Bug Prediction Techniques incorporating the involvement of humans

BY

Raghavender Sahdev

(2011A7PS257H)

B.E. (Hons) Computer Science Engineering

SUBMITTED IN FULLFILLMENT OF THE REQUIREMENTS OF

CS F266: STUDY ORIENTED PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI

(RAJASTHAN)

HYDERABAD CAMPUS

(November 2014)

A Project Report

On

Bug Prediction Techniques incorporating the involvement of humans

BY

RAGHAVENDER SAHDEV

2011A7PS257H

B.E. (Hons.) Computer Science Engineering

Under the supervision of

Dr. NL BHANUMURTHY

HOD, Computer Science Department

SUBMITTED IN FULLFILLMENT OF THE REQUIREMENTS OF

CS F266: STUDY ORIENTED PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI

(RAJASTHAN)

HYDERABAD CAMPUS

(November 2014)

ACKNOWLEDGEMENT

In performing my project, I had to take the help and guideline of some respected people, who deserve my greatest gratitude. The completion of this project gives me much pleasure. I would like to show my gratitude towards Dr. NL Bhanumurthy, BITS Pilani Hyderabad campus for giving me a guideline for assignment throughout numerous consultations. I would also like to expand my gratitude to all those who have directly / indirectly guided me in completion of this project.

I would also like to thank Mr. Pranav Ramarao who helped me in extraction of data at some point of time in the course of my project. In addition I would also like to thank Mr. Rishabh Kohli whose suggestions also played a part in this project.

Birla Institute of Technology and Science-Pilani,

Hyderabad Campus

Certificate

This is to certify that the project report entitled

“Bug Prediction Techniques incorporating involvement of humans”

submitted by **RAGHAVENDER SAHDEV** (ID No. 2011A7PS257H) in fulfilment of the requirements of the course CS F266, Study Oriented Project course, embodies the work done by him under my supervision and guidance.

Date: **Dr. NL BHANUMURTHY**

HOD, Computer Science Department

ABSTRACT

The goal of the investigation is to determine whether information about which particular developer modified a file and the cumulative number of developers changing the file are able to improve defect predictions. We investigate whether files in a large system that are modified by an individual developer consistently contain either more or fewer faults than the average of all files in the system. We also continue a study to evaluate the use of counts of the number of developers who modified a file as predictors of the file's future faultiness. We also continue an earlier study to evaluate the use of counts of the number of developers who modified a file as predictors of the file's future faultiness.

CONTENTS

1. INTRODUCTION

Bug prediction has generated widespread interest for a considerable period of time. The growth of empirical software engineering techniques has led to increased interest in bug prediction algorithms. These algorithms predict areas of software projects that are likely to be bug-prone: areas where there tend to be a high incidence of bugs appearing (these are also sometimes called fault-prone areas).

The driving scenario is resource allocation: Time and manpower being finite resources, it makes sense to assign personnel and/or resources to areas of a software system with a higher probable quantity of bugs. A variety of approaches have been proposed to tackle the problem, relying on diverse information, such as code metrics (lines of code, complexity), process metrics (number of changes, recent activity) or previous defects. Previous research has provided evidence that a combination of static code metrics and software history metrics can be used to predict with surprising success which files in the next release of a large system will have the largest numbers of defects. In contrast, very little research exists to indicate whether information about individual developers can profitably be used to improve predictions.

2. PRIOR WORK (RELEVANT WORK):

Static code metrics:

A growing number of studies have been carried out on the subject of automated software defect prediction using static code metrics. Such studies typically involve observing the performance achieved by classifiers in labelling software modules (functions, procedures or methods) as being either defective or otherwise. Although such a binary labelling toward module defectiveness is clearly a simplification of the real world, it is hoped that such a classification system could be an effective aid at determining which modules require further attention during testing. An accurate software defect prediction system would thus result in higher quality, more dependable software that could be produced more swiftly than was previously possible.

The predictions that are made in typical defect prediction studies are usually assessed using confusion matrix related performance measures, such as recall and precision. In some studies a defect prediction experiment was carried out to allow a manual Analysis of the classifications made in terms of each module's: original metrics, corresponding classification result (one of either: true positive, true negative, false positive or false negative) and corresponding decision value; a value output by the classifier which can be interpreted as its certainty of prediction for that particular module. The experiment carried out involved assessing the performance of Support Vector Machine (SVM) classifiers against the same data with which they were trained. The purpose of this was to gain insight into how the classifiers were separating the training data, and to see whether this separation appeared consistent with current software engineering beliefs (i.e. that larger, more complex modules are more likely to be defective.

Change Bursts:

In an empirical study on Windows Vista, researchers found that the features of such change bursts have the highest predictive power for defect-prone components. With precision and recall values well above 90%, change bursts significantly improve upon earlier predictors such as complexity metrics, code churn, or organizational structure. As they only rely on version history and a controlled change process, change bursts are straight-forward to detect and deploy.

3. DATA EXTRACTION

Initial part of the project focuses on data extraction. Following options exist for extracting data

- Writing scripts in languages like R or Python
- Using Data extraction tools available on www.Bugzilla.com

We here extract data using R scripts. We have chosen R as the data extraction tool due to the numerous libraries available in R.

R Code Explanation

Initially we get the data from the URLS. We initially import the required libraries. Here we extract data from 11 pages at one time. This takes 3-4 minutes to get the entire data from the 11 URLs. Following is the code to assign appropriate URLS for the project 'lucene-solr':

```
library(httr)

url0 = "https://github.com/apache/lucene-solr/commits/trunk?page=33"


url1 = "https://github.com/apache/lucene-solr/commits/trunk?page=34?"

url2 = "https://github.com/apache/lucene-solr/commits/trunk?page=35?"

url3 = "https://github.com/apache/lucene-solr/commits/trunk?page=36?"

url4 = "https://github.com/apache/lucene-solr/commits/trunk?page=37?"

url5 = "https://github.com/apache/lucene-solr/commits/trunk?page=38?"


url6 = "https://github.com/apache/lucene-solr/commits/trunk?page=39?"

url7 = "https://github.com/apache/lucene-solr/commits/trunk?page=40?"

url8 = "https://github.com/apache/lucene-solr/commits/trunk?page=41?"

url9 = "https://github.com/apache/lucene-solr/commits/trunk?page=42?"

url10 = "https://github.com/apache/lucene-solr/commits/trunk?page=43?"


# update, fix, add, modify, execut,
```

Now we iterate over the 11 entries in the for loop, after each iteration we get approximately 400 commits. So after running this code once we get approximately 4000 commits.

```
for(i in 1:10)

{

  if(i==1)
```

```
{
  html2 = GET(url0)
} else if(i==1) {
  html2 = GET(url1)
} else if(i==2) {
  html2 = GET(url2)
} else if(i==3)
{
  html2 = GET(url3)
}else if(i==3)
{
  html2 = GET(url3)
}else if(i==4)
{
  html2 = GET(url4)
}else if(i==5)
{
  html2 = GET(url5)
}else if(i==6)
{
  html2 = GET(url6)
}else if(i==7)
{
  html2 = GET(url7)
}else if(i==8)
{
  html2 = GET(url8)
}else if(i==9)
{
  html2 = GET(url9)
}else if(i==10)
{
  html2 = GET(url10)
}
```

Following part of the code is used to get the sha s from the source code of the page that is stored in html2 variable. We then parse the source code using htmlParse function.

```
library(XML)
parsedHtml = htmlParse(html2,asText=TRUE)
shaz <- xpathSApply(parsedHtml,"//div//div//button",xmlGetAttr,"data-clipboard-text")
library(stringr)
```

Following is the URL used to extract the required information about the files, their committers, the commit message, etc. Here initially we faced an issue with the rate limit. Github allows 60 requests per hour for unauthenticated requests and allows upto 5000 request per hour for unauthenticated requests. The process of authentication is done by generating an access token from a registered github account and then copying the access token generated at the end of the URL being used for extracting data.

Github uses an API provided for extraction of data. We here leverage the github API in combination with the ‘jsonlite’ library to extract data. R provides support for this library and we here take advantage of this library to extract data.

```
a<-"https://api.github.com/repos/apache/lucene-
solr/commits/sha?access_token=081090653e76e914592beea67f159d6841ec1574"
len2 = length(shaz)
c_urls <- list()
```

Here this loop iterates over all the urls that are generated out of the shas extracted from the content stored in html2 variable as discussed above. In this loop the sha is replaced by a alpha-numeric long string which is extracted from the information contained in html2.

```
for (i in 3:len2)
{
  b<-shaz[i]
  c_urls[i-2] <- str_replace(a,"sha",b)
}
library("jsonlite")
for (i in 3:len2)
```

```
 {
  jsonData <- fromJSON(c_urls[[i-2]][1])
  len = length(jsonData$files)
  for (i in 1:len)
  {
    p1 <- jsonData$commit$author$name
    p2 <- jsonData$commit$author$date
    p3 <- jsonData$files$filename[i]
```

Here we take out the release dates for a given project by going to github and check the release dates for that particular project. We then check the dates and compare them with the standard release dates and assign each file a particular release.

We later extract commit messages for the code. We have used a simple technique of text mining. We check for existence of some keywords specific to a particular project. The presence of such keywords indicates that the commit message is due to a bug in the file. The absence of such keywords signifies it being an update. This information is later written in an excel files. The excel file has 4 columns the username, the filename, the release number and the commit nature (bug or update).

```
    if(p2>"2014-10-30T12:00:00Z") {
      release <- 34
    } else if(p2>"2014-09-28T12:00:00Z") {
      release <- 33
    } else if(p2>"2014-09-21T12:00:00Z") {
      release <- 32
    } else if(p2>"2014-09-02T12:00:00Z") {
      release <- 31
    } else if(p2>"2014-06-24T12:00:00Z") {
      release <- 30
    } else if(p2>"2014-05-19T12:00:00Z") {
      release <- 29
    } else {
      release <- 28
    }
    p4 <- jsonData$commit$message

    write(p4,file="C:/Users/Raghavender Sahdev/Desktop/BITS/Bug
Prediction/lucene_solr_msgs2.csv", append=TRUE)

    write("$$",file="C:/Users/Raghavender Sahdev/Desktop/BITS/Bug
Prediction/lucene_solr_msgs2.csv", append=TRUE)

    message <- p4

    word1 <- "LUCENE-"

    word2 <- "SOLR-"

    word3 <- "Fix"

    word4 <- "fix"

     word5 <- "typo"

    word6 <- "warning"

    word7 <- "potential"

    word8 <- "Warning"

    word9 <- "Correct"

    word10 <- "Fx"

    word11 <- "#"

    word12 <- "Conflict"

    word13 <- "Fixed"

    word14 <- "change"

    word15 <- "Change"

    word16 <- "error"

    word17 <- "bugr"

    word18 <- "Error"

    if(grepl(word1,message)|grepl(word2,message)|grepl(word3,message)|grepl(word4,message)){

      answer <-"bug"

    } else
if(grepl(word5,message)|grepl(word6,message)|grepl(word7,message)|grepl(word8,message)|grep
l(word9,message)|grepl(word10,message)){

      answer <- "bug"
    } else
if((grepl(word11,message)|grepl(word12,message)|grepl(word13,message)|grepl(word14,message)
|grepl(word15,message))){

      answer <- "bug"

    } else if((grepl(word16,message)|grepl(word17,message)|grepl(word18,message))){

      answer <- "bug"

    } else {

      answer <- "update"

    }


    if(p1 == null)

    p6 = paste(p1,p3,release, answer,sep=",")

    write(p6,file="C:/Users/Raghavender Sahdev/Desktop/BITS/Bug
Prediction/commits_lucene2.csv", append=TRUE)

    }

  }

}
```

4. APPROACH:

At a time we focus on only 3 releases dubbed prior release, current release and next release.

We need to find out which files are buggy for which we have written a short macro which mines the message and uses key words like fix, bug, corrected, etc to find out whether it is a bug or not.

We store the User name, File name and the release number (Calculated using the commit date and the Dates each release has been released.) in an excel file that will be used for all further data manipulation.

Prior Release
Current Release
Next Release

In the prior release we associated programmers who have made changes to the file with the file regardless of whether the file was buggy or not.

We used the previous association in order to find the bug ratio for each programmer. Bug ratio is the ratio of the files a programmer has worked on in the prior release is to the number of those files that turn out to be buggy. We then made another association of files with the programmers for current release to match with the next release. We used this bug ratio in order to get the error quality of the programmer (5 classes). We then found the overall error possibility of all the files in this release by adding the error quality of programmer. Error quality is found by dividing the entire bug ratio domain into 5 classes on the basis of mean and standard deviation.

In the next release we look into the top 20 percent of the files in the current release which have the greatest error possibility. We then find out how many of these files are actually buggy. We calculate three accuracy ratios which give us an idea about how good our prediction model is.

Accuracy 1: W want to look at how many bugs did we catch in top 20% of the predicted file list arranged in descending order with respect to the points given to each file. After arranging in descending order, we look at the files in top 20%. We then compare them with the total number of bugs in the current release of the file.

Accuracy 1 = (top 20 % of the files in the current release after arranging in descending order) / (all bugs in the current release)

Accuracy 2:

Suppose t1 denotes the top 20 percentage of the files in the current files, and suppose t2 denotes out of those 20 % of the files how many files appeared to be buggy.

Acurracy 2 = t2 / t1

5. IMPLEMENTATION:

1) Following code checks how many files have been touched by a particular user in prior release. Odd index in the array list contains user name and the next even index contains the files separated by commas of that particular user.

```
for(int i=0 ; i< brr.length ; i++)  //here brr.length returns length of the number of rows
            {if(Integer.parseInt(brr[i][2]) == PRIOR_RELEASE &&
!files_of_users.contains(brr[i][0]))
                  {       files_of_users.add(brr[i][0]);
                          String temp= "";
                          for(int j=0 ; j<brr.length ; j++)
                          {       if(brr[i][0].equalsIgnoreCase(brr[j][0]) &&
Integer.parseInt(brr[j][2]) == PRIOR_RELEASE) // made change here *
                                  {temp = temp + brr[j][1] + ",";}
                          }
                          files_of_users.add(temp);
                  }
            }
```

2) Variable files_of_users_with_number have usernames in odd indexes and names of files seperated by commas touched by them in even indexes.

3) Variable files_of_users_with_number have usernames in odd indexes and number of files touched by them in even indexes.

4) Calculation of Bug Ratio of each user. Iterate over all prior release files of a particular user and then for each such iteration find the file in the current release.

```
 for(int i=0 ; i< number_of_users ; i++)
                    {
                              buggyFiles_of_users.add(files_of_users.get(i*2));
                              String temp2 = (String)files_of_users.get((i*2)+1);
                              String temp_buggy_files = "";
                              StringTokenizer tkn = new StringTokenizer(temp2,",");
                              while(tkn.hasMoreTokens())
                              {
                                        String temp = tkn.nextToken();
                                        for(int j=0 ; j<brr.length ; j++)
                                        {
                                                  // check the release of the file and check if it is of the ith contributor
                                                  if(Integer.parseInt(brr[j][2]) == CURRENT_RELEASE &&
temp.equalsIgnoreCase(brr[j][1])
                                                  {
                                                            if(brr[i][3].equalsIgnoreCase("bug"))
                              {
                                                                      temp_buggy_files = temp_buggy_files + brr[j][1] + ",";

                                                                      break;
                                                            }}}}
```

5) Then associate points with all of the developers in the prior release by dividing them into 5 classes using mean and standard deviation.

6) Add the points and bug ratios of developers to each file and then arrange in descending order.

```
for(int i=0 ; i<users_of_files.size() ; i = i+2)
                    {
                              String temp_userNames = (String)users_of_files.get(i+1);
                              StringTokenizer break_it = new StringTokenizer(temp_userNames,",");
```

```
                int no_of_users = break_it.countTokens();

                double bratio_sum = 0.0;

                int users_cnt=0; // here we divide to take average of the bugratios of people

                while(break_it.hasMoreTokens())

                {

                        String t1 = break_it.nextToken();

                        //System.out.println(t1+"ddsaa!@!");

                        bratio_sum = bratio_sum + findBugRatio(username_bratios,t1);

                        users_cnt++;

                }


                String temp_fileName = (String)users_of_files.get(i);

                //System.out.println(temp_userName+": "+no_of_users);


                users_of_files_with_number.add(temp_fileName);

                users_of_files_with_number.add(bratio_sum/users_cnt);

        }
```

7) Then see how many in the top 20% of these files have bugs and few more accuracy points.

6. RESULTS

We performed analysis of 3 projects using our code.

- Lucene-solr
- Eclipse Acceleo
- Firefox

We see that the results of our experiments to be on a lower side. Prior work through the paper "Programmer Based Fault Prediction", has also shown similar results.

We here define 2 accuracies as defined above:

Accuracy 1: W want to look at how many bugs did we catch in top 20% of the predicted file list arranged in descending order with respect to the points given to each file. After arranging in descending order, we look at the files in top 20%. We then compare them with the total number of bugs in the current release of the file.

Accuracy 1 = (top 20 % of the files in the current release after arranging in descending order) / (all bugs in the current release)

Accuracy 2:

Suppose t1 denotes the top 20 percentage of the files in the current files, and suppose t2 denotes out of those 20 % of the files how many files appeared to be buggy.

Accuracy 2 = t2 / t1

For lucene-solr project

<u>Accuracies for release 29, 30 , 31</u>

Taking the sum of bug ratios for arranging files in descending order we get
Accuracy 1 = 5/83 = 6%
Accuracy 2 = 5/50 = 10%

For releases 30,31,32

Accuracy 1 = 7/52 = 14%
Taking the sum of bug ratios for arranging files in descending order we get
Accuracy 2 = 7/23 = 30%

For Acceleo Project

<u>Accuracies for release 30, 31 , 32</u>

Taking the sum of bug ratios for arranging files in descending order we get

Accuracy 1 = 5/45 = 11%
Accuracy 2 = 5/58 = 8.6%

For releases 31,32,33

Accuracy 1 = 4/59 = 6.7%
Taking the sum of bug ratios for arranging files in descending order we get
Accuracy 2 = 9/63 = 14.2%

For Firefox project

Taking the sum of bug ratios for arranging files in descending order we get
Accuracy 1 = less than 5 %
Accuracy 2 = less than 5 %

For releases 30,31,32

Accuracy 1 = less than 5 %
Taking the sum of bug ratios for arranging files in descending order we get
Accuracy 2 = less than 5 %

7. CONCLUSION:

We have tried to replicate the paper Programmer-based Fault Prediction on open source software. The paper implements their technique on proprietary software that is used in the industry we have tried to use the same technique on open source software like acceleo, lucene, firefox etc. Open source software operates with rolling releases; this means as and when a few small bug are corrected an update is triggered. However for more appropriate results we need to wait till sufficient bugs are discovered, corrected and tested before a release. Hence the results were not as encouraging as the paper but there exists a similarity in the results.

REFERENCES

Journals

O Does Bug Prediction Support Human Developers? Findings From a Google Case Study.

Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, E. James Whitehead Jr.

O Predicting Bugs from History

Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller

O Programmer-based Fault Prediction

Thomas J. Ostrand, Elaine J. Weyuker, Robert M. Bell

Websites

- git.eclipse.org/c/jdt/

Courses on Coursera

The Data Scientist Toolbox by *John Hopkins University* https://class.coursera.org/datascitoolbox-009

Getting and Cleaning Data by *John Hopkins University* https://www.coursera.org/course/getdata