Int J Syst Assur Eng Manag (November 2017) 8(Suppl. 2):S599–S616
DOI 10.1007/s13198-016-0479-2

ORIGINAL ARTICLE

An empirical study of software entropy based bug prediction using machine learning

Arvinder Kaur[1] · **Kamaldeep Kaur**[1] · **Deepti Chopra**[1]

Received: 29 April 2015 / Revised: 14 April 2016 / Published online: 18 May 2016

Abstract There are many approaches for predicting bugs in software systems. A popular approach for bug prediction is using entropy of changes as proposed by Hassan (2009). This paper uses the metrics derived using entropy of changes to compare five machine learning techniques, namely Gene Expression Programming (GEP), General Regression Neural Network, Locally Weighted Regression, Support Vector Regression (SVR) and Least Median Square Regression for predicting bugs. Four software subsystems: mozilla/layout/generic, mozilla/layout/forms, apache/httpd/modules/ssl and apache/httpd/modules/mappers are used for the validation purpose. The data extraction for the validation purpose is automated by developing an algorithm that employs web scraping and regular expressions. The study suggests GEP and SVR as stable regression techniques for bug prediction using entropy of changes.

Keywords Software bug prediction · Regression · Software entropy · Gene expression programming · General regression neural network · Locally weighted regression · Support vector regression · Least median square regression

✉ Deepti Chopra
dchopra27@gmail.com

[1] University School of Information and Communication Technology (U.S.I.C.T), Guru Gobind Singh Indraprastha University (G.G.S.I.P.U.), New Delhi, India

1 Introduction

Software bug prediction is an active and continuously evolving research field. Bug Prediction is defined as the act of identifying files or software code modules that are most likely to contain bugs before formal testing, so that that testing time and resources, can be allocated optimally. Only the files that are more likely to contain bugs should be tested more thoroughly. Accurate and reliable bug prediction can help software industry, as companies seek out ways to deliver extremely high quality software systems at lower costs of software quality assurance activities such as testing (Catal and Banerjee 2010). Other benefits of bug prediction models are that they can be used to identify refactoring candidates (Catal and Banerjee 2010), architectural improvements (Catal and Banerjee 2010) and selection of best design approaches (Catal and Banerjee 2010). Bug prediction is also useful for software project managers as it helps in quantitative planning and steering of projects (Ekanayake et al. 2012).

Bug prediction literature contains many seminal studies where bug prediction models are based on static code metrics (Agarwal 2009; Basili 1996; Gyimothy 2005). Basili et al. (1996) concluded that a relationship exists between bugs and Chidamber and Kemerer metrics. Their conclusions were based on studying eight medium sized systems. Aggarwal et al. (2009) concluded that import coupling and size metrics were related to bugs. Their conclusions were based on studying 12 software systems. Gyimothy et al. (2005) also found object oriented metrics to be influential in bug prediction. Their results were based on study of open source web software Mozilla. Despite existence of many seminal studies on empirical validation of code metrics (Agarwal 2009; Basili 1996; Gyimothy

2005) in bug prediction, code metrics based bug prediction has faced criticism from some researchers (Fenton and Ohlsson 2000). Fenton et al. (2007) constructed a bug prediction model based on project and process metrics. Moser et al. (2008) also showed that process metrics outperform code metrics in bug prediction. Graves et al. (2000), Khoshgoftaar et al. (1999) and Nagappan and Ball (2007) have shown that number of previous modifications to a file are good predictors of bugs. Radjenovic et al. (2013) conducted a systematic literature review of bug prediction metrics and found that object-oriented metrics were twice more popular as compared to traditional source code metrics or process metrics. They reported that object-oriented and process metrics were more successful in finding faults compared to traditional size and complexity metrics. Recently, Hassan (2009) introduced the concept of complexity of code change by applying information theory principles. He conceptualized that code change process of a software system can be viewed as a system that emits data, where he defined data as the feature introducing changes (FIC) to source code files. This allows concepts from Shannon's information entropy theory to be applied to quantify the complexity of code change. As per Hassan(2009), in a software system consisting of n source code files, if changes are monitored and it is found that the probability of modification of a single file(for example file1), is one and all other files is zero, then the complexity of code change or software entropy is minimum. On the other hand if the probability of modification of all files (file1….file n) is equal ($= 1/n$), then software entropy is maximum. Hassan (2009), further defined history complexity metrics (HCM) based on software entropy concepts and showed that bugs could be more accurately predicted using HCM as predictors. Hassan (2009) used statistical linear regression to build bug prediction models with HCM as predictors (Hassan 2009). Another conspicuous research direction in software bug prediction emphasizes that the selection of learning methods is very important to accurately predict software bugs (Menzies et al. 2007). Menzies et al. (2007) established a baseline experiment by utilizing rich developments in machine learning and data mining to demonstrate that the selection of a machine learning method greatly affect the accuracy of a defect prediction model. Later, Lessmann et al. (2008) extended Menzies et al.'s (2007) experiment and evaluated 22 machine learners on defects data sets of ten large scale NASA projects. However, the experiments by Menzies et al. (2007) and Lessmann et al. (2008) are based on static code metrics only. Very recently, Malhotra (2015) performed systematic literature review machine learning techniques for software fault/bug prediction and they concluded that the machine learning techniques had the ability to predict software bugs. They further conclude that more number of studies should be carried out in order to obtain well formed and generalizable results. Hassan (2009) did not evaluate any learning techniques in the context of entropy based bug prediction. Although a large number of machine learning techniques have been evaluated in static code metrics based bug prediction (Menzies et al. 2007; Lessmann et al. 2008; Malhotra 2014; Kaur and Kaur 2014), no comparative study is available in entropy based bug prediction. This motivates us to evaluate machine learning techniques in entropy based bug prediction. It is also imperative to note that the dependent or variable in static code metrics based bug prediction (Menzies et al. 2007; Lessmann et al. 2008; Malhotra 2014; Kaur and Kaur 2014) considered in most previous studies is binary. In this paper, the dependent or response variable is continuous. To the best of our knowledge there is only one study that considers the application of only a single machine learning technique, that is support vector regression in software entropy based bug prediction (Singh and Chaturvedi 2012). This motivates us to investigate various machine learning techniques in software entropy based bug prediction. Collection of data for empirical studies for bug prediction is another challenge. Therefore a tool for automatic data extraction is developed. Thus, the contribution of this paper is twofold:

(1) A tool for automatic data collection and classification of software changes is developed.
(2) Concept of complexity of code change as proposed by Hassan (2009) is used and performance of the following machine learning techniques for predicting bugs is compared:

- Gene Expression Programming (GEP)
- General Regression Neural Network (GRNN)
- Locally Weighted Regression (LWR)
- Support Vector Regression (SVR)
- Least Median Square Regression (LMSR)

The performance of these machine learning techniques is compared for two subsystems of Mozilla and two subsystems of Apache Http Server. The results are analyzed to arrive at general conclusions regarding the applicability of the techniques.

The rest of this paper is organized as follows: Sect. 2 presents an overview of related work in bug prediction. Section 3 describes the concept of Entropy of changes. The data extraction algorithm and metrics calculation is explained in Sect. 4. Section 5 describes the regression techniques that have been compared. Results are analyzed in Sect. 6 while Sect. 7 discusses the threats to validity. The study is finally concluded in Sect. 8.

2 Related work

Many Bug Prediction approaches have been developed by distinguished researchers. Mende and Koschke (2009) verified that a trivial defect prediction model such as large files are more prone to bugs performs well when a classic evaluation metric is used, but fails badly when an effort-aware performance metric is used. D'Ambros et al. (2012) have compared the performance of various such techniques. They developed a benchmark for bug prediction that includes process metrics, system metrics and defect history of five open source software projects: Eclipse JDT Core, Eclipse PDE UI, Equinox framework, Mylyn and Apache Lucene. The approaches are evaluated using a binary classification scenario, a ranking-based evaluation and an effort-aware ranking-based evaluation. Also, two effort-aware models were also evaluated and compared with a classical prediction model. Khoshgoftaar et al. (1996) used the number of past modifications to the module to predict bug-prone entities in the software system. It was concluded by them that the number of modifications in the past reliably predict future bugs. Nagappan and Ball (2005) also conducted a study on the influence of code churn or the number of changes to the system on the defect density. Their study was validated for Windows Server 2003 and it was found that relative code churn predicted better than absolute churn. Zhou and Leung (2006) built bug prediction models that could classify bugs according to two levels of severity-high and low. Later, Singh et al. (2010) used machine learning techniques to classify bugs in three severity levels-low, medium and high.

A lot of work has been done to determine the best techniques for prediction. Khoshgoftaar et al. (1997) applied neural networks for bug prediction using procedural static code metrics as predictor variables. Thwin and Quah (2005) applied generalized regression neural networks and used object-oriented static code metrics as predictor variables to predict bugs. Kanmani et al. (2007) used object-oriented static code metrics as predictor variables and applied two different kinds of neural network techniques for bug prediction. Menzies et al. (2007) conducted a study that suggested that the category of static source code metrics employed is not as important as the learning algorithm that is used for prediction. They used the datasets from NASA Metrics Data Program (MDP) to conclude this. Their study compared the impact of using various categories of software metrics like Halstead metrics, Lines of code, McCabe complexity metrics with the impact of using various learning algorithms like J48, Naive Bayes and OneR. Tosun et al. (2011) performed a study on bug prediction on embedded software projects using classifier ensembles and found that 70 % defects could be detected. Their study is based on static code metrics. Rodrigues et al. (2013) utilized static code metrics datasets from PROMISE (Menzies et al. 2016) data repository and bug prediction datasets developed by D'Ambros et al. (2012) and suggested an evolutionary subgroup based descriptive approach for defect prediction rather than the precise classification techniques. Malhotra (2014) performed comparative analysis of statistical and machine learning techniques for bug prediction using static code metrics. They found that decision tree method was better than logistic regression and other machine learning techniques. Okutan and Yildiz (2014) applied Bayesian networks in code and process metrics bug prediction and found that there was positive correlation between number of developers and level of defects. Dejaeger et al. (2013) applied fifteen different Bayesian network classifiers in static code metrics based bug prediction and found that they had better comprehensibility than other machine learning techniques. We have presented a brief review of machine learning techniques in bug prediction. There are three noteworthy systematic literature reviews on bug prediction (Catal 2011; Radjenovic et al. 2013; Malhotra 2015). Two significant observations from these reviews are:

- Most bug prediction studies use static code metrics as predictors or independent variables
- The dependent or response variable is binary in most bug prediction studies.

There are only a few studies (Afzal and Torkar 2008) where dependent variable is continuous but they use only one technique that is genetic programming.

Hassan (2009) introduced a novel concept of bug prediction by quantifying the complexity of code changes and using them to develop bug prediction models. He applied Shannon's information entropy principles to complexity of changes. He devised three code change models namely: Basic Code Change (BCC) Model, Extended Code Change (ECC) Model and File Code Change (FCC) Model. Entropy of changes is calculated using Shannon's entropy (Hassan, 2009). Hassan (2009) proposed a new entropy-based complexity metric which is termed as history complexity metric (HCM) and used it as independent or predictor variable for prediction of bugs. He concluded that history complexity metrics (HCM) predicted bugs more accurately than the code churn metrics and prior faults. Hassan (2009) developed bug prediction models using Statistical Linear Regression (SLR) techniques but did not evaluate any machine learning techniques in the context of entropy based bug prediction. Singh and Chaturvedi (2012) also employed the same concept of history complexity metrics to arrive at the conclusion that Support Vector

Regression performs better than Statistical Linear Regression (SLR). They have considered only one software system and one machine learning technique, but in our current study on software entropy based bug prediction, subsystems from two different software systems are considered and five machine learning techniques have been compared. We consider two subsystems of Mozilla and two subsystems of Apache Http Server and compare the performance of the five machine learning techniques for predicting bugs: Gene Expression Programming (GEP), General Regression Neural Network (GRNN) Locally Weighted Regression (LWR), Support Vector Regression (SVR), and Least Median Square Regression (LMSR).

3 Entropy of changes

Entropy of changes as proposed by Hassan (2009) is used to quantify the complexity of code changes. A software file is altered:

- when a bug is to be removed,
- when a new functionality is introduced,
- when some comments or coding standards are changed.

The changes that take place when a new functionality is introduced are the most complex type of changes. The complexity of such changes is what is quantified in terms of entropy of changes. This entropy of changes is then used to derive History Complexity Metrics (HCM) for predicting bugs.

3.1 Measurement of entropy

Entropy of changes for a period in a system/subsystem is calculated by the Shannon's Entropy formula specified in (1). The period is taken as 1 year for this study.

$$SE_n(P) = -\sum_{k=1}^{n} (P_k \times \log_2 P_k) \qquad (1)$$

where $P_k \geq 0$ and $\sum_{k=1}^{n} P_k = 1$

P_k is taken to be the probability of change for the *kth* file in the specified period i.e. the number of times *kth* file is modified divided by the total number of modifications. For example, let us assume as shown in Fig. 1, that there are 14 changes that occurred in four files and divided into three periods. For a first period, there are six changes that occurred across all four files. The probability of change occurrence for files F1, F2, F2, and F4 will be 2/6 (=0.33), 1/6 (=0.17), 1/6 (=0.17) and 2/6 (=0.33) respectively. These probabilities are also shown in Fig. 1. The value of Entropy for the first period is calculated as

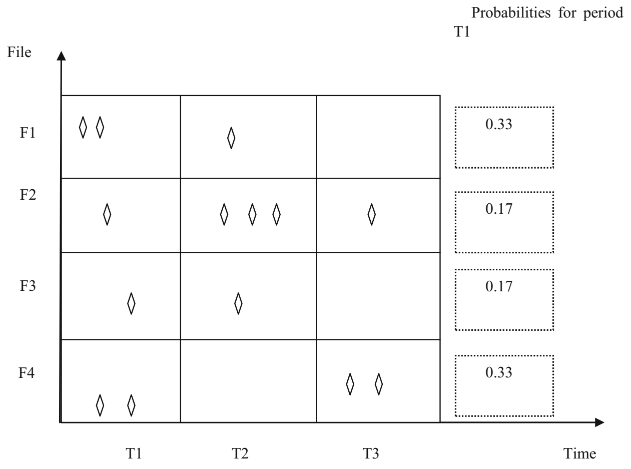


Fig. 1 Probability of change for a file in a specified time period

$$-(0.33 \times \log_2 0.33 + 0.17 \times \log_2 0.17 + 0.17 \times \log_2 0.17 + 0.33 \times \log_2 0.33) = 1.924819.$$

The Entropy is normalized using (2), so that it the entropy of subsystems that contain different number of files or totally different software systems can be compared easily.

$$SE(P) = \frac{1}{\text{Maximum Entropy}} \times SE_n(P) = -\frac{1}{\log_2 n} \times \sum_{k=1}^{n} (P_k \times \log_2 P_k) \qquad (2)$$

such that, $0 \leq SE \leq 1$ *SE* is the value of Normalized Entropy. For the example given in Fig. 1 the value of Normalized Entropy (SE) for the first period is calculated as $1.924819/\log_2 4 = 0.962409$.

3.2 History Complexity Metric

Entropy of Changes is then used to compute the History Complexity Metric (HCM). It is a measure for the effect of complexity of changes that is assigned to each file in the software subsystem/system. But, first the History Complexity Period Factor $HCPF_i(j)$ for a file *j* during period *i* is calculated using (3).

$$HCPF_i(j) = \left\{ \begin{matrix} C_{ij} \times SE_i, & j \in F_i \\ 0, & otherwise \end{matrix} \right\} \qquad (3)$$

where SE_i is the Entropy of changes for the system/subsystem over period *i* and C_{ij} is the portion of SE_i which is assigned to every file *j* that is modified in period SE_i. The definition of variants of C_{ij} is varied to arrive at the three variants of HCPF that are used in computing HCM. Figure 2 describes the three variants of HCM.

HCM1	HCM2	HCM3
$HCPF_i(j) = \begin{cases} C_{ij} * SE_i, & j \in F_i \\ 0, otherwise \end{cases}$ C_{ij}=1 For all files modified during the period *i* $HCM_{\{f,\ldots,g\}}(j) = \sum_{i \in \{f,\ldots,g\}} HCPF_i(j)$ $HCM_{\{f,\ldots,g\}}(S) = \sum_{j \in S} HCM_{\{f,\ldots,g\}}(j)$	$HCPF_i(j) = \begin{cases} C_{ij} * SE_i, & j \in F_i \\ 0, otherwise \end{cases}$ C_{ij}=P_j Where P_j is probability of change of file *j* over period *i* $HCM_{\{f,\ldots,g\}}(j) = \sum_{i \in \{f,\ldots,g\}} HCPF_i(j)$ $HCM_{\{f,\ldots,g\}}(S) = \sum_{j \in S} HCM_{\{f,\ldots,g\}}(j)$	$HCPF_i(j) = \begin{cases} C_{ij} * SE_i, & j \in F_i \\ 0, otherwise \end{cases}$ C_{ij}=1/NF_i where NF_i Is the number of files modified during the period *i* $HCM_{\{f,\ldots,g\}}(j) = \sum_{i \in \{f,\ldots,g\}} HCPF_i(j)$ $HCM_{\{f,\ldots,g\}}(S) = \sum_{j \in S} HCM_{\{f,\ldots,g\}}(j)$

Fig. 2 Variants of HCM

In the example given in Fig. 1 the HCPF calculated for file 1in the first period is different for the three variants of HCM. For HCM1 the HCPF calculated is 1 × 0.962409 = 0.962409, for HCM2 the HCPF is 0.33 × 0.962409 = 0.317595 whereas for HCM3 the HCPF is 1/4 × 0.962409 = 0.240602. The *HCM* for a file *j* for the evolution period set *{f,…,g}* is defined as

$$HCM_{\{f,\ldots,g\}}(j) = \sum\nolimits_{i \in \{f,\ldots,g\}} HCPF_i(j) \quad (4)$$

Similarly, HCM for a system/subsystem S for the evolution period set *{f,…,g}* is the sum of HCMs for all files in that subsystem as specified in (5).

$$HCM_{\{f,\ldots,g\}}(S) = \sum\nolimits_{j \in S} HCM_{\{f,\ldots,g\}}(j) \quad (5)$$

4 Empirical data

Data Extraction is the most crucial step in any empirical study. It is very important to correctly collect the data from software repositories for predicting bugs. The prediction is accurate only if the data collected is reliable and does not contain errors. The following subsections describe the software subsystems used for this study, the development of the tool used for data extraction and how the metrics are calculated.

4.1 Data sources

In this study, we extract bugs data for two subsystems each of Mozilla and Apache Http Server for evaluating the performance of machine learning techniques in software entropy based bug prediction. Mozilla is a popular web browser whereas Apache Http Server is a popular web server. The data for the subsystems of Mozilla is extracted from Mozilla-central which is a Mercurial repository of Mozilla project. On the other hand GitHub, a web-based hosting service for Git repositories is used to extract the data for Apache Http Server. After extracting the year-wise bugs and changes for each file in each subsystem, the entropy and History Complexity Metric (HCM) are calculated for a time period of each year using Eqs. (1)–(5). The number of changes and Normalized Entropy for each year for the four subsystems is given in Table 2.

Since manual data collection is a tedious process and prone to errors, we develop a tool for automatic data collection as explained in next Sect. 4.2

4.2 Automating data extraction

The first step in data extraction is to browse the revision history of each file and record the year-wise bugs and changes. A change record typically defines the reason for the change along with date of change, name of the committer, lines of code added/deleted/modified etc. These changes are categorized as follows:

- Bug Repairing Changes (BRC): These are the changes that take place when a bug is to be corrected.
- Functionality Introducing Changes (FIC): These changes take place whenever new functionality or a feature needs to be added in the software system.

Fig. 3 Algorithm for data extraction

```
1. Begin
2. Input the URL of the repository page
3. Set Bug Repairing Changes (BRC), Functionality Introducing Changes (FIC) and General
   Changes (GC) count to zero.
4. For year in considered time period
      a. Get content from the specified URL.
      b. While regular expression pattern is found
            i. Assign variables desc and date the value of extracted reason of change and
               date of commit.
            ii. If date contains year
                  I. if desc contains any of the following substrings:
                     "bug","patch","reintroduce","issue,"revert","fix","fault".
                     then increment BRC.
                  II. else if desc contains any of the following substrings:
                     "introduce","implement","add","new"","merge".
                     then increment FIC.
                  III. else increment GC.
5. Print value of year, BRC, FIC and GC.
6. Set BRC, FIC and GC to zero.
7. End for.
8. End.
```

- General Changes (GC): These changes are maintenance related and do not add any feature or rectify a bug. This class of changes includes changes in comments, copyrights, formatting changes etc.

Feature introducing changes (FIC) are those that introduce new features or enhance existing features and are related to adaptive maintenance of a software system. FIC cause modifications to software code which may be highly scattered in large number of source code files and modules and cause code decay. This code decay is responsible for increase in overall complexity and thus FIC are used for calculating entropy of changes (Hassan 2009). General Changes (GC) are not related to introduction of new features (Hassan 2009). Some examples of general changes are like addition of authors in comments, update of copyright notice in comments or re-indentation of source code to make it more readable. Thus general changes are localized and do not lead to code decay. Bug Repairing changes (BRC) are logically related to the number of faults or bugs in a file or module of source code. Thus, BRCs are used for validation of results. This methodology is consistent with the methodology established by Hassan (2009).

Change classification is automated by developing a Java application named Change Classifier, which inputs the URL of the repository, the repository name and classifies change records listed on that page. The output generated is year-wise frequency of each type of change. Web scraping and regular expressions are used for the purpose of extracting the date and reason of change. A regular expression is a text string used for pattern matching. Two regular expressions are formulated, one for Mozilla-Central repository and the other for GitHub Repository. The main reason behind writing different regular expression for GitHub and Mozilla-Central is that the website of these repositories are structured and formatted differently hence requiring a different regular expression for pattern matching. The regular expression used for Mozilla-Central in the Change Classifier is:

expr = “<td class =\“age\”> .*? <i>([^ <]+) </i> .*? <td> (.*?) </strong > </td>”

The regular expression used for GitHub in the Change Classifier is:

expr = “<a href = .*?class =\“message\”.*?title = (.*?)> .*? <time datetime = [^ >]*> (.*?) </time>”

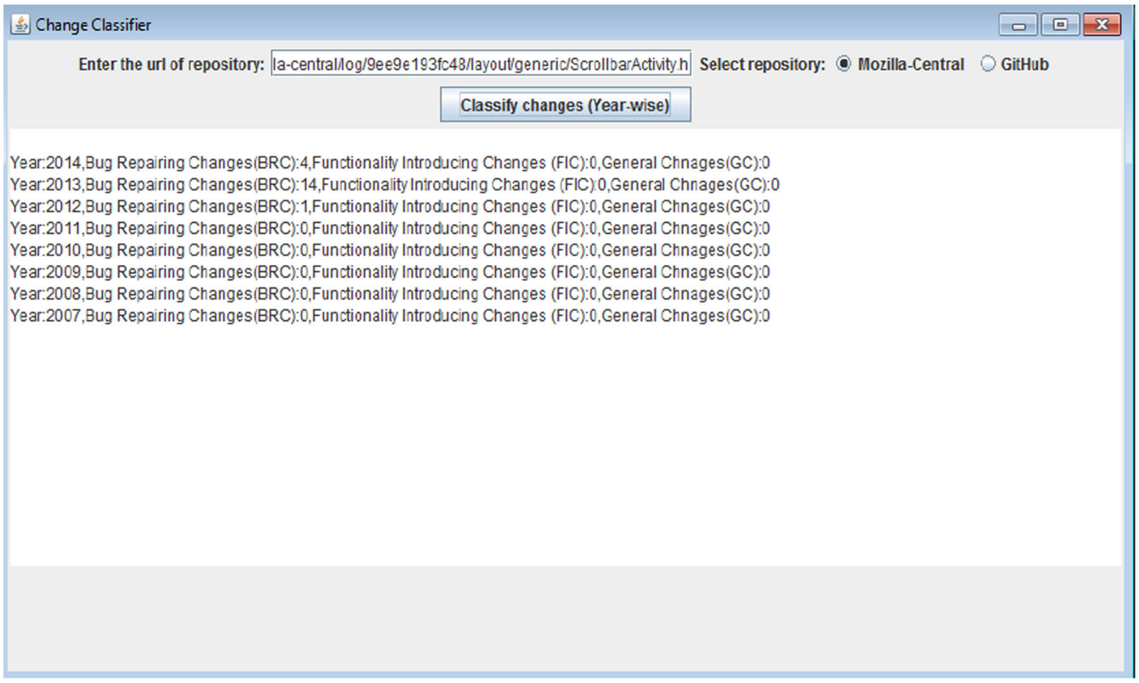
The content matching the highlighted expression is extracted. After extracting the date and reason of change, the changes are classified as specified in the algorithm given in Fig. 3. The output of Change Classifier for Mozilla-Central and GitHub is as shown in Figs. 4 and 5.

4.3 Metrics calculation

After extracting the year-wise bugs and changes for each file in the subsystem, the entropy and History Complexity Metrics are calculated for each year using Eqs. (1)–(5). The number of changes and Normalized Entropy for each year for the four subsystems is given in Table 2.

4.4 Independent and dependent variables

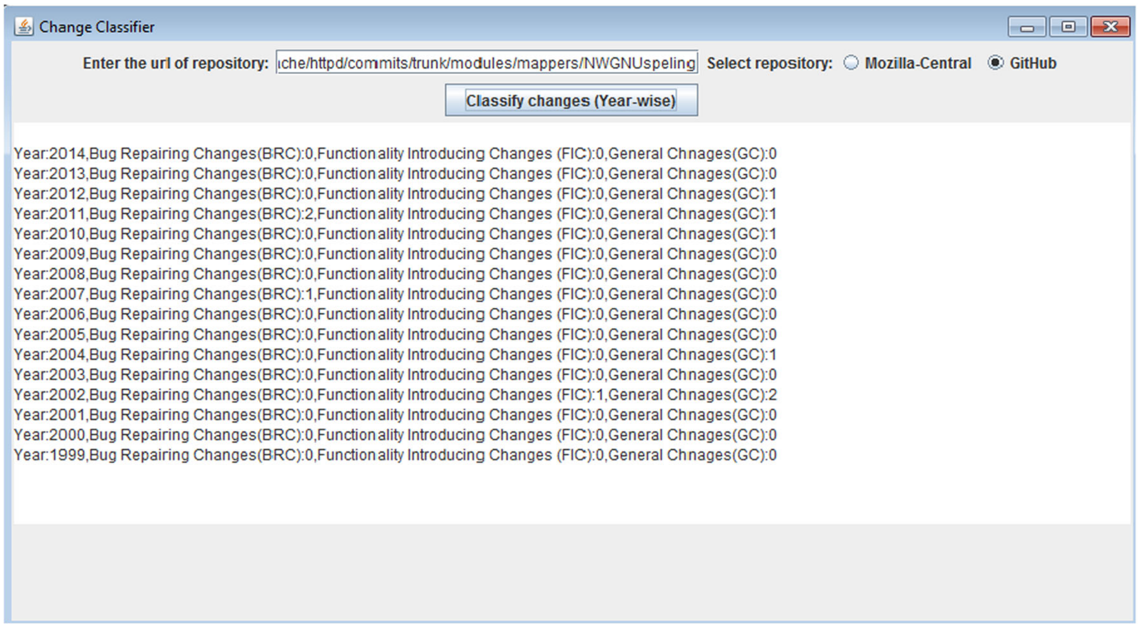
In this study bug prediction models are built using data of History Complexity Metrics (HCM1, HCM2 and HCM3) as independent variables and number of bugs as dependent



Enter the url of repository: la-central/log/9ee9e193fc48/layout/generic/ScrollbarActivity.h Select repository: [✓ Mozilla-Central] GitHub

```
Year:2014,Bug Repairing Changes(BRC):4,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2013,Bug Repairing Changes(BRC):14,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2012,Bug Repairing Changes(BRC):1,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2011,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2010,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2009,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2008,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2007,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
```

Fig. 4 Output of change classifier for Mozilla-Central



Enter the url of repository: iche/httpd/commits/trunk/modules/mappers/NWGNUspeling Select repository: Mozilla-Central [✓ GitHub]

```
Year:2014,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2013,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2012,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):1
Year:2011,Bug Repairing Changes(BRC):2,Functionality Introducing Changes (FIC):0,General Chnages(GC):1
Year:2010,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):1
Year:2009,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2008,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2007,Bug Repairing Changes(BRC):1,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2006,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2005,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2004,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):1
Year:2003,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2002,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):1,General Chnages(GC):2
Year:2001,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:2000,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
Year:1999,Bug Repairing Changes(BRC):0,Functionality Introducing Changes (FIC):0,General Chnages(GC):0
```

Fig. 5 Output of change classifier for GitHub

variable. The dependent variable is continuous in this study. The calculations of History Complexity Metrics (HCM1, HCM2 and HCM3) are explained in Sect. 3. The bug prediction models are built using five machine learning based regression techniques. The accuracy of five machine learning techniques is evaluated on the data sets of five software subsystems described in Table 1. For each of the five software subsystems mentioned in Table 1, bug prediction models are built using three HCM metrics as predictors

Table 1 Software subsystems for evaluation

Software system	Application type	Programming Language	Subsystem	Repository URL	Number of files
Mozilla	Web browser	C++	mozilla/layout/generic	http://hg.mozilla.org/mozilla-central/file/9ee9e193fc48/layout/generic	132
			mozilla/layout/forms	http://hg.mozilla.org/mozilla-central/file/9ee9e193fc48/layout/forms	43
Apache Http server	Web server	C	apache/httpd/modules/ssl	https://github.com/apache/httpd/tree/trunk/modules/ssl	34
			apache/httpd/modules/mappers	https://github.com/apache/httpd/tree/trunk/modules/mappers	38

Table 2 Normalized entropy and number of changes

Year	Mozilla/layout/generic		Mozilla/layout/forms		Apache/httpd/modules/ssl		Apache/httpd/modules/mappers	
	Changes	Entropy	Changes	Entropy	Changes	Entropy	Changes	Entropy
1999	–	–	–	–	–	–	30	0.725
2000	–	–	–	–	–	–	40	0.591
2001	–	–	–	–	119	0.796	41	0.631
2002	–	–	–	–	120	0.678	35	0.479
2003	–	–	–	–	17	0.592	30	0.541
2004	–	–	–	–	32	0.671	14	0.344
2005	–	–	–	–	17	0.615	4	0.381
2006	–	–	–	–	13	0.553	8	0.458
2007	138	0.849	37	0.896	25	0.734	4	0.286
2008	73	0.633	3	0.292	38	0.639	8	0.202
2009	58	0.558	3	0.169	33	0.645	6	0.341
2010	61	0.556	2	0.184	41	0.682	20	0.442
2011	504	0.850	85	0.818	72	0.182	20	0.504
2012	1005	0.821	220	0.816	37	0.637	5	0.137
2013	22	0.546	3	0.169	30	0.669	2	0.191

5 Machine learning based regression techniques

Regression analysis is a statistical process used for estimating the relationships among variables. There are many different types of techniques that model and analyze variables to derive a relationship between a target variable that is dependent on one or more independent predictors. Performance of some of these regression techniques for predicting bugs using Entropy-based metrics have been compared in this study. The following subsections describe the regression techniques which are being compared in this study.

5.1 Gene expression programming (GEP)

Gene Expression Programming (GEP) (Ferreira 2001) is a procedure based on biological evolution. It creates a computer program for modeling some phenomenon. The different types of models that can be created by GEP include neural networks, decision trees and polynomial constructs. A simplified representation of GEP algorithm is given in Fig. 6.

GEP is similar to Genetic Algorithms (GA) and Genetic Programming (GP) as it uses a population of individuals, computes their fitness to select them and introduces genetic variations by using genetic operators such as mutation, transposition, recombination etc. But the basic difference between the three is that:

- in GA the individuals are linear strings having fixed length.
- in GP the individuals are non-linear entities having different sizes and shapes.

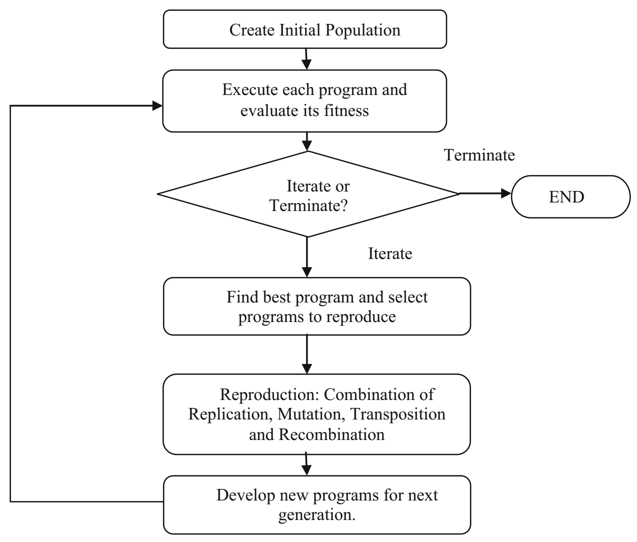

Create Initial Population
Execute each program and evaluate its fitness
Iterate or Terminate?
Terminate
END
Iterate
Find best program and select programs to reproduce
Reproduction: Combination of Replication, Mutation, Transposition and Recombination
Develop new programs for next generation.

Fig. 6 Simplified algorithm of GEP

- in GEP the individuals are encoded using linear strings having fixed length which are later expressed using non-linear entities of different sizes and shapes.

DTREG (Predictive Modeling Software) tool is used to implement GEP. The type of GEP modeled by DTREG is Symbolic Regression. Symbolic Regression is a subset of non-parametric regression in which the form of the function to be fitted is not given in advance but the function is restricted to be mathematical or logical expressions. It is the goal of the procedure to find the function that best fits the data.

The goal of GEP for Symbolic Regression is to find the expression that performs well for all fitness cases within a certain minimum error of the correct target value. An evolutionary strategy is used for discovering a very good solution without halting the evolution process. So, the system finds the best possible solution within minimum error. The founder individuals are very unfit but their modified descendents are reshaped by selection and then the population adapts wonderfully by finding better solutions that ultimately approach a perfect solution. The fitness f_i of a program i is calculated using (6) if absolute error is considered or by (7) if relative error is considered.

$$f_i = \sum_{j=1}^{C_t} \left(M - \left| C_{i,j} - T_j \right| \right) \tag{6}$$

$$f_i = \sum_{j=1}^{C_t} \left(M - \left| \frac{C_{i,j} - T_j}{T_j} .100 \right| \right) \tag{7}$$

where M is the range of selection, $C_{i,j}$ is the value returned by individual i for fitness case j out of total C_t fitness cases and T_j is the target value for fitness case j.

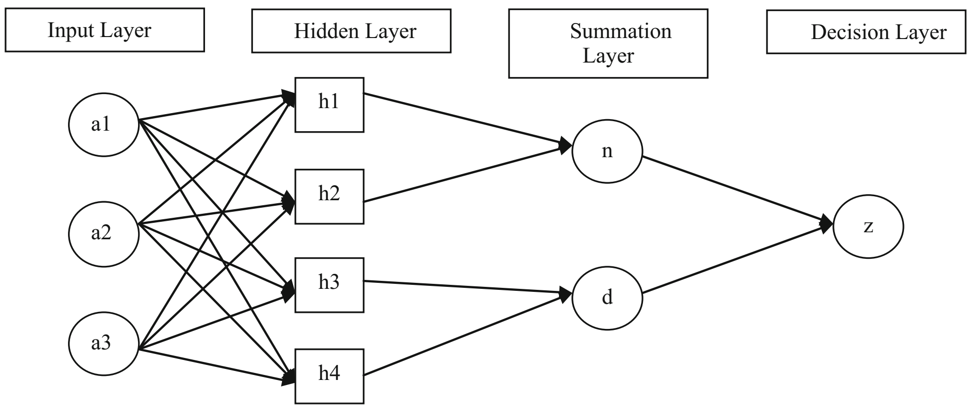
5.2 General regression neural network (GRNN)

The general regression neural network (GRNN) (Specht 1991) is a memory-based network that provides estimates for continuous target variable. It is a single-pass learning algorithm having a highly parallel structure. The architecture of GRNN is as shown in Fig. 7.

GRNN consists of the following four layers:

1. *Input layer* The input layer consists of one neuron per predictor. The input neuron standardizes the values of input variables by subtracting the median and then dividing the result by the interquartile range. Then these values are fed into the neurons of the hidden layer.
2. *Hidden layer* There is one neuron for each case of training data set in the hidden layer. Each neuron stores the value of the predictors for the case along with the value of target variable. This hidden neuron when given a vector of input values from the input layer, calculates the Euclidean distance of the test case from the center point of the neuron and then applies the

Fig. 7 GRNN architecture



kernel function to compute its weightage. This value is passed into the summation layer.

3. *Summation layer* Summation layer consists of only two neurons, namely the denominator summation unit and the numerator summation unit. The denominator summation unit sums up the weightage value calculated by each of the hidden neuron. While the numerator summation unit sums up the product of weightage values and the actual value of the target variable for each of the hidden neuron.
4. *Decision layer* The decision layer calculates the predicted value of target variable by dividing the value calculated in the numerator summation unit by the value calculated in the denominator summation unit.

GRNN is implemented using DTREG (Predictive Modeling Software) tool. The tool provides a choice of two kernel functions: Gaussian and Reciprocal. The performance of GRNN for predicting bugs using Entropy of changes is recorded for both kernel functions.

5.3 Locally weighted regression (LWR)

Locally Weighted Learning (LWL) (Atkeson et al. 1997) employs a lazy learning method since the processing is deferred unless a query needs to be answered. This is a local method in the sense that it attempts to fit the training data only about the query point. The weights for the training instances are calculated using a distance function. The nearby points have greater weight. The weighting function is also called the Kernel function (K).

In general, there are two methods of weighting:

- *Weighting the error criterion* In this method the error criterion is assigned weights. The aim is to minimize the error criterion given in (8).

$$C(q) = \sum_i (\hat{y}_i - y_i)^2 K(d(x_i, q)) \tag{8}$$

- *Direct Data Weighting* In this method the weights are directly assigned to the training data using the Kernel function as specified in (9).

$$\hat{y}(q) = \frac{\sum y_i K(d(x_i, q))}{\sum K(d(x_i, q))} \tag{9}$$

where, x_i is the *ith* input vector, y_i is the *ith* training data and $d(x_i, q)$ is the distance function.

The tool used to implement LWR is Weka (Witten et al. 2011). Weka provides six kernel functions for LWR namely, Linear, Epanechnikov, Tricube, Inverse-distance, Gaussian and Constant weighting. The performance of LWR for predicting bugs using Entropy of changes is analyzed for all six kernel functions.

5.4 Support vector regression (SVR)

The goal of SVR is to estimate the function *f(x)* specified in (10) for the training dataset $\{x_i, d_i\}$where x_i is the *ith* input vector and d_i is the *ith* target value, such that it predicts the actual target value as closely as possible and is also as flat as possible in order to provide good generalization.

$$f(x) = w.\varphi(x) + b \tag{10}$$

where, b denotes the bias and w denotes the coefficient vector. Also, $z = \varphi(x)$ specifies the feature space vector. All computations are done using the Kernel function defined in (11)

$$K(x, \hat{x}) = \varphi(x).\varphi(\hat{x}) \tag{11}$$

where · represents the dot product in feature space. The primal optimization problem of SVR given ε-intensive loss function is to minimize Eq. (12).

$$\frac{1}{2}w^2 + C\sum\left(\xi_i + \hat{\xi}_i\right) \tag{12}$$

such that: $d_i - w.z_i - b \le \varepsilon + \xi_i$, $w.z_i + b - d_i \le \varepsilon + \hat{\xi}_i$ and $\xi_i, \hat{\xi}_i \ge 0$.

It is difficult to solve the primal optimization problem due to the fact z and w are infinite-dimensional. Hence, a finite-dimensional optimization known as the dual optimization problem is defined as in (13) using Lagrange multipliers (α_i, .$\hat{\alpha}_i$).

$$\text{Maximize } \sum_i d_i(\alpha_i - \hat{\alpha}_i) - \varepsilon \sum_i \alpha_i + \hat{\alpha}_i - \frac{1}{2} w(\alpha, \hat{\alpha}_i)^2 \tag{13}$$

where, $w(\alpha, \hat{\alpha}_i) = \sum_i (\alpha_i - \hat{\alpha}_i) z_i$ such that: $\sum_i (\alpha_i - \hat{\alpha}_i) = 0$ and α_i, .$\hat{\alpha}_i \in [0, C]$ for each *i*. *C* denotes the coefficient of smoothness.

Weka (Witten et al. 2011) is used to implement SVR. It uses SMOReg to implement SVR, the explanation of which is given by Shevade et al. (2000). Weka provides four different kernels for numeric data in SVR: PolyKernel, NormalizedPolyKernel, Puk and RBFKernel and which have been used to perform the analysis.

5.5 Least median square regression (LMSR)

Least Median Square Regression (LMSR) generates functions from random samples of data. The final model is the least squared regression with the lowest median squared error. Consider the data generation process given by (14).

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i \tag{14}$$

where ε_i is independently and identically distributed. So when a realization of n observations is given in X, Y pairs called the sample, the aim is to estimate the parameters β_0 and β_1. This is done by fitting a line to the observations in the sample. The fitted line's intercept is β_0 and slope is β_1. Least Square fits the line by finding the intercept and slope that minimizes the sum of squared residuals. This technique is also implemented using Weka (Witten et al. 2011) based on the algorithm given by Leroy and Rousseeu (1987).

6 Result analysis

This section presents the result of performance of the machine learning techniques described in the previous section in software entropy based bug prediction. The machine learning techniques are compared for each of the four selected subsystems and then general conclusions are derived.

Performance of machine learning techniques is compared using Mean Absolute Error (MAE) and Root Mean Square Error (RMSE). These measures are defined in Witten et al. (2011) as follows:

- Mean Absolute Error (MAE):

$$\frac{|p_1 - a_1| + |\cdots| + |p_n - a_n}{n} \tag{15}$$

- Root Mean Square Error (RMSE):

$$\sqrt{\frac{(p_1 - a_1)^2 + \cdots + (p_n - a_n)^2}{n}} \tag{16}$$

where p_n predicted number of bugs and a_n actual number of bugs.

The results of the regression techniques for each subsystem are presented in Tables 3, 4, 5 and 6. The best results for each technique are highlighted.

Table 3 presents the results for the mozilla/layout/generic subsystem. The best cases are compared for each of the techniques. The best case MAE and RMSE values are plotted in Fig. 8. It is noticed that SVR gives least MAE (125.089) followed by LWR and then LMSR, GEP and GRNN respectively. Also based on RMSE the best results are obtained by LWR followed by SVR, GEP, LMSR and GRNN in that order.

The results for the mozilla/layout/forms subsystem are presented in Table 4. The best cases are compared for each of the techniques. It is noticed that LMSR gives least MAE (65.518) followed by GEP, SVR, GRNN and LWR respectively. The least RMSE is observed for LMSR (95.694) followed by SVR, GEP, GRNN and LWR respectively. These best case MAE and RMSE values are plotted in Fig. 9.

Table 5 lists out the results for the apache/httpd/modules/ssl subsystem. The best cases are compared for each of the techniques. It is observed that GEP gives least MAE (12.995) followed by LWR, SVR, GRNN and LMSR in that order. The least value of RMSE is obtained using GEP (19.039) followed by GRNN, LWR, SVR and LMSR in that order. These best case MAE and RMSE values are plotted in Fig. 10.

Table 6 shows the results for the apache/httpd/modules/mappers subsystem. The best cases are compared for each of the techniques. The best case MAE and RMSE values are plotted in Fig. 11.The least values of both MAE and RMSE are obtained using GRNN (MAE = 10.343 and RMSE = 12.951) followed by SVR, LWR, GEP and LMSR in that order.

Although it is difficult to conclude which machine learning based regression technique performs best it is noticed LMSR performs worst for both the Apache Http Server subsystems. Also the techniques that give adequate performance for all the subsystems are GEP and SVR. Thus, it is our suggestion to employ GEP and SVR for predicting bugs using Entropy of changes.

7 Threats to validity

Empirical studies in bug prediction are subject to factors that affect the correctness of results. These are called threats to validity. Broadly there are two kinds of threats-internal and external validity threats. Threats to internal validity occur if there is misinterpretation of true causes that affect the experimental results. External validity refers to ability to generalize results of a study. For this purpose, we include multiple data sets extracted from four open-source software subsystems and performing empirical analysis on them. This study does not take into consideration the bugs that are reported but not corrected while counting the year-wise number of bugs. But since there are hardly any bugs that are not fixed, the results of this study stand justified. Also the tool we developed and deployed, matches keyword substrings in the extracted reason of change from the repository, and hence it does not apply any human like reasoning for classification of the change. The tool uses a simple, but robust keyword matching algorithm to classify the changes with least possible chances of misclassification. Since the number of studies on applicability of machine learning in entropy based bug prediction is very few, it is suggested that more studies be carried out on various application domain, industrial settings and programming languages to obtain better generalization of results. Another threat to validity is that the four software

Table 3 Results for mozilla/layout/generic

Regression technique	Parameters	Kernel function	Metric	MAE	RMSE
Gene Expression Programming (GEP)	Population size = 50	–	HCM1	**163.405**	**178.449**
	Gene per chromosome = 4		HCM2	217.889	268.594
	Generations required to train the model = 246		HCM3	236.283	261.558
	Generations required for simplification = 29				
	Gene head length = 8				
General Regression Neural Network (GRNN)	Min sigma = 0.0001	Gaussian	HCM1	**175.377**	**201.549**
	Max sigma = 10		HCM2	187.563	213.904
	Number of search steps = 20		HCM3	194.614	250.483
		Reciprocal	HCM1	268.639	312.131
			HCM2	227.751	262.109
			HCM3	227.658	262.064
Locally Weighted Regression (LWR)	Classifier = decision stump Search algorithm = nearest neighbor	Linear	HCM1	135.910	177.004
			HCM2	136.630	171.991
			HCM3	136.623	171.974
		Epanechnikov	HCM1	136.802	178.251
			HCM2	135.159	172.05
			HCM3	135.149	172.025
		Tricube	HCM1	136.292	178.071
			HCM2	132.311	166.870
			HCM3	**132.305**	**166.847**
		Inverse-distance	HCM1	137.006	177.976
			HCM2	138.705	177.556
			HCM3	138.704	177.555
		Gaussian	HCM1	136.962	178.344
			HCM2	136.793	174.941
			HCM3	136.792	174.933
		Constant weighting	HCM1	137.952	179.348
			HCM2	137.952	179.348
			HCM3	137.952	179.348
Support Vector Regression (SVR)	Complexity parameter c = 1.0	Poly	HCM1	**125.089**	**173.618**
	Cache size = 250,007		HCM2	186.815	199.650
	Exponent value = 1.0		HCM3	186.708	199.507
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	NormalizedPoly	HCM1	687.686	719.967
	Cache size = 250,007		HCM2	591.696	656.849
	Exponent value = 1.0		HCM3	682.933	718.99
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	Puk	HCM1	172.609	196.124
	Cache size = 250,007		HCM2	228.223	260.065
	Exponent value = 1.0		HCM3	227.806	259.581
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	RBF	HCM1	577.473	604.443
	Cache size = 250,007		HCM2	579.861	606.200
	Exponent value = 1.0		HCM3	579.852	606.191
	Epsilon = 1.0E−12				
	Gamma = 0.01				
Least Median Square Regression (LMSR)	Random seed = 0	–	HCM1	274.021	309.012
	Sample size = 4		HCM2	152.085	180.249
			HCM3	**151.812**	**180.025**

Table 4 Results for mozilla/layout/forms

Regression technique	Parameters	Kernel function	Metric	MAE	RMSE
Gene Expression Programming (GEP)	Population size = 50	–	HCM1	88.178	**112.841**
	Gene per chromosome = 4		HCM2	85.309	122.702
	Generations required to train the model = 246		HCM3	**85.181**	122.423
	Generations required for simplification = 29				
	Gene head length = 8				
General Regression Neural Network (GRNN)	Min sigma = 0.0001	Gaussian	HCM1	91.412	**114.228**
	Max sigma = 10		HCM2	97.939	123.631
	Number of search steps = 20		HCM3	97.939	123.631
		Reciprocal	HCM1	99.934	121.089
			HCM2	92.615	122.551
			HCM3	**88.923**	119.178
Locally Weighted Regression (LWR)	Classifier = decision stump	Linear	HCM1	133.576	144.736
	Search algorithm = nearest neighbor		HCM2	116.176	135.249
			HCM3	116.176	135.249
		Epanechnikov	HCM1	137.331	146.672
			HCM2	117.165	135.673
			HCM3	117.165	135.673
		Tricube	HCM1	133.703	143.335
			HCM2	131.320	141.551
			HCM3	131.320	141.551
		Inverse-distance	HCM1	117.189	136.183
			HCM2	102.299	118.591
			HCM3	102.299	118.591
		Gaussian	HCM1	116.047	135.004
			HCM2	102.700	118.810
			HCM3	102.700	118.810
		Constant weighting	HCM1	114.786	134.520
			HCM2	**101.286**	**117.456**
			HCM3	**101.286**	**117.456**
Support Vector Regression (SVR)	Complexity parameter c = 1.0	Poly	HCM1	104.146	124.597
	Cache size = 250,007		HCM2	87.298	111.946
	Exponent value = 1.0		HCM3	**87.298**	**111.945**
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	NormalizedPoly	HCM1	186.019	204.127
	Cache size = 250,007		HCM2	204.724	216.43
	Exponent value = 1.0		HCM3	201.848	219.735
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	Puk	HCM1	92.948	121.102
	Cache size = 250,007		HCM2	89.323	119.047
	Exponent value = 1.0		HCM3	89.457	119.097
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	RBF	HCM1	168.285	195.022
	Cache size = 250,007		HCM2	169.196	195.566
	Exponent value = 1.0		HCM3	169.196	195.566
	Epsilon = 1.0E−12				
	Gamma = 0.01				
Least Median Square Regression (LMSR)	Random seed = 0	–	HCM1	84.161	110.690
	Sample size = 4		HCM2	**65.518**	**95.694**
			HCM3	**65.518**	**95.694**

Table 5 Results for apache/httpd/modules/ssl

Regression technique	Parameters	Kernel function	Metric	MAE	RMSE
Gene Expression Programming (GEP)	Population size = 50	–	HCM1	18.283	24.432
	Gene per chromosome = 4		HCM2	**12.995**	**19.039**
	Generations required to train the model = 246		HCM3	**12.995**	**19.039**
	Generations required for simplification = 29				
	Gene head length = 8				
General Regression Neural Network (GRNN)	Min sigma = 0.0001	Gaussian	HCM1	24.564	39.526
	Max sigma = 10		HCM2	23.270	42.942
	Number of search steps = 20		HCM3	24.443	**39.007**
		Reciprocal	HCM1	24.198	44.652
			HCM2	**22.207**	42.622
			HCM3	24.289	43.137
Locally Weighted Regression (LWR)	Classifier = decision stump	Linear	HCM1	24.049	39.886
	Search algorithm = nearest neighbor		HCM2	24.046	39.788
			HCM3	24.046	39.788
		Epanechnikov	HCM1	23.771	39.856
			HCM2	23.991	39.877
			HCM3	23.991	39.877
		Tricube	HCM1	23.857	39.817
			HCM2	23.794	39.819
			HCM3	23.794	39.819
		Inverse-distance	HCM1	22.105	39.334
			HCM2	22.084	**39.323**
			HCM3	22.084	**39.323**
		Gaussian	HCM1	22.083	39.407
			HCM2	**22.036**	39.399
			HCM3	**22.036**	39.399
		Constant weighting	HCM1	22.392	39.368
			HCM2	22.392	39.368
			HCM3	22.392	39.368
Support Vector Regression (SVR)	Complexity parameter c = 1.0	Poly	HCM1	24.926	50.754
	Cache size = 250,007		HCM2	24.873	50.072
	Exponent value = 1.0		HCM3	24.873	50.072
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	NormalizedPoly	HCM1	25.448	49.259
	Cache size = 250,007		HCM2	25.172	49.734
	Exponent value = 1.0		HCM3	25.172	49.734
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	Puk	HCM1	23.565	44.759
	Cache size = 250,007		HCM2	**22.101**	**43.704**
	Exponent value = 1.0		HCM3	**22.101**	**43.704**
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	RBF	HCM1	24.898	49.639
	Cache size = 250,007		HCM2	24.958	49.628
	Exponent value = 1.0		HCM3	24.958	49.628
	Epsilon = 1.0E−12				
	Gamma = 0.01				
Least Median Square Regression (LMSR)	Random seed = 0	–	HCM1	**27.039**	**52.581**
	Sample size = 4		HCM2	27.204	52.686
			HCM3	27.204	52.686

Table 6 Results for apache/httpd/modules/mappers

Regression technique	Parameters	Kernel function	Metric	MAE	RMSE
Gene Expression Programming (GEP)	Population size = 50	–	HCM1	**14.432**	**17.135**
	Gene per chromosome = 4		HCM2	16.388	21.752
	Generations required to train the model = 246		HCM3	17.220	22.571
	Generations required for simplification = 29				
	Gene head length = 8				
General Regression Neural Network (GRNN)	Min sigma = 0.0001	Gaussian	HCM1	11.061	14.477
	Max sigma = 10		HCM2	11.479	13.931
	Number of search steps = 20		HCM3	**10.343**	**12.951**
		Reciprocal	HCM1	11.160	15.799
			HCM2	13.214	16.994
			HCM3	14.834	18.378
Locally Weighted Regression (LWR)	Classifier = decision stump	Linear	HCM1	11.253	14.840
	Search algorithm = nearest neighbour		HCM2	11.689	15.381
			HCM3	11.689	15.381
		Epanechnikov	HCM1	11.457	15.132
			HCM2	11.803	15.565
			HCM3	11.803	15.565
		Tricube	HCM1	**11.173**	**14.806**
			HCM2	11.830	15.567
			HCM3	11.830	15.567
		Inverse-distance	HCM1	11.598	15.317
			HCM2	11.711	15.487
			HCM3	11.711	15.487
		Gaussian	HCM1	11.561	15.270
			HCM2	11.793	15.579
			HCM3	11.793	15.579
		Constant weighting	HCM1	11.831	15.702
			HCM2	11.831	15.702
			HCM3	11.831	15.702
Support Vector Regression (SVR)	Complexity parameter c = 1.0	Poly	HCM1	13.268	15.393
	Cache size = 250,007		HCM2	17.534	21.052
	Exponent value = 1.0		HCM3	17.534	21.052
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	NormalizedPoly	HCM1	22.776	34.350
	Cache size = 250,007		HCM2	23.449	34.922
	Exponent value = 1.0		HCM3	24.955	35.264
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	Puk	HCM1	**10.734**	**14.526**
	Cache size = 250,007		HCM2	12.079	15.391
	Exponent value = 1.0		HCM3	12.084	15.401
	Epsilon = 1.0E−12				
	Complexity parameter c = 1.0	RBF	HCM1	23.437	33.468
	Cache size = 250,007		HCM2	23.222	33.375
	Exponent value = 1.0		HCM3	23.222	33.375
	Epsilon = 1.0E−12				
	Gamma = 0.01				
Least Median Square Regression (LMSR)	Random seed = 0	–	HCM1	25.211	**30.760**
	Sample size = 4		HCM2	**24.046**	32.172
			HCM3	**24.046**	32.172

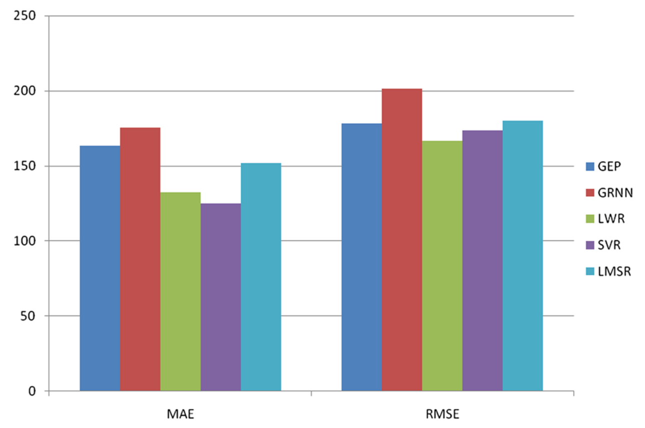


Fig. 8 Best case MAE and RMSE values for mozilla/layout/generic subsystem

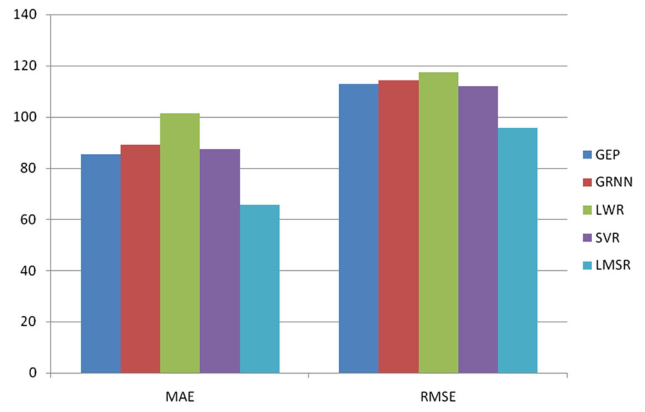


Fig. 9 Best case MAE and RMSE values for mozilla/layout/forms subsystem

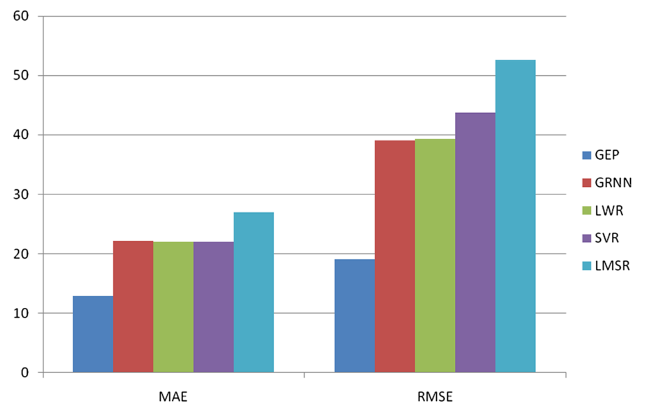


Fig. 10 Best case MAE and RMSE values for apache/httpd/modules/ssl subsystem

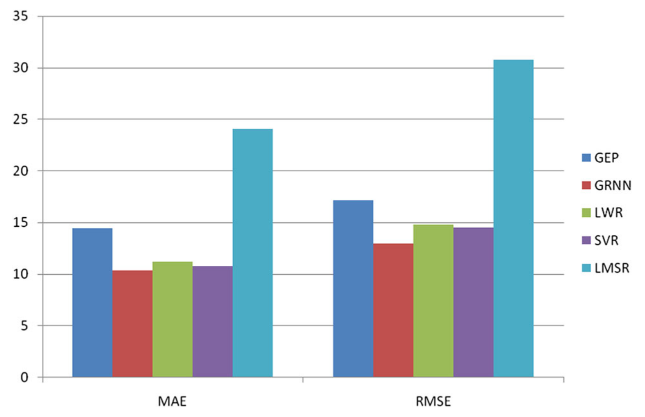


Fig. 11 Best case MAE and RMSE values for apache/httpd/modules/mappers subsystem

subsystems considered in this study mozilla/layout/generic, mozilla/layout/forms, apache/httpd/modules/ssl, apache/httpd/modules/mappers cannot be considered representative of systems written in programming languages different from C and C++.

8 Conclusions and Future direction

Empirical software engineering is concerned with developing accurate models to support various phases of software development. Bug prediction models support the testing phase by helping to optimize software testing costs through early identification of modules that require more rigorous testing. It is encouraged to build replicable and refutable models in empirical software engineering (Menzies et al. 2016). The contribution of this study is two-fold for research community and industry practitioners. This study proposes and implements an algorithm that automates the data extraction process for conducting software entropy based bug prediction studies. The concept of software entropy is grounded in information theory principles which is both intuitive and has a strong mathematical foundation (Hassan 2009). It is evident from three recent systematic literature reviews (Catal 2011; Radjenovic et al. 2013; Malhotra 2015) that there are no benchmarking studies till date that evaluate the applicability of machine learning in software entropy based bug prediction. The second important contribution of this study is that compares machine learning based regression techniques for predicting bugs using entropy of changes. The study explains how entropy of changes is calculated for a software system/subsystem, how metrics are derived using it and finally compares results obtained by using the five regression techniques namely: Gene Expression Programming (GEP), General Regression Neural Network (GRNN), Locally Weighted Regression (LWR), Support Vector Regression (SVR) and Least Median Square Regression (LMSR). Even though a single best technique that performs better than all regression techniques for every case is not observed, nevertheless it is noticed that GEP and SVR give adequate results in all cases. Hence it is suggested that GEP and SVR should be employed for bug prediction using Entropy of changes. An important extension of our work will be to extract software entropy data for large scale software systems developed using other programming languages such as Java, python and other modern and upcoming programming languages such as Go, Rust and Ruby.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

Afzal W, Torkar R (2008) A comparative evaluation of using genetic programming for predicting fault count data. In: The third international conference on software engineering advances (ICSEA'08), pp 407–414

Aggarwal KK, Singh Y, Kaur A, Malhotra R (2009) Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study. Softw Process Improv Pract 14:39–62

Atkeson CG, Moore AW, Schaal SA (1997) Locally weighted learning. AI Rev 11:75–113

Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng 22(10):751–761. doi:10.1109/32.544352

Catal C (2011) Software fault prediction: a literature review and current trends. Expert Syst Appl 38:4626–4636

Catal C, Banerjee S (2010) Application of artificial immune systems paradigm for developing software fault prediction models. In: Evolutionary computation and optimization algorithms in software engineering Hershey, USA: IGI Global, pp 76–93

D'Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir Softw Eng 17(4–5):531–577

Dejaeger K, Verbraken T, Baesens B (2013) Toward comprehensible software fault prediction models using Bayesian network classifiers. IEEE Trans Softw Eng 39:237–257

Ekanayake J, Tappolet J, Gall HC, Bernstein A (2012) Time variance and defect prediction in software projects. Empir Softw Eng 17(4–5):348–389. doi:10.1007/s10664-011-9180-x

Fenton N, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 26(8):797–814. doi:10.1109/32.879815

Fenton N, Neil M, Marsh W, Hearty P, Radlinski L, Krause P (2007) Project data incorporating qualitative factors for improved software defect prediction. In: Proceedings of the 29th international conference on software engineering workshops, IEEE computer society, Washington, DC, USA (ICSEW'07), pp 69. doi:10.1109/ICSEW.2007.171

Ferreira C (2001) Gene expression programming a new adaptive algorithm for solving problems. Complex Syst 13(2):87–129

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661. doi:10.1109/32.859533

Gyimothy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans Software Eng 31:897–910

Hassan AE (2009) Predicting faults using the complexity of code changes. In: 31st international conference on software engineering, IEEE computer society pp 78–88

Kanmani S, Uthariaraj VR, Sankaranarayanan V, Thambidurai P (2007) Object-oriented software fault prediction using neural networks. Inf Softw Technol 49(5):483–492

Kaur A, Kaur K (2014) An empirical study of robustness and stability of machine learning classifiers in software defect prediction. Adv Intell Inf 320:383–397

Khoshgoftaar TM, Allen EB, Goel N, Nandi A, McMullan J (1996) Detection of software modules with high debug code churn in a very large legacy system. In: Proceedings of seventh international symposium on software reliability engineering, pp 364–371

Khoshgoftaar TM, Allen EB, Hudepohl JP, Aud SJ (1997) Application of neural networks to software quality modeling of a very large telecommunications systems. IEEE Trans Neural Netw 8(4):902–909

Khoshgoftaar TM, Allen EB, Jones WD, Hudepohl JP (1999) Data mining for predictors of software quality. Int J Softw Eng Knowl Eng 9(5):547–563

Leroy AM, Rousseeu PJ (1987) Robust regression and outlier detection. Wiley, New York

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Softw Eng 34:485–496

Malhotra R (2014) Comparative analysis of statistical and machine learning methods for predicting faulty modules. Appl Soft Comput 21:286–297

Malhotra R (2015) A Systematic literature review of machine learning techniques for software fault prediction. Appl Soft Comput 27:504–518

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th international conference on predictor models in software engineering. doi:10.1145/1540438.1540448

Mende T, Koschke R (2010) Effort-aware defect prediction models. In: 14th European conference on software maintenance and reengineering (CSMR), pp 107–116

Menzies T, Jeremy G, Frank A (2007) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(1):2–13

Menzies T, Krishna R, Pryor D (2016) The promise repository of empirical software engineering data. North Carolina State University, Department of Computer Science [Online]. http://openscience.us/repo

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th international conference on software Engineering, ACM, New York, NY, USA, pp 181–190. doi:10.1145/1368088.1368114

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of 27th international conference on software engineering pp 284–292

Okutan A, Yildiz OT (2014) Software defect prediction using Bayesian networks. Empir Softw Eng 19(1):154–181

Radjenovic D, Herico M, Torkar R et al (2013) Software fault prediction metrics: a systematic literature review. Inf Softw Technol 55:1397–1418

Rodriguez D, Ruiz R, Riqelme JC, Harrison R (2013) A study of subgroup discovery approaches for defect prediction. Inf Softw Technol 55 (10):1810–1822

Shevade SK, Keerthi SS, Bhattacharyya C, Murthy KRK (2000) Improvements to the SMO algorithm for SVM regression. IEEE Trans Neural Netw 11(5):1188–1193

Singh VB, Chaturvedi KK (2012) Entropy based bug prediction using support vector regression. In: Proceedings of 12th international conference on intelligent systems design and applications, pp 746–751

Singh Y, Kaur A, Malhotra R (2010) Empirical validation of object-oriented metrics for predicting fault proneness models. Softw Qual J 18(1):3–35

Specht DF (1991) A general regression neural network. IEEE Trans Neural Netw 2(6):568–576

Thwin MMT, Quah TS (2005) Application of neural networks for software quality prediction using OO metrics. J Syst Softw 76(2):147–156

Tosun MA, Bener AB, Turhan B (2011) An industrial case study of classifier ensembles for locating software defects. Software Qual J 19(3):515–536

Witten IH, Frank E, Hall MA, Holmes G (2011) Data mining practical machine learning tools and techniques. Morgan Kaufmann, Burlington

Predictive Modeling Software-DTREG- https://www.dtreg.com/download

Zhou Y, Leung H (2006) Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Trans Softw Eng 32(10):771–789