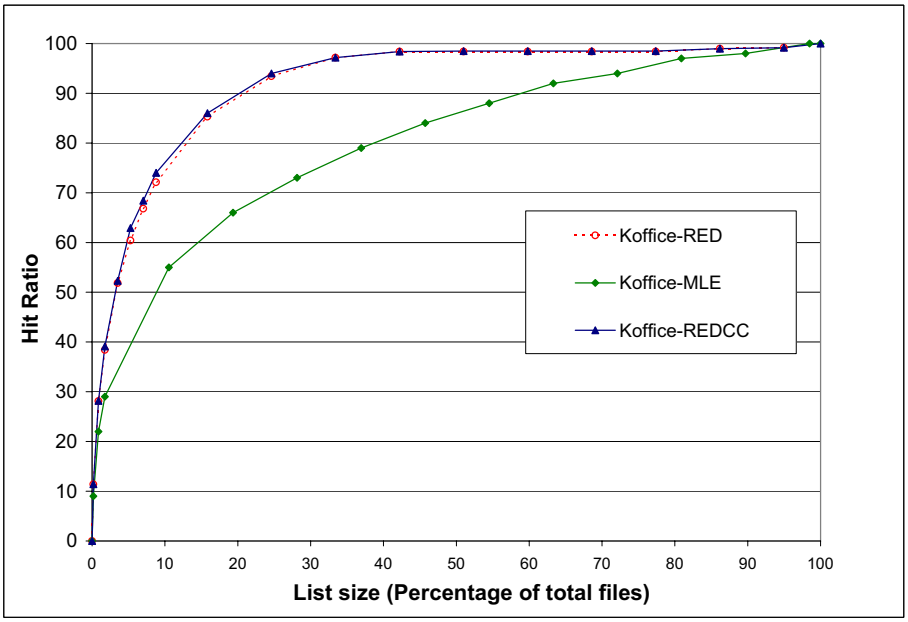

Figure 6.10: KOffice- Hit rate growth due to increasing list size for different models.

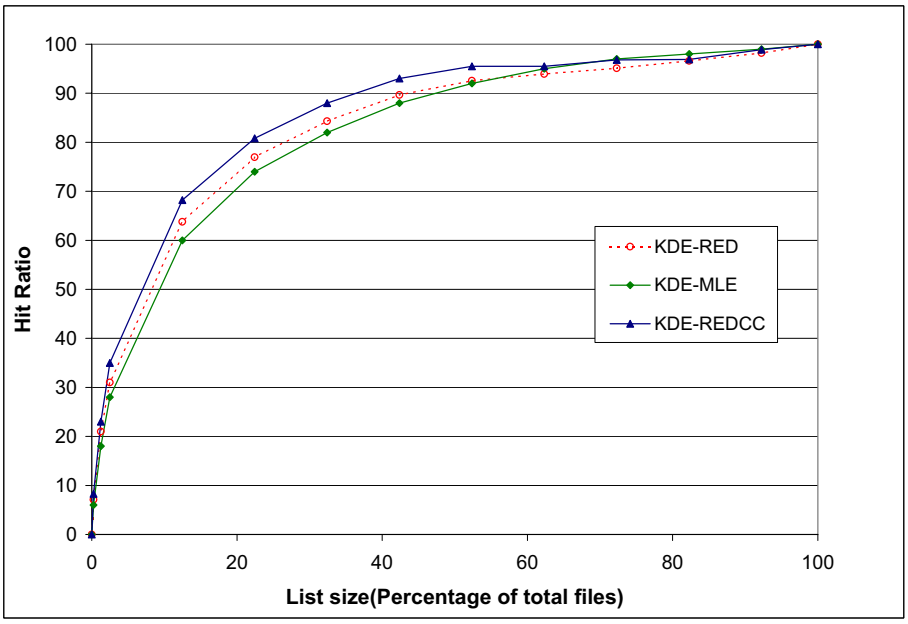

Figure 6.11: KDE- Hit rate growth due to increasing list size for different models.

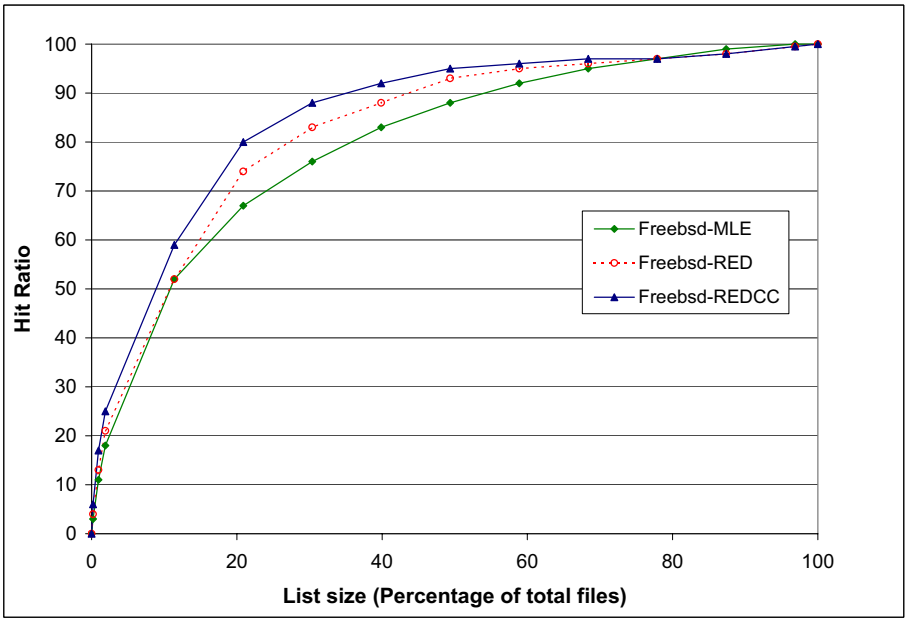

Figure 6.12: FreeBSD- Hit rate growth due to increasing list size for different models.

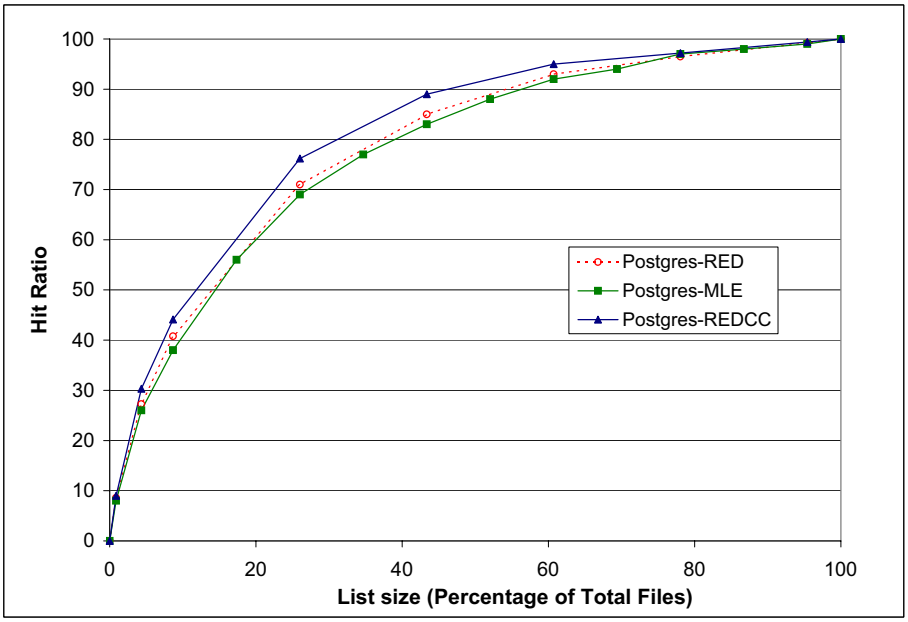

Figure 6.13: Pstgres- Hit rate growth due to increasing list size for different models.

until present time. For all experiments, we have the sequence of events $e_1, e_2, ..., e_n$, occurring respectively at monotonically increasing times $t_1, t_2, ..., t_n$ and we suppose that they have an underlying distribution which may be non-stationary.

6.2.1 Comparing the Prediction Models using Corpus Cross Entropy

Based on our proposed approach, when we compare two probability distributions, if Corpus Cross Entropy of one probability distribution model with a sequence of events $e_1, e_2, \ldots, e_N$ has a lower CCE, this model is closer to the underlying distribution of the data and hence is a better model.

In the previous sections we have explained how we build our models using the data. We have three models: MLE, RED and REDCC.

To apply this approach on the sequence, there are two important parameters:

- The size of sequence used to make the models for each experiments
- The size of the corpus which is used in Corpus Cross Entropy, for evaluating the model. Based on the theoretical part, when the size of corpus is larger, the approach can be more accurate.

Regarding the size of corpus that is used in Corpus Cross Entropy for evaluating the models we expect that the longer the corpus is the more accurate the results will be. This means that as the corpus grows, the results become more reliable. In this section we will show the effects of corpus size on the amount of corpus cross entropy for each model.

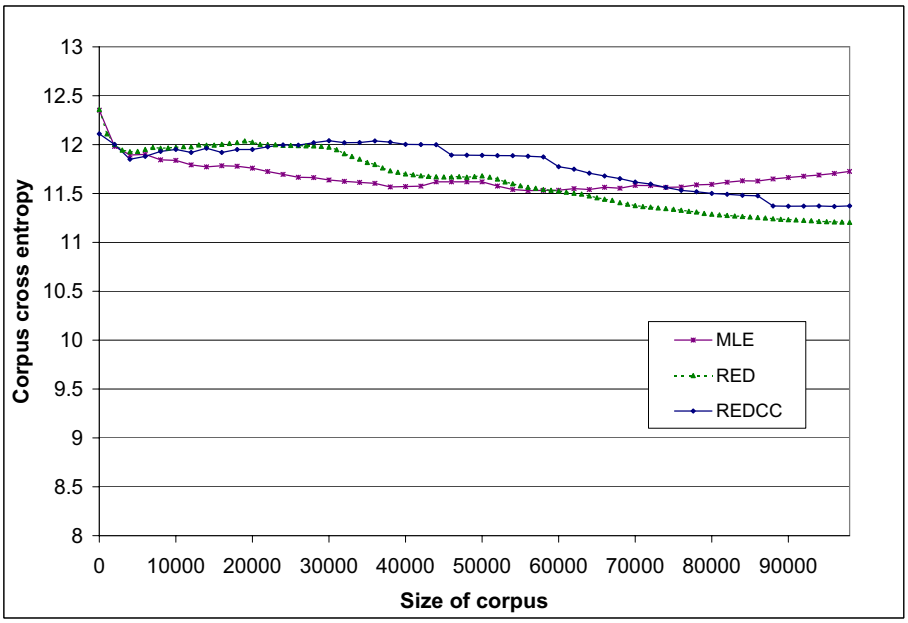

Figure 6.14: FreeBSD- Corpus Cross Entropy for the 3 proposed models.

Here, we use the sequence of events for making the models and compute the probability model distribution for each event using that. With the occurrence of each event, we make our three prediction probability models (MLE, RED and REDCC) and we use corpus cross entropy equation for each of them:

$$Hc(m) = -(1/N)\sum \log m(e_i) \qquad i = 1, \ldots, N \tag{6.2}$$

Where m is the prediction model.

We computed the Corpus Cross entropy for increasing corpus sizes, from 1 to N.

Figure 6.14 to 6.19 show the variation of Corpus Cross Entropy for the three models applied on different systems.

As can be seen in the figures, when the size of corpus becomes big enough for

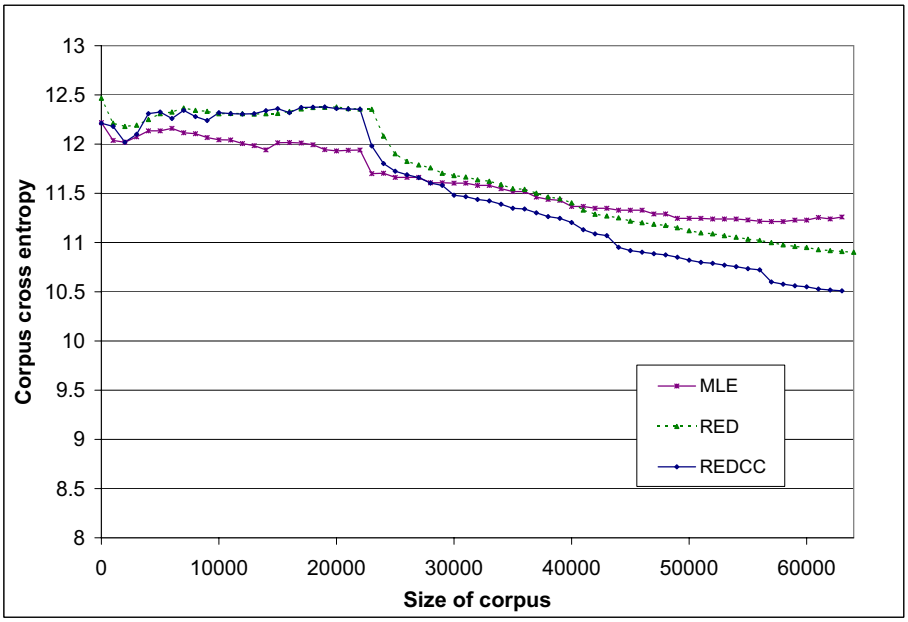

Figure 6.15: KOffice- Corpus Cross Entropy for the 3 proposed models.

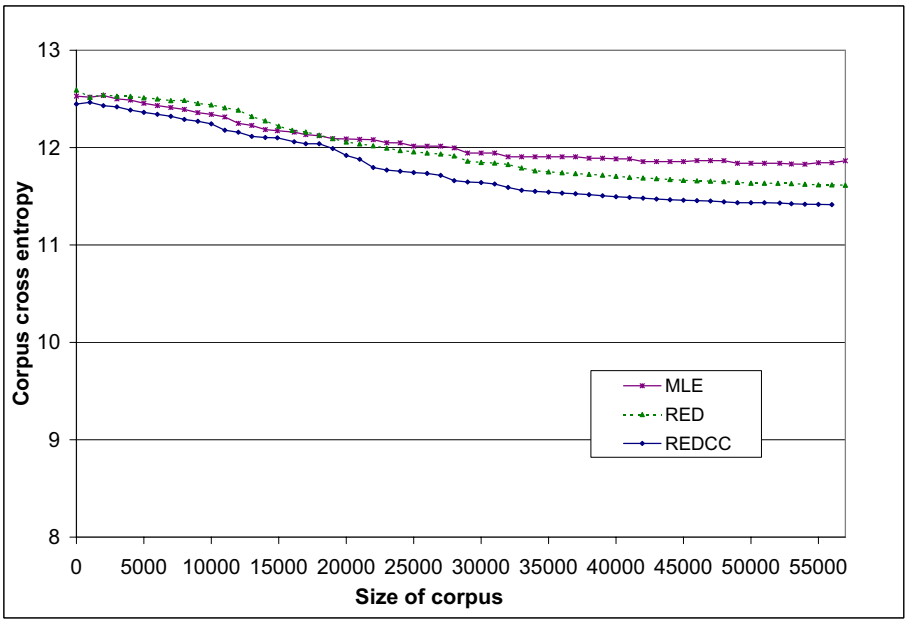

Figure 6.16: OpenBSD- Corpus Cross Entropy for the 3 proposed models.

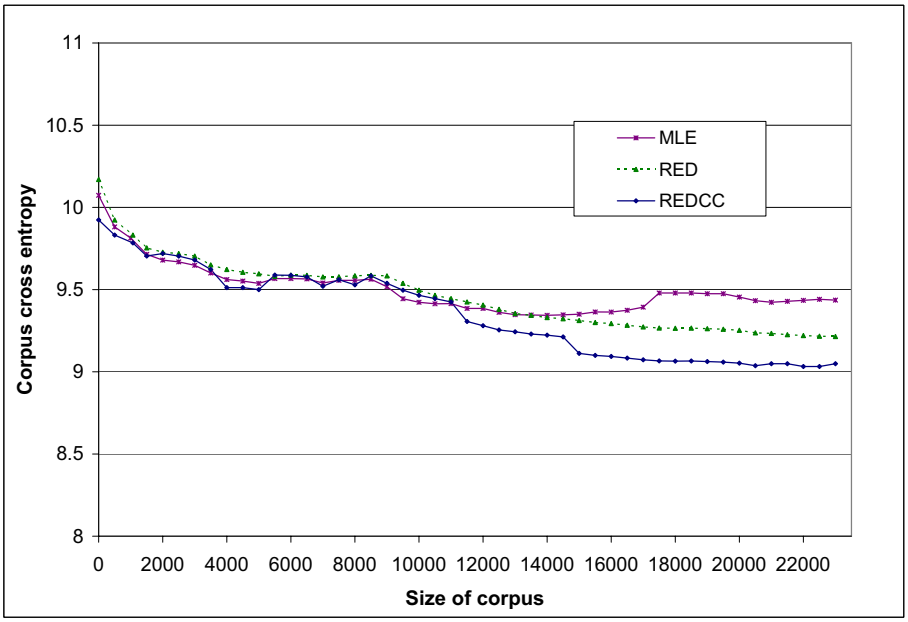

Figure 6.17: Postgres- Corpus Cross Entropy for the 3 proposed models.

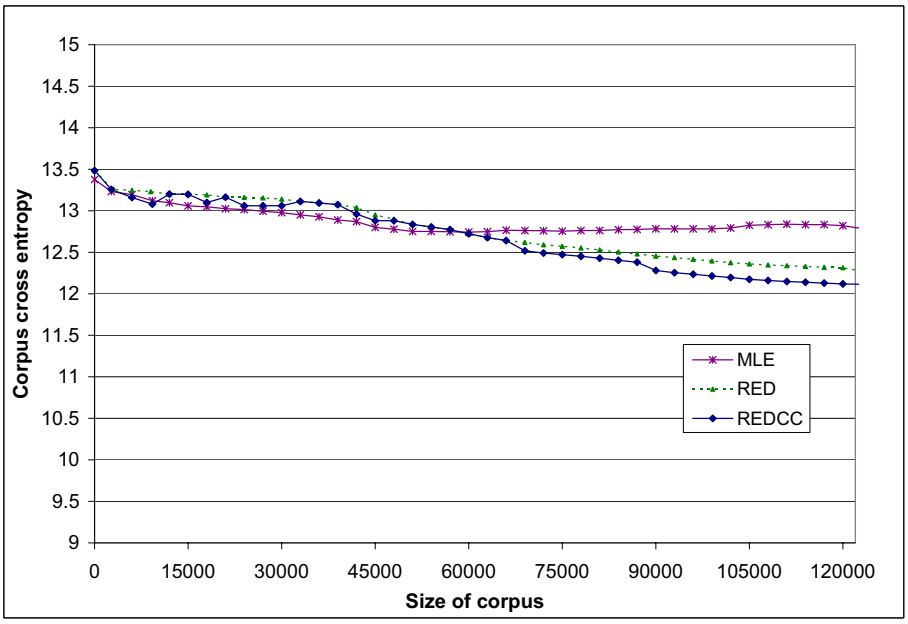

Figure 6.18: NetBSD- Corpus Cross Entropy for the 3 proposed models.

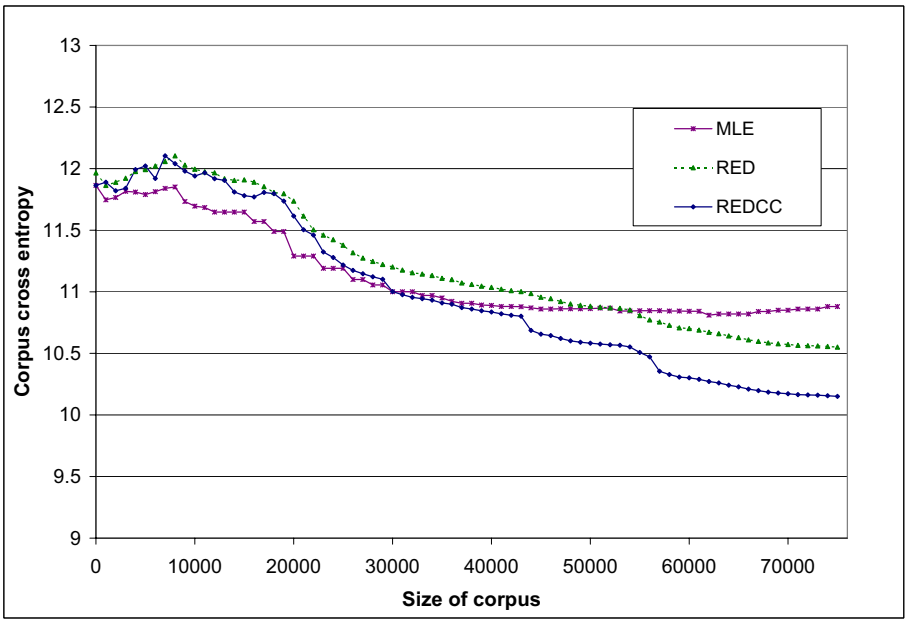

Figure 6.19: KDE- Corpus Cross Entropy for the 3 proposed models.

relying on CCE, REDCC has the lowest Corpus Cross Entropy, which means it has the closest distribution to the actual distribution of data. We observe that with this approach we have the same results as Top Ten List. Using the information theoretic approach the amount of goodness is measurable as bits of information and in this approach we do not need any list size. For example, in KOffice, when the size of corpus is big enough, the Corpus Cross Entropy for REDCC, RED and MLE are $10.5, 10.9 and 11.3$ respectively. This means that at this specific time, REDCC distribution has .8 bits less uncertainty than MLE. This can be compared to the maximum uncertainty of a distribution which is equal to $\log k$, where k is the number of files in the system. For KOffice system which has 6312 files, $H_{Max} = \log 6312 = 12.473$. It means that REDCC has $(0.8 * 100/12.473) = 6.4$ percent better performance than MLE.

We can also see except for FreeBSD, as the size of corpus increases, the Corpus

Cross Entropy of MLE distribution gets farther from Corpus Cross Entropy of RED and REDCC distributions. It shows that MLE gets worse when the size of corpus increases.

RED's half life

Here again in our experiments for using RED models, we were interested in estimating the best half life for using in other experiments.

Figure 6.20 shows the Corpus Cross Entropy for KOffice for different half lives: 1, 6, 12, and 18, during 5 years of development. Again it can be seen that when the half life is 6, RED model has a lower Corpus Cross Entropy and hence is a better prediction model. Based on these experiments, we again chose 6 months as the half life for RED. We got similar results for other systems as well, which are not shown here.

6.2.2 Comparing the Entropy of Prediction Models

Suppose that we use the sequence of events for making the models and compute the probability model distribution using that. With each event, we first make our three prediction probability models, and having these prediction probability models we can use entropy equation to compute the entropy of these distributions at that time:

$$H(m) = -\sum m(f_i) \log m(f_i) \qquad i = 1, \ldots, |D| \tag{6.3}$$

We computed the entropy for the three prediction models for different corpus sizes from 1 to N.

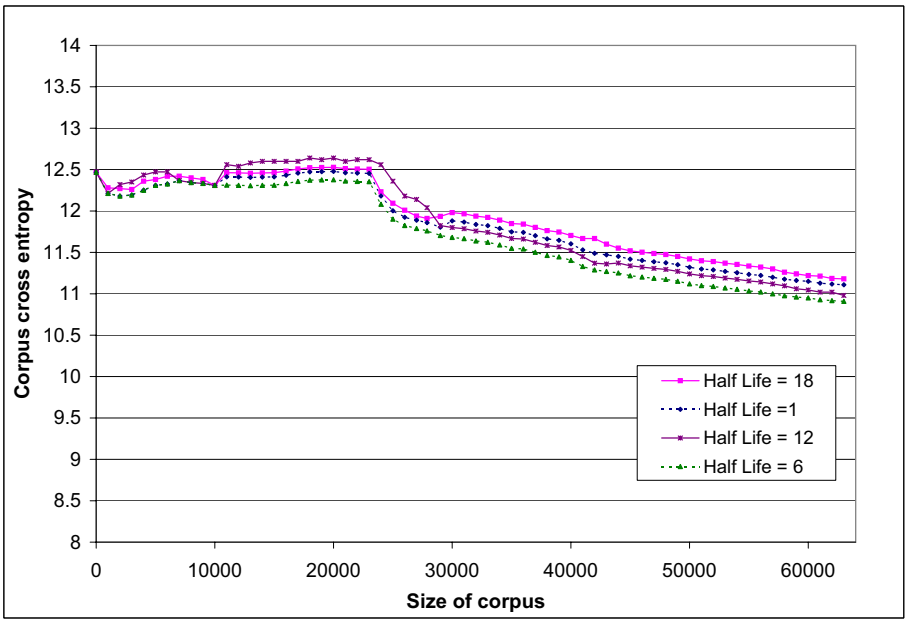

Figure 6.20: KOffice- Corpus Cross Entropy for RED Model with Different Half Lives.

Figures 6.21 to 6.26 show the variation of Entropy for different models applied on different systems.

In general we might expect that if a model's distribution has a lower Entropy, it has less uncertainty and hence is a better predictor. Even though this is a sufficient condition, it is not a necessary condition- that is, if a model has lower entropy it does not follow that it will be a better prediction model too. However, we see that in all cases, REDCC has a lower entropy during the development. It means that not only the CCE of REDCC is lower than others which makes its probability distribution closer to true distribution of data, but also its distribution is more far from uniform distribution than other studied prediction models.

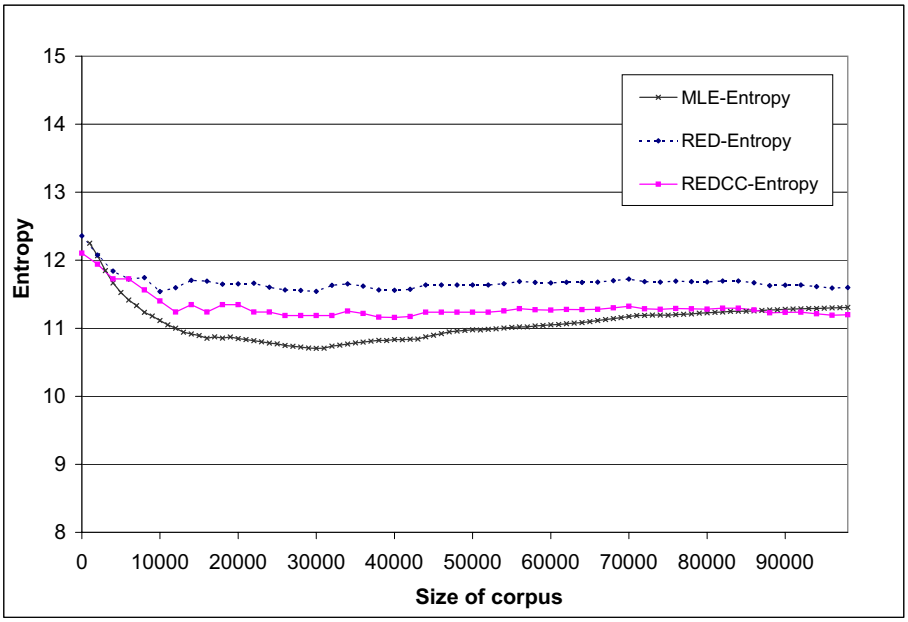

Figure 6.21: FreeBSD- Entropy for the 3 proposed models.

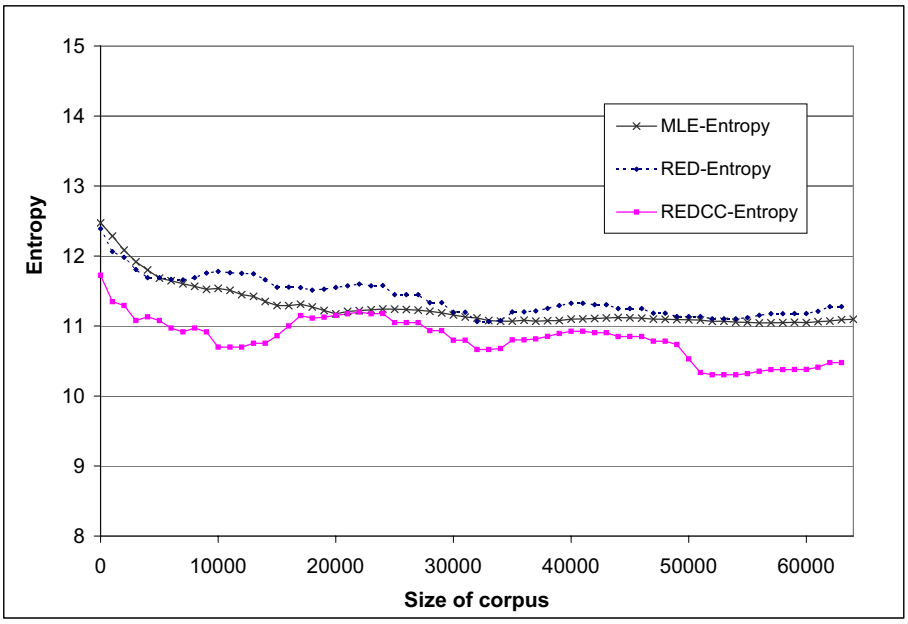

Figure 6.22: KOffice- Entropy for the 3 proposed models.

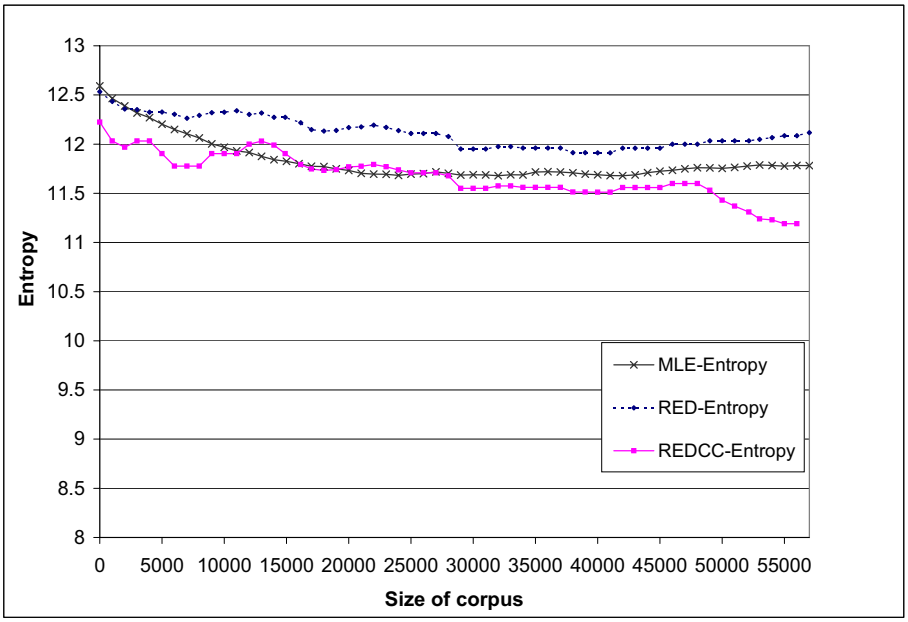

Figure 6.23: OpenBSD- Entropy for the 3 proposed models.

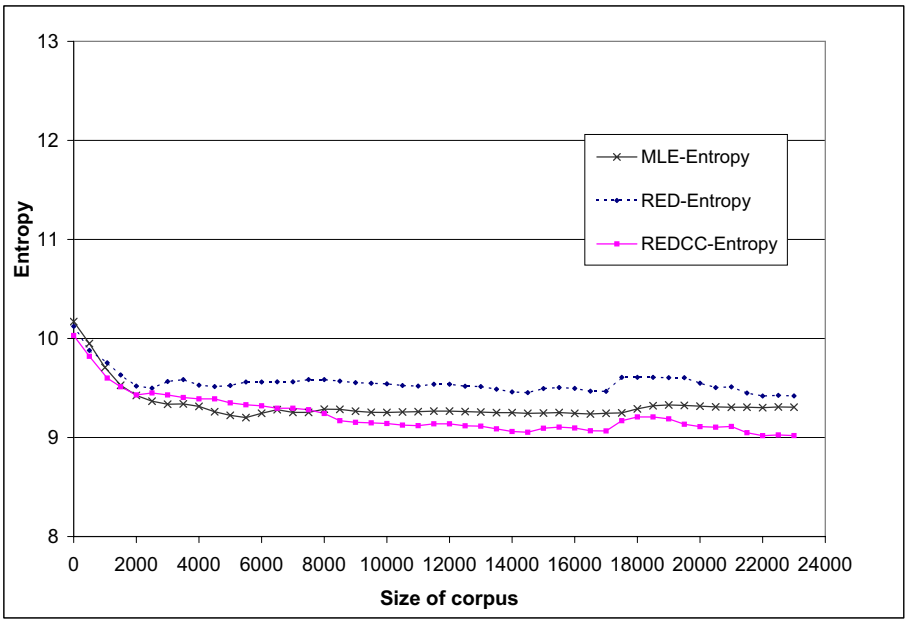

Figure 6.24: Postgres- Entropy for the 3 proposed models.

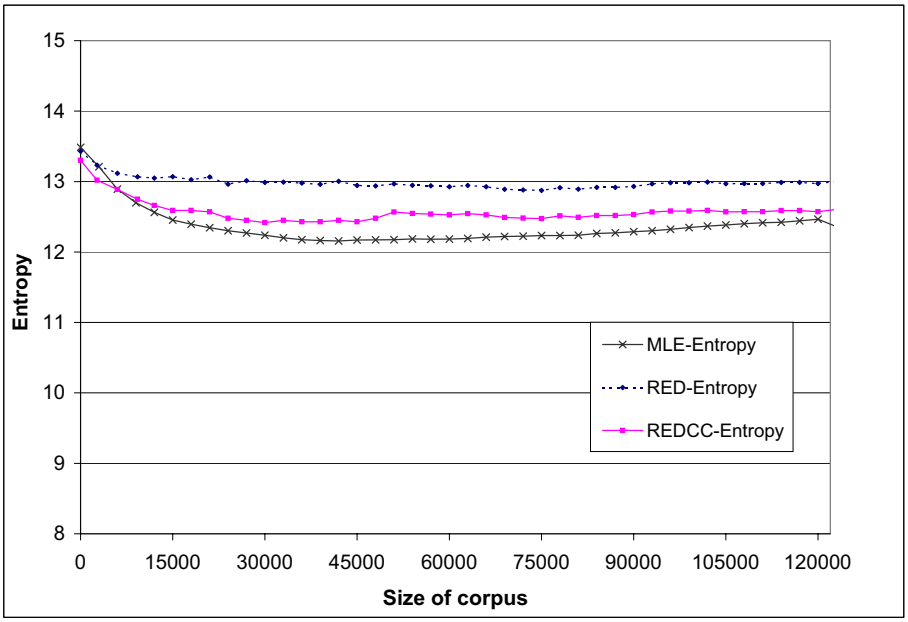

Figure 6.25: NetBSD- Entropy for the 3 proposed models.

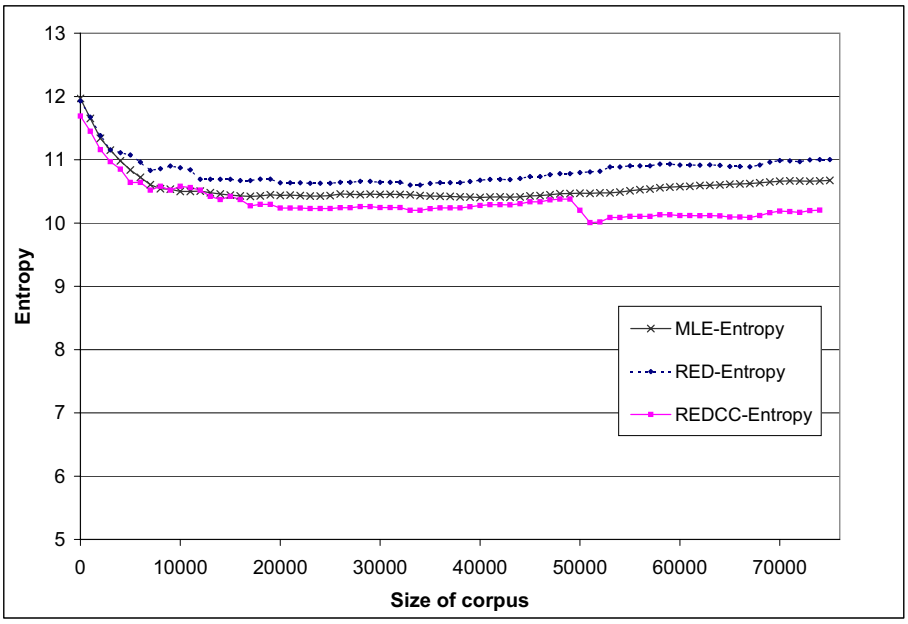

Figure 6.26: KDE- Entropy for the 3 proposed models.

6.3 Summary

In a previous chapter, we explained two different approaches that can be used for evaluating the change prediction models. In this chapter we applied these two approaches on the extracted information of our studied systems.

Using Top Ten List [3], we found that REDCC model has the highest Hit Ratio (between 70-90 percent when the size of list is 20 percent of the total files in the system)in all studied systems. Based on this Hit Ratio, REDCC considered as the best model.

In the second part, we used our information theoretic approach to evaluate prediction models. Using this approach, it is possible to determine which model predicts better and how much better it is. We got two important results using this approach. The first result, which is consistent with the TTL approach, is that REDCC has the lowest CCE in most of studied systems. It shows that it is the best predictor between theses three models. The second result is that CCE of these three models is decreasing as time goes on (the size of corpus increases) and it become stabilized in some pont. We can conclude that from the point that CCE get stabilized, the size of corpus is big enough and so the results of CCE is valid.

Chapter 7

Conclusion

7.1 Summary

In this thesis first we analyzed the information that is generated during the development process and can be obtained through mining the repositories of the software artifacts. We observed that the change data follows a Zipf distribution and exhibits self-similarity. Based on the extracted data, we then developed models for predicting future modification based on available change histories of software. We proposed a rigorous approach for evaluating these predictive models. We call it an information theoretic approach because in this approach, the closeness of model distribution to actual unknown probability distribution of the system is measured using cross entropy. We evaluated our proposed prediction models empirically using two approaches for six large open source systems. First we used the Top Ten List [3] approach to see which model predicts more accurately. Using this approach we showed that REDCC model works better. Then using our information theoretic evaluation approach, we observe that the REDCC model has the distribution that

is closest to the actual distribution for all the studied systems. The advantage of our approach over the Top Ten List approach is that using our approach we know quantitatively how much worse is the prediction model compared to what should be the best result. We hope that our approach can be used by developers to better predict possible future changes and bugs, based on the history of their software. Our approach also can be used by researchers who have developed new prediction models to evaluate them using our information theoretic approach.

In summary the main contributions in this thesis are:

- We investigated the characteristics of software modifications data and found that it has a self-similarity nature and follows a Zipf's distribution.
- We formulated three prediction models that use the information extracted from software repositories and predict probability distribution of future modifications of files in the system.
- We proposed using an information theoretic approach for evaluating different prediction models using a corpus of data.
- We performed experiments on CVS logs from six different systems to evaluate three different prediction models.

7.2 Possible Future Work

There are several extensions that can also be applied to our work in future.

We could investigate which other artifact sources, such as email logs, could be used by prediction models and how much information would need to come from

other sources to get better predictions. How can we get this information automatically?

It would also be interesting to investigate the maximum performance that a model can probably give us based on the information it is using. We expect that the performance of any model cannot exceed some specific amount, regardless of what the model is. We also believe that there is a relation between the exponential increase of the performance of the models using Top N List for different sizes and the cumulative frequency distribution of the systems.

In our study we were confined to applying the techniques and ideas presented in this thesis on open source systems. Open source software has a different nature due to different approaches used for developing them. It could be of interest to see if the techniques and approaches presented in this thesis could also be applied to commercial software systems. It would let us determine if the presented findings and results hold for such systems or if they are specific to open source systems.

Finally, it can be investigated to make and evaluate other prediction models similar to the RED in which substitute some other function instead of exponential function to implement "forgetting" the effects of changes and bugs.

Bibliography

[1] A. E. Hassan and R. C. Holt, "Studying the chaos of code development," in *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 123.

[2] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Data mining for predictors of software quality." *International Journal of Software Engineering and Knowledge Engineering*, vol. 9, no. 5, pp. 547–564, 1999.

[3] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the International Conference on Software Maintenance (ICSM 2005)*, 2005, pp. 25–30.

[4] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[5] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. M. Flass, "Using process history to predict software quality," *Computer*, vol. 31, no. 4, pp. 66–72, 1998.

[6] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy, "Predicting fault incidence using software change history," *Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[7] D. E. Perry and C. S. Stieg, "Software faults in evolving a large, real-time system: a case study," in *ESEC '93: Proceedings of the 4th European Software Engineering Conference on Software Engineering*. London, UK: Springer-Verlag, 1993, pp. 48–67.

[8] A. E. Hassan, Mining Software Repositories to Assist Developers and Support Managers. PhD Thesis, 2005, University of Waterloo, Ontario, Canada. [Online]. Available: http://plg.uwaterloo.ca/~aeehassa/home/pubs/phdthesis.pdf

[9] T. M. Khoshgoftaar and E. B. Allen, "Ordering fault-prone software modules," *Software Quality Control*, vol. 11, no. 1, pp. 19–37, 2003.

[10] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, 1999.

[11] Reliability Analysis Center (RAC), "Introduction to Software Reliability: A state of the Art Review. http://rome.iitri.com/rac/," 1996.

[12] M. R. Lyu, Ed., *Handbook of software reliability and system reliability*. Hightstown, NJ, USA: McGraw-Hill, Inc., 1996.

[13] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, Massachusetts: The MIT Press, 1999.

[14] Software Reliability. [Online]. Available: http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/

[15] J. Musa, *Software Reliability Engineering.* New York, New York: McGraw-Hill, 1998.

[16] J. Gaffney and C. Davis, "An automated model for software early error prediction," in *13th Minnowbrook Workshop on Software Reliability*, July 1990.

[17] J. D. Musa, A. Iannino, and K. Okumoto, *Software reliability: measurement, prediction, application.* New York, NY, USA: McGraw-Hill, Inc., 1987.

[18] Rome Labratory), "A History of Software Measurement at Rome Laboratory . http://www.dacs.dtic.mil/techs/history/toc.html."

[19] R. W. Selby and A. A. Porter, "Learning from examples: Generation and evaluation of decision trees for software resource analysis," *IEEE Trans. Softw. Eng.*, vol. 14, no. 12, pp. 1743–1757, 1988.

[20] M. M. T. Thwin and T.-S. Quah, "Application of neural networks for software quality prediction using object-oriented metrics," *J. Syst. Softw.*, vol. 76, no. 2, pp. 147–156, 2005.

[21] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 886–894, 1996.

[22] M. Pighin and R. Zamolo, "A predictive metric based on discriminant statistical analysis," in *ICSE '97: Proceedings of the 19th international conference on Software engineering.* New York, NY, USA: ACM Press, 1997, pp. 262–270.

[23] T. M. Khoshgoftaar, N. Seliya, and A. Herzberg, "Resource-oriented software quality classification models," *J. Syst. Softw.*, vol. 76, no. 2, pp. 111–126, 2005.

[24] D. J. Hand, "Data mining: Statistics and more?" *The American Statistician*, vol. 52, no. 2, pp. 112–??, 1998.

[25] Cuncurrent System Version), "CVS Tutorial - Concurrent Versions System. http://www.csc.calpoly.edu/ dbutler/tutorials/winter96/cvs/."

[26] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572.

[27] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster, "Visualizing software changes," *IEEE Trans on Software Engineering*, vol. 28, no. 4, pp. 396–412, 2002.

[28] A. T. T. Ying, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, 2004, member-Gail C. Murphy.

[29] V. R. Basili and B. Perricone, "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[30] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history." in *Proc. 14th International Conference on Software Maintenance*, Bethesda, Washington D.C., November 1998.

[31] A. T. T. Ying, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, 2004, member-Gail C. Murphy.

[32] A. Mockus and L. Votta, "Identifying reasons for software change using historic databases," in *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*. IEEE Computer Society Press, 2000, pp. 120–130.

[33] A. Mockus, D. M. Weiss, and P. Zhang, "Understanding and predicting effort in software projects," in *International Conference on Software Engineering (ICSE 2003).* Portland, Oregon: ACM Press, May 2003, pp. 274–284.

[34] D. E. Perry and W. M. Evangelist, "An empirical study of software interface faults - an update," in *20th Annual Hawaii International Conference on Systems Sciences*, Hawaii, USA, 1987, pp. 113–136.

[35] Pareto Law. [Online]. Available: http://www.it-cortex.com/Pareto_law.htm

[36] G. K. Zipf, *Human Behavior and the Principle of Least-Effort.* Cambridge, MA: Addison-Wesley, 1949.

[37] A. E. Hassan, J. Wu, and R. C. Holt, "Visualizing historical data using spectrographs," in *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005)*, Como, Italy, September 2005.

[38] Relative Frequency. [Online]. Available: http://en.wikipedia.org/wiki/Relative_frequency

[39] Allen, J. F. Using Entropy for Evaluating and Comparing Probability Distributions. [Online]. Available: ttp://www.cs.rochester.edu/u/james/CSC248/Lec3.pdf

[40] D. Beyer and A. Noack, "Mining Co-Change Clusters from Version Repositories," Tech. Rep., 2005.

[41] J. Han, "Supporting impact analysis and change propagation in software engineering environments," in *STEP '97: Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (in-*

cluding CASE '97). Washington, DC, USA: IEEE Computer Society, 1997, p. 172.

[42] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 284–293.

[43] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories." in *IEEE METRICS*, 2005, p. 29.

[44] Allen, J. F. Using Entropy for Evaluating and Comparing Probability Distributions. [Online]. Available: ttp://www.cs.rochester.edu/u/james/CSC248/Lec6.pdf

[45] C. E. Shannon, "A mathematical theory of communication," *Bell Sys. Tech. J.*, vol. 27, pp. 379–423, 623–656, 1948.