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

Mina Askari
School of Computer Science
University of Waterloo
Waterloo, Canada
maskari@uwaterloo.ca

Ric Holt
School of Computer Science
University of Waterloo
Waterloo, Canada
holt@uwaterloo.ca

ABSTRACT
In this paper, we analyze the data extracted from several open source software repositories. We observe that the change data follows a Zipf distribution. Based on the extracted data, we then develop three probabilistic models to predict which files will have changes or bugs. The first model is Maximum Likelihood Estimation (MLE), which simply counts the number of events, i.e., changes or bugs, that happen 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 third model is called RED-Co-Change. With each modification to a given file, the RED-Co-Change 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 RED-Co-Change model predicts the distribution that is closest to the actual distribution for all the studied systems.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement- *Version control*

D.2.8 [**Software Engineering**]: Metrics- *Performance measures, Process metrics*

General Terms
Performance, Reliability, Theory

Keywords
Prediction Models, Evaluation approach, Information Theory

MSR'06, May 22–23, 2006, Shanghai, China.

1. INTRODUCTION
Software systems are continuously being changed to adapt to meet the needs of their users or to correct the faults appearing in systems during development or after deployment. There has been extensive research on new processes and approaches for developing software systems to minimize these new modifications. The idea is that during software development by following some specific principles, the probability of certain kinds of modifications can be decreased. Despite this progress, new changes and bugs are inevitable during software development. However, if software developers were able to forecast the occurrence of changes and bugs then they could mitigate their impact. Therefore, developing accurate techniques to predict the future behavior of changes and bugs can be valuable for software development and maintenance.

The idea for predicting which files/subsystems are most susceptible to having a fault in the near future is a well-known idea. There exist several prediction models [5][6][8][9][10][14] and more are emerging. However, many of these fault prediction models have not been evaluated in practice and some of them are not applicable to large-scale software systems. The majority of fault prediction models are applicable to deployed systems only. The general approach for evaluating these models is to run the system and collect the observed information during its execution and then compare it with the results predicted by the models [18]. The problem is that too often these models are not general and hence, they are not applicable to different software systems. In many cases, because the models measure different metrics, the results are not comparable [18]. There are many questions with respect to the validity of the underlying assumptions, accuracy, and applicability of software prediction models. In this paper, our goal is to contribute toward more general and realistic assessment and prediction of software modifications based on theoretical and empirical studies. We are interested in methods and models that have two properties. First, they use data collected during development process and second, their distance from the actual but unknown distribution of the collected data can be measured. Our goal is twofold: first, to develop prediction models driven by software repositories and second, evaluate and compare different models using a mathematical approach. We use historical records, from source control repositories of large software systems, to develop prediction models and to estimate how much information is captured by the models.

Figure 1 illustrates the problem we are trying to solve. Suppose we have a list of all the events that have so far occurred on different files of a software system during development process.

These events are file changes to fix bugs, or to add new features or change existing features. We have extracted these events from the history of the software. For example f_{21} shows that one modification has happened on file 21 at the specific time (see Figure 1). We ask this question: To what degree are these changes unpredictable? More particularly, what will be the uncertainty of the next sample, if all past samples are known? In some cases, it may be impossible to say anything about the next sample regardless of how many past samples are already known. In other cases, the process may be much less uncertain about the next sample when given the history of the changes. Having extracted historical data, we want to determine how much information this data provides us about the future. Our results indicate that CVS log data contains information about the past that can help to predict the future. The questions include: How much information is buried in the CVS logs and how can we capture this information? How good are the prediction models that use this information to predict the future?

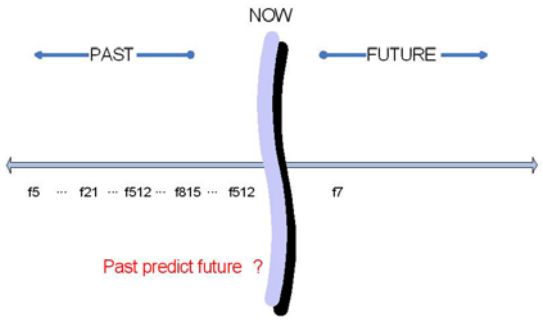


Figure 1. Does past predict future?

After introducing related work in Section 1.1, the rest of the paper is organized as follows. Section 2 presents the techniques and approaches we used to perform different experiments in order to analyze the data extracted from several open source software repositories. In Section 3, we present our three prediction models and describe the steps involved in developing the models. In Section 4, we present an information theoretic approach for evaluation prediction models. In Section 5 presents the results of using two approaches for evaluating our proposed prediction models: the Top Ten List approach proposed by Hassan et al. [8] and our information theory based approach. Finally, Section 6 concludes the paper and discusses possible future works.

1.1 Related Work

Many researchers [2][5][6][8][9][16][17] in software development area have realized the value of historical data and have used them in their research ranging from software design to software understanding, software maintenance, development process and many more areas.

There is considerable research on developing tools to recover such historical data. Hassan et al. [7] developed C-REX tool, an evolutionary code extractor, which recovers information from source control repositories. Zimmermann et al. [19] 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. [5] used software repositories. They uncovered the dependencies and interrelations between classes and modules (logical dependencies) which can be used by developers in maintenance phase of a system.

Graves et al. [6] showed that there is a relation between the number of changes a subsystem has with the future faults in that subsystem. Hassan et al. [8] presented various heuristics using historic version control data to create the Top Ten List. Top Ten List highlights to managers the ten most susceptible subsystems to have a fault. They also developed techniques to measure the performance of these heuristics.

Mockus et al. [12] 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 [13]. Khoshgoftaar et al. [9][10] 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. Eick et al. [4] presented visualization techniques to explore change data to help engineers understand and manage the software change process. Ostrand, et al. [14] 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. CHARACTERISTICS OF THE DATA

The prediction models and the evaluation methods presented in this paper are based on change history data. Change data is the information generated during development process and can be obtained through mining the repositories of the software. We began by analyzing the extracted data to understand its statistical properties. In particular, we observed that history data has Zipf distribution [20].

2.1 Studied Systems

To perform our study we used several CVS logs of open source software systems. Table 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. We were looking for any kind of change and bug which happens to different files of a system. The process of acquiring such specific data is very challenging, since CVS logs are mainly designed as record keeping repositories and commits aren't atomic and large amount of data stored in these repositories complicates the data extracting 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 software systems in a standard format that is easier to process and not developing tools that automatically recover data from these repositories. So we obtained and used the data which were extracted from these CVS logs by tools developed by Hassan et al. [7]. This let us concentrate on analyzing the extracted data instead of spending time developing tools to recover the data.

Table 1. Number of events available for different systems

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

2.2 Zipf's Law

We started by counting the number of modifications which 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. Figure 2 illustrates the number of modifications for each file for different systems we studied. Different systems have different number 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 2 to 4.

As it can be seen from the figure 2, 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 [15] and Zipf's law [20].

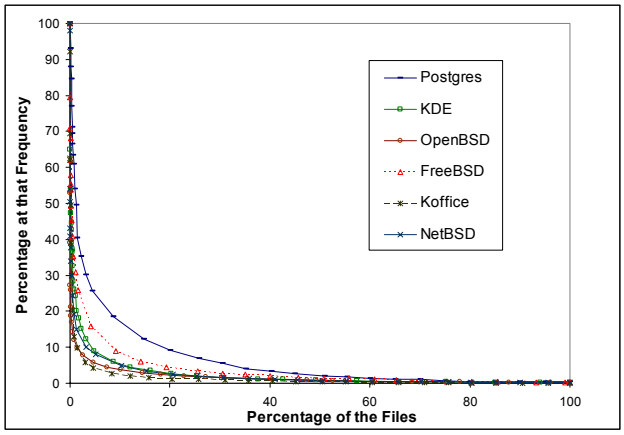


Figure 2. Change data follows Zipf's law.

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 in Figure 3. It can be seen from the figure that almost 20% of the files in the systems have (almost) 80% of activities during development. To show that Zipf's law holds for the data, we plotted the log-log scale of the cumulative frequency distributions; see Figure 4. 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. CHANGE PREDICTION MODELS

The prediction of future modifications in a large software system is an important part in software evolution. Since most 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. Our expectation was that if we could build a model applicable to different range of systems based on the information which is generated during development process, e.g. CVS logs, it would be useful for software developers. In this Section we will show several prediction models which can use the CVS logs to predict the future bugs and changes in any arbitrary system. These models are generally in form of probability models.

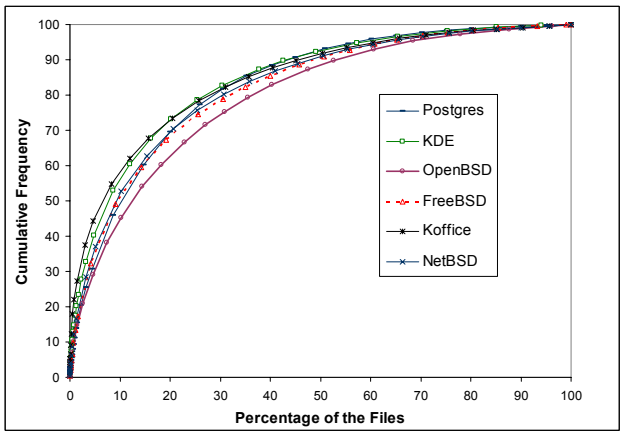


Figure 3. Cumulative frequency distributions.

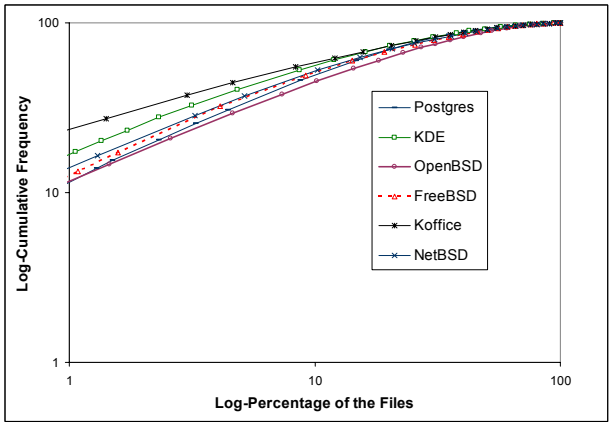


Figure 4. Log-log scale of cumulative distributions.

After extracting the changes and bugs that occurred in the various files of a system during development, we created a sequence of events showing file changes to fix bugs or to add features. Having this sequence of events our goal is to predict future comparable events. There are many files in the systems

we studied, ranging from 1000 to 20000 files. We wanted to construct a probabilistic model of this process, in other words, to define a probabilistic model that characterizes the result of the next element in the sequence. We assume that we know that the possible value space, i.e., the Domain D (i.e., sample space) for event *e* (considered as a random variable). In our work, D is the set of files in the system. We denote the elements of this Domain as $f_1, f_2 \ldots f_m$. Our goal is to define a good probability model to give the probability that the i^{th} (i.e., next) element in the sequence will have a particular value (will be a particular file); in other words for finding the probability distribution of random variable e_i, what we need to do is to decide on the form of the underlying model of the sequence of events. Ideally this would be a conditional probability function of form $P(e_i \mid e_1, e_2, \ldots, e_{i-1})$. Our work is complicated by the fact that, in general, a new probability function is needed for each e_i. Based on this approach, we will now present three probabilistic models.

3.1 Most Likely Estimation (MLE) Model

Our first model, maximum likelihood estimate (MLE), simply uses the counts from the sequence to estimate the distribution.

$$P_{MLE}(e = f_i) = Count(f_i) / N \qquad (1)$$

In (1), $fi \in D$, N is the size of sequence, and $Count(fi)$ is the number of occurrences of fi in the sequence.

The proportion of times a certain event f_i occurs is called the relative frequency of the event. In the MLE model, we compute (predict) the relative frequency of each new event based on the preceding sequence. Empirically for our data we observed if one performs a large number of trials, the relative frequency (for each file) tends to stabilize around some number.

In our experiments, instead of definition (1), we computed our MLE probability distributions [1] using this formula:

$$P_{MLE}(E = f_i) = (Count(f_i) + 1) / (N + d) \qquad (2)$$

In (2), $fi \in D$, N is the size of sequence, $Count(fi)$ is the number of occurrences of fi and d is the size of domain D.

We use this equation because equation (1) has two computational problems. The first problem is that it implicitly 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 sequence containing a previously unseen element. The second problem is that it does not distinguish between different levels of certainty based on the amount of evidence we have seen. One solution is to assign a small probability to each possible observation at the start. We do this by adding a small number (we use l) to the count of each outcome to get the estimation formula. This technique, using value 1, is called Laplace estimation [1]. If we never see a token of a type f in a corpus of size N and domain size d, the probability estimate of a token of f occurring will be $1/(N+d)$. For the second problem, using Laplace formula, our prior knowledge that there is D different types of events makes our estimate stay close to the uniform distribution [1].

3.2 Reflexive Exponential Decay (RED) Model

Our second model relies on the idea that when a change is observed in a file, it is likely that more changes will be observed in that file, but that this effect decreases (decays) with time. We are given a sequence of events called $e_1, e_2 \ldots e_n$, occurring respectively at monotonically increasing times $t_1, t_2 \ldots t_n$. We assume that events probabilistically predict events, e.g., 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. More generally, we call it the reflexive exponential decay (RED) model. A particular event occurring at time t_i on the file f_j, implies (predicts) a future frequency rate $R_t(j)$ for that file at future time t. Our model defines $R_t(j)$ as follows:

$$R_t(j) = I\, e^{k(t-t_i)} = I\,(1/2)^{(t-t_i)/h} \qquad (3)$$
$$where \quad k = -\ln(2)/h \ and \ t > t_i$$

In formula (3), h is the half life (measured typically in months) and I is the "impact" of an event (measured typically in events per month). This means that if in the sequence of events, e_i happens at time t_i and e_i is a modification of file f_j, for all time $t > t_i$, the predicted incremental frequency for *file will be* $R_j(t)$.

A larger half life h means that the effects of a change last longer. Figure 5 shows $R_t(j)$ for different half lives and with impact of $I = 1$ and $t_i = 0$.

Based on $R_t(j)$ for each event on file f_j, we define the RED model as the summation of the effects all (historical) events happening to each file.

We now formalize the RED frequency model. Suppose that the sequence of events, $e_0, e_1, e_2, .., e_l$ has happened on file f_j up time t. Then RED predicts that the future frequency of changes to this file will be:

$$R_t(j) = I\, e^{k(t-t_0)} + I\, e^{k(t-t_1)} + I\, e^{k(t-t_2)} + \ldots + I\, e^{k(t-t_l)}$$
$$for\ all\ t \geq t_l \qquad (4)$$

Figure 6 shows how the effect of each event is added to the previous ones for a specific file. In the figure, specific 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 these 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.

RED Distribution Model

We will now convert our RED model so that it predicts probability distribution rather than frequency. Given a sequence of events: $e_0, e_1, e_2, .., e_m$, having $R_t(j)$ for all files $j = 1..n$, we can define the distribution of RED at time t as follows:

$$RED_t(e_{m+1} = f_i) = \frac{R_t(i)}{\sum_{j=1..n} R_t(j)} \quad for\ t \geq t_m \qquad (5)$$

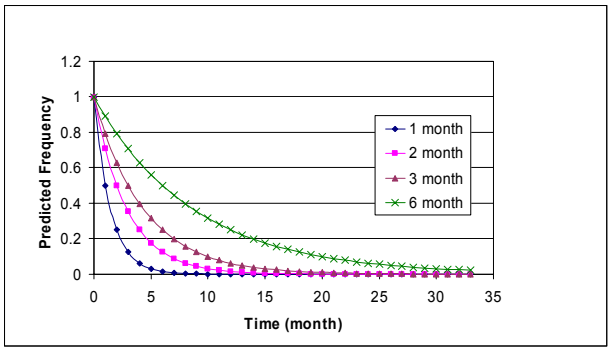


Figure 5. Exponential decay for different half lives.

3.3 RED Co-Change (REDCC) Model

Our third model is an enhanced version of RED. When each event occurs we update the probability not only for the changed file but also for the *co-changed* files. There are several different approaches for concluding that (or defining that) the files change (co-change) together during software development.

Developers commonly modify files together (co-change them) to introduce new features or fix bugs. Developers should ensure that when one file is changed, other related files in the software system are updated to be consistent with the modifications. We use a definition of co-change that is inspired by the literature [8]. If file f_1 and f_2 changed together (on the same day) in previous change sets, then they are candidates to be considered as co-changed files. We will define that co-change files are those sets of files which have changed on the same day in the past at least 3 times within the preceding 7 days. We now define the *RED Co-Change* (REDCC) model. We assume that at time t, the sequence of events, $e_0, e_1, e_2, .., e_m$ has happened on file f_i or on the co-change files of f_i up to this time.

$$REDCC_t(j) = I\,e^{k(t-t_0)} + I\,e^{k(t-t_1)} + \ldots + I\,e^{k(t-t_m)} \qquad (6)$$

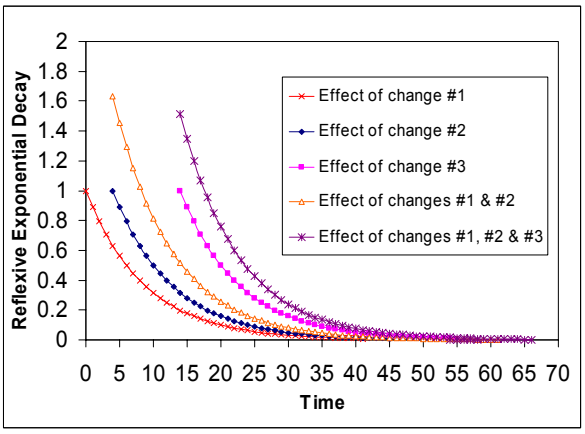


Figure 6. Reflexive exponential decay for a file.

Using $REDCC_t(j)$ frequency model, we convert it to the probability model $REDCC_t(e_{m+1}=f_i)$ in the same way we converted the RED frequency model to a probability model.

4. EVALUATION OF PREDICTION MODELS

In this section we present an information theoretic approach to quantify the goodness or fitness of a guessed probability g (g is a prediction model) compared to the actual probability p. Our approach uses entropy concepts to evaluate prediction models.

Our goal is to compare these predictive models (distributions) to see how good they are. By "good" we mean how close they are to the true distributions of the events. It also could mean that how well they predict the occurrence of the next event. The approach we take is well known in Natural Language Processing (NLP) area, where a sequence of words in language is called *corpus*, but to our knowledge has not been used in the field of Mining Software Repositories. NLP uses information theory to find the distance between prediction models and actual distribution of corpus [11].

4.1 Entropy and Cross Entropy

Before introducing our information theoretic approach, we will review some related concepts. Information theory techniques define the amount of information in a message. The theory measures the amount of uncertainty/entropy in a distribution. Shannon entropy [11], given probability p(x), is defined as:

$$H(p) = -\sum p(x)\log p(x)$$

Larger values of H(p) imply that more bits are needed for coding messages.

There is a related concept called *cross entropy* which allows us to compare two probability functions. (Cross entropy is closely related to Kullback-Leibler divergence [11].) The cross entropy between two probability distributions measures the overall difference between the two distributions p and m and is defined as:

$$H(p,m) = -\sum p(x)\log m(x)$$

Where p(x) is the true distribution and m(x) the model distribution.

The cross entropy is minimal when p and m are identical, in which case it reduces to simply H(p). The closer the cross entropy is to entropy proper, the better m is an approximation of p. 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 approximation to distribution to p.

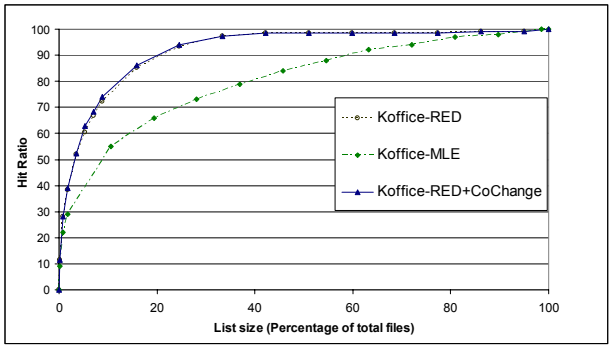
This approach seems to require that we know p, the actual distribution of data, which unfortunately we do not know. One of the central problems we face in using probability models is obtaining the actual distribution p(x) of data. The true distributions are not known, yet we want to estimate predictive models and validate them using the existing data.

Here there is a paradox: if we had p(x) in advance, we wouldn't need to make any model for estimating p(x).

4.2 Corpus Cross Entropy

We solve this problem with using *corpus cross entropy* (CCE). Given a sequence c with of length N consisting of events $e_1 \ldots e_N$, the corpus cross entropy of a probability function m is defined as follows:

$$H_c(m) = -(1/N)\sum \log m(e_i)$$



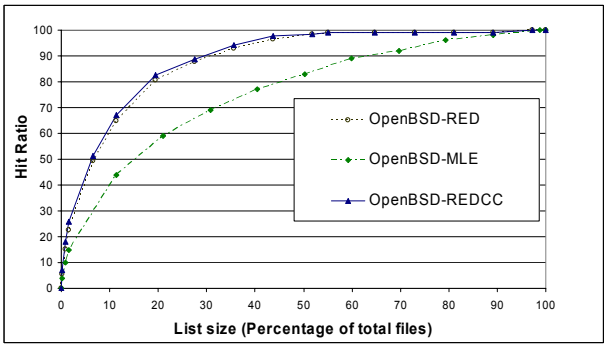


Figure 7. Evaluation of 3 models based on Hit Ratio of Top Ten List, with varying size of list.

It is straightforward to prove that corpus cross entropy $H_c(m)$ approaches cross entropy H(p,m) as N approaches infinity, given that p is the true distribution of corpus c and given that p is stationary. We can compute $H_c(m)$, as an approximation to H(p,m), even though we do not know distribution p. As is done in NPL literature [11], we assume that given two models m_1 and m_2 we can compare $H_c(m_1)$ and $H_c(m_2)$ to determine which of m_1 and m_2 is the better model, even though we do not know the true distribution p, given that p is reasonably stationary. That is, when $H_c(m_1) < H_c(m_2)$ we conclude that m_1 is a closer distribution to the true distribution and hence is a better model.

5. EMPIRICAL STUDIES

In this section we evaluate our three proposed prediction models (MLE, RED and REDCC) empirically, using two approaches, for six large open source systems. Table 1 summarizes the details of the software systems we studied. Due to space limitation we will only shown the results for two systems (Koffice and NetBSD). The other systems had similar behavior.

5.1 Top Ten List evaluation

For evaluating the quality of our three models, first we use the Top Ten List [8] approach. This approach evaluates which model predicts more accurately.

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, call it file f_i. We record whether file f_i is in the event's Top Ten List. We define the Hit Ratio as the fraction of events in which file f_i was observed to be in its Top Ten List. Models with higher Hit Ratios are considered to be better models.

We applied the Top Ten List approach to evaluate our three proposed models, for all the studied system; see Figure 7 for the results for two of these systems: Koffice and OpenBSD. (Results for the other studied systems are comparable.) As can be seen, for both systems, REDCC and RED have very similar results, with REDCC being slightly superior. By contrast, MLE's results are considerably worse. In other words, the Top Ten List approach evaluates REDCC is slightly better than RED, and both of these considerable better than MLE.

As can be seen in Figure 7, as the size of list increases, we have a higher hit ratio. Interestingly, using REDCC or RED model, when we use 20 percent of total files in the system, the hit ratio is almost 80 percent.

5.2 Information theoretic evaluation

We also applied the information theoretic approach to compare our three prediction models. Due to space limitations, we only present the result for one of the studied system, Postgres. The results for the other systems are similar.

Using historical Postgres data, we developed instances of our three models: MLE, RED and REDCC. To develop the MLE model, we used the first 10000 events and kept it fixed for the remaining corpus.

Figure 8 shows the corpus cross entropy of our three predictive models when applied to Postgres. As it can be seen in the figure, REDCC has the lowest corpus cross entropy which means its distribution is the closest to the actual distribution of the data. The next closest (see middle curve in Figure 9) is RED, and the worst (top curve) is MLE. Note that this ordering is the same that we observed when our evaluations were based on the Top Ten List.

As can be seen in Figure 8, as the size of corpus increases the MLE distribution gets farther from the real distribution of data but for two other models, RED and REDCC, the opposite is true. This suggests that the RED and REDCC models benefit by updating their distributions based on the events in the corpus as time passes.

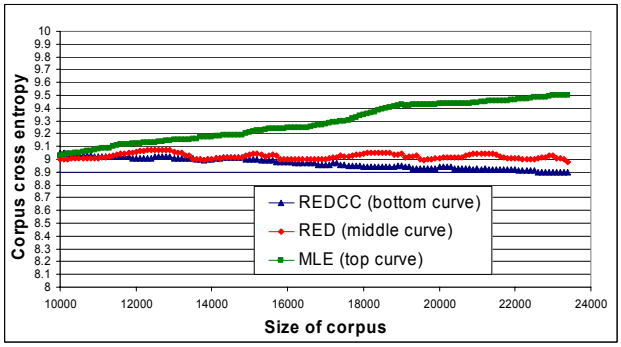


Figure 8. Evaluation of 3 models using corpus cross entropy on Postgres.

6. CONCLUSION

We developed three models (MLE, RED and REDCC) for predicting future modification of files based on available change histories of software. We proposed a rigorous approach for evaluating such predictive models. This approach has been used in Natural Language Processing, but not in Mining Software Repositories, as far as we know. This is an information theoretic approach in that the closeness of a predictive model distribution to an actual but 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 [8] approach to see which model predicts more accurately. Using this approach we showed that the REDCC model works best of our three models. Then using our information theoretic evaluation approach, we observe that the REDCC model again has the distribution that is closest to the actual distribution for all the studied systems. An advantage of our information theoretic approach over the Top Ten List approach is that using our approach we know quantitatively, as measured by cross entropy, how much better or worse is the prediction model compared to ideal result.

Our hope is that our approach can be used to help better predict future changes and bugs, based on the history of software. Our approach also can be used by researchers who have developed new prediction models to evaluate them using a information theoretic approach.

7. ACKNOWLEDGMENT

The authors would like to thank Ahmed Hassan. This paper would not have been possible without his generous help and his data. We also would like to thank the referees for their extremely helpful suggestions.

8. REFERENCES

[1] Allen, J. F. Using Entropy for Evaluating and Comparing Probability Distributions, available at: http://www.cs.rochester.edu/u/james/CSC248/Lec6.pdf

[2] Basili, V. R., and Perricone, B. Software errors and complexity: An empirical investigation. Communications of the ACM, 27(1):42 – 52, 1984.

[3] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J.S., and Mockus, A. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. on Software Engineering,* 27(1):1–12, 2001.

[4] Eick, S.G., Graves, T.L., Karr, A.F., Mockus, A., Schuster, P. Visualizing Software Changes, *IEEE Trans. on Software Engineering*, vol. 28, no. 4, pp. 396-412, April, 2002.

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

[6] Graves, T. L., Karr, A. F., Marron, J. S. and Siy, H. P. Predicting fault incidence using software change history. *IEEE Trans. on Software Engineering,* 26(7):653–661, 2000.

[7] Hassan, A. E., *Mining Software Repositories to Assist Developers and Support Managers*. PhD Thesis, University of Waterloo, Ontario, Canada, 2004

[8] Hassan, A. E. and Holt, R. C., The Top Ten List: Dynamic Fault Prediction, *Proceedings of ICSM 2005: International Conference on Software Maintenance,* Budapest, Hungary, Sept 25-30, 2005.

[9] Khoshgoftaar, T. M., Allen, E. B., Halstead, R., Trio, G. P. and Flass, R. M. Using Process History to Predict Software Quality. *Computer,* 31(4), 1998.

[10] Khoshgoftaar, T. M., Allen, E. B., Jones, W. D., and Hudepohl, J. P. Data Mining for Predictors of Software Quality. *International Journal of Software Engineering and Knowledge Engineering,* 9(5), 1999.

[11] Manning, C. and Schütze, H. Foundations of Statistical Natural Language Processing, MIT Press. Cambridge, MA: May 1999.

[12] Mockus, A. and Votta, L. G. Identifying reasons for software change using historic databases. In *International Conference on Software Maintenance*, pages 120-130, San Jose, California, October 11-14 2000

[13] Mockus, A., Weiss, D. M., and Zhang, Ping. Understanding and predicting effort in software projects. In *2003 International Conference on Software Engineering*, pages 274-284, Portland, Oregon, May 3-10 2003. ACM Press.

[14] Ostrand, T. J., Weyuker, E. J., Bell, R. M. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Trans. Software Eng.* 31(4): 340-355 (2005)

[15] Pareto Law: http://www.it-cortex.com/Pareto_law.htm

[16] Perry, D. E. and Evangelist, W. M. An Empirical Study of Software Interface Faults — An Update. *In Proceedings of the 20th Annual Hawaii International Conference on Systems Sciences,* pages 113–136, Hawaii, USA, January 1987.

[17] Perry, D. E. and Steig, C.S. Software Faults in Evolving a Large, Real-Time System: a Case Study'. *In Proceedings of the 4th European Software Engineering Conference,* Garmisch, Germany, September 1993.

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

[19] Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A. Mining Version Histories to Guide Software Changes, *IEEE Trans. on Software Engineering*, vol. 31, no. 6, pp. 429-445, June, 2005.

[20] Zipf, G. K. Human Behavior and the Principle of Least Effort.Addison-Wesley, 1949.