INDIAN INSTITUTE OF INFORMATION TECHNOLOGY ALLAHABAD

Information Technology
Allahabad 211012, India
September 2019

SEMESTER PROJECT REPORT

for

Indian Institute of Information Technology Allahabad
(Information Technonolgy)

by
Prabhakar Kumar , Vardhan Malik , Kumar Raju Bandi
Prasanna Venketesan, Harsh Gupta

Graph Entropy Based Software Defect Prediction

under the guidance of

Dr. Sonali Agarwal
Assistant Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

CERTIFICATE FROM SUPERVISOR

We hereby declare that the work presented in this end semester project report of B.Tech (Information Technology) 5th Semester entitled "GRAPH ENTROPY BASED SOFTWARE DEFECT PREDICTION", submitted by us at Indian Institute of Information Technology, Allahabad, is an authenticated record of our original work carried out from August 2019 to November 2019 under the guidance of Dr. Sonali Agarwal.

Due acknowledgements have been made in the text to all other material used. The project was done in full compliance with the requirements and constraints of the prescribed curriculum.

Submitted by :

- Prabhakar Kumar (IRM2017008)
- Kumar Raju Bandi (IWM2017502)
- Vardhan Malik (IWM2017007)
- Harsh Gupta (ISM2017501)
- Prasanna Venkatesan V S (IIT2017101)

Date :
Place : Allahabad

Supervisor:
Dr Sonali Agarwal

Abstract

This report describes the 5th semester project our group is working on, titled "Graph Entropy based Software Defect Prediction". The following paper talks about predicting the number of bugs using graph entropy of changes, a measure for the complexity of code changes with the use of dependency graph based on the modular structure of files. A lot has been explored in this field using the trivial entropy measures like Shannon entropy or Extended entropy, but they lack the use of software dependency structure which is supposed to affect the propagation of bugs, and hence the propagation of change in a software. The approach discussed, uses the dependency of directory structure of the software modules for calculation of entropy, for a better modelling of the available data.

The work is carried out using the technologies Python, SQLite and DBBrowser. The model predicts the number of bugs in the files and calculates the error in the predicted number of bugs and the original number of bugs on R^2 parameter. Hence as an end result, for the given software if we have the change and version history, then we train our model on that data and then when we are approached with the History Complexity Metric caused by the addition of new features or refinement of pre-existing features, we can predict how many bugs are supposed to be present in the software.

Content

1 Introduction:

Software bug Prediction is an interesting and continuous evolving research field.

Bugs are generated when code is modified, old bugs are fixed or new features are introduced due to miscommunication among the active users, programming errors and several other reasons.

Faults in the software code are a major source of software product problems. It is really beneficial to predict the number of defects or bug prone entities, so that more stress can be given on these entities.

With the advances of Software Bug Prediction, it will be easier for organizations to allocate efforts according to the characteristics of the bugs.

In this project, we attempt to improve one of the metrics used to predict bugs based on cyclic entropy.
In software development, resources for quality assurance are limited by time and by cost. In order to allocate resources effectively, managers need to rely on their experience backed by code complexity metrics. Software errors cost the U.S. industry 60 billion dollars a year according to a study conducted by the National Institute of Standards and Technology[4] . One contributing factor to the high number of errors is the limitation of resources for quality assurance (QA). Such resources are always limited by time, e.g., the deadlines that development teams face, and by cost, e.g., not enough people are available for Quality Assurance. Based on their experience, managers predict the quality of the product to make further decisions on testing, inspections, etc.

Hassan**[1]**(2009) introduced the concept of complexity of code change using information theory principles. Hassan**[1]**(2009), introduced History Complexity Metrics also shortly known as HCM and showed that using HCM as predictors, bugs can be predicted more accurately.
Thomas Zimmermann and Nachiappan Nagappan**[3]** (2008) proposed the concept of using the concept of using dependency of files on other files as a feature to predict bugs in the system.
Till date, no studies have been done on using History Complexity Metrics on a dependency graph to predict bugs more accurately.

2 Motivation:

In the field of Bug Prediction even though the most trivial pipeline was the peer review by the testers or the project managers, who used to analyse part of the software code and make decisions on the change of software code quality and whether to allocate developers to refine the code or not.

The use of only Line of Code change to predict the decrease in the software bug quality then came in which was then replaced when Prof. Ahmad Hassan came out with his entropy based method. His methodology was able to predict the bugs using a parameter that he called as History Complexity Metric, without much deviation from the correct value.

In recent years even the bug proneness of software modules or files were explored using the Dependency Structure of the software. Various research work has been made on different types of dependency structures, the most famous being the Call Dependency of the modules and also the Directory Structure of the software.

But even though the above methods were proving to be correct in their measures, not much work has been done to couple the two methods to form one because both of them have correlation to the bug prediction mechanisms. If we see the Line of Change then greater the LOC count greater would be the chance of creation of bugs, and the greater the dependency greater the chance of bug to be introduced due to change. Hence we got motivated to find a way to combine the two different methodologies such that we not only consider the Line of Change, but also take into account the Dependency Structure of our project.

3 Problem Statement:

We aim to extend the work done by Prof. Ahmed E. Hassan, Research Chair at Queen's University, done in the field of deriving a parameter for the problem of prediction of bugs with the help of the version history of the software. The problem that we have approached is to derive a parameter that would not only depend on the Line of Code of change or the version history of the software but also takes into account the dependency graph, particularly that formed by the directory structure of the software. Even though a lot of work has been done on the two but on two different tables and our problem was to bring them together into a single parameter.

The model at the end would analyse the version history of the software based on the Lines of Code change through-out the version history and the dependency graph, and would give a Support Vector Regression model such that it takes the parameter called History Complexity Metric and outputs the possible number of bugs to be present. Hence lets say, after a release a software company decides to add some features or refine some, resulting in change in the software codes, we will ask the company of their change log for the period, calculate the metric and predict the number of bugs that may have been introduced. We also aim at ranking the files on the basis of probability of which file is most probable to have the highest number of bugs.

Having the ranked list of probable file, the software firm could then make fair decisions of the amount of resources that should be allocated for the review and refinement of those files, such that there is least wastage of the resources.

4 Literature Review:

With the help of recent papers in the study of Entropy and software defects , we realised that following results have already been worked over and proposed in previous papers.

Study by Ahmed E. Hassan **[1]** (2009) in his paper has**:**

Proposed the concept of entropy of changes, a measure for the complexity of code changes. The performance of Entropy based metrics have been compared to the number of changes and the number of previous bugs. The entropy based metrics were evaluated on six open-source systems: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL. The entropy based metrics were found to be better predictors of bugs than the number of changes and the number of previous bugs.

The base of the project is from his study and further extended using dependency structure based on modules to predict number of bugs in the system.

Study by Arvinder Kaur et. al **[2]** (2016) in their paper have:

Compared five machine learning models to predict bugs namely Gene Expression Programming(GEP), General Regression Neural Network(GRNN), Locally Weighted Regression(LWR), Support Vector Regression(SVR) and Least Median Square Regression(LMSR) and suggested GEP and **SVR to be used, for bug prediction using Entropy of changes**. They did this empirical study using the data of two subsystems each of Mozilla and Apache Http Server.

They also proposed a tool for automatic data collection and classification of software changes.

Study by Singh and Chaturvedi **[3]** (2012), in their paper have:

Extended Hassan's concept **[1]** and proposed an entropy based bug prediction approach using support vector regression (SVR). They have compared the results of proposed models with many existing conventional models on the basis of R^2 and found that the proposed models have shown significant improvement of the performance based on R^2. The value of R^2 is over 95% for all the proposed models except few cases. Their study shows that the proposed **SVR models can be applicable in predicting the future occurrence of bugs** based on the entropy of the current year, which will also help in testing effort allocation and bug scheduling.

Study by Qinbao Song, Yuchen Guo and Martin Shepperd **[4]** (2018), in their paper have:

They have addressed the inconsistency issues of data by paying attention to the relationship between the data set and the predictor, secondly they have integrated all the analysis into a single consistent and comprehensive experimental framework, and thirdly by they have avoided biased measures of prediction performance.

Study by Thomas Zimmermann and Nachiappan Nagappan**[3]** (2008), in their paper have:

Zimmermann and Nagappan proposed **the concept of using dependency of files on other files as a feature to predict bugs in the system**. The paper takes the already known and used methods to predict bugs such as code complexity metrics and compares it with the above proposed method. It concludes that the network metrics predict 60% of the critical binaries while code complexity metrics predict only 30%. They proposed many different types of dependency graphs that can be used for the purpose of bug prediction, that included the In-Out Call Graph for the software files and modules, the Directory Structure in which the software modules has been defined, or the dependency formed due to the developers that make the change in the software bugs.

We use the above proposed network metric method using the Dependency Graph formed by the In-Out Call Graph of a part of the Windows Server 2003 and tried to predict the number of bugs in the system. They concluded with the result a software module having high density of connections with other modules have a higher probability of having a bug.

Study by Kamaldeep Kaur et. al**[9]** (2017), in their paper have:

The paper evaluates imbalanced learning algorithms with the entropy of code changes as the predicting variables and later trained them in six machine learning models. It concludes how better do imbalanced learning algorithms perform than classical learning methods on various measures in each of the machine learning models.

Study by Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Akinori Ihara, Kenichi Matsumoto**[4]** (2016), in their paper have:

Defect prediction models may have misleading results if redundant metrics issues are not solved. Redundant metrics issue is such a phenomenon in which one metric is highly correlated with another. 3 detection techniques will be used for identifying redundant metrics on collected data- (a) Variation inflation factor (b) Variable clustering (c) Redundancy Analysis. Collected 101 publicly available defect datasets from open source domain. Concluded that 10% - 67% of metrics of defect dataset are redundant.

Study by Jitesh Shetty and Jafar Adibi,**[8],** in their paper have:

Tackled the problem of hidden organisational structure and selection of interesting influential members(files in our case). **It shows how entropy models of the graph are relevant for studying the information flow**. They showed that entropy is highest when the probabilities are uniform and it decreases as the probability becomes less uniform. They worked over Eron email dataset which is publicly available. Concluded that important nodes are those which have the most effect on the graph entropy on being removed from the graphs.

Study by Talat Parveen and H. D. Arora(2017)**[11]**, in their paper have:

Instead of using months or time period as a parameter to form the dataset, they proposed that the data points for the HCM calculations should be accumulated over different version releases of the software.
They carried their analysis on the Bugzilla Project which had ample data corresponding to versions also, using which they were able to not only predict the number of bugs using the HCM metrics, but they were also able to state that once we have the HCM meric and the corresponding bugs, the total time that could be taken to solve the bugs, i.e., the predicted time for the next version could also be predicted.

5 Proposed Methodology:

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.

Prior to Hassan's work the software bug prediction model was made only on the basis of Lines of code change, and when Hassan introduced his new approach where he used the Shannon's Entropy to define HCM metrics for a software, he could establish a linear relation between the number of bugs that a software faced and the HCM metric for a period or version whatsoever may be involved.

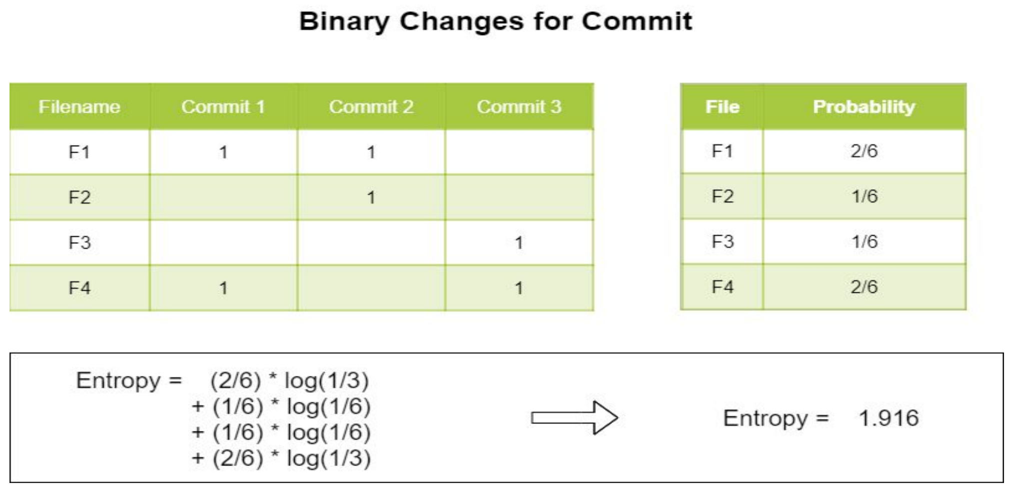
Entropy in general is defined as the degree of randomness, the lack of predictability and the gradual decline into disorder brought to any system under consideration. In case of a software, software entropy defines a measure of the randomness and inherent instability that has been brought to a software over a period of time. In case when we are discussing of a software, a change in entropy may be brought by any change in code that has been made, either as a part of feature addition or bug prediction. Here it should be noted that often there is a decrease in the entropy of software when periodic review of the codes is done to refactor and redesign the code, and decrease the randomness. Since we wanted to use the additive nature of entropy, we did not take such reviews under the consideration of our study.

5.1 Measurement of entropy

Entropy of changes for a period in a system/subsystem is calculated by the Shannon's Entropy formula**[1]** specified below:

$$H_n(P) = -\sum_{k=1}^{n} p_k * log_2(p_k)$$

Here 'p_i' is taken to be the probability of change for the ith file in the specified period i.e. the number of times ith file is modified divided by the total number of modifications.



Binary Changes for Commit

Filename	Commit 1	Commit 2	Commit 3
F1	1	1	
F2		1	
F3			1
F4	1		1

File	Probability
F1	2/6
F2	1/6
F3	1/6
F4	2/6

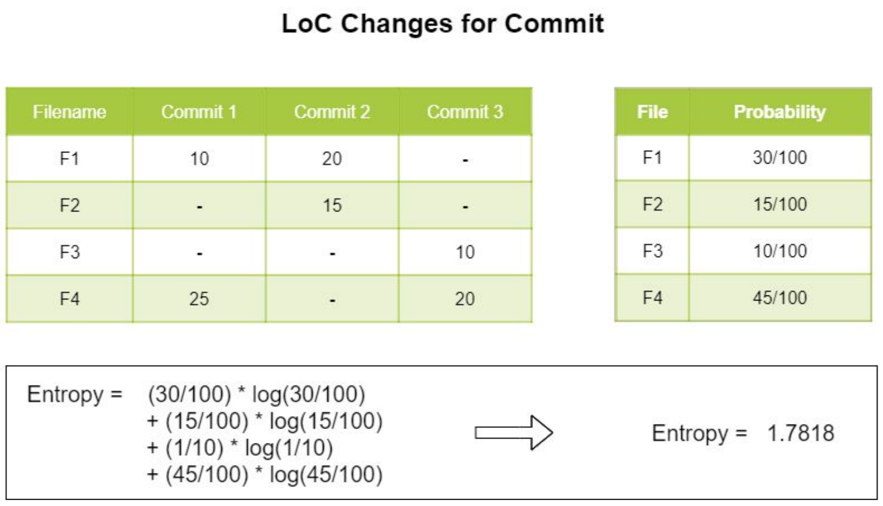
Entropy = (2/6) * log(1/3) + (1/6) * log(1/6) + (1/6) * log(1/6) + (2/6) * log(1/3) $\Rightarrow$ Entropy = 1.916

Fig. Figure showing entropy calculation as per Hassan's Proposed Method.

For example, let us assume 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.

Here we can see that what Hassan had done was that he treated all file changes as a binary parameter, contributing the same value to the calculation of entropy. We here proposed that when two file changes have different code change data, like they have different number of lines that have been added or removed in the code change, or if the two files have different number of interdependencies with other files that also has been changed, then both the file changes should not be treated the same.

The first step that we took was to quantify the code change firstly by the sum of the number of lines added and removed. The intuition behind this was that the greater the number of lines changed in the code, the greater will be the contribution in imparting randomness to the software.



LoC Changes for Commit

Filename	Commit 1	Commit 2	Commit 3
F1	10	20	-
F2	-	15	-
F3	-	-	10
F4	25	-	20

File	Probability
F1	30/100
F2	15/100
F3	10/100
F4	45/100

Entropy = (30/100) * log(30/100)
+ (15/100) * log(15/100)
+ (1/10) * log(1/10)
+ (45/100) * log(45/100)

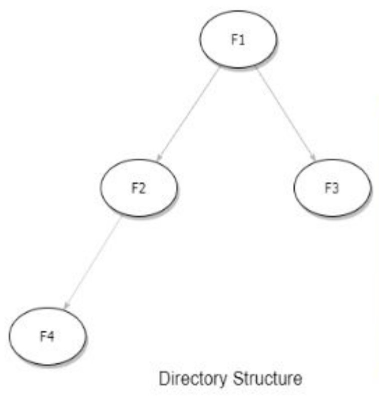
$\Longrightarrow$ Entropy = 1.7818

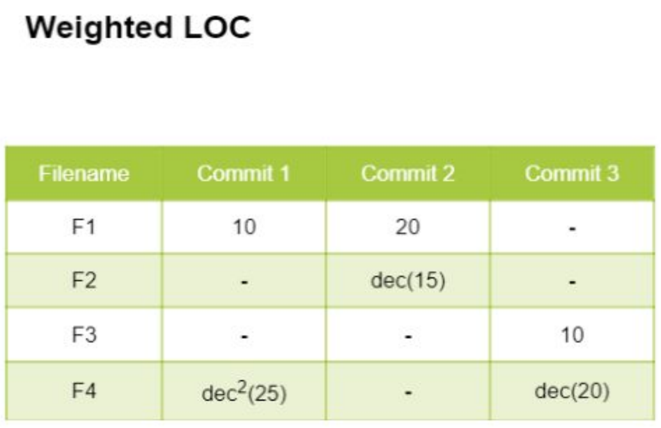
Fig. Figure showing entropy calculation using Line of Code change parameter.

Then with the works of ………… we had got the idea that the bug prediction and the propagation of entropy is also affected by the several dependencies that lie within the software and also the dependencies that is formed while the file changes were done. The software dependencies as proposed by ………… would include Directory Structure, Call Structure, etc. whereas other kinds of dependencies may include the developer dependencies that was involved during the file change process.

We hence proposed that lets say we have a file F1 that has relation with another file F2, that may be a relation due to call relation or a module and sub-module relation, then any file change happening in file F1 would also cause some file change in the file F2, and hence the changes will be propagated down the dependency graph, if any. One trivial form of dependency graph as was introduced by ……. Is the directory structure, such that when a software is defined in modules then it is organised in such a manner that a module that has dependency on any sub-module, then that sub-module is defined in a directory lying at the same level as the parent module in the directory structure. This is done to increase the Cohesion of the software modules and to reduce the Coherence as much as possible.

Hence we also introduced a new method of using the above dependency such that if there is a change in a module then we would not treat the changes that have happened in the sub-modules the same. We hence introduced a decline factor through which we define the propagation of change and its decreasing contribution in imparting randomness. Let we have the following dependency graph formed using the directory structure for the same example that we have been using, the following is the new method that we used for the calculation of entropy.




Weighted LOC

Filename	Commit 1	Commit 2	Commit 3
F1	10	20	-
F2	-	dec(15)	-
F3	-	-	10
F4	$dec^2(25)$	-	dec(20)

Fig. Figure showing the entropy calculation based on Line of Code Change weighted with the directory dependency structure of the software.

Here the *dec( )* function that we had defined was an abstract function being treated as a Black Box function, but later upon analysis and comparison with several functions we concluded that a good function would be to multiply the incoming parameter with 0.9 everytime the function is called, and then return the value. Hence with that our change metric and entropy will be calculated in the same manner as those
Once we have the change entropy formulated by our proposed method, we followed the same path as Hassan had proposed of calculating the HCPF(History Complexity Period

Factor) for each file and then the HCM(History Complexity Metric), such that there should be a strong linear dependency between HCM and the number of bugs.

5.2 Calculation of History Complexity Metric

History Complexity Metric of a system is defined by Hassan as an empirical measure for complexity of change assigned to each file in the software system. Calculation of History Complexity Metric involves the entropy concept in the code change process. To compute the HCM for a file we first find the History Complexity Period Factor for the file during a given commit or at a period and then sum the HCPF over the whole of a span of time, so that we get the total HCM for the file over that period. This History Complexity Metric defines how much randomness the changes in this particular file introduced.

The calculation of HCPF can be done in three different manners with three different values for the parameter C_{ij} , such that they define three different kinds of HCPF mertics and then the HCM metrics, as follows-

$$\boldsymbol{HCPF_i = C_{ij} * H_i \;\; for \;\; j \, \varepsilon \, F_i}$$

Here $HCPF_i$ denotes the HCPF for the ith period and *j* dontes each file and C_{ij} denotes the coefficient for ith period and jth file.

HCM1 with $\mathbf{HCPF_i}$ being defined with parameter $\mathbf{C_{ij} = 1}$, denoting that we assign full complexity for each modified file.

HCM2 with $\mathbf{HCPF_i}$ being defined with parameter $\mathbf{C_{ij} = P_j}$, denoting that we distribute the complexity to the jth file according to the probability of code change that we had earlier used to calculate Shannon Entropy.

HCM3 with $\mathbf{HCPF_i}$ being defined with parameter $\mathbf{C_{ij} = 1/F_i}$, where F_i denoted the number of files that has been changes in the ith period.

$$\mathbf{HCM_{p,\dots,q}(j)} = \sum_{i \, \varepsilon \, p,..q} \mathbf{HCPF_i\,(j)}$$

The HCM for a subsystem or a software is found by summing up the HCPF factors for all the files that are present in the software. It should be noted that the files that have not been affected by any code change would have a zero HCPF value, denoting no contribution in increase in entropy.

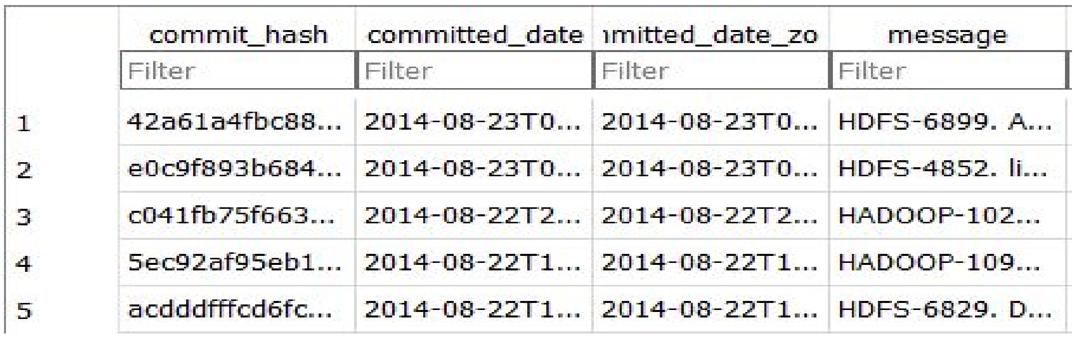
5.3 Dataset Used -

The dataset used in the project was acquired from Harvard Dataverse-The SEOSS Database - Requirements, Bug Reports, Code History, and Trace Links for Entire Projects. (https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/PDDZ4Q).
The data taken is data from Hadoop software. Hadoop is a collection of open-source projects that uses a network of multiple computers to solve problems involving tremendous amount data and computation.

The data has been compiled by the version history of the software being managed by Bugzilla which helps firms to manage the process of software development by keeping a detailed track of the changes and issues.

The dataset consists of **nine tables** where the smallest table is meta of ten rows which consists of meta information of the dataset and the largest table is issue_comment of 4,53,765 rows. The size of the file was **585 MB**. From this dataset, four tables were used in the project as discussed below.

1) change_set

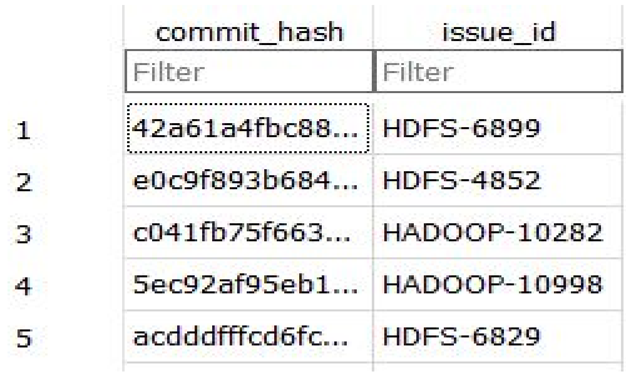


	commit_hash	committed_date	ımitted_date_zo	message
	Filter	Filter	Filter	Filter
1	42a61a4fbc88...	2014-08-23T0...	2014-08-23T0...	HDFS-6899. A...
2	e0c9f893b684...	2014-08-23T0...	2014-08-23T0...	HDFS-4852. li...
3	c041fb75f663...	2014-08-22T2...	2014-08-22T2...	HADOOP-102...
4	5ec92af95eb1...	2014-08-22T1...	2014-08-22T1...	HADOOP-109...
5	acdddfffcd6fc...	2014-08-22T1...	2014-08-22T1...	HDFS-6829. D...

Fig. Figure showing a snapshot of the *change_set* table of the dataset.

The change_set table consists of commit_hash, committed_date, committed_date_zone, message, author, author_email, is_merge. It has 27,776 rows of data. The primary key of this table is commit_hash table which is used to cross-reference with other tables.

2) change_set_link

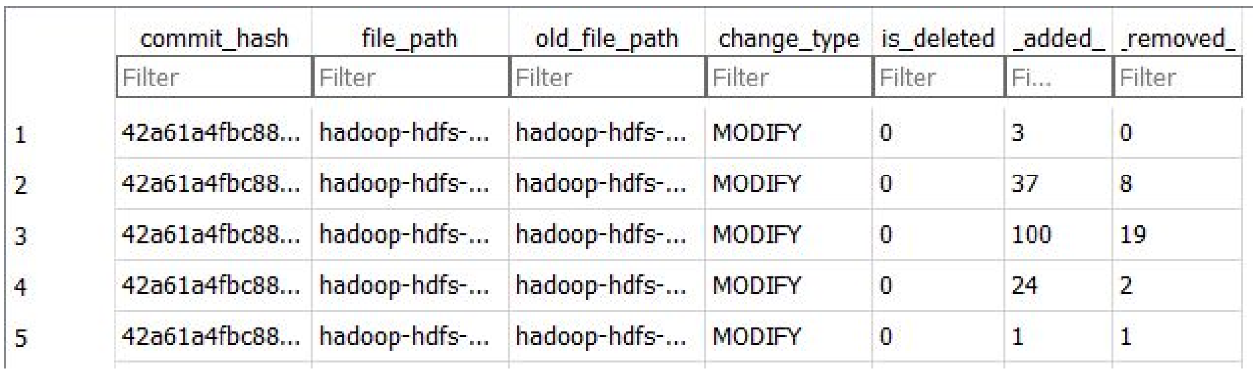


	commit_hash	issue_id
	Filter	Filter
1	42a61a4fbc88...	HDFS-6899
2	e0c9f893b684...	HDFS-4852
3	c041fb75f663...	HADOOP-10282
4	5ec92af95eb1...	HADOOP-10998
5	acdddfffcd6fc...	HDFS-6829

Fig. Figure showing a snapshot of the *change_set_link* table of the dataset.

The change_set_link table consists of commit_hash and issue_id. It consists of 29,309 rows of data. The primary key of this table is a combination of commit_hash and issue_id which is used to cross-reference with other tables.

3) code_change:



	commit_hash	file_path	old_file_path	change_type	is_deleted	_added_	_removed_
	Filter	Filter	Filter	Filter	Filter	Fi...	Filter
1	42a61a4fbc88...	hadoop-hdfs-...	hadoop-hdfs-...	MODIFY	0	3	0
2	42a61a4fbc88...	hadoop-hdfs-...	hadoop-hdfs-...	MODIFY	0	37	8
3	42a61a4fbc88...	hadoop-hdfs-...	hadoop-hdfs-...	MODIFY	0	100	19
4	42a61a4fbc88...	hadoop-hdfs-...	hadoop-hdfs-...	MODIFY	0	24	2
5	42a61a4fbc88...	hadoop-hdfs-...	hadoop-hdfs-...	MODIFY	0	1	1

Fig. Figure showing a snapshot of the *code_change* table of the dataset.

The code_change table consists of commit_hash, file_path, old_file_path, change_type, is_deleted, sum_added_lines, sum_removed_lines columns. It consists of 3,62,802 rows of data. The primary key this table is commit_hash which is used to cross-reference with other tables. The file_path column of this table helped us with the formation of the directory structure to define the dependency and the sum of the sum_added_line and sum_removed_lines gave the total Line of Code change parameter.

4) issue

The table consists of 18 columns. It consists of 39,086 rows of data. The primary key is issue_id which is cross-reference with other tables. We used issue_id and priority columns to assign the severity of the issues.

5.4 Data Preprocessing -

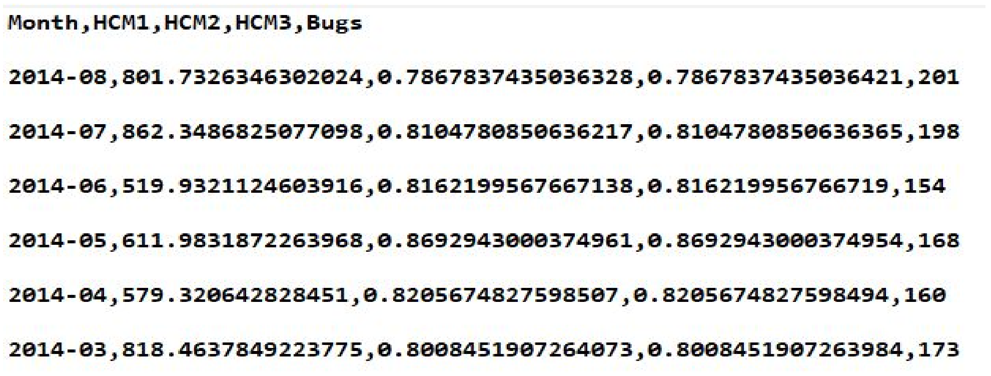
The dataset had multiple tables that we had to use in parallel. One simple method that we could have resolved to was to take a join of the 4 tables that were of use for us. But the data tables that we have used had rows ranging from 27,000 to 2,60,000 in numbers and making join of such large tables was computationally very tough and because of that several times our system stopped responding in between.

Hence we then resolved to loading the data from all the tables using *python* scripting, and then using the data as per our requirement. It was then discovered that there were multiple incosistancies in the tables, such that an entry in one table had no corresponding entry in another table. As an example, if there is a list of file changes corresponding to a commit in the *code_change* table, then corresponding to that there may have been no entry in the *change_set_link* from where the commit could be associated to a particular issue.

Once we had made the four tables consistent by removing the entries from the table for which we could have got a ‘missing data’ or ‘data not found’ error, we then used a python script to pre-process the data since as the end result we wanted the total HCM of the system after every version or at the end of the period under consideration.

For this in the *python* script first clubs the commits according to the month in which it was made, since the period for which we are accumulating the data for each datapoint is of a month. We then for each commit we form the dependency structure using the directory structure of the files involved in the commit. We then also associate the commit with the corresponding issue or bug for which it was made, since a single issue may be attended by multiple commits and innumerous file changes.

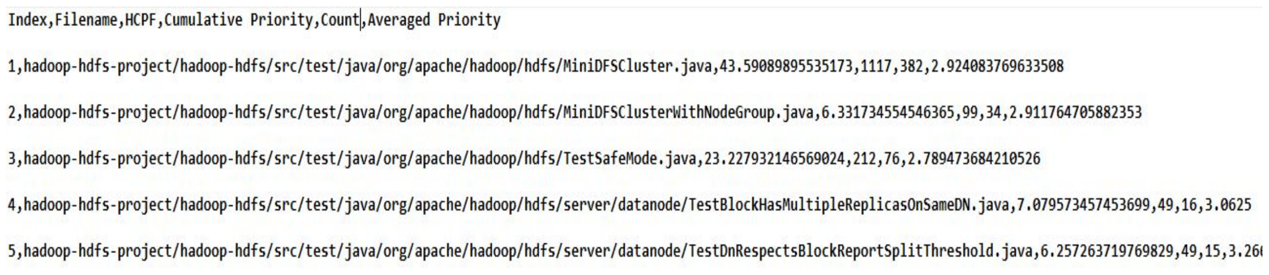
We then for a month accumulated the HCPF for each file over the period by appropriately applying the weighted approach that we had proposed using the dependency structure, which could be seen to be performed in the *decrease_weight( )* function. Once we have the HCPF of all the file, we apply a summation over all the files to get the HCM for the whole system and also the total number of issues for the whole month. It should be noted that the dependency structure is being formed at each commits, since storing the already composed structure would have taken up a lot of space and would not have contributed in making any help in reducing complexity of the script.



```
Month,HCM1,HCM2,HCM3,Bugs

2014-08,801.7326346302024,0.7867837435036328,0.7867837435036421,201

2014-07,862.3486825077098,0.8104780850636217,0.8104780850636365,198

2014-06,519.9321124603916,0.8162199567667138,0.816219956766719,154

2014-05,611.9831872263968,0.8692943000374961,0.8692943000374954,168

2014-04,579.320642828451,0.8205674827598507,0.8205674827598494,160

2014-03,818.4637849223775,0.8008451907264073,0.8008451907263984,173
```

Fig. Figure showing the processed data with month, corresponding HCM1, HCM2, HCM3 and bugs.

We also accumulated the bug priority data using the already existing data from the above procedure and the issue data available in the *issue* table, since we also wanted to see if there happens to be any relation between the priority of the issue or bugs and the corresponding contributions in the entropy.



```
Index,Filename,HCPF,Cumulative Priority,Count,Averaged Priority

1,hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/MiniDFSCluster.java,43.59089895535173,1117,382,2.924083769633508

2,hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/MiniDFSClusterWithNodeGroup.java,6.331734554546365,99,34,2.911764705882353

3,hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/TestSafeMode.java,23.227932146569024,212,76,2.789473684210526

4,hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/server/datanode/TestBlockHasMultipleReplicasOnSameDN.java,7.079573457453699,49,16,3.0625

5,hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/server/datanode/TestDnRespectsBlockReportSplitThreshold.java,6.257263719769829,49,15,3.26
```

Fig. Figure showing the processed data with filename,its HCPF, cumulated priority and count.

5.5 Model Training and Testing-

Once we have the HCM metric for a given set of periods and the number of bugs corresponding to them, we then went forward to establishing a Linear relationship between the two sets of data. Hassan in his paper had proved that there should be a linear relationship between the HCM metric and the bugs, using a simple linear regression model. Singh and Chaturvedi**[3]**, in their paper has also stated that since the data of bug report is a very skewed data, and there are a number of outliers that would be present while considering the bug report for any software over a period, hence a Machine Learning Model that would differentially treat the outliers data that does not follow the general trend, would be a better predictor of the result.

The available data is trained in two different models. The first one is Linear Regression. Here the data consists of History Complexity Matrix(HCM) of three different types which is the independent variable and the corresponding number of bugs in that particular version as the dependent variable. The data is trained by randomly picking nine-tenths of the available data and testing the model on the remaining one-tenth of the model.

The second and the preferred model is the Support Vector Regression(SVR). This model is preferred as shown in the paper by Singh and Chaturvedi**[2]**. The Support Vector though had a relatively bad performance when we trained the model after normalising the HCM value, but then when we trained the model on raw data we had a very significant increase in accuracy.

Since our dataset produced only about 100 data-points hence to test the model we looped for 100 iterations and for each iteration we split the data into $^9/_{10}$ as training and $^1/_{10}$ of testing data and evaluated the R^2 parameter. This happened for all the iterations and final evaluations were done based on the result of the 100 iterations

6 Challenges:

Finding a suitable dataset was very difficult as most of the datasets available consisted only of simple entropy and not graph data and vice versa. Finding a suitable dataset that can be processed was difficult.

The suitable dataset was very large making it impossible to use SQL operations as the smallest table contained about 27,000 rows of data and the largest table about 3,60,000 rows of data.

The dataset was inconsistent and corresponding entries were missing between the tables. The version count issues and commits were less documented so forced to used month count data.

The suitable dataset was in different dimensions and not in standard dimensions, which made it difficult to train the models and plot the data.

7 Software Requirements & Tools:

Python : Python is an interpreted and general purpose programming language.This high level language is used as the primary programming language for this project. Modelling, training and testing was done completely using python.

SQLite3 : SQLite3 is a relational database management system. It is embedded into the program rather than acting as a client-server database engine. The datasets of hadoop version history were stored in SQLite and used for training, testing and validation tasks.

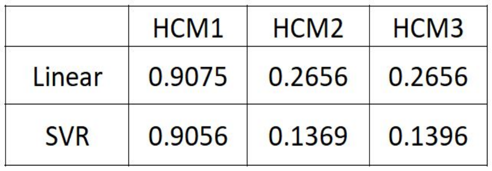
DBBrowser : DBBrowser is an open source visual tool to visualize, create and edit database files compatible with SQLite.

Libraries :

- **Matplotlib:** Matplotlib is a library used for plotting in the programming language Python and its library NumPy.
- **ScikitLearn:** Scikit-Learn is a machine learning library used in the Python programming language.
- **SeaBorn:** SeaBorn is a data visualization library based on matplotlib for Python programming language.
- **Pandas:** Pandas is a library used for data manipulation analysis in Python programming language.

8 Inference and Results:

Since our project was a Research Project in which we had proposed some improvisations in the already existing methodology that has been used for bug prediction and has been explored by many researchers, hence we have a lot of graphs to compare our model with and to justify that the end model with all the hyper-parameters that we have proposed is able to make substantially better results.
For every step we had implemented both a Linear Regression Model and Support Vector Regression model with Linear Kernel Function. The first and foremost decision that we had to make was to decide which of the HCM matrics out of HCM1, HCM2 and HCM3, had the best fitting relation with the data. As per Hassan and ……., out of the three HCM1 was supposed to be the give the best fitting model, and that could be seen in the table below-



	HCM1	HCM2	HCM3
Linear	0.9075	0.2656	0.2656
SVR	0.9056	0.1369	0.1396

Fig. Table of comparison of R^2 obtained for different types of HCM metric with the two ML Models.

One anomaly that we saw was that the HCM2 and the HCM3 metrics were too close to each other in values and also gave a very bad fitting for the bug prediction dataset, since other researchers were able to get a better fitting with HCM2 and HCM3 with the entropy calculation as per Hassan's proposal. From this analysis we concluded that HCM1 metric gave has the best fitting relation with the bug prediction.

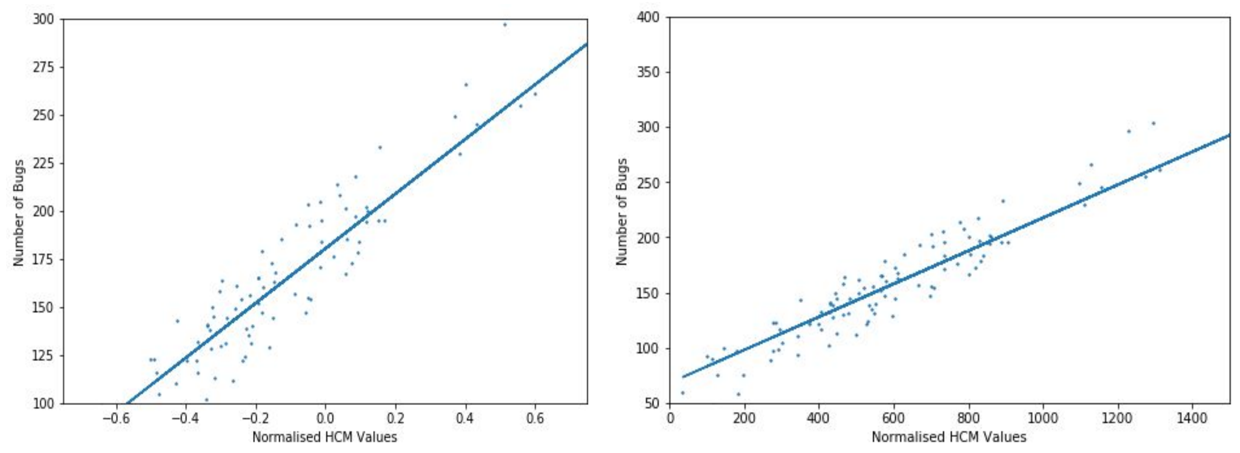
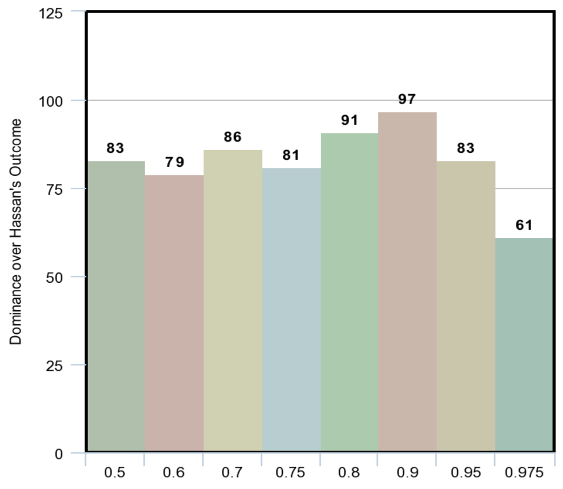


Fig. Figure showing a comparison of the best fitting line of Linear Regression Model (Left) and the Support Vector Regression Model(Right).

The two figures above shows the scatter plot of the same data points and the best fitting line formed by the Linear Regression Model on the left and the Support Vector Regression Model on the right. It should be noted that the two graphs seem to look different since we had normalised the input parameters for Linear Regression Model while not for the Support Vector Regressor. The two plots are also on the different scales and limits.

Then in our entropy calculation, as we have mentioned earlier we had treated the *dec_weight( )* as a black box such that it performs the task of decreasing the impact of code changes of files that lie lower in the dependency structure. We then used a few number of parameters using which we had to decrease the weight, ranging from 0.5 to 0.975, since the value 1 would have resulted in un-weighted entropy.



The table on the side is showing histogram of results of comparison of our proposed model such that the value of weight that we use as the hyper-parameter is being plotted on the x-axis and the y-axis denoting the count of how many times our model predicted better than on data using our proposed model as compared to the one using Hassan's proposed method.

Fig. Figure showing a comparison of accuracy
For different hyper-parameter values for *dec_weight()*
function

Now we needed to compare the two models with two different regression techniques on two sets of processed data, one using the Hassan's proposed methodology and the other on our proposed methodology mad entropy calculation formula. Through this we had a total of four output data to be compared and we had made four comparisons to conclude to our result.

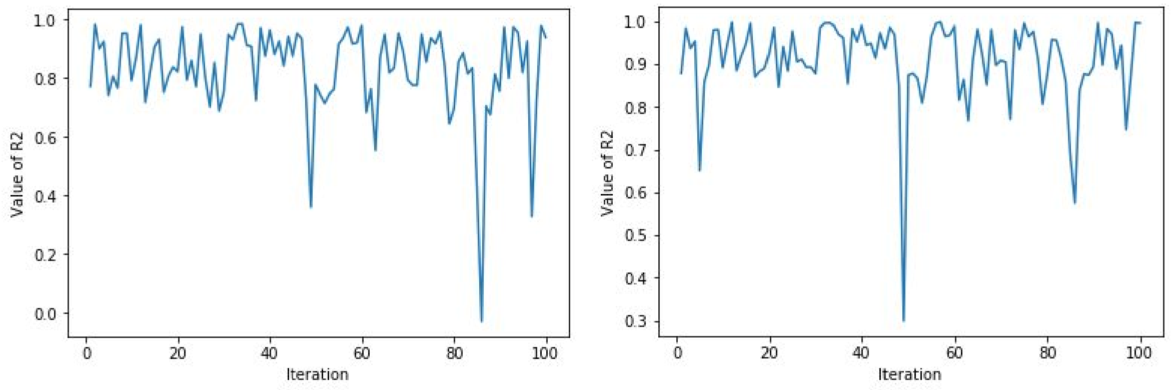


Fig. Figure showing R^2 values for Hassan's Model(Left) vs our proposed model(Right) for Linear Model

The above is the R^2 value of Linear Regression Model with the data prepared by Hassan's proposed method and on the left is that on the right is that trained and tested on the data by the proposed method.

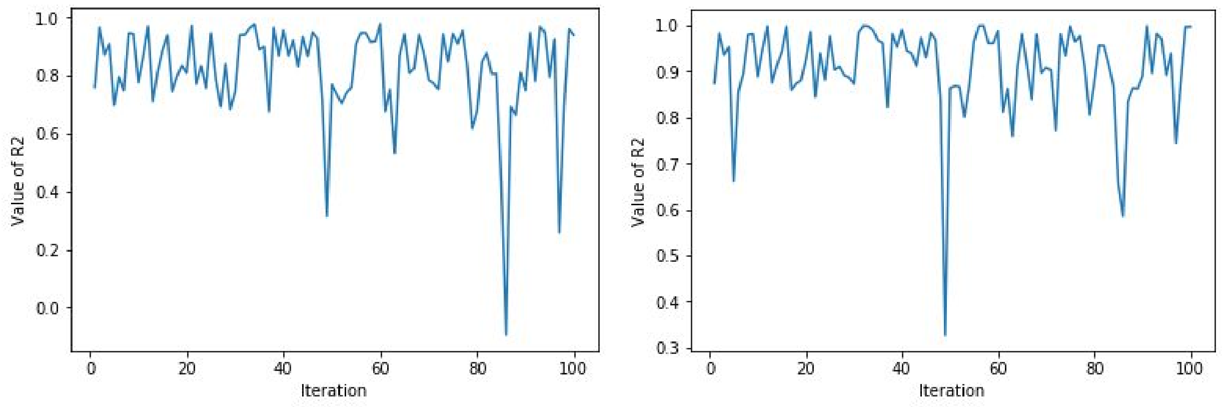


Fig. Figure showing R^2 values for Hassan's Model(Left) vs our proposed model(Right) for SVM Model

The above is the R^2 value of Support Vector Regression Model with Linear Kernel

Function with the data prepared by Hassan's proposed method and on the left is that on the right is that trained and tested on the data by the proposed method.

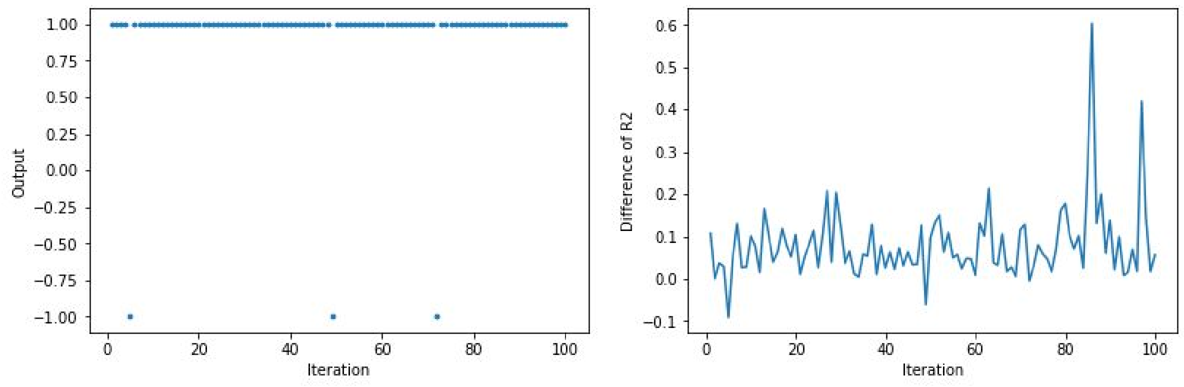


Fig. Figure showing confidence(Left) and the difference of R^2 on the Right for Linear Regression.

The above plot on the left shows how well data through our proposed method was able to predict the bugs as compared to data through Hassan's method, using the Linear Regression model. The +1 on the plot signifies that our data was a better predictor, and vice versa. Now even to quantify the difference of R^2 value, plotted we value of difference with the iterations.

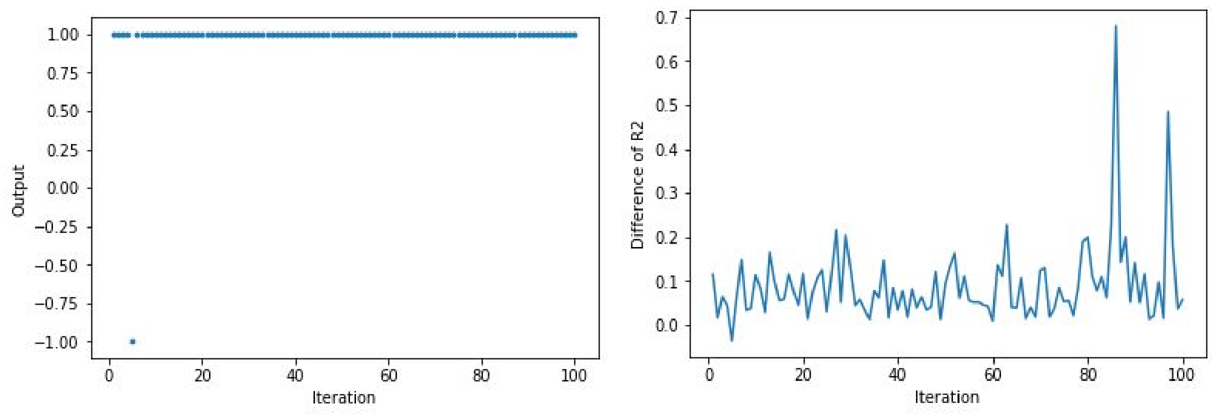


Fig. Figure showing confidence(Left) and the difference of R^2 on the Right for SVR.

The above plot on the left signifies the same plot as that as previous such that the model used was Support Vector Regression. On the right plot we again showed the difference of R^2 value for the model trained and tested in our data and on Hassan's data, plotted with value of difference on y-axis and iteration on x-axis.

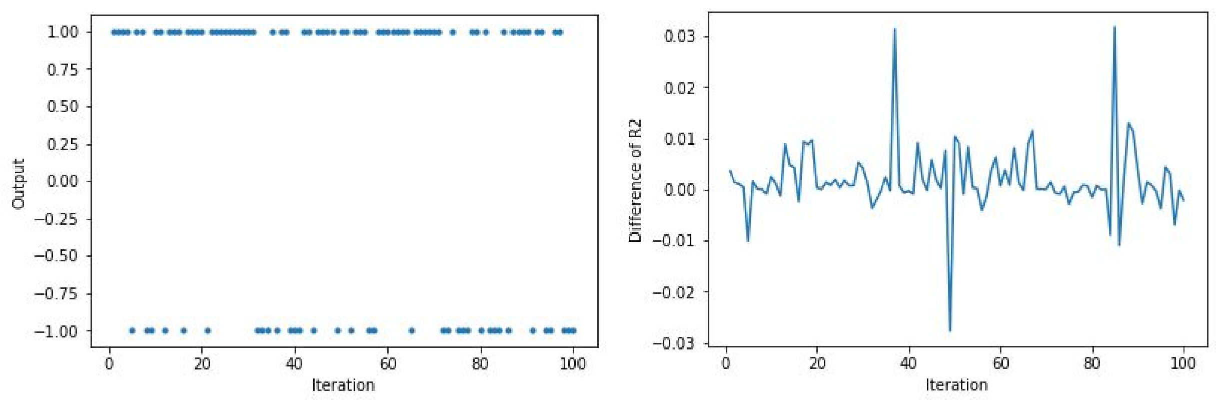


Fig. Figure showing confidence(Left) and the difference of R^2 on the Right for Linear vs SVR.

Now finally we compared the Linear Regression Model with Support Vector Regression model trained and tested on our data being retrieved using our proposed methodology. Hence from here we can conclude that Support Vector Regression is a better prediction model for prediction of bugs using HCM, when compared to simple Linear Regression Model.

Types of Models compared	Performance
Proposed Weighted Lines of Code vs Hassan's Code Change (Linear Regression)	**97/100**
Proposed Weighted Lines of Code vs Hassan's Code Change (Support Vector Regression)	**99/100**
Proposed Weighted Line of Code Support Vector Regression vs Linear Regression	**65/100**

Fig. Table showing confidence of our proposed method and Hassan's method for Linear and SVR.

The above table shows the final output of the models with the processed data using the two different methods, one with the one proposed by Hassan and the one with our proposed method of where we had weighted the change according to the dependency graph. The fraction in the second column is defined as : the numerator is the number of times the first method in the first column for the corresponding row dominates the second method, and the denominator is the total number of iterations for which the testing was done.

We then also tried to establish a correlation between the priority of the bug that has been assigned by the Bugzilla software and their corresponding contribution to the entropy using the HCM. We supposed that their would a linear correlation, most probably signifying that the bug with higher priority contributed higher in imparting entropy to the software and hence would also require a larger effort to be removed.

Priority of Bugs : Blocker(5) , Critical(4) , Major(3) , Minor(2) and Trivial(1).

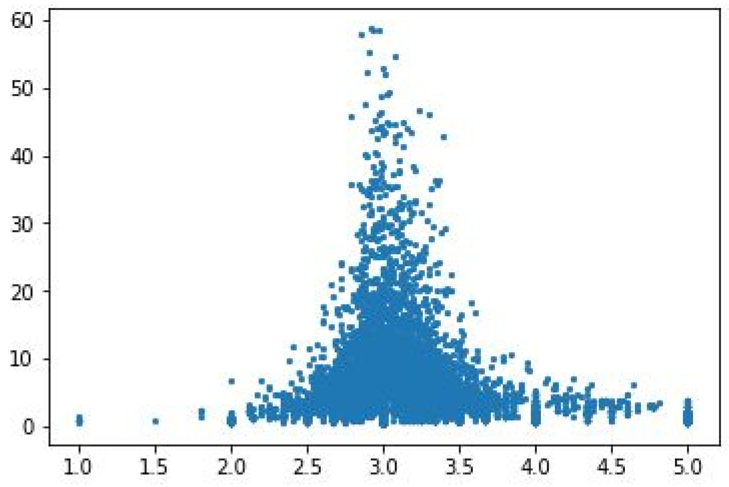


Fig. Plot of HCM contribution(y-axis) and average priority(x_axis).

The above plot with the average priority of the files on the x-axis and their contribution to the entropy on the y-axis, came out to follow some what a Gaussian relation with the center being that of priority 3. If there would have a linear correlation then we would have used that to rank the files in order of their priority, so that a higher ranked file should be reviewed before a lower ranked file

9 Conclusion :

Hence from the various comparisons that we had seen in the Inference and Result section, we can conclude to the fact that -

- Software Dependency Graphs, like In-Out Call graphs, Directory Structure, Developer Dependency Graphs, certainly have effects on the propagation of code change in the software.
- The use of Line of Code change after scaling down the changes made in files such that a file higher in the dependency structure is also changed, is a better measure of prediction of bugs using the History Complexity Metric evaluated upon

the calculated entropy.

- We also verified the fact that HCM1 is a better metric as compared HCM2 and HCM3, for bug prediction.
- We also verified that the use of Support Vector Regression with Linear Kernel gave a better fitting model to the data formed by proposed method, than the Linear Regression model, which also comply with the conclusion made by Singh and Chaturvedi in their study**[3].**
- We also concluded that we could not establish a considerable correlation between the bug priority and its effect on the software entropy, and would need to incorporate data from more dependency structures to get a successful result.

10 Implementation Plan :

Task	**Start Date**	**End Date**
Literature Review	23-08-2019	01-09-2019
Finalizing Problem Statement and Goals	01-09-2019	04-09-2019
Gathering and Pre-Processing of data	04-09-2019	05-10-2019
Training of multiple ML models	05-10-2019	15-10-2019
Cross-validation and comparison of ML models	15-10-2019	23-10-2019
Evaluation and testing of models	23-10-2019	30-10-2019
Hyperparameter tuning and optimization	30-10-2019	08-11-2109
End-Semester Evaluation	21-11-2019	21-11-2019

Pre-Midsem - Before the Mid-Semester break we had successfully read 12 research papers related to the issue of Bug Prediction. This included both the general Entropy Based Method and the Dependency Graph based approach. Then we discussed our proposed method that we had formulated with our mentor Mr. Amit Tyagi, who approved our proposed method after hearing the justification of why our method would give a better parameter. Hence by the Mid-Semester evaluation we had our problem statement ready.

Post-Midsem - After the mid-semester exams our first aim was to search for a dataset to which we could apply our proposed methodology, since most of the datasets that were mentioned in the works of other researchers were pre-processed data, to which we could not make any changes. We had received our dataset to work by the last week of September, after which we had to pre-process the dataset.
The pre-processing took a lot of effort because we had to use data from four different tables, such that they did not have a consistent entries. By the second week of October we had our dataset being processed in a form that can be used to train and test using Machine Learning Model. We then trained our dataset and the dataset that we received from Hassan's approach, on two regression model and also iterated over our hyper-parameters to make sure that the end model with the hyper-parameters that we proposed is the best of what we have explored.
We also tried to establish a relationship between the priority of bugs and its corresponding contribution in entropy, and we hoped that we would get a linear relationship with the correlation coefficient being close to either -1 or 1. But instead we received a sort of like Gaussian Distribution, hence we could not conclude to our method to predict the files to look at in order of priority.

11 Future Works :

Using the above conclusions we could though be satisfied by the new Entropy Metric for the bug prediction, but when it came to file prediction we could not have a successful correlation using the Dependency Graph from the Directory structure.
We wish to extend the entropy calculation method to incorporate other dependency structures like In-Out Call graph and the developer dependency too. This should help our HCM2 and HCM3 metric also to be a better predictor, which could be used to convert this problem into multidimensional regression problem. This should help us to predict the files that should be allocated the most immediate attention by the reviewers.
Next, we also wish to continue this into our next semester and use the version release data also to predict the time that the next version should take to come out as had been defined by Talat Parveen and H. D. Arora in their paper**[11]**.

12 References:

1. A.E. Hassan, "Predicting Faults based on complexity of code change", proceedings of 31st Intl. Conf. on Software Engineering, pp. 78-88, 2009.
2. V.B. Singh, K.K. Chaturvedi, "Entropy based Bug Prediction using Support Vector Regression", proceedings of 2012 12th International Conference on Intelligent Systems Design and Applications (ISDA), pp. 746-751, 2012.
3. T. Zimmermann, N. Nagappan, "Predicting defects using network analysis on dependency graphs", Software Engineering 2008. ICSE'08. ACM/IEEE 30th International Conference on, pp. 531-540, 2008.
4. Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Akinori Ihara, Kenichi Mat-sumoto, "A study of redundant metrics in defect prediction datasets", Software Reliability Engineering Workshops (ISSREW) 2016 IEEE International Symposium on, pp. 51-52, 2016.
5. S.Young, T.Abdou, A.Bener, “A Replication Study: Just-In-Time Defect Prediction with Ensemble Learning”, 2018 ACM/IEEE 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering
6. Q. Song, Y. Guo, M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction", IEEE Trans. Softw. Eng., [online]
7. H.Hatami, S.Janson, B.Szegedy, “Graph Properties, Graph Limits and Entropy”
8. J.Shetty, J.Adibi, “Discovering Important Nodes through Graph Entropy The Case of Enron Email Database”, LinkKDD '05 Proceedings of the 3rd international workshop on Link discovery
9. K.Kaur, J.K.Name, J.Malhotra, “Evaluation of imbalanced learning with entropy of source code metrics as defect predictors”, 2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions) (ICTUS)
10. A. Kaur, K. Kaur, D. Chopra, "Entropy based bug prediction using neural network based regression", IEEE International Conference on Computing Communication and Automation, pp. 168-174, May. 2015.
11. Talat Parveen, H. D. Arora, “Estimating Release TIme and Predicting Bugs with Shannon Entropy Measure and Their Impact on Software Quality”, Proceeding of Thai Journal of Mathematics, pp. 91-105, 2017.
12. G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," National Institute of Standards and Technology 2002.
13. M. D'Ambros, M. Lanza, R. Robbes, "An extensive comparison of bug prediction approaches", MSR'10: Proceedings of the 7th International Working Conference on Mining Software Repositories, pp. 31-41, 2010.

14. T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system", Proceedings of ISSRE 1996, pp. 364, 1996.
15. N. Nagappan, T. Ball, "Use of relative code churn measures to predict system defect density", Proc. of the 27th International Conference on Software Engineering, pp. 284-292, 2005.
16. Tim Menzies, Jeremy Greenwald, Art Frank, "Data mining static code attributes to learn defect predictors", IEEE Trans. Software Eng., vol. 33, no. 1, pp. 2-13, 2007.
17. Thilo Mende, Rainer Koschke, "Revisiting the evaluation of defect prediction models", PROMISE ‘09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, pp. 1-10, 2009.
18. Thilo Mende, Rainer Koschke, "Effort-aware defect prediction models", CSMR '10: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, pp. 109-118, 2010.