On Predicting Rediscoveries of Software Defects

by

Mefta Sadat

Bachelor of Science, Islamic University of Technology, 2013

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2017

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my dissertation may be made electronically available to the public.

On Predicting Rediscoveries of Software Defects

Master of Science 2017

Mefta Sadat

Computer Science

Ryerson University

Abstract

The same defect may be rediscovered by multiple clients, causing unplanned outages and leading to reduced customer satisfaction. One solution is forcing clients to install a fix for every defect. However, this approach is economically infeasible, because it requires extra resources and increases downtime. Moreover, it may lead to regression of functionality, as new fixes may break the existing functionality. Our goal is to find a way to proactively predict defects that a client may rediscover in the future. We build a predictive model by leveraging recommender algorithms. We evaluate our approach with extracted rediscovery data from four groups of large-scale open source software projects (namely, Eclipse, Gentoo, KDE, and Libre) and one enterprise software. The datasets contain information about ≈ 1.33 million unique defect reports over a period of 18 years (1999-2017). Our proposed approach may help in understanding the defect rediscovery phenomenon, leading to improvement of software quality and customer satisfaction.

Acknowledgements

This master's thesis is submitted to fulfill the requirements of the MSc of Computer Science at Ryerson University in Toronto, Canada. The work carried out in this thesis was supervised by Dr. Andriy Miranskyy and Dr. Ayse Bener.

Foremost, I would like to express my sincere gratitude to Dr. Miranskyy for the continuous support of my MSc study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in the research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my MSc studies.

My sincere thanks also goes to Dr. Bener, who always provided great guidance related to my research and inspiration to work hard and aim for the best. I thank her also for leading me to work on diverse exciting projects.

Besides my supervisors, I would like to thank the rest of my thesis committee.

I thank my fellow labmates in Ryerson AMiR Lab: Mujahid Sultan, Sokratis Tsakiltsidis, and Jorge Lopez for the stimulating discussions in our group meetings.

Last but not the least, I would like to thank my family and my wife Nadira, whose constant love and support made this possible.

Dedication

To my beloved mother, who always inspired me to go for higher studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software quality assurance is the process that ensures that the software being developed meets all the expected quality standards [73]. During software quality assurance and maintenance, a significant amount of time is invested on detecting, analyzing, and correcting software defects [19]. A Software defect is an anomaly in the software product that causes the software to perform incorrectly or to behave in an unexpected way [16]. Defect reports are software engineering artifacts that contain description of software defects.

When a defect is reported for the first time pointing to a problem in the software that was never identified before, we call it Defect Discovery [9, 60]. If multiple users report the same defect, we call it a Defect Re-discovery [9, 60] of the original discovery. The occurrence of a large number of rediscovery causes an avalanche of support tickets, increased downtime, and reduced customer satisfaction. Preventing a defect rediscovery (i.e., by applying a fix beforehand as a preventive measure) is costly because in software projects a large number of defects is reported on a regular basis (Figure 1.1) and creating so many special builds is expensive and time consuming. Thus, the necessity to develop optimised techniques to prevent defect rediscovery arises.

1.1 Terminology

Throughout the study the following **terminology** (adopted from [16, 15, 65, 58]) is used.

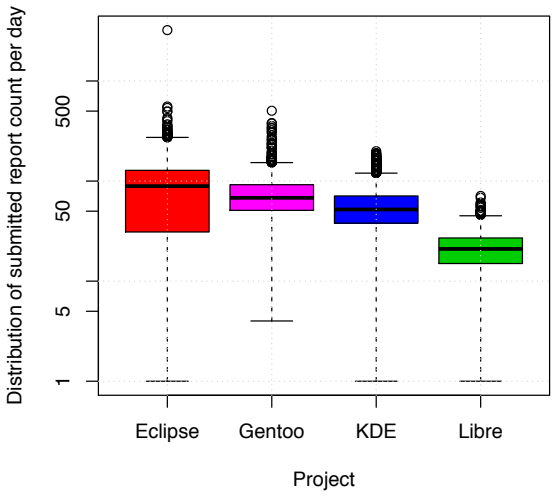

Figure 1.1: The distribution of number of defect reports submitted per day for four different software projects in the last 18 years. The Y-axis in log scale.

1.1.1 Software Defect

The users of a software product often encounter a problem or a *fault* in the software that leads to an undesired outcome or even a software *failure*. This problem or flaw is also known as software defect. Even though there are multiple ways to define the term defect, typically in Software Engineering, it means the deviation from an expected outcome or requirement [16].

1.1.2 Defect Rediscovery

Original defect *discovery* can be defined as the moment when a customer encounters a defect in the software for the very first time. Encounter is manifested by a problem or a fault in the software that leads to an undesired outcome or even a software *failure*. The customer then submits a *report* to a defect tracking system describing the problem.

If another customer encounters the same defect again, it is called Defect Rediscovery [60]. Defect *rediscovery* may occur when the fix is not ready or the fix is ready and yet to be installed by the customer. Sometimes, the administrators of the software system may delay the fix request as they are preoccupied with other crucial tasks related to the

overall functionality of the system or are awaiting for planned maintenance downtime [9].

1.1.3 Graph of Rediscoveries

After encountering a defect, a customer typically submits a new report to the defect tracking system of the software provider. During report triaging, developers identify if a new report relates to a discovery of a new defect or to a rediscovery of an existing one. If it is a rediscovery, then developers typically mark the most recent report as a duplicate and link it to the original report (in some cases the link may be established incorrectly: "to err is human"). They then choose one of the linked reports as a *master report* and the rest of the reports associated with this particular failure will be deemed *duplicates* of the master report. Note that the report associated with the first discovery does not necessarily become a master report – sometimes developers choose a report of one of the rediscoveries as a master one. Given that there can be more than one rediscovery of the same defect, the network linking the original report with duplicate ones (which we call the *graph of rediscoveries*) may become complex [15]. For example, Figure 1.2 shows the graph of rediscoveries for Eclipse project's defect report #4671 of product Platform. Note that the master report (#6325) in this case is not the original report. The defect tracking system used by Eclipse project numbers defect reports sequentially with an integer *id*, with the first *id* set to 1.

Summing up original discovery and rediscovery count yields *total number or reports for a given failure*. If a given report was discovered in total once, then it means that it was never rediscovered; discovered twice – means that it was rediscovered once, and so on. In the case of Figure 1.2, report #4671 was rediscovered 14 times. Thus, the total number of reports for a failure associated with report #4671 is 15.

1.1.4 Preventive Service

Preventive Service (PS) generally means the installation of a fix for a defect beforehand, in order to prevent the defect rediscovery [9]. Preventive Service is provided to counter defect rediscovery. Sometimes, customers may ask proactively for preventive services after getting notification from the software manufacturer about potential defect rediscovery [60].

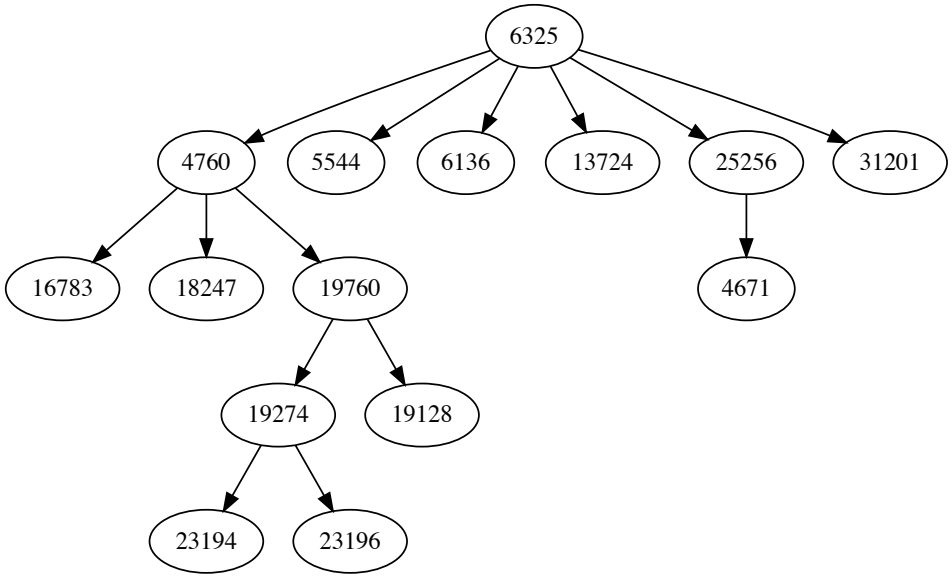

Figure 1.2: Graph of rediscoveries of Eclipse report #4671. Report B being duplicate of report A is denoted by $A \rightarrow B$. Note that even though report #4671 is the original discovery, a later report #6325 was chosen by developers as the master report. We can say that the failure associated with report #4671 was discovered 15 times in total (counted as the total number of vertices/reports in the graph) and rediscovered 14 times (total number of duplicate reports).

1.2 Motivation

Defects are injected in the software during the development cycle. When the test team inadvertently fails to detect the injected defect, it escapes the test cycle, resides in the shipped product and ultimately reaches the end customer. The detection of all the defects before release is hardly ever achieved in real-life software projects [39]. So, the customers "trip" on the dormant defects that are part of the released product. After encountering such a defect, a customer usually reports it to the software provider by opening a ticket in a defect reporting system of the software provider. When a ticket is opened, it is analysed and then assigned to a single developer or a group of developers to fix the problem. The process is generally known as bug fixing [88]. When the fix is ready and sent to the

customer, it is applied to the product to eliminate the defect.

A defect injected in a frequently used software component is more likely to be rediscovered multiple times. On the contrary, a defect residing in an infrequently used software component has a lower chance of rediscovery. The group of customers who use the same set of software features excessively, may rediscover the same defects more often because users expect the software to behave similarly in similar situations [59]. Some customers may use the software more extensively than the others, contributing to the higher probability of defect rediscovery, as they 'traverse' through a higher number of execution paths. Essentially, they have higher number of inputs, outputs, and configurations. For example, one group of customers may use a spreadsheet software only for storing data and another group of customers may use the same software for both storing and analysing the data. The latter group may rediscover more defects.

Defect rediscoveries affect clients, as they lead to unplanned outages and reduced customer satisfaction. Moreover, rediscoveries drain resources of the support personnel [9, 60], as they have to analyse each request before providing a solution. Since both parties suffer from the problems caused by rediscoveries, it is essential to reduce the number of rediscoveries per defect in a software product. Defect rediscovery can be countered by applying Preventive Service (PS) [9]. If a client applies the fix proactively, before rediscovering the defect, they will never encounter it. By applying PS, one may minimise the total number of rediscoveries significantly. However, this strategy has some disadvantages as per[9, 60]:

- PS is expensive, as it requires allocation of additional resources (e.g., to apply patches),
- Some customers are resistant to apply PS (i.e., they prefer not to update until a major service pack is released),
- PS itself may inject a new defect, causing regression of functionality, and
- Each customer will not rediscover every defect. As a result, it is not practical to install fix for every discovered defect.

Therefore, if we take all the above mentioned constraints into account, we can come to the conclusion that this strategy is not always desirable. Rather than applying fixes to

all defects, we need to identify a subset of defects that the client will rediscover in the future and target only this subset of defects for PS.

1.3 Objective

The primary objective of this study is to build predictive models that can predict defect rediscoveries in order to proactively eliminate them before a customer finds. We reach the primary objective by answering the following research question.

RQ1: How can we proactively predict defects that a client will rediscover in the future?

The secondary objectives include (1) understanding the defect rediscovery phenomenon in both commercial and open source software projects and (2) identifying the essential factors that influence the accuracy of the predictive models. We address these objectives by answering the following questions.

RQ2: How prevalent is the defect rediscovery phenomenon in commercial and open source software projects?

RQ3: What are the factors that influence the accuracy of the predictive models?

1.4 Proposed Solution

In order to address RQ1, first, we extract data from one commercial and several open source software projects. We write a custom-built web scraper to extract defect report data from the defect tracking system (Bugzilla) of the open source software projects. We mine the Bugzilla engine of Eclipse, Gentoo, KDE, and LibreOffice (details of the data extraction presented in Section 4.1). We collect the data from the Enterprise software project through its structured relational database system.

Second, we implement a Defect Recommender System (DRS) to proactively identify the defects a client may encounter in the future. In case of recommender systems, the basic idea is to predict the items that the user of the system is going to be interested in based on some historical data, such as, previous interactions with the system, what like

minded users are interested in, and items that are very similar to the previously liked items. For example, a movie recommender recommends movies to the user based on the movies watched by the user in the past, currently popular or trending movies, and the preference of the like-minded users (what similar users also watched). In our case, we have built our defect recommender system (DRS) based on the defect reports (which in recommender system terminology becomes item) by the users. Basically we predict defect rediscovery based on the previously reported defects by a user, the preference of the similar reporters, and the most commonly reported defects.

To address RQ2, we provide an in-depth analysis of the defect rediscovery data collected from both commercial and open source software projects in Section 4.2. We compare the data from different projects using various statistical techniques. Finally, to answer RQ3, we examine the cases where the model fails to perform in Section 4.3.4. We apply a Random Forest (RF) and a Naive Bayes classifier to distinguish such cases and use the variable importance measure of RF to identify the driving factors of the failed cases.

1.5 Novelty

To the best of our knowledge, there is no other work that implements recommender systems to predict defect rediscoveries by the customers. Although other researchers have studied the prediction/detection of duplicate/rediscovered defects, ours is the first study that predicts defect rediscoveries from the perspective of the customers with the intention to increase the customer-satisfaction of the software project. Furthermore, we investigate the inter-relations of rediscovered defects, which has not been studied before.

We analyze $\approx$ 1.3 million unique defect reports from one commercial and four different open source large scale software projects each of them having thousands of users worldwide. We provide a solution on reducing the sparsity for such sparse data in the context of software engineering and how recommender systems can be used to proactively eliminate potential defect rediscoveries. We show that partitioning the data using product-component yields the highest accuracy in terms of the recommender systems.

We present a comprehensive and reproducible approach using commonly used recommender systems including popularity based and k-Nearest Neighbors based recommender

algorithms. We show that simply recommending the most frequently reported defects for a given product-component may reduce the number of defect rediscoveries because the popularity-based recommender is the best performing algorithm. We find empirical evidence that there exists similarity among the users of a software product because the k-Nearest Neighbour based recommender performs as the second best algorithm while predicting rediscoveries.

1.6 Contribution

The major contributions of this work can be summarized as:

- A set of techniques to connect duplicate defects and identify defect rediscovery information in software projects.
- A novel approach, to reduce number of defect rediscovery leveraging recommender systems algorithms in order to increase customer satisfaction and better manage resource allocation.
- Three extracted rediscovery datasets from open source projects shared with the Software Engineering community in CSV, MySQL, and Neo4j formats [67, 66, 68]. The data is available in open access data sharing repository [68]: `https://doi.org/10.5281/zenodo.400614`
- A prototype tool to extract rediscovery data from Bugzilla-based defect report tracking systems, listed in Appendix B.
- A prototype tool implementing the novel approach, core features[1] of which are listed in Appendix C.

1.7 Outline

In Chapter 2, we provide related works and a brief literature review on defect report analysis. In Chapter 3, we introduce the methodologies that we use in order to built

[1]Full code base of the tool will be shared via GitHub.

our predictive model and the approaches followed to evaluate the model. In Chapter 4 we present the data analysis and the analysis of the results of our model. Finally, in Chapter 5, we provide a summary of this study, as well as a conclusion and a direction towards future work.

Chapter 2

Literature Review

There has been multiple studies done in the past in software engineering literature that leverage duplicate defect reports. For example, one can detect duplicate reports to speed up report triaging (deduplication) [65, 10] and identification of the root cause of failure [15], or to predict defect rediscoveries in order to proactively eliminate defects before a customer finds [9], or to improve resource allocation to optimally manage the workforce [60], or to predict bug priority to improve planning [77], or to build customer profiles to improve quality assurance processes [58], or to automatically assign defect reports to owners to speed up time-to-fix of defects [13].

In this chapter, we review the related research works that either leverage rediscovered/duplicate defect reports or the research works that highlight the importance of reducing the number of defect rediscovery (or defects in general) in a software project. Additionally, even though we differentiate our study from defect report analysis studies, we review the literature from this field of study as they are related. This includes studies on defect report prioritization, duplicate defect report detection, quality of defect reports, misclassified defect reports, and predicting the severity of defect reports.

2.1 Triage Leveraging Duplicate Defect Reports

Many researchers have investigated the methodologies to detect duplicate defect reports in order to speed up triaging [65, 10]. Most of the studies focus on either the textual similarity between the defects or the stack trace information. Hiew [36] was first to

use Natural Language Processing (NLP) to detect duplicate defects. Later, Runeson et al. [65] used a more sophisticated approach leveraging a vector space model as well as combined multiple textual attributes. Alipour et al. [10] applied existing NLP based techniques along with a set of contextual word lists to detect duplicate reports. Sureka and Jalote [76] introduced a n-gram based detection approach instead of typical word based approach.

Another group of studies focus on the stack trace data in order to detect duplicates [81, 74, 25]. Stack trace contains information about the software execution as a crash report. Such information typically helps to understand the characteristics of a software defect. Therefore, many researchers used this technique to identify duplicates.

The main difference between our study with the studies of duplicate detection is that we do not predict if a defect report relates to a duplicate report/defect rediscovery. Rather we predict, for a given set of rediscovered defects, which are the defects a specific user may rediscover in the future. The prediction problem we deal with is more user-centric; and thus we choose recommender systems as the proposed solution.

Avik et al. [13] proposed a machine learning system to automatically assign defect reports to developers (automatic traiaging). The authors observed that the *assigned_to* field does not update often and point to the actual developer (who fixes the defect) in case of defects labelled with the *duplicate* tag. Therefore, they derived heuristics such as using the information from the original defect the duplicate is related to.

Researchers analysed defect report prioritization using rediscovery information. Tian et al. [77] leveraged duplicate reports data while predicting priority level of a defect report since the similar defects may share the same priority level. Therefore, during model-training time they used a set of duplicate defects. The authors adopted a modified version of REP which is a state-of-the-art technique to measure similarity between two defects, proposed by Sun et al. [75]. The modified version includes textual summary and description fields as well as the product and component of the defects. They discarded the priority field because their main objective was to predict the priority level for a given software defect.

The main difference between our study with defect report prioritization related studies is that, we proactively identify defects the users may encounter in the future so that some preventive actions can still be recommended. However, defect reports prioritization studies focus on the final solution or fix without eliminating immediate risk of potential

defect rediscovery. Our results may be used for prioritization before a defect is fixed. But, this is not the main goal of this research.

2.2 Reducing Defect Rediscovery

There are not many studies done in the area of reducing defect rediscoveries in software projects; however, the importance of minimising the number of defects in a software is critical to its success: some authors have suggested in the past that the number of defects existing in a software is closely related to customer satisfaction [18]. Researchers have developed different metrics related to defect rediscovery in order to benefit software providers and increase customer satisfaction. An earlier study presented a set of metrics to estimate the risk associated with defect rediscovery [60]. Another metric was proposed to improve product support that measures the probability of a customer detecting a defect within a short period of time [61].

In the seminal paper in 1984, Adams [9] assessed when preventive services should be applied to avoid defect rediscovery. He reported that by applying preventive services for the defects that have been rediscovered many times early in the life cycle of the software, one can achieve the most benefits. Adams also discussed how the maximisation of preventive services is not desirable and how one should optimise preventive services.

Researchers have developed different metrics related to defect rediscovery in order to benefit software providers and increase customer satisfaction. A study presented a set of metrics to estimate the risk associated with defect rediscovery [60]. According to this study the information on number of defect rediscoveries can be leveraged by the software maintenance, support, and quality assurance teams. Upon receiving a support ticket, the support team must verify first if the ticket is a rediscovery of a existing defect or not. Increased number of rediscoveries cause slower transfer of support tickets to maintenance teams. Similarly, maintenance teams need to know when to expect maximum defect rediscovery in order to better prepare for creating special builds for clients and to estimate allocation of personnel. A large number of defect rediscovery in a frequently used software component can be used to pinpoint software testing related failure by the Quality Assurance Team [60]. Another study proposed a metric to improve product support that measures the probability of a customer detecting a defect within a

short period of time [61].

In order to better understand software reliability and system outages, researchers have analysed the data from a software product that had geographically distributed user base (which is also the case in our study) and suggested metrics such as the number of rediscoveries per defect and the time window between first and last rediscovery of a given defect [22].

In order to prioritise fixes and allocate staffing, many researchers investigated distributions of defect rediscoveries. They observed that the distribution can be either thin-tailed [80] or heavy-tailed [9, 62, 60], depending on the data under study. In general, software engineering processes, it is common to have a heavy-tailed distribution [53].

To the best of our knowledge, there is no other work that implements recommender systems for defect rediscoveries. The work, that is closest to ours, identifies several methods to reduce the number of defect rediscovery by customers [85]. These methods include 1) making fixes available quickly for severe defects (defects that cause a large number of rediscovery), 2) releasing fixes that are available, in the soonest possible update rather than waiting to complete more fixes, 3) making announcements about availability of fixes and creating easily installable fixes, 4) taking preventive measures against severe defects by doing root cause analysis for existing severe defects. However, this work is complementary to ours, as they focused only on severe defect and generalised to all the customers, whereas, we take into account all the defects and create personalised predictions for each customer.

In our case study in this research, we used different defect rediscovery metrics previously used by other researchers in order to understand how we can reduce the number of rediscoveries [9, 22, 61, 59].

2.3 Defect Report Analysis

Apart from defect report triage and defect rediscovery studies, there are general defect report analysis studies. This type of studies deal with defect report optimization problem (improving report quality) [14, 47, 86, 40, 12, 35].

Bettenburg et al. [14] analysed the quality of defect reports by conducting a developer survey for Eclipse project. According to the developers the inclusion of reproducibility

and stack trace in the report are the most helpful data. In addition, the authors reported that the developers typically do not consider duplicate reports as harmful, because sometimes these reports contain additional description.

Lamkanfi and Demeyer [47] and found that defect reports often contain incorrect information in some of the fields of the reports. The main reason is that the users of the software are not completely aware of all the technical aspects of the software (i.e., product, component, etc.). As a result, the triager may need to manually correct this information. The authors proposed machine learning approach to predict incorrect component field in defect reports.

Xie et al. [86] analysed the impact of introducing non-developer-triagers in optimizing defect reports. The optimizing activities involve filtering defect reports, completing incomplete reports, and mapping products to reports. The authors observed that the traigers were efficient filtering invalid defects whereas not so efficient in mapping products to reports.

Wu et al. [40] also found that defect reports are often incomplete. They built a machine learning model using Support Vector Machine (SVM) in order to predict the values of the missing fields based on the previous data of the software projects under study. In addition, they detect if a new report is a duplicate or not based on textual similarity measures such as cosine distance.

Another group of studies focus on identifying defect reports that do not relates to actual defects. The problem can be termed as defect report misclassification problem as per Antoniol et al. [12]. The authors differntiated defects from other issues using Decision Tree, Naive-Bayes, and Logistic Regression based approaches. Herzig et al. [35] reported that two out of five defect reports are wrongly classified as defects, especially in open source projects.

Chapter 3

Methodology

In this chapter we provide general overview of recommender system (RS), followed by discussion on RS used in our study. In Section 3.1 we present a brief overview of RS. Then, we discuss different aspects of RS, including the type of user feedback used in RS in Section 3.2, the Top-N recommendations in Section 3.3, the input user-item rating matrix to RS in Seciton 3.4, and the recommender algorithms used in our study in Section 3.5. RS often suffers from lack of information, so called sparsity problem, which we discuss in Section 3.6. We tackle the problem by partitioning the data, as discussed in 3.6.1. The evaluation of the RS's performance is given in Section 3.7. Throughout the chapter, we use the term *item* and *defect* interchangeably because item is a more common term in RS terminology and in our context an item and a defect are equivalent.

3.1 Recommender Systems

Recommender System (RS) is a popular technology used by different organisations (e.g., Amazon [1], Netflix [5], and Spotify [7]) in various domains such as e-commerce [72], news [20], and entertainment websites [45]. RS uses statistical and knowledge discovery approaches in order to create recommendations based on historical transaction data [69]. These recommendations help users to find relevant products or items from a plethora of choices. The basic idea is to predict the products or items that the user of the system is going to be interested in, based on the user's previous interactions with the system. For example, e-commerce website Amazon recommends products to a user based on the

user's interests [51]. Some recommender systems also consider the interactions of the similar users with the system, while presenting recommendations to a user. For example, when a user visits the web page of a movie in Internet Movie Database or IMDb [3], it recommends a list of other movies that people — who liked a given movie — also liked.

3.2 User Feedback for Recommender Systems

Recommender Systems rely heavily on users' feedback. The quality of the user feedback is the key to successful recommendations [70]. It is crucial to study the characteristics of user feedback data in order understand the design and evaluation of recommender algorithms based on different types of user feedback data. Generally, there are two types of user feedback in recommender systems [52]:

- Explicit User Feedback,
- Implicit User Feedback.

3.2.1 Explicit User Feedback

Explicit user feedback tells us how much the user likes or is interested in an item. This type of user feedback is readily available, whereas the implicit user feedback is gathered by observing the user's interactions with the system [52]. The explicit user feedback is typically ordinal and more common when the user expresses specific opinion or preference about the product using a rating interface [23]. The rating interface is a scale that depicts how much the user likes or dislikes the product. The scale can be finite (e.g., 1 to 5 Stars Ratings in Amazon), continuous (e.g., any real value within -10 to 10 in Jester Online Joke Recommender [4]) or even binary (e.g., like and dislike for YouTube [8] videos).

3.2.2 Implicit User Feedback

The implicit feedback is more common when the user's interest or opinion is inferred from implicit user actions, such as clicks and browsing history (i.e., when a customer browses or buys a product on Amazon or plays a song on Spotify) [37, 52]. Implicit user feedback can be collected by observing the user's interactions with the system. In many

cases, the implicit user feedback is unary or positive-only [37], which means that we have items that the user may like but we do not have the items that the user dislikes. For example, when a user plays a song, it means that the user may like the song. However, it does not mean that the user dislikes thousands of other songs that s/he did not play.

We have built our Defect Recommender System (DRS) based on defect (which in RS terminology becomes item) reports by the users. This type of data can be classified as positive-only-implicit user feedback.

3.3 Top-N Recommendations

RS returns Top-N recommended items, sorted from most desirable to least desirable. For example, in the case of DRS system, Top-3 returned items would represent ids of three defect reports that this user may rediscover in the future. The first defect in the Top-3 list has the highest probability of rediscovery, the second one – the second highest, and the third one – the third highest.

Essentially, these are the defects for which PS should be applied. For example, let us assume that there are 100 unique defects and 50 unique customers for a software product. A subset of the customers made a defect report and no customer made more than one defect report. In a traditional setting, the worst case assumption is that each of the 50 customers may rediscover the 100 defects (if defect reports by that customer are not taken into acoount). Therefore, it will require 100 preventive services for each customer and in total $100 \times 50 = 5000$ PS for the software product. This naive solution maximises PS to minimize rediscovery. The total number of required PS is also very high and it is nearly impossible to provide so many PS to the customers due to increased downtime and resource allocation issues. Moreover, the 100 defects will not be rediscovered concurrently, which may cause multiple time windows of downtime. On the contrary, the DRS only recommends the Top-N most probable defects for each customer by taking into account all the defect reports made by the customer. Therefore, it will minimise the number of rediscoveries without maximising the number of PS.

In case of DRS, the Top-N values, for the sake of brevity, were set to $N = 1, 3, 5, 10, 20$ (so that they may apply preventive services for the Top-N defects and minimise the number of rediscoveries without maximising the number of preventive services). Depending

Table 3.1: Sample rating matrix capturing information about defect (re)discoveries.

	d_1	d_2	d_3	d_4	d_5	d_6
u_1	0	0	1	0	1	0
u_2	0	1	1	1	0	0
u_3	1	1	0	0	0	0
u_4	0	1	0	0	0	0
u_5	1	0	1	0	0	1

on the amount of resources that a support team has and the tolerance to false positives, a given development shop can pick a value of N that suits their needs. One can argue that in the case of Top-20, all 20 patches will be installed simultaneously, thus not increasing downtime needed for patch application significantly. Of course, the threat here is that installing 20 patches rather than one leads to increase of probability of regressing functionality.

3.4 Rating Matrix for RS

The input to the recommender algorithms is a $m \times n$ rating matrix which is also known as the Utility Matrix; where m denotes the number of users and n denotes the number of items. In case of positive only implicit user feedback, a non-zero cell in the rating matrix corresponds to a rating. A rating in RS terminology means the user rated/liked/bought an item (in our context, it becomes a user rediscovered a defect). An example of a rating matrix, storing information about defect (re)discoveries, is shown in Table 3.1. In this case we have a 5×6 rating matrix with five users $U = \{u_1, u_2, u_3, u_4, u_5\}$ and six defects $D = \{d_1, d_2, d_3, d_4, d_5, d_6\}$. Ones in the table represent defect discovery/rediscovery by users; zeroes – the opposite. For example, u_2 (re)discovered defect d_3, but u_3 – did not.

3.5 Recommender Algorithms

Below we provide information about the four approaches used in our DRS: namely, Random-, Popular-, User-, and Naive-Bayes-based, given in Sections 3.5.1 – 3.5.4, respectively. The first three approaches are implemented in `recommenderlab` [32] R package. We implemented the fourth, Naive-Bayes-based approach as an extension to the

`recommenderlab` package, for consistency of our experiments.

3.5.1 Random items

In the case of random items algorithm, DRS chooses at random N defects and returns them to a user as Top-N defects that the user may rediscover in the future. For example, as per Table 3.1, the Random-items will recommend the Top-2 defects for user u_5, by randomly sampling 2 defects without replacement from the following set of defects:

$$\{\{d_1, d_2, d_3, d_4, d_5, d_6\} - \{d_1, d_3, d_6\}\}. \tag{3.1}$$

Random-items will not recommend defects from $\{d_1, d_3, d_6\}$ as they have been already reported by user u_5. This is the most naive approach and is used as a baseline in our study.

3.5.2 Popular items

In the case of popular items algorithms, DRS sorts defects by the historic number of rediscoveries in descending order and return Top-N defects as the ones that a given user may rediscover in the future. The algorithm is based on the assumption that a frequently rediscovered defect resides in a commonly executed path, which suggests that it relates to a popular or core functionality of the product [59]. Therefore, this user may rediscover it as well.

For example, as per Table 3.1, let's assume we are predicting the re-discoveries for the user u_5. The total number of report counts for each defect are:

$$\{(d_1, 2), (d_2, 3), (d_3, 3), (d_4, 1), (d_5, 1), (d_6, 1)\}. \tag{3.2}$$

Popular-items sort these defects on the historic number of rediscoveries in descending order. As a result, we get the following sorted list of defects:

$$\{(d_2, 3), (d_3, 3), (d_1, 2), (d_4, 1), (d_5, 1), (d_6, 1)\}. \tag{3.3}$$

The Top-2 defect recommended for u_5 by Popular-items will be d_2 and d_4 since u_5 already

reported d_1, d_3, and d_6.

3.5.3 User-based Collaborative Filtering

Collaborative filtering (CF) is one of the most common recommendation techniques and is being used for many years [44]. CF systems use the knowledge of the crowd [42, 33]. It means that an item is recommended to a user based on the preferences of a set of users who have some degree of similarity with the target user. This set of similar users are referred to as the crowd or the neighbours and the system uses their predilections as a knowledge base while providing a new recommendation to the target user [33]. The neighbours are identified by applying statistical techniques on the historical feedback data. Typically, the neighbours have similar opinions or preferences with the target user. They may rate products similarly, listen to the same songs, or buy items of the same type more frequently.

In a User Based Collaborative Filtering (UBCF) scenario, there is a set of users, $U = \{u_1, u_2, \ldots, u_n\}$ and a set of items $I = \{i_1, i_2, \ldots, i_n\}$ (items becomes defects in our context). Each user rates some of the items. The rating is a numeric value (explicit feedback) or unary value (implicit feedback). The rating r_{ui} represents a rating expressed by the user u_i for the item i_j and is typically stored in a rating matrix. The active user u_a is the person for whom the rating is being predicted. The basic idea behind the prediction is that the active user u_a will have similarity in preferences with a similar set of users.

The neighbourhood S_a of the active user u_a is a set of users similar to u_a. It is typically formed by calculating similarity scores between the active user and every other user, then comparing the scores with a threshold score or, alternatively, considering the k users most similar to u_a (k nearest neighbours) [33]. The result is an ordered set of users, based on the similarity score. So, the first user in the set S_a will be the most similar user to the active user u_a. Pearson correlation coefficient or cosine similarity can be used to compute similarity in case of explicit feedback [48].

Similarity Measure

In the case of implicit feedback or positive only data, we know only which items are favoured by the user, which is generally expressed by a Boolean value (i.e., TRUE-FALSE

or 1-0). Calculation of similarity using the Jaccard similarity index is recommended in such a situation [31]. The Jaccard similarity index can be expressed as:

$$Sim_J(u_a, u_i) = \frac{|U_a \cap U_i|}{|U_a \cup U_i|}, \tag{3.4}$$

where u_a is the active user, u_i is the other user, U_a is the set of items favoured or rated positively by the active user and U_i is the set of items favoured by the other user. The numerator in Eq. 3.4 yields the number of items that the active user and the other user rated positively simultaneously (intersection of the sets). The denominator yields the total number of all the items rated positively by both of the users (union of the sets). This is the measure of similarity that we are going to use in our implementation of the UBCF, since what we have is positive-only-implicit user feedback: defects found by users are marked by 1s and not found – by 0s. For example, the similarity between u_2 and u_3 in Table 3.1 is:

$$Sim_J(u_2, u_3) = \frac{|\{d_2\}|}{|\{d_1, d_2, d_3, d_4\}|} = 1/4 \tag{3.5}$$

Rating Calculation

After the neighbourhood S_a of the active user u_a is defined, the missing rating $\hat{r_a}(j')$ of an item j' for the active user u_a is predicted by aggregating the ratings for that item in the neighbourhood:

$$\hat{r_a}(j') = \frac{1}{|S_a|} \sum_{k \in S_a} r_k(j') \tag{3.6}$$

The final output of the UBCF is a set of Top-N items/defects with the highest predicted ratings. These are the items that the active user is likely to prefer most.

As an example, let us assume the user u_5 in Table 3.1 is the active user for whom we want to predict defect rediscoveries.

First, we calculate the similarity between the active user (u_5) and all other users (u_1, u_2, u_3, u_4) as per Section 3.5.3 using Jaccard similarity measure. The similarities are: $Sim_J(u_5, u_1) = \frac{1}{4}, Sim_J(u_5, u_2) = \frac{1}{5}, Sim_J(u_5, u_3) = \frac{1}{4}$, and $Sim_J(u_5, u_4) = \frac{0}{4}$.

Second, in case of a 3-Nearest Neighbour system we aggregate the ratings of the three

most similar users (based on Sim_J) to u_5: u_1, u_2, and u_3, for the defects that are not yet reported by u_5 using $\hat{r_a}(j')$. The valuse of $\hat{r_a}(j')$ are $d_2 : \frac{2}{3}, d_4 : \frac{1}{3}, d_5 : \frac{1}{3}$. Therefore, the Top-1 defect for user u_5 by UBCF would be defect d_2.

3.5.4 Naive-Bayes-based

We also created a simple probabilistic classifier based on Naive-Bayes (N-BAYES) approach. In the scope of this classifier, each defect can have two classes associated with it: defect-reported class C_r (when a defect is reported by a user) and defect-not-reported class C_n (when a defect is not reported by a user). For example, let us assume that we want to compute the two class probabilities for the cell $[1, 3]$ in the rating matrix provided in Table 3.1. This can be expressed as:

- $P(C_r|u = u_1, d = d_3)$ referring to the probability of defect-reported, given the user id $u = u_1$ and defect id $d = d_3$.
- $P(C_n|u = u_1, d = d_3)$ referring to the probability of defect-not-reported, given the user id, $u = u_1$ and defect id, $d = d_3$.

.

We apply Bayes theorem [11], to calculate $P(C_r|u, d)$ and $P(C_n|u, d)$:

$$P(C_r|u_i, d_j) = \frac{P(C_r) \times P(u_i|C_r) \times P(d_j|C_r)}{P(u_i) \times P(d_j)}, \tag{3.7}$$

where, $P(u_i|C_r)$ represents the number of times user u_i reported a defect over the total number defects reported by all users, when we consider only the reported-defects and $P(d_j|C_r)$ represents the number of times defect d_j was reported over the number of times all the defects were reported, when we consider only the reported-defects. Analogously, $P(C_n|u_i, d_j)$ is computed as

$$P(C_n|u_i, d_j) = \frac{P(C_n) \times P(u_i|C_n) \times P(d_j|C_n)}{P(u_i) \times P(d_j)}, \tag{3.8}$$

where, $P(u_i|C_n)$ represents the number of times user u_i did not report a defect over the total number defects not reported by all users, when we consider only the not-reported-defects and $P(d_j|C_n)$ represents the number of times defect d_j was not reported over

the number of times all the defects were not reported, when we consider only the not-reported-defects.

Finally, we compare the probabilities of a given defect and user belonging to each class: if $P(C_r|u_i, d_j) > P(C_n|u_i, d_j)$, then we assume that a given defect will be rediscovered, else it will not be. Note that since denominators in Eqs. 3.7 and 3.8 are the same, we do not have to compute $P(u_i)$ and $P(d_j)$ for the purpose of comparison, setting it to a dummy value of 1.

We then generate personalised list of defect, returning Top-N defects that have the highest values of $P(C_r|u_i, d_j)$ (sorted in descending order).

As an example, for user u_5 let us see what is the Top-1 recommendation by N-BAYES. First, we need to calculate the class probabilities for the defects that are not yet reported by u_5. These defects are d_2, d_4, and d_5. For, defect d_2 and user u_5 the probabilities based on Eq. 3.7 and 3.8 are:

$$P(C_r|u_5, d_2) = P(C_r) \times P(u_5|C_r) \times P(d_2|C_r) = \frac{11}{30} \times \frac{3}{11} \times \frac{3}{11} \approx 0.03, \tag{3.9}$$

$$P(C_n|u_5, d_2) = P(C_n) \times P(u_5|C_n) \times P(d_2|C_n) = \frac{19}{30} \times \frac{3}{19} \times \frac{2}{19} \approx 0.01. \tag{3.10}$$

Because $P(C_r) > P(C_n)$, we can conclude that the probability that u_5 will report d_2 is ≈ 0.03. Similarly, for (u_5, d_4) and (u_5, d_5), we calculate $P(C_r|u_5, d_4)$, $P(C_n|u_5, d_4)$ and $P(C_r|u_5, d_5)$, $P(C_n|u_5, d_5)$, respectively. In both of these cases, $P(C_r) < P(C_n)$ and, thus, d_4 and d_5 are discarded from potential Top-N recommendations. For user u_5, N-BAYES recommends d_2 as the Top-1 recommendation[1].

3.6 Sparsity Problem

Recommender systems in general, and collaborative filtering systems in particular suffer from the lack (sparsity) of information [70], especially in a high-dimensional space [38].

[1] If d_4 or d_5 had $P(C_r) > P(C_n)$, the Top-1 would have been the defect with the highest $P(C_r)$ among defects d_2, d_4, and d_5.

The rating matrices in real world are sparse. There is a large number of users and items. However, only a few items will be rated/encountered by a user. For example, most users will buy small number of items sold on Amazon (out of hundreds of millions items being offered [30] or watch a handful of movies on Netflix (out of thousands being offered [54]). Similarly, in a good quality software product a client rediscovers only a handful of defects [9, 60].

3.6.1 Reducing the Sparsity of the Datasets

To tackle the sparsity problem, researchers have proposed several solutions to reduce the sparsity of the rating matrix [29]. The users who rated only handful of items or items that have very few ratings can be removed [29]. There exists other sophisticated techniques, such as, Latent Semantic Analysis [26] and clustering-based approaches [49, 71].

Partitioning by software product-component

One way to overcome sparsity of the data is to partition the data by software product and component. By product in this study, we denote the software subsystem the report belongs to (i.e., JDT is a product of Eclipse project). By component we mean a group of functions (in C/C++ sense of the term) dedicated to implementing a specific functionality. For example[2], code compiler can be split into multiple components, such as, scanner, parser, and optimiser.

The split by component allows to partition dataset by functionality, which becomes important as not every customer will use each and every feature. Of course, there is quite a number of common components that will be executed by all the users (e.g., scanner or parser in the case of compiler). Defects uncovered in such components will be of interest to all the clients.

However there exist optional components too. For example, Intel C++ compiler is shipped with code coverage tool that can provide information about code covered during execution of test cases [2]. We can think of this tool as an individual component. The defects exposed in this component will be of interest only to the clients using it; the

[2]Another example would be Eclipse project, which has a product JDT consisting of multiple components, such as, UI, Debug, Text, and Core.

rest of the clients can safely ignore most of them (example of exception: critical security defect).

From computational perspective, per product-component partitioning allows to significantly reduce both (users and items/defects) dimensions of the rating matrix; which, hopefully, will improve predictive power of the models. Results of partitioning by product-component are discussed in Section 4.3.

In addition, we explore some clustering based approaches with limited success which we describe in Appendix A.

3.7 Evaluation

There are several evaluation metrics to evaluate the performance of a recommender system. For explicit user feedback or numeric rating based recommender systems, error measures, such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE), are often used [24]. However, for implicit feedback data sets in recommender systems, classification metrics [46] popular in the field of Information Retrieval, such as Precision and Recall, are used. On the other hand, these metrics are more useful in comparing the performance of different recommender algorithms, rather than evaluating how good a recommender is [34]. This is because each user typically rates a very small fraction of the available items in the data set and these metrics are dependable upon the number of items rated by the user. Therefore, in this study, we have used F-measure and a graphical evaluation technique called Receiver Operating Characteristics (ROC) curve [57] as our evaluation metrics.

3.7.1 Accuracy Metrics

The ROC curve visually represents the trade-off between two metrics, the True Positive Rate (TPR) and the False Positive Rate (FPR) across the different thresholds (i.e., the numbers of Top-N recommendations [42, 31]). TPR is defined as the proportion of correctly classified positive instances and is plotted on the Y-axis of the ROC curve plot:

$$TPR = \frac{TP}{(TP + FN)}, \tag{3.11}$$

where TP is the number of true positive results, FP is the number of false positive results, TN is the number of true negative results, and FN is the number of false negative results. FPR is defined as the proportion of the incorrectly classified negative instances and plotted on the X-axis of the ROC curve plot:

$$FPR = \frac{FP}{(FP + TN)}. \tag{3.12}$$

TPR and FPR ranges between 0 and 1. The ideal curve would have $TPR = 1$ and $FPR = 0$ for all N.

For selecting the best algorithm, we use F-measure, which combines precision and recall (to obtain a balanced measure) into a single metric:

$$F - measure = \frac{2 \times TP}{(2 \times TP + FN + FP)}. \tag{3.13}$$

3.8 Validation

As our validation scheme, we have used a temporal splitting technique. We divide the training and testing set based on the temporal attribute in the data in order to simulate the real-world setting where the predictive model would be applied as new data arrives. This kind of chronological splitting based evaluation scheme has been used by many previous studies that analyze defect reports [77, 36, 63, 65].

We develop four different temporal splitting schema in order to understand how the models perform (in terms of the predictive accuracy) while varying the amount of historical training and testing data. For each schema, we split the the dataset into several folds. The number of folds for each schema depends on the time-interval-increment dt, measured in years. The smallest value of $dt = 1$. Details of the schema are given below.

Schema #1 is defined in Algorithm 1. In this schema we keep accumulating historical training data and test on 'immediate' future test data. In the example given in Figure 3.1, in fold #1, we train on the data of year 1 and test on the data of year 2. In fold #2, we train on the data of years 1 and 2 and test on the data of year 3. In fold #3, we train on the data from years 1-3 and test on the data gathered in year 4, etc.

Schema #2 is shown in Algorithm 2. In this schema we train the model on recent historical data and test on 'immediate' future test data. Going back to the example in

Figure 3.1, in fold #1, we train the model on the data of year 1 and test the model on the data of year 2. In fold #2, we train on the data of year 2 and test on the data of year 3. In fold #3, we train on the data from year 3 and test on the data from year 4, and so on.

Schema #3 is given in Algorithm 3. In this case, we always train on 'recent' historical data and test on all subsequent future test data. Note that, the last fold of the Schema #3 and #2 are the same. In the example shown in Figure 3.1, in fold #1, we train on the data of year 1 and test on the data of years 2-10. In fold #2, we train on the data of year 2 and test on the data of years 2-10. In fold #3, we train on the data of year 3 and test on the data of years 4-10, etc.

Schema #4 is depicted in Algorithm 4. In this schema we accumulate both training data and testing data. Note that, the last fold of schema #4 and #1 are the same. In the example given in Figure 3.1, in fold #1, we train on the data of year 1 and test on the data of years 2-10. In fold #2, we train on the data of years 1-2 and test on the data of years 3-10. In fold #3, we train on the data from years 1-3 and test on the data from years 4-10, and so on.

input : A list of unique year values (year_list)
output: The training and testing set time intervals

$t_0 \leftarrow min(year\_list)$;
$t_f \leftarrow max(year\_list)$;
$dt \leftarrow 1$;

```
/* [ or ] closed interval                                    */
/* ( or ) opened interval                                    */
```

for $i \leftarrow t_0 + dt$ **to** t_f **by** dt **do**
  $training\_set\_time\_interval \leftarrow [t_0, i)$;
  $test\_set\_time\_interval \leftarrow [i, i + dt)$;
end

Algorithm 1: Temporal Split: Schema 1

input : A list of unique year values (year_list)
output: The training and testing set time intervals

$t_0 \leftarrow min(year\_list)$;
$t_f \leftarrow max(year\_list)$;
$dt \leftarrow 1$;

```
/* [ or ] closed interval                                    */
/* ( or ) opened interval                                    */
```

for $i \leftarrow t_0 + dt$ **to** t_f **by** dt **do**
    $training\_set\_time\_interval \leftarrow [i - dt, i)$;
    $test\_set\_time\_interval \leftarrow [i, i + dt)$;
end

Algorithm 2: Temporal Split: Schema 2

input : A list of unique year values (year_list)
output: The training and testing set time intervals

$t_0 \leftarrow min(year\_list)$;
$t_f \leftarrow max(year\_list)$;
$dt \leftarrow 1$;

```
/* [ or ] closed interval                                    */
/* ( or ) opened interval                                    */
```

for $i \leftarrow t_0 + dt$ **to** t_f **by** dt **do**
    $training\_set\_time\_interval \leftarrow [i - dt, i)$;
    $test\_set\_time\_interval \leftarrow [i, t_f]$;
end

Algorithm 3: Temporal Split: Schema 3

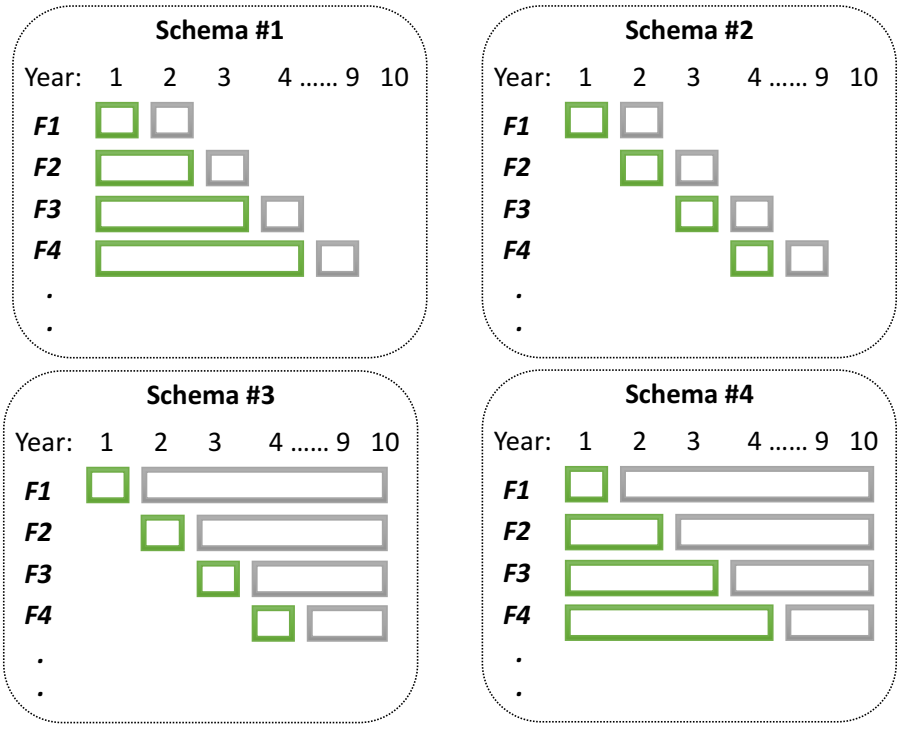

Figure 3.1: Temporal Splitting: Schematic Diagram. The diagram represents how we split a dataset containing defect reports from ten consecutive years into training and testing sets. Time-interval-increment (dt) is set to 1 year. The green boxes represent the training-set-time-interval and the grey boxes represent the testing-set-time-interval.

input : A list of unique year values (year_list)
output: The training and testing set time intervals

$t_0 \leftarrow min(year\_list)$;
$t_f \leftarrow max(year\_list)$;
$dt \leftarrow 1$;

```
/* [ or ] closed interval                                    */
/* ( or ) opened interval                                    */
```

for $i \leftarrow t_0 + dt$ **to** t_f **by** dt **do**
  $training\_set\_time\_interval \leftarrow [t_0, i)$;
  $test\_set\_time\_interval \leftarrow [i, t_f]$;
end

Algorithm 4: Temporal Split: Schema 4

Chapter 4

Evaluation

In this chapter, we apply our proposed solution to our research problem using the methodologies presented in the previous chapter. We start with discussing how we extract the defect rediscovery data from the Bugzilla based defect tracking system of several open source software projects in Section 4.1. We provide an analysis of the datasets focusing on the defect rediscovery phenomena in Section 4.2. We present the results of our proposed approach; and a discussion on how we apply our predictive models, and evaluate them as well as the limits beyond which our models do not work in Section 4.3. We conclude the chapter by discussing the threats to validity in Section 4.4.

4.1 Data Extraction

We mined bug repositories of four groups of open source software projects: Eclipse, Gentoo, LibreOffice, and KDE. We extract the defect rediscovery information from the defect reports publicly available in these repositories. The final datasets contain information about approximately 1.3 million defects that have been reported in the last 15-18 years (depending on the project). Some of the resulting datasets are located at `http://doi.org/10.5281/zenodo.400614`.

In addition, we mined the defect rediscovery information of a large-scale enterprise software product. We extract the data from a defect database system of this software manufacturer. This dataset contain information about defects that have been reported between the year 2007 and 2015. We denote this software in the rest of the study as

ENT.

For each group of the open source software projects, the set of attributes that we extracted from each defect report are described below.

- **id:** The unique integer identifier that identifies a report.
- **product:** The name of the software subsystem the report belongs to.
- **component:** The name of the component the report is associated with.
- **reporter:** The unique username of the person who opened the report.
- **bug_status:** The current status of the report.
- **resolution:** The current resolution of the report.
- **priority:** It represents how quickly the defect described in the report should be fixed.
- **bug_severity:** It refers to the degree of impact the reported defect has on the whole system.
- **version:** The version the defect was observed in.
- **short_desc:** A short textual summary of the report.
- **opendate:** The date when the report was opened.
- **dup_list:** The list of *id*s of duplicates of a given report; if the report does not have any duplicates – the value is an empty string.
- **root_id:** The *id* of the root vertex of the graph of rediscoveries, which typically resembles the master report. If the report does not have any duplicates – the value is an empty string. This is a derived attribute.
- **disc_id:** The *id* of the oldest defect (i.e., the one that is opened first) in the graph of rediscoveries. If the defect does not have any duplicates – the value is an empty string. This is a derived attribute.

Data Extraction Procedure

We performed the following four extraction and transformation steps to obtain the attributes described in Section 4.1.

Step 1: Retrieval of report *id*s. For each of the software projects we selected, we mined its Bugzilla defect tracking system which numbers defect reports sequentially with an integer *id*, with the first *id* set to 1.

Given the sequential nature of the data, we query a given Bugzilla engine for reports opened within the last seven days (at the day of data gathering) and select the maximum *id* value, denoted by $I_{\max}$ returned by the engine. Thus, for a given engine the range of reports *id*s is set to $[1, I_{\max}]$.

Step 2: Data mining and extraction. The data were extracted using a custom-built web scraper. The input to the scraper was the range of *id*s to be mined - identified in the previous step. The scraper outputs all the attributes mentioned in Section 4.1 (except the two derived attributes) in *CSV* format (one line per report), saving intermediate results, as the extraction process takes several days to complete.

Step 3: Construction of the dataset. First, we aggregate all intermediate results for a given project in a single *CSV* file.

Second, we eliminate rows from the *CSV* file for which a report either does not exist or is not available. The former may happen because the report may get cancelled by a user before submission or may be erased by a bug tracker administrator. The latter may happen because we do not have sufficient permissions to access a given report. The former case cannot bias our dataset, as the data does not exist. However, the latter case may lead to bias, if the number of reports that we cannot access is large. We built a script that computed the number of *id*s associated with each case (by analysing error messages returned by the bug tracking engine). Details of our analysis are provided in Table 4.1.

Step 4: Construction of derived attributes. In order to construct derived attributes, we built a directed graph G linking *id* with its duplicates using information stored in the *dup_list* attribute. Going back to example given in Figure 1.2, report

#19274 has two duplicates linked to it (#23194 and #23196), as per the *dup_list* attribute. Thus, we will add to the G two edges: $19274 \rightarrow 23194$ and $19274 \rightarrow 23196$. We repeat this process for each report in a given dataset. We then use Graphviz software [27] to identify all 'connected components' (in the graph theory sense of the term) in the G. The resulting connected components represent the graph of rediscoveries for each of the original defects. An example of such connected component is given in Figure 1.2.

We then analyze each graph of rediscoveries (connected component) and identify the root vertex (typically, this report is a master report) and the vertex associated with the *id* with the oldest *opendate*. The former becomes *root_id* value for each report associated with a given graph of rediscoveries; the latter value becomes *disc_id*. For example, in case of Figure 1.2, the *root_id* value for all the reports will be set to 6325 and *disc_id* to 4671 (since, by design of the Bugzilla defect tracking system, the smaller the defect *id* – the older the defect). Then, we merge the original dataset with the derived attributes and store the resulting dataset in the *CSV*, *SQL*, and *Neo4j* formats.

4.2 Dataset Analysis

The summary statistics of the datasets are given in Table 4.1. The number of reports that we gathered (column 'Total accessible reports count') ranges from ≈ 13 thousands for ENT to ≈ 504 thousands for Eclipse. The reports were opened between years 1999 and 2017.

The total count of unique defect reports and reporters are illustrated in Figure 4.1. The Y-axis in the barplot is in log scale. For example, in case of Eclipse there are $503,935$ unique defect reports reported by $46,993$ unique reporters. The Eclipse, KDE, and Gentoo dataset have lower unique reporter to report ratio, whereas the ENT and LibreOffice dataset have much higher unique reporter to report ratio.

The distributions of the total number of reports (obtained by combining rediscovery and original defect count, as discussed in Section 1.1.3) for a given failure are given in Figure 4.2. The distributions are heavy-tailed as evident from the linear structure of the data plotted on the log-log scale.

As discussed in Section 4.1, we could not access some of the reports. The percentage of such reports (shown in column 'Inaccessible reports count') is small: 0.1% for Eclipse,

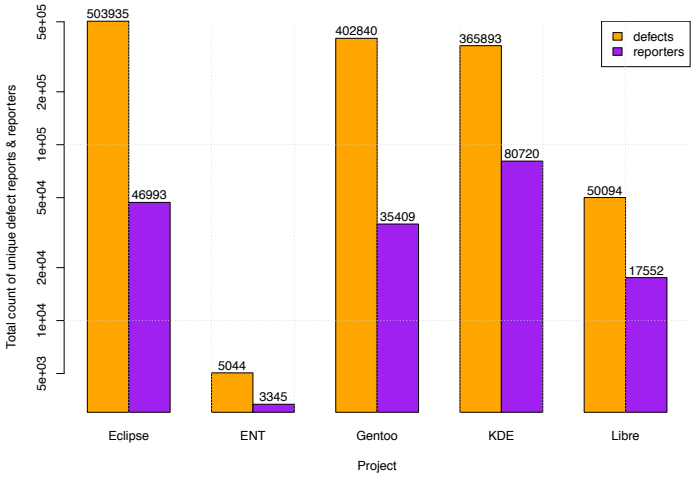

Figure 4.1: Unique defect reports and reporters count for each project. Note that y-axis has log scale.

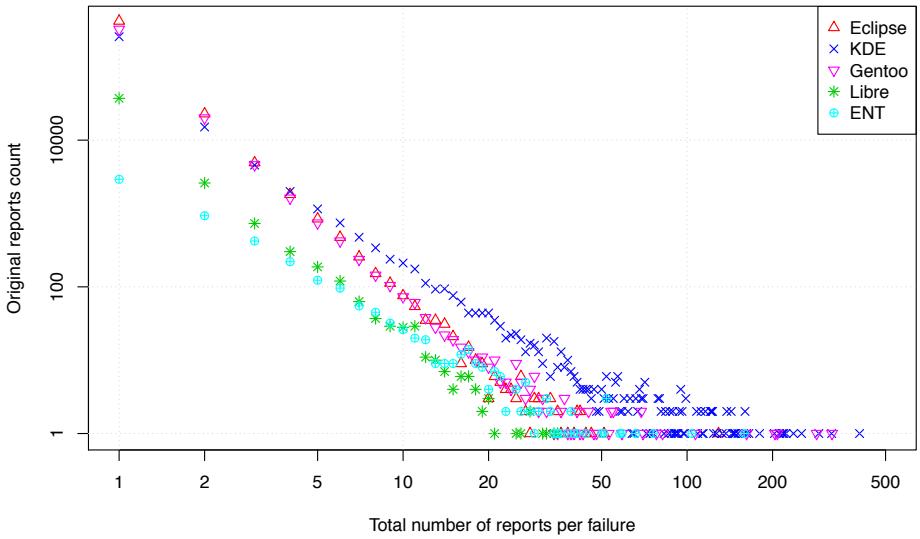

Figure 4.2: Count of the total number or reports for a given failure vs. count of original reports. If a given failure was reported once, then it means that it was never rediscovered; reported twice – means that it was rediscovered once, and so on (see Section 1.1.3 for details). For example, KDE dataset has 257420 reports that were never rediscovered (i.e., discovered once) and 15106 reports that were rediscovered once (i.e., discovered twice).

Table 4.1: Summary statistics.

Project name	Total accessible reports count	Inaccessible reports count	Rediscoveries count	Distinct *disc_id* count	Min report *opendate* (YY-MM-DD)	Max report *opendate* (YY-MM-DD)	Max number of rediscoveries	Distinct *products* count	Distinct *product-components* count	Non-rediscovered reports (% of total)
Eclipse	503,935	560	52,499	31,811	01-10-10	17-02-07	128	232	1,486	83
KDE	365,893	4,818	82,359	26,114	99-01-21	17-02-13	405	584	2,054	70
Gentoo	402,840	205,014	50,082	28,333	02-01-04	17-01-31	324	15	168	81
Libre	50,094	55,881	8,718	4,192	10-08-03	17-02-13	35	12	43	74
ENT	13,112	0	8,068	2,120	07-06-11	15-10-02	159	1	185	22

1.3% for KDE and is moderate: 34% for Gentoo, 53% for LibreOffice. In case of ENT all reports were accessible.

To gather information about original discoveries and rediscoveries of reports, as discussed in Section 4.1, we analysed graphs of rediscoveries (similar to the one shown in Figure 1.2). Such graphs can become fairly large: based on Table 4.1, the maximum number of rediscoveries of an original report (per graph of rediscoveries) ranges from 35 for LibreOffice to 405 for KDE.

The percentage of the original reports that were rediscovered at least once ranges from 7% (28333/402840) for KDE to 16% (2120/13112) for ENT . The distributions of the total number of reports (obtained by combining rediscovery and original defect count, as discussed in Section 1.1.3) for a given failure are given in Figure 4.2. The distributions are heavy-tailed as evident from the linear structure of the data plotted on the log-log scale.

In Figure 4.3 and 4.4, we present the per year analysis. The data are current as of February 2017, thus the dataset for year 2017 is incomplete, hence the "dip" in reports for year 2017. By construction, zero observations for a given year are not shown.

The number of reports per year changes, as seen in Figure 4.3. Magnitude-wise, the number of reports ranges from thousands for ENT to tens of thousands for Eclipse, Gentoo, and KDE (with the exception of the first and last reporting year for each project).

Overall, percentage of reports that are not rediscovered ranges between 70% for KDE and 83% for Eclipse. However, these values change from year to year, as shown in Figure 4.4. This figure may suggest that for the last seven years percentage of non-rediscovered reports grows up (albeit non-monotonically). For example, for defects opened in 2016, the percentage of non-rediscovered defects ranges from 75% for KDE to 92% for Eclipse (compare these numbers with the average values of 70% and 83%, respectively).

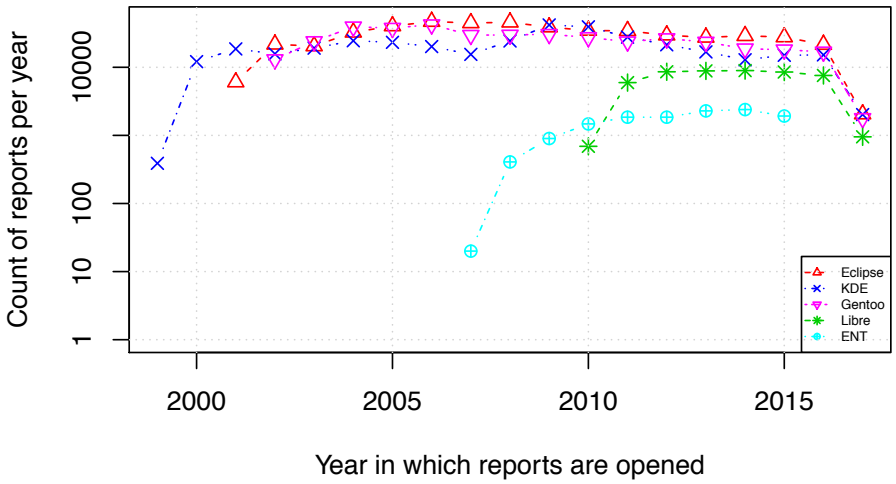

Figure 4.3: Per-year analysis: Number of reports per year.

However, in the future, users may encounter and report some of the defects discussed in these non-rediscovered reports. This will lead to reduction of the number of non-rediscovered reports opened in previous years. To confirm this conjecture, we plot the distribution of time intervals between the opening dates of the original discovery and the latest rediscovery, shown in Figure 4.5. The figure suggests that some reports get rediscovered years after the original discovery.

Even for the graph of rediscoveries shown in Figure 1.2, the time interval between open dates of the original report #4671 and its latest rediscovery #31201 was ≈ 1.3 years. The number of *product*s per open source project ranges from 12 for LibreOffice to 584 for KDE; the number of *product-component* tuples per project – from 43 for LibreOffice to 2054 for KDE. The percentage of reports that are not rediscovered per product-component is given in Figure 4.6. For the open source the software projects, the median percentage ranges between 84% for KDE to 96% for Eclipse. For the enterprise software project, the median percentage is 31%. However, there are outliers with low percentage of non-rediscovered defects, suggesting that different components may exhibit different behaviour. Therefore, various *product-component*s may be studied independently.

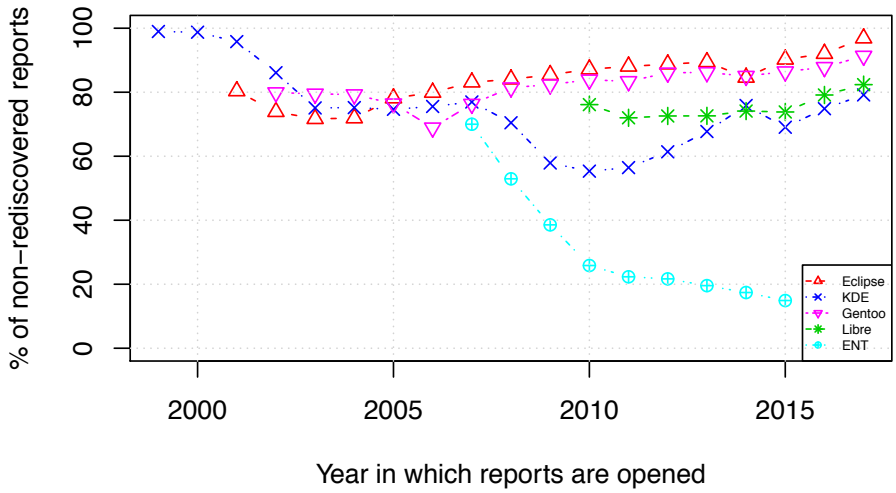

Figure 4.4: Per-year analysis: Percent of reports that have not been (yet) rediscovered.

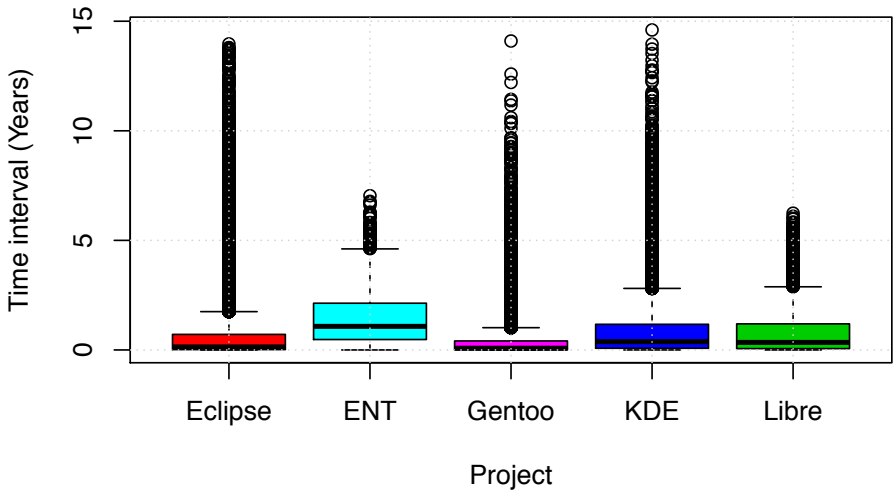

Figure 4.5: Distributions of time intervals between the original discovery and the latest rediscovery for a given graph of rediscoveries.

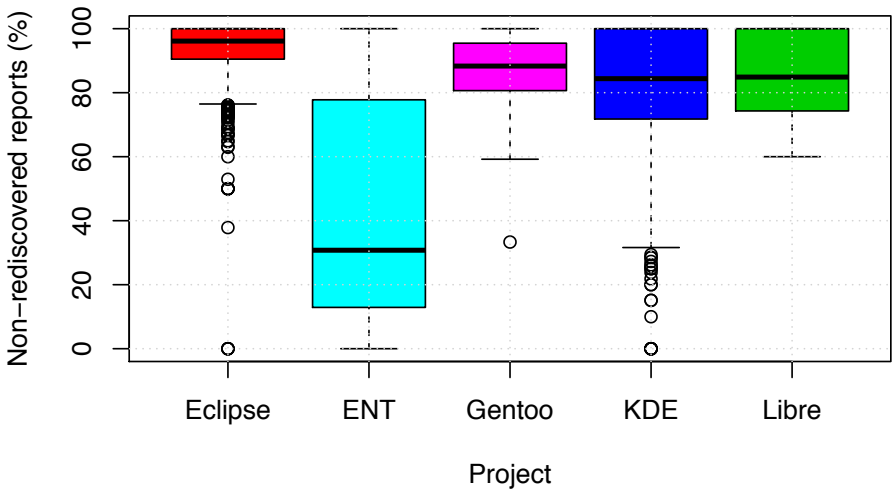

Figure 4.6: Distribution of non-rediscovered reports per *product-component*.

4.3 Discussion

In this section, we discuss how we apply our proposed approach to reduce defect rediscovery on the datasets that we gathered and the results that we achieve in Section 4.3.1. We also discuss how we evaluated the models' performance in terms of the schemas (Section 4.3.2), algorithms (Section 4.3.3), and the failed cases (Section 4.3.4).

4.3.1 Rediscovery Prediction

In this section, we provide details about the application of RS algorithms (presented in Section 3.1).

We measure the sparsity of the datasets by calculating the percentage of non-zero elements in the rating matrix. Non-zero elements refer to ratings in recommender system terminology (which becomes discovery/rediscovery in our context). This is essentially the positive implicit user feedback (discussed in Section 3.2). The percentage of non-zero

elements in the rating matrix for each project, denoted by α, is computed as:

$$\alpha = \frac{|r_{ui} = 1|}{|U| \times |D|} \times 100, \tag{4.1}$$

where $|r_{ui} = 1|$ is the number of reported (re)discoveries, $|U|$ is the number of unique users, and $|D|$ is the number of unique defects. The values of α for each project are presented in Table 4.2 as Pre-split α. The numbers suggests that we do have a very sparse rating matrix, as discussed in Section 3.6. Therefore, we apply the '*partitioning by product-component*' technique described in Section 3.6.1 in order to reduce the sparsity of the rating matrix. The resultant median α values across product-components are presented in Table 4.2 as Post-split α. These α values suggests, our partitioning technique produces much denser rating matrix.

The distribution of the percentage of non-zero elements in the rating matrix of each subset of data partitioned by product-component for each project is given in Figure 4.8. The median value range between $\approx 1\%$ and $\approx 3\%$, which are significantly higher values than in the case of the original matrix (depicted in Table 4.2). The distributions suggest that in case of the enterprise software the split by product-component yields the maximum median non-zero elements in each split. In case of the open source projects, KDE has the highest median non-zero elements when we split by the product-components.

After partitioning the data for each project by product-components, we divide the resulting data into several folds of temporal training and testing sets based on the four time-split schema as discussed in Section 3.8. The number of folds depends on the time range (in years) of the software project and the time-interval-increment (dt). We experiment the time-interval-increment by setting $dt = 1, 2$, or 3 years and comparing the prediction accuracy of the model with respect to dt. We find that setting $dt = 1$ year yields good results for the temporal splitting schema, as shwon in Figure 4.7.

We run the Defect Recommender System for each project, for each selected product-component, for each temporal splitting schema, for each fold of the schema, and for each Top-N value. We store all configuration attributes in a log file along with the accuracy metrics (discussed in Section 3.7). We calculate the accuracy metrics by evaluating the predicted defect rediscoveries by DRS with the actual data. Essentially, we follow, the $given-x$ experimental protocol introduced by Breese et al. [17]. As per this protocol, for each user, out of n items, we give the recommender x items and withhold $n-x$ items from

the recommender. Then, we evaluate the predictive performance of the recommender only on the withhold items.

```
for project in {Eclipse, ENT, Gentoo, KDE, Libre} do
    for product_component in all product_components of project do
        for schema in {schema1, schema2, schema3, schema4} do
            for temporal_fold = 1 to number_of_folds do
                get train_data and test_data of temporal_fold ;
                for N in {1, 3, 5, 10, 20} do
                    run POPULAR with train_data and test_data;
                    run UBCF with train_data and test_data;
                    run RANDOM with train_data and test_data;
                    run NBAYES with train_data and test_data;
                    get best_performer among
                     {POPULAR, UBCF, RANDOM, NBAYES} ;
                end
            end
        end
    end
end
```

Algorithm 5: Selection of the DRS-best performing algorithm for each case

Lastly, using the accuracy metrics, we select the DRS-best performing algorithm for each case. By case, we mean each time we split the data into train and test set by product-component, temporal-splitting schema, and fold as depicted by Algorithm 5. We use the best performing algorithm's predictive accuracy to evaluate different schemas and recommender algorithms in the following sections.

4.3.2 Which Schema is the Best One?

The accuracy of the models varies across the temporal splitting schemas (we describe the temporal splitting in Section 3.8). We illustrate this variation using Figure 4.9. The figure, essentially, represents ROC curve plot, where we change the value of N (in $Top-N$) to adjust the 'threshold'.

Each data point in the ROC plot represents the best performing algorithm's mean FPR and mean TPR values for a given Top-N, across all the product-components and

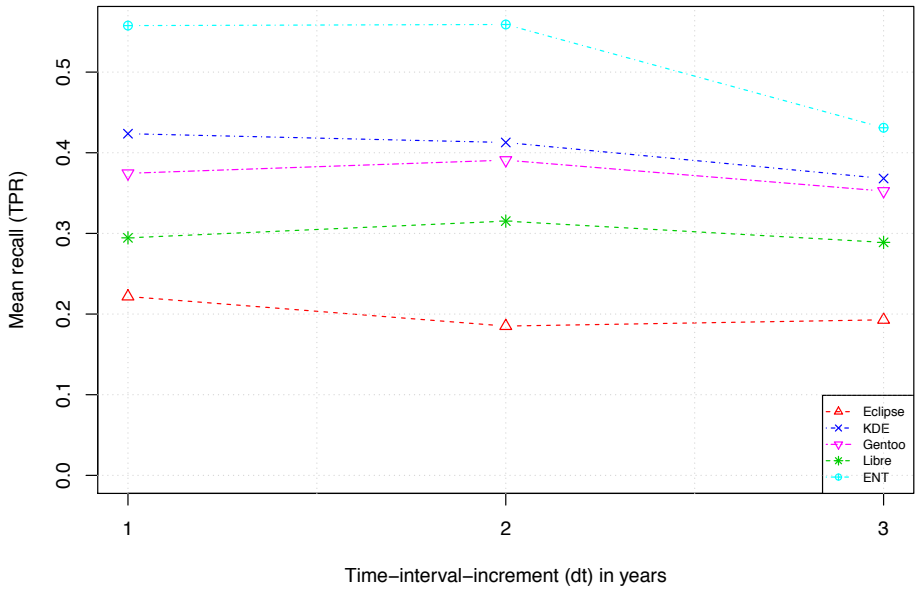

Figure 4.7: Mean TPR while changing dt=1 to dt=3, by 1 year.

Table 4.2: Percentage of non-zero elements (α) for each project without splitting and median α after splitting by product-components

Project	Pre-split α	Post-split α
Eclipse	0.0021	0.4456
Gentoo	0.0028	0.1536
KDE	0.0012	0.7205
Libre	0.0057	0.4292
ENT	0.0777	2.9759

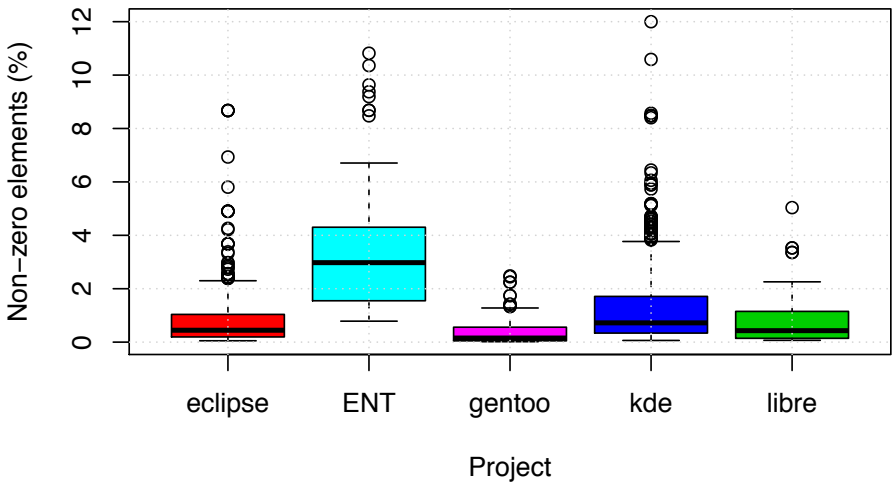

Figure 4.8: Distribution of non-zero elements in the per component analysis.

all the folds of the given temporal split schema. The data points for the best performing algorithm is ordered: the left-most data point has $N = 1$, the next one – $N = 3$, then $N = 5, 10$, until we reach $N = 20$. The vertical and horizontal error bars show one standard deviation of the mean TPR and FPR.

Both FPR and TPR increase monotonically with the increase of N. However, TPR grows faster than FPR. This can be explained by the fact that increasing the number of defects returned gives more chances to get the correct result. For example, in the case of $N = 20$, the model needs to return at most 1 defect that was actually (re)discovered out of 20, in order for this outcome be deemed as true positive.

For Eclipse dataset, according to Figure 4.9, schema #2 and #3 have overall higher TPR and at the same time lower FPR, consistently for all values of Top-N. So for Eclipse, when we train on 1 year of data and test on 1 or more years of data, DRS yields the best results. We reach maximum mean TPR $\approx 60\%$ and FPR $\approx 37\%$ with a standard deviation of 0.41 and 0.25 respectively for Top-20 recommendations.

For Gentoo dataset, schema #2 and #3 are again performing better in terms of TPR and FPR. In case of schema #2, we reach maximum mean TPR $\approx 60\%$ and FPR $\approx 20\%$ with a standard deviation of 0.43 and 0.19 respectively for Top-20 recommendations. In

the case of schema #3, we reach maximum mean TPR ≈ 66% and FPR ≈ 20% with a standard deviation of 0.36 and 0.18 respectively.

For ENT dataset, although all four schema have high TPR, the FPR is also high. In contrast to the open source projects, where we observe a more significant difference between some of the schemas. Across the four schemas, the maximum mean TPR ranges between ≈ 90% and ≈ 100%, at the same time with a maximum mean FPR that ranges between ≈ 60% and ≈ 100%. The maximum TPR and FPR standard deviation ranges between 0.01 and 0.21 and 0.03 and 0.29 respectively. However, for ENT schema #1 and #4 gives better TPR to FPR ratio. So, for the enterprise software, accumulating training data from the previous years yield better performance. In the case of ENT dataset, we achieve higher TPR at the expense of FPR (which also increases).

For KDE dataset, similar to the cases of Eclipse and Gentoo, schema #2 and #3 have overall higher TPR and at the same time lower FPR in comparison with schema #1 and #4. When we train on 1 year of data and test on 1 or more years of data, DRS yeilds the best results. In case of schema #2 and #3 the maximum mean TPR ranges between ≈ 80% and ≈ 83% with standard deviation between 0.28 and 0.29, whereas the mean FPR ranges between ≈ 54% and ≈ 60% with standard deviation of 0.34.

For LibreOffice dataset, considering all the values of Top-N, schema #2 performs best with a maximum mean TPR ≈ 60% and FPR ≈ 46% with a standard deviation of 0.49 and 0.37 respectively.

As we can see from Figure 4.9, the decision to pick to the best temporal splitting schema depends on the size and type of the project. For example, in case of large scale open source project schema #2 and #3 yields the better results, whereas in case of enterprise software, accumulating rediscovery data from previous years slightly improves the accuracy. Therefore, it would be better to leave it to an analyst to select the appropriate temporal splitting schema for a particular dataset. The decision will depend on the business goals and the comfort of organization with different TPR and FPR values.

4.3.3 Which algorithm is the Best One?

We begin our evaluation of DRS by applying Random-, Popular-, User-, and N-Bayes-based algorithms (discussed in Sections 3.1) on the partitioned data for each software project. The performance of the top performing algorithms of DRS are shown in Fig-

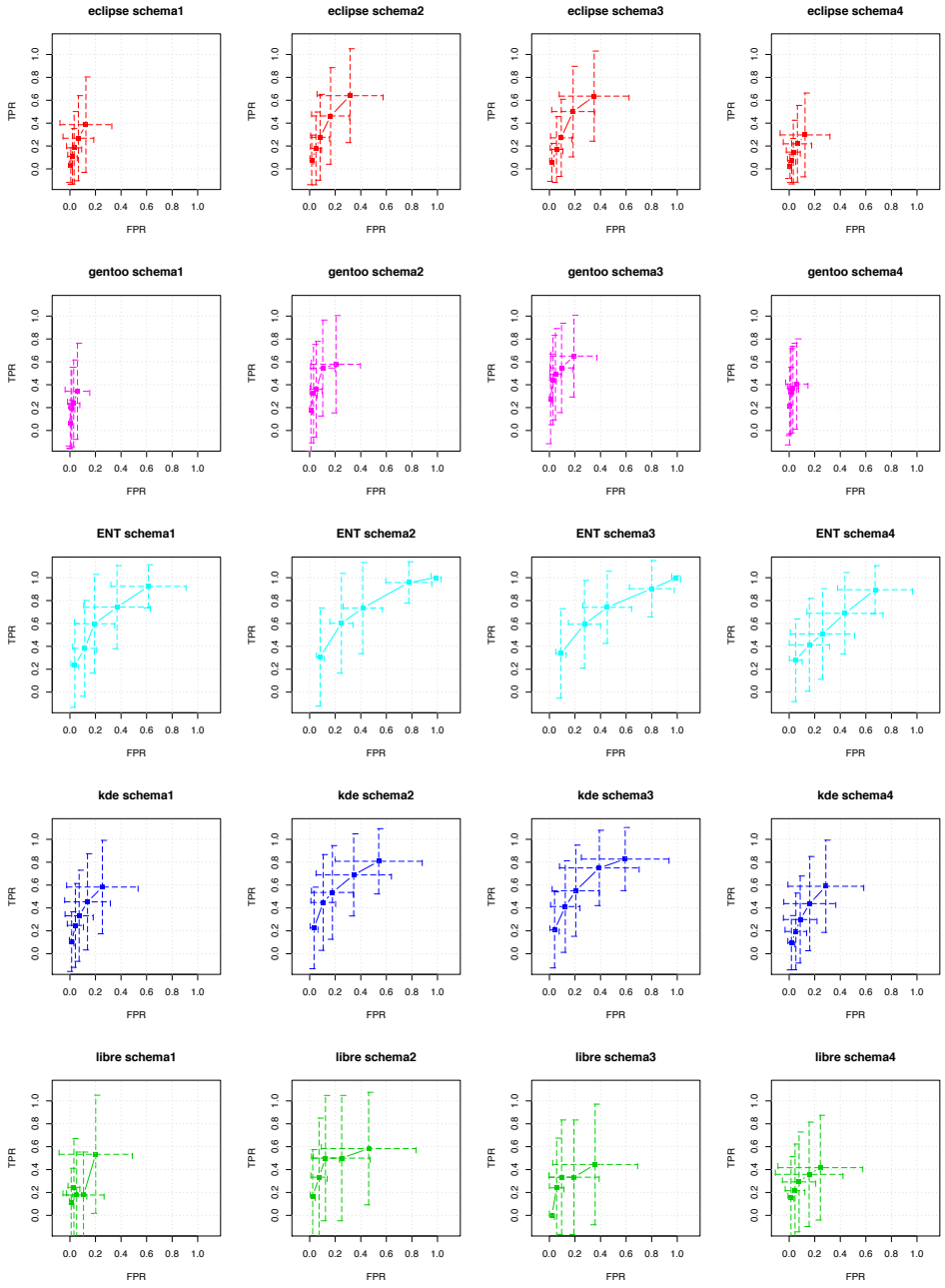

Figure 4.9: ROC plots for each temporal-splitting-schema and for each dataset. The thresholds of the curve are the values of Top-N = 1,3,5,10,20. The Y and X axis represent the mean FPR and TPR respectively. Error bars represent one standard deviation spread from either side of the mean.

ure 4.10.

In case of the user-based approach, given the nature of our data, we use Jaccard Similarity (Eq. 3.4) as our measure of similarity between the users. A series of experiments were conducted to identify the ideal size (k) of the neighbourhood as per [11]. We varied the value of k from 3 to 35 incrementing by 2 and $k = 25$ gave us the best results. Thus, $k = 25$ was chosen as the neighbourhood size.

To asses performance of the algorithms, we examined their performance for all possible splits and across all components for a given dataset and schema; examples are shown in Figure 4.10. The figure shows that the performance of the algorithm may change with N. Moreover, the shape of the curves fluctuates from dataset to dataset and schema to schema, however, the Popular- and User- based are the top performer in most times than the other two algorithms. To aggregate the performance data, Figure 4.10 shows the percentage of the winning cases across all components, where each of the algorithms performs best for a given dataset, schema, and N. Performance is evaluated using F-measure (Eq. 3.13).

Table 4.3 shows the summary of the top-performing algorithms combining all temporal splitting schemas. The Popular- and User- based algorithm are the two top performing algorithms. For example, in case of Eclipse, POPULAR is the top-performer in $\approx 79\%$ of the cases and UBCF is the top-performer in $\approx 12\%$ of the cases. For each algorithm, we show the breakdown of the actual number of cases per schema in separate tables. Tables 4.9, 4.10, 4.11, and 4.12 show the actual number of top-performing cases per schema for each algorithm, for each dataset, and all values of Top-N.

We conjecture that the potential reason for the Popular-based algorithm to be the best performer is the rediscovery phenomenon. As shown in Figure 4.2, there exist defects in each software project that have been rediscovered hundreds of times. Thus, the frequentist reasoning prevails in such cases. The User-based approach is the second best one, which suggests that there exists some similarity between the users of a software product. However, the number of such cases is not high, because we do not supply the model with enough information about the users. This is due the fact that 75% of the users[1] reported, at the most, one or two defects in the lifetime of the software

[1]Note that when we say 'users' we mean a fraction of the customer base that actually reported a problem. Obviously, we do not have information about customers who use the product without ever encountering a problem.

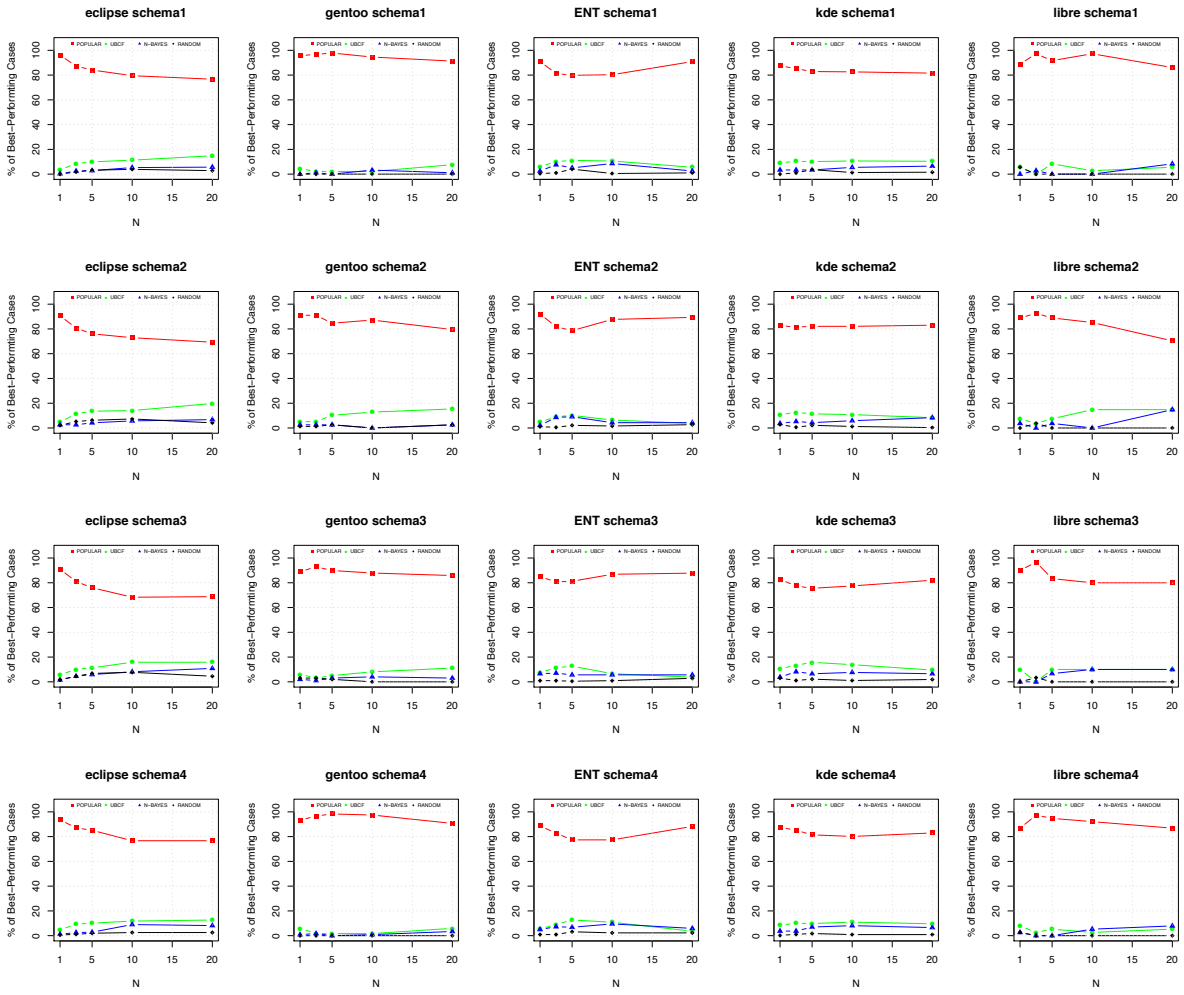

Figure 4.10: Best Performing Algorithm for each schema and dataset

projects under study. As a result, we do not have sufficient information about users' characteristics.

4.3.4 What drives models' failure?

After we complete the evaluation of top-performing schemas and algorithms, we examine the cases where DRS can not predict. By case, we mean a unique split in terms of product-component, temporal splitting schema, and fold. In this section, we present our analysis of the factors leading to models' failure.

Table 4.3: Summary of the best-performing algorithms (in %) incorporating all schemas

	N-BAYES	POPULAR	RANDOM	UBCF
Eclipse	3.71	79.37	5.12	11.80
Gentoo	0.67	89.13	1.74	8.46
ENT	2.44	79.56	7.89	10.11
KDE	1.34	82.06	5.56	11.05
Libre	0.85	91.06	3.83	4.26

Table 4.4: Confusion Matrix of the Random Forest Classifier

		Predicted as,	
		can-not-predict	can-predict
Actual,	can-not-predict	1302	42
	can-predict	95	318

As we run DRS on the partitioned datasets (unique splits), we store the characteristics of the input data structure (the rating matrix) in a log file for further analysis of the models' performance. Essentially, we have two input rating matrix, one for the training and the other for testing. The attributes that we keep track of are listed in Table 4.5. If the models fail to predict a single defect rediscovery (True-Positive), we assign the case to the class: *can-not-predict*, and *can-predict* otherwise. We show the frequency of cases belonging to each class in Table 4.8.

Then, we merge all cases and in order to identify the important attributes, we train a Random Forest classifier using the attributes mentioned in Table 4.5. We use the implementation of the classifier from `randomForest` package in **R** with a configuration of 100 trees [50]. We randomly split 70% of the cases into training set and 30% data to test. The overall accuracy of the classifier on the test set is $\approx 92\%$. We present the predictive performance on the test set in a confusion matrix in Table 4.4:

The Random Forest classifier creates multiple decision trees with a different combination of a random subset of variables. The classifier takes into account the vote of each tree while making a classification. The finally predicted class is obtained by aggregating the predictions from all trees. Random Forest classifier can estimate the importance of the attributes used. The attributes that have more predictive power are likely to influence more. This helps us to identify the important factors leading to models' failure.

To identify the most influential factors that drive models' failure, we use Random

Table 4.5: List of factors potentially influencing models' performance

Attribute	**Description**
tr_users	Number of unique users in training set
tr_defects	Number of unique defects in training set
tr_ratings	Number of unique ratings in training set
tst_users	Number of unique users in testing set
tst_defects	Number of unique defects in testing set
tst_ratings	Number of unique ratings in testing set
present_tr_present_tst	Number of defects rediscovered by both train and test users
present_tr_absent_tst	Number of defects rediscovered by only training users
absent_tr_present_tst	Number of defects rediscovered by only testing users
class	Two classes: whether model *can-predict* or *can-not-predict*

Forest's variable importance measure [28]. The coefficient that we use is called Mean Decrease in Gini. It measures the effect of each variable in the homogeneity of the decision node. Every time a variable is used to partition the tree into children, the difference in Gini coefficient between the child node and the parent node is calculated. A predictor variable with a higher decrease in Gini indicates that it has more influence in partitioning the data into classes.

We present the importance of the factors in terms of the gini coefficient in Figure 4.11. We can see that the *tst_ratings*, the *tst_users*, and *present_tr_present_tst* are the three most important factors in terms of predictive power. The *tst_ratings* and the *tst_users* correspond to the number of unique rediscoveries and users in the testing set respectively. The *present_tr_present_tst* represents the number of common rediscoveries in both training and testings set .

Table 4.6: Statistical Analysis of the three most important factors. The table shows means of the attributes plus-minus standard deviation (s.d.).

Attribute	Can-predict (mean ± s.d.)	Can-not-predict (mean ± s.d.)
tst_users	59.36 ± 75.1	8.8 ± 12.07
tst_ratings	63.96 ± 82.33	8.83 ± 12.1
present_tr_present_tst	19.42 ± 17.86	4.19 ± 3.28

To verify our findings, we again train a different classifier. We use the Naive-Bayes classifier with a 10-fold validation scheme using the three most important factors returned

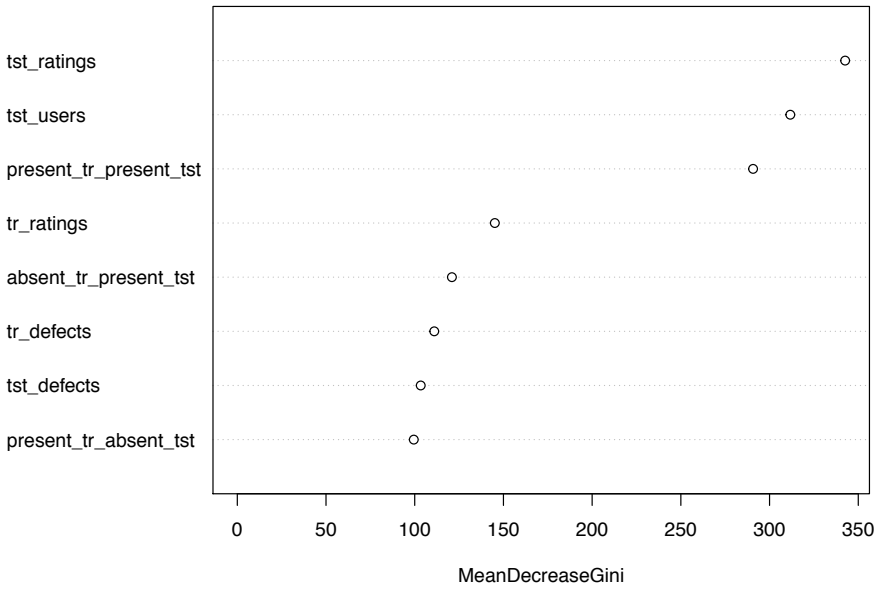

Figure 4.11: Most influential factors as per Random Forest classifier

Table 4.7: Confusion Matrix of the Naive Bayes Classifier

		Predicted as,	
		can-not-predict	can-predict
Actual,	can-not-predict	4326	164
	can-predict	580	785

by Random-Forest. We get overall accuracy of $\approx$ 87% with the Naive-Bayes. The confusion matrix is shown in Table 4.7.

We compare the statistical measures (namely, mean and standard deviation) of the three most important factors against each class. We present this analysis in Table 4.6. According to the analysis, we find that the model belongs to can-not-predict class, if we do not have sufficient data about defect rediscovery. For example, compare the number of unique users in the test set (tst_users) for the can-not-predict class ($\approx 9 \pm 12$) with the ones for the can-predict class ($\approx 59 \pm 75$). Similar picture holds for the number of unique rating in testing set tst_ratings ($\approx 64 \pm 82$ vs. $\approx 9 \pm 12$) and the number of defects rediscovered by both train and test present_tr_present_tst ($\approx 19 \pm 18$ vs. $\approx 4 \pm 3$).

Table 4.8: Frequency of the two class-attributes for each dataset

Project	Can-not-predict	can-predict
KDE	2453	554
Eclipse	1015	410
ENT	411	194
Gentoo	458	160
Libre	153	47

Therefore, the larger is the number of users, the higher is the number of rediscoveries (i.e., the less sparse the matrix is), and the more rediscoveries of the same defect are there between train and test set – the better the performance of the models.

4.4 Threats to Validity

In this section we discuss threats to validity, classified as per [87, 84].

Internal Validity

We do not have access to a number of reports, which may bias our dataset (as discussed in Section 4.2). However, given that the percentage of such reports is small: 0.1% for Eclipse, 1.3% for KDE and is moderate: 34% for Gentoo, 53% for LibreOffice, the dataset should not be affected significantly.

In addition, some of the reports that are currently non-rediscovered may be rediscovered in the future (as discussed in Section 4.2). This has to be taken into consideration during data analysis. In the case of ENT, all the reports were available.

Customers may under-report defect rediscoveries, skewing the distribution of rediscoveries (shown in Figure 4.2). Two main types of defects are not reported to the service desk: 1) defects with low severity with obvious workarounds; and 2) rediscoveries of known defects that were not fixed last time. Under-reporting may affect quality estimates of the software, but it will not affect resource allocation of in-house service and maintenance team. As far as they are concerned, a defect that is not reported does not exist. These situations are a nuisance to the clients, but the clients typically find solutions in a minimum amount of time. Thus, from practical perspective, absence of these data can be ignored.

Construct Validity

In order to reduce the threat to validity related to *human errors*, we automated the process of data gathering and analysis, reducing the risk of human error. Python, R, and SQL-based scripts were created to extract, transform, and analyse the data.

Conclusion Validity

In order to prevent *over-fitting* of the models, we have used four different temporal splitting techniques. We pick $dt = 1$ year which ensures we have the maximum number of folds for each temporal splitting schema.

External Validity

As described by Wieringa and Daneva [83], software engineering studies suffer from the variability of the real world, and the *generalisation* problem cannot be solved completely. As they indicate, to build a theory we need to generalise to a theoretical population and have adequate knowledge of the architectural similarity relation that defines the theoretical population. In this study, even though we have used the data from five different software projects (include both open source and enterprise) to build the DRS, we can not generalise our results to all software projects. Our goal of this study was not building a new theory, rather we wanted to achieve a deeper understating of how established machine learning techniques perform in the rediscovery domain. Our approach on reducing defect rediscoveries can be applied to other software products with well-designed and controlled experiments.

Table 4.9: Best Algorithms for Schema-1 for each dataset and for each Top-N value

dataset	TopN	N-BAYES	POPULAR	RANDOM	UBCF
Eclipse	1	0	106	0	4
Eclipse	3	3	93	4	10
Eclipse	5	6	86	4	14
Eclipse	10	8	82	6	14
Eclipse	20	4	78	7	21
ENT	1	0	40	2	3
ENT	3	1	35	2	7
ENT	5	2	30	5	8
ENT	10	0	33	6	6
ENT	20	0	38	3	4
Gentoo	1	0	35	0	1
Gentoo	3	0	32	2	2
Gentoo	5	0	34	0	2
Gentoo	10	0	34	0	2
Gentoo	20	0	31	2	3
KDE	1	0	125	2	12
KDE	3	0	117	8	14
KDE	5	0	118	5	16
KDE	10	0	116	7	16
KDE	20	2	123	4	10
LibreOffice	1	1	13	0	1
LibreOffice	3	0	13	1	1
LibreOffice	5	0	14	0	1
LibreOffice	10	0	14	0	1
LibreOffice	20	0	13	0	2

Table 4.10: Best Algorithms for Schema-2 for each dataset and for each Top-N value

dataset	TopN	N-BAYES	POPULAR	RANDOM	UBCF
Eclipse	1	0	59	1	2
Eclipse	3	4	54	2	2
Eclipse	5	4	49	3	6
Eclipse	10	5	42	5	10
Eclipse	20	4	40	6	12
ENT	1	1	28	1	2
ENT	3	2	23	4	3
ENT	5	1	25	5	1
ENT	10	0	28	2	2
ENT	20	0	32	0	0
Gentoo	1	1	22	0	3
Gentoo	3	0	22	1	3
Gentoo	5	0	20	0	6
Gentoo	10	0	20	1	5
Gentoo	20	0	21	0	5
KDE	1	3	80	3	11
KDE	3	1	76	9	11
KDE	5	2	78	6	11
KDE	10	3	78	4	12
KDE	20	1	82	4	10
LibreOffice	1	0	5	0	1
LibreOffice	3	0	6	0	0
LibreOffice	5	0	5	0	1
LibreOffice	10	0	6	0	0
LibreOffice	20	0	5	1	0

Table 4.11: Best Algorithms for Schema-3 for each dataset and for each Top-N value

dataset	TopN	N-BAYES	POPULAR	RANDOM	UBCF
Eclipse	1	2	69	0	6
Eclipse	3	2	59	8	8
Eclipse	5	4	51	9	13
Eclipse	10	8	43	9	17
Eclipse	20	5	56	4	12
ENT	1	2	34	3	2
ENT	3	1	30	6	4
ENT	5	1	31	5	4
ENT	10	0	35	5	1
ENT	20	0	40	1	0
Gentoo	1	2	29	0	5
Gentoo	3	1	30	2	3
Gentoo	5	1	30	0	5
Gentoo	10	0	31	2	3
Gentoo	20	0	31	2	3
KDE	1	6	98	4	17
KDE	3	2	96	9	18
KDE	5	1	90	12	22
KDE	10	1	91	14	19
KDE	20	1	107	10	7
LibreOffice	1	0	9	0	0
LibreOffice	3	0	8	1	0
LibreOffice	5	0	8	1	0
LibreOffice	10	0	8	1	0
LibreOffice	20	0	8	1	0

Table 4.12: Best Algorithms for Schema-4 for each dataset and for each Top-N value

dataset	TopN	N-BAYES	POPULAR	RANDOM	UBCF
Eclipse	1	0	155	1	5
Eclipse	3	1	142	2	16
Eclipse	5	5	128	10	18
Eclipse	10	5	117	12	27
Eclipse	20	6	118	12	25
ENT	1	2	47	3	10
ENT	3	1	47	4	10
ENT	5	4	45	3	10
ENT	10	3	45	6	8
ENT	20	1	50	5	6
Gentoo	1	0	48	0	3
Gentoo	3	0	48	1	2
Gentoo	5	0	49	0	2
Gentoo	10	0	49	0	2
Gentoo	20	0	48	0	3
KDE	1	0	165	8	20
KDE	3	3	163	6	21
KDE	5	4	154	15	20
KDE	10	4	156	13	20
KDE	20	3	160	11	19
LibreOffice	1	1	15	0	1
LibreOffice	3	0	17	0	0
LibreOffice	5	0	15	1	1
LibreOffice	10	0	15	2	0
LibreOffice	20	0	17	0	0

Chapter 5

Conclusions and Future Work

In this study, we present defect rediscovery datasets collected from several groups of open source projects (Eclipse, Gentoo, LibreOffice, and KDE) as well as an enterprise software project (ENT), aimed at capturing information associated with duplicate / rediscovered defects. We describe the schema of the datasets, extraction and transformation process, and present analysis of the datasets. Then, we build a predictive model leveraging common recommender system algorithms in order to predict defect rediscoveries based on the extracted data.

We apply recommender systems to reduce the number of defect rediscoveries by the users in large scale open source and enterprise software projects. We identified that for large scale software projects, the defect rediscovery data is highly sparse, that is only a small number of defects out of thousands of defects are rediscovered frequently by the users of the software product. In order to provide accurate recommendations about defect rediscoveries to the users, the sparsity must be reduced. We introduce a product-component based data partitioning technique to reduce the sparsity in defect rediscovery datasets.

We investigate several temporal splitting schema to build our predictive models. The performance of the temporal splitting varies. Generally, in the case of open source software projects, training the model on one year of rediscovery data and testing on one or more years of data yields the best results. In the case of the enterprise software project, accumulating rediscovery data from previous years, typically result in higher accuracy while predicting future defect rediscoveries.

We describe the advantages of our data partitioning techniques. However, they have limitations. Namely, the product-components with limited defect rediscovery data and the time-windows with a small number of defect rediscoveries may result in inability to recommend defects for a given client.

Our primary research question, RQ1 was: *How can we proactively predict defects that a client will rediscover in the future?* To address this question we applied recommender system techniques to predict defects rediscovered by a given client. The recommender system achieved a maximum TPR between 60% and 92% with a maximum FPR between 20% and 60% for each project, while predicting Top-20 rediscovered defects. To achieve this result, we applied four temporal splitting techniques in combination with partitioning the data by product-components.

Our secondary research questions, RQ2 and RQ3 were, *How prevalent is the defect rediscovery phenomenon in commercial and open source software projects?* and *What are the factors that influence the accuracy of the predictive models?*. To address RQ2, we analyzed the inter-relations between rediscovered defects and connected all rediscoveries related to the same discovery. We also presented a statistical analysis on the defect rediscovery data of five different software projects. We found the rediscovery phenomenon to be widespread among the projects.

To address RQ3, we applied two classifiers to identify the important factors leading to models' failure. We found that lack of information about users and rediscoveries in the test data, as well as the lack of overlap of rediscoveries in train and test data are the main reasons that cause the predictive model to fail.

We believe that our approach for predicting defect rediscoveries is of interest to practitioners, as they may use this approach to proactively identify a subset of defects that a particular client may encounter in the future. It is also of interest to researchers, as it may help in understanding phenomenon of rediscovery data, leading to creation of new models for predicting rediscoveries.

Moreover, we believe that the defect rediscovery datasets that we extracted, will aid researchers and practitioners in gathering insight into usage of duplicate reports in various areas of software engineering.

5.1 Future Work

Going forward, we would like to extend our work to include data from other software projects as well as experiment with other recommender algorithms and data partitioning techniques.

We consider this study as a starting point in the application of recommender systems to reduce defect rediscovery. Generally, recommender systems are widely used in other domains and real world use cases. We plan to build more sophisticated recommender systems (including context-aware and content-based ones) leveraging the insights we got from this study. For example, defect reports not labelled as duplicates but with high textual similarity may get rediscovered by similar clients in the future.

The defect rediscovery phenomena should be investigated more. As we build the graph of rediscoveries for each connected set of duplicate defects, we notice how duplicate/rediscovered defects are interrelated with each other across different products and components. Unraveling the unique characteristics of the duplicate/rediscovered defects would be another line of our future research.

Finally, we look forward to leveraging our proposed approach to above-mentioned and relevant research problems including defect report prioritization and defect report optimization.

Appendices

Appendix A

Reducing the Sparsity using Clustering

As we discuss in Secition 3.6.1 that there exists several techniques to reduce the sparsity of the rating matrix, we explore some of these techniques in combination with partioning by product-components. However, we exclude them from our analysis because these techniques did not yield superior results than the partitioning by product-components. Moreover, we use the geo-location based feature for this purpose which is available only in the ENT dataset. Therefore, we share our findings on the clustering based sparsity reduction techniques on the ENT dataset here in the appendix.

A.1 Clustering

As mentioned above, clustering based techniques reduce the sparsity of the dataset by forming groups of defect reports using unsupervised learning approaches. In order to group defect reports into clusters, we have investigated the location attribute in the ENT dataset that is associated with each defect report. The location attribute was chosen because the behaviours of the users may vary from country to country [79].

For example, people from some countries may be not very keen on reporting defects or they may not use the software extensively, resulting in rediscovering less number of defects. On the contrary, users from some other locations may always report defect as soon as they find or they may use the software more extensively, resulting in rediscovering

higher number of defects in the software.

Our location data set contains country-level location information with 72 unique countries (geo-locations). To understand the properties of each geo-location, at first we have extracted a set of derived attributes for each country listed below:

1. Average number of defects per customer,
2. Average number of rediscoveries per defect,
3. Rediscovery window,
4. Average defect arrival rate.

Attributes 1 and 2 were found useful in operational profiling of customers [59]. A high *average number of defects per customer* suggests that users from such locations use the software extensively and a high *average number of rediscoveries per defect* in a location suggests that large number of users are using the product in a similar manner. To compute Attribute 3, *rediscovery window*, for each country we calculated the number of days between the first and last defect, as per [22], indirectly providing information on the length of usage of this product in a given country. Attribute 4, the *average defect arrival rate*, is computed as the number of defects reported per day for a given country. This attribute is linked to product quality [41], hence its inclusion in the set.

We use these four attributes to perform unsupervised learning using two techniques: agglomerative hierarchical clustering (AHC) and self-organising map (SOM), discussed in Sections A.1.1 and A.1.2, respectively.

A.1.1 Agglomerative Hierarchical Clustering

The AHC or bottom-up hierarchical clustering technique groups geo-locations by merging pairs of locations first (based on a measure of distance) and then greedily pairing the resulting clusters pair-wise, as we move up the hierarchy [56]. The resulting hierarchy of clusters is called dendrogram. We used Euclidean Distance to compute distance between each pair of objects; in order to determine distance between clusters we used average linkage criterion. Built-in R function `hclust` [64] was used to obtain the dendrogram. We used Gap Statistic method [78] to identify optimal number of clusters, implemented in `clustGap` R function from the `cluster` package [55]. In our case optimal number of

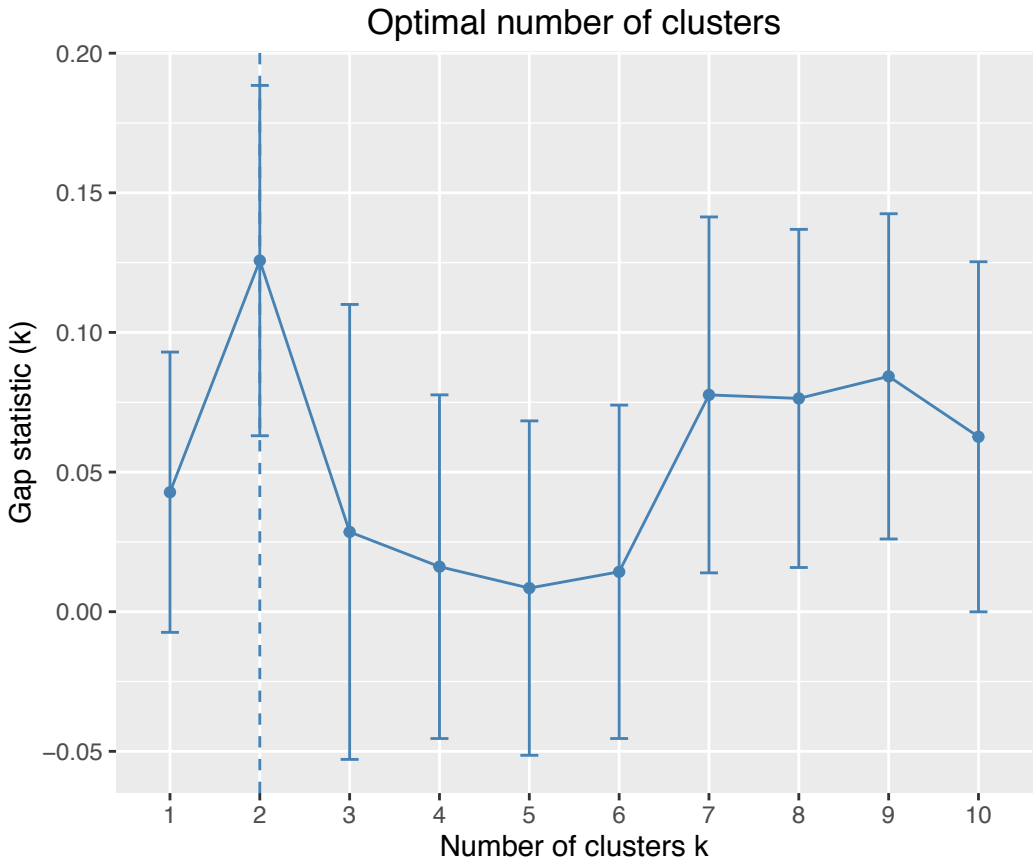

Figure A.1: Identification of optimal number of clusters for AHC using gap statistic.

clusters is two. Hence, we cut the dendrogram (Figure A.2) at a height that splits the tree into two clusters, which we will further discuss in Section A.1.3.

In order to identify optimum number of clusters, we used the Gap Statistic method [78]. The gap statistic measures the total intra-cluster differences for different values of k with corresponding expected values under reference distribution of the data. For several values of k (the number of clusters), the intra-cluster difference or variation between the actual data and the reference data is computed. The value of k that returns the maximum variation or gap statistic is the optimum number of clusters. Based on the `clustGap` R function from the `cluster` package[55]

We used the `clustGap` R function from the `cluster` package[55] to identify optimal number of clusters using the gap statistic automatically. Figure A.1 illustrates that in our data, for $k = 2$ we get the maximum gap statistic.

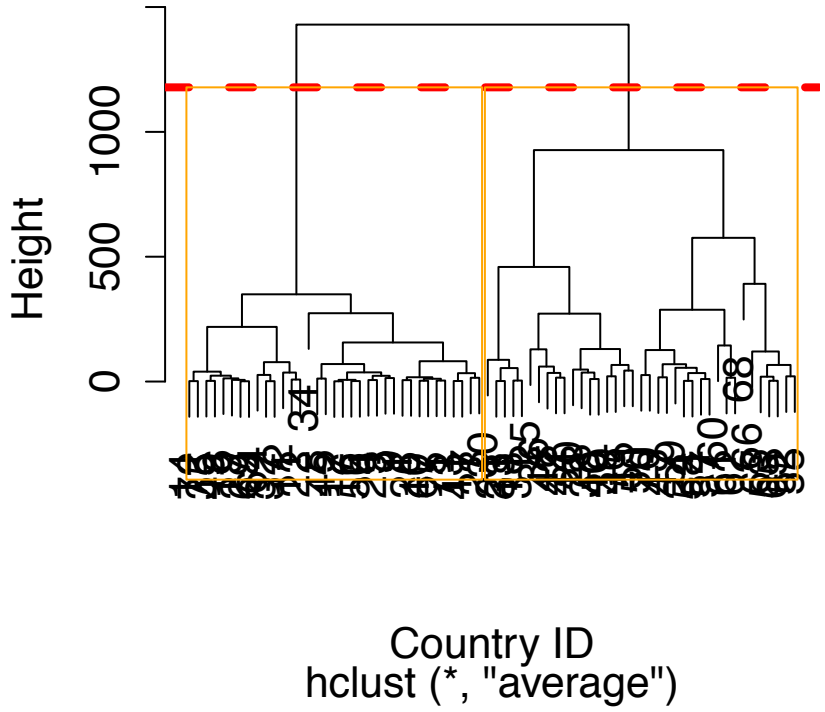

Figure A.2: Cluster Dendrogram and Tree Pruning for AHC.illustrates the dendrogram obtained by AHC. The red dotted line shows the height at which the tree pruning was done. The two orange rectangular boxes represent the two clusters, obtained after tree pruning.

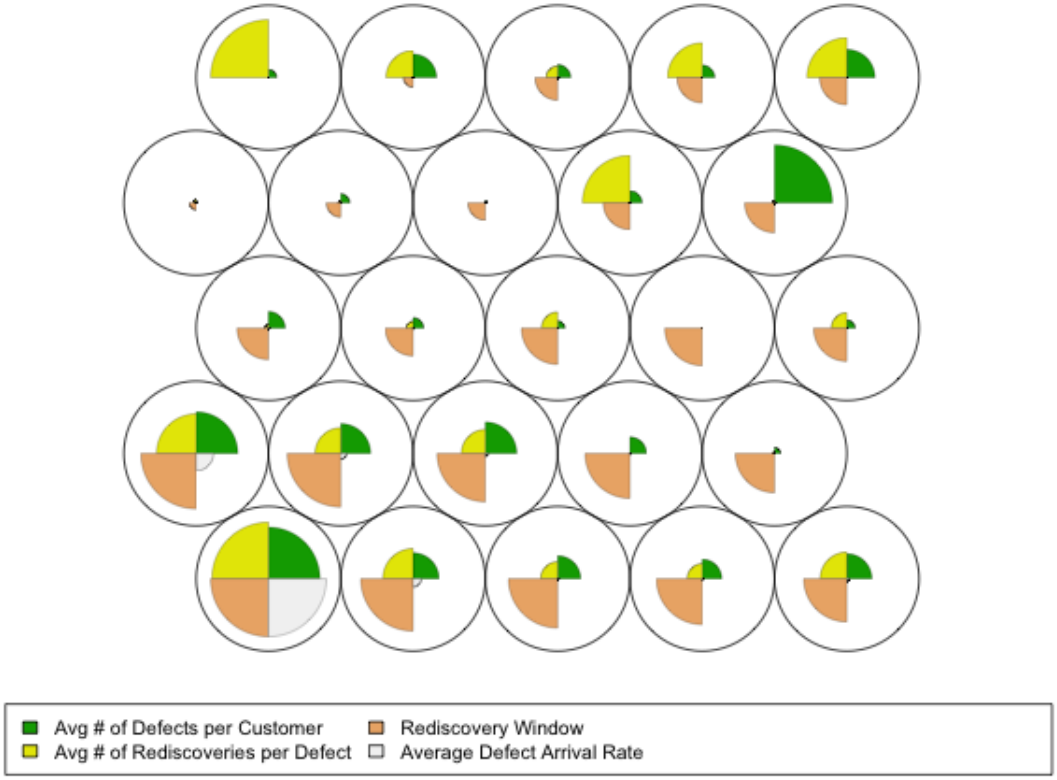

Figure A.3: Illustrates the self-organising map in 5×5 grid. Each circle represents one of the 25 neurons of the grid. The diagram is called a Fan-Diagram. Each fan represents the magnitude of each attribute in the weight vector.

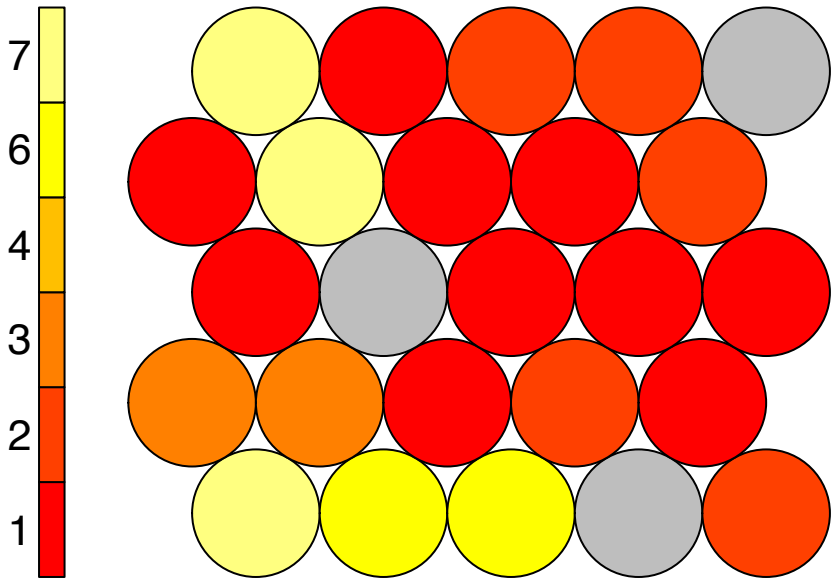

Figure A.4: Counts plot. Shows the number countries associated with each neuron on the self-organising map in 5×5 grid. Grey colour represents zero countries.

A.1.2 Self-Organising Map (SOM)

The Kohonen self-organising map is a special type of Artificial Neural Network [43] . It can visualise high dimensional data in a low-dimensional space; typically the space is reduced to a two-dimensional map. By inspecting the map, we can understand the underlying characteristics of the geo-locations and identify similar ones.

The building block of SOM is a unit or a neuron. The number of neurons to be used must be decided before training. There are weights associated with each of the neurons. The number of weights are equal to the number of attributes in the input space. The neurons are typically represented in a rectangular grid system. At first, random weights are assigned to the neurons in SOM. During the training process, in each iteration all the data points are presented to the SOM. The similarity in weights between a data point and all the neurons is computed and the most similar neuron becomes the winning neuron. The weights of the winning neuron is adjusted in each iteration as the neuron becomes more and more similar to the input data. The weights of the other neurons in the neighbourhood of the winning neuron are also adjusted. This process is repeated for a fixed number of iterations.

The output of the SOM are clustered data points (i.e., every data point will be associated with a certain neuron in the grid) [21]. By design, the data points in adjacent neurons of the grid are similar to each other. Visual inspection of the resulting grid (map) allows to identify the clusters.

We trained SOM implemented in `kohonen` [82] R package on our 4-dimensional geo-location data. To construct the SOM, as discussed above, we need to choose the number of neurons and the number of iterations. To choose the number of the neurons, we used the following heuristic formula: $\beta \approx 5 \times \gamma^{0.54321}$ [6], where β is the number of units in the map and γ is the number of observations in the data set. We had 72 observations and according to this heuristic our map size should be 25. The neurons were configured in a 5×5 grid. The maximum number of iterations was set to 100; the algorithm converged after $\approx$ 45 iterations. In Figure A.5, we can see that in the case of our dataset the algorithm converges after $\approx$ 45 iterations. The resulting visual inspection of the data suggested to aggregate data from four neurons into one cluster. The analysis of the data associated with this cluster is given in Section A.1.3.

These clusters can be identified by visual inspection of the map.

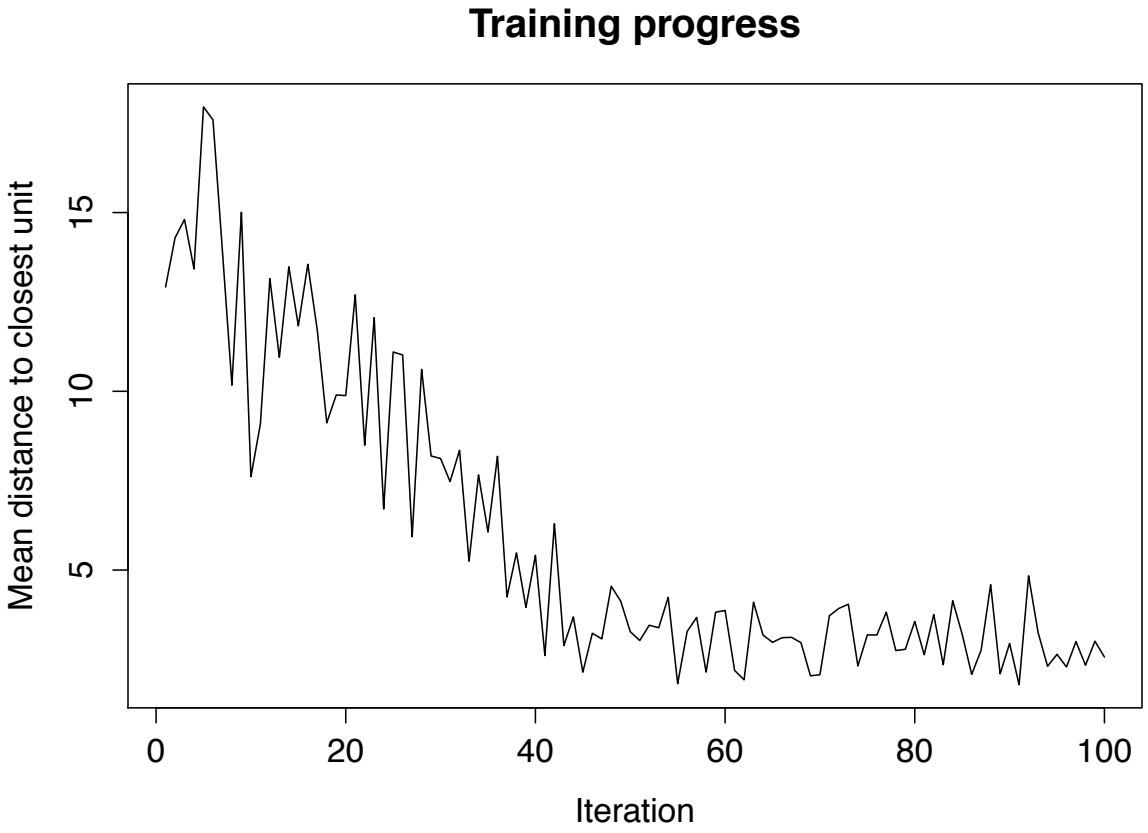

Figure A.5: SOM Training.

The learning rate decreases during the training and the map converges, which means that the neighbourhood size around the best matching unit decreases with each iteration until the neighbourhood is only the best matching unit [43].

To choose the size of the grid we have used the following heuristic formula, $munits \approx 5 \times dlen^{0.54321}$ [6]. Where $munits$ is the number of units in the map and $dlen$ is the number of observations in the data set. We had 72 observations and according to this heuristic our map size should be 25.

A.1.3 Results

We partition the dataset, reducing sparsity of the rating matrix, using clustering techniques (shown in Sections A.1.1 and A.1.2) and apply RS algorithms to the resulting clusters in Sections A.1.3 and A.1.3 . Lastly, we further reduce the sparsity by partitioning the data per product-component and applying the RS algorithms to the resulting subsets of data.

Clustering – AHC

We first cluster the complete dataset (grouped by country as discussed in Section A.1) using AHC (details of the algorithm are provided in Section A.1.1). Figure A.2 shows the resulting dendrogram generated by the AHC. The dendrogram was split into two clusters (using the gap statistic approach described in Section A.1.1). The resulting clusters are marked using orange boxes in Figure A.2.

Each cluster has $\approx 50\%$ of the countries associated with it. However, even though the right cluster on Figure A.2 has $\approx 50\%$ of the countries associated with, $\approx 99\%$ of the defect reports originated from these countries. Therefore, we use the defect reports from this cluster for our experiments.

Clustering – SOM

In this section we cluster the complete dataset using SOM (details of the algorithm are shown in Section A.1.2). Upon training on the complete dataset, SOM returned 5×5 grid of neurons with each of the 72 countries associated with a particular neuron. Figure A.4

show the number of countries associated with each neutron. The number of countries per neuron ranges from 0 to 7.

Each neuron in the SOM has a weight vector (one weight value per each of the four attributes discussed in Section A.1). Figure A.3 visualises these weights using fan-diagrams. Each fan represents the magnitude of each attribute in the weight vector. For example, the bottom-left neuron in Figure A.3 contains countries with the highest *average number of defects per user*, *average number of rediscoveries per defect*, *rediscovery window*, and *average defect arrival rate*.

Inspection of the fan diagram helps to uncover some patterns. Note that by design of SOM (see [43]), countries in adjacent neutrons of the grid possess some similarities. In particular, four neurons at the bottom left corner of the grid (the bottom left neuron and three other neurons surrounding it) possess interesting characteristics: the magnitude of the weights for all the attributes is relatively higher in this region of the grid, in comparison with other regions.

Although, 19 countries out of the total 72 countries are mapped to the four neurons mentioned above, $\approx 94\%$ of the defect reports came from these 19 countries. The other 21 out of 25 neurons contain 53 out of 72 countries, with $\approx 6\%$ of the defect reports coming from these. Therefore, we apply the DRS to the subset of data reported from these 19 countries. The percentage of non-zero elements in the rating matrix for this subset of data is 0.08%, which is higher than 0.071% of the full matrix. Thus, as in the case of AHC described in Section A.1.3, we expect that performance of DRS should increase (in comparison with the performance of the DRS on the complete dataset).

Per product-component

We partitioned the dataset by product-component in combination with the two clustering based approaches mentioned above. We select 28 out of 185 components, covering $\approx 80\%$ of defect reports.

Essentially, we took the complete dataset, filtered defects associated with 28 components, and split the filtered dataset by component. We then applied four DRS algorithms to each of the 28 subsets of data and gathered the summary statistics. We then repeated this process twice: first, replacing the complete dataset with 'cluster 1' generated by AHC, and second, replacing the complete dataset with the cluster of data identified

by SOM. In this section we will refer to the datasets originated from complete dataset as *Complete-based*, from AHC 'cluster 1' as *AHC-based*, and from the SOM cluster as *SOM-based*.

To assess usefulness of clustering techniques, we "collapse" distribution of TPR and FPR per component, by computing means and 95% asymmetric confidence interval[1] (CI) of FPR and TPR data for each dataset and N. Results are given in Figure A.6. AHC dataset has consistently[2] higher mean TPR values while maintaining similar or lower mean FPR rates. Analysis of confidence intervals suggests that AHC-based approach outperforms the other two for $N \leq 5$; SOM-based approach prevails for $N > 5$.

For example, in the case when $N = 3$, TPR's 95% CI for AHC-based approach ranges between 0.05 and 0.48, with the mean of 0.20; FPR's – between 0.01 and 0.08, with the mean of 0.4. In the case when $N = 20$, TPR's 95% CI for SOM-based approach ranges between 0.26 and 0.88, with the mean of 0.53; FPR – between 0.09 and 0.54, with the mean of 0.25.

[1] Computed based on 0.025 and 0.975 empirical quantiles.

[2] The only exception is $N = 20$, where SOM TPR value is higher than AHC TPR values by 0.01; however, this increase comes at expense of FPR, which is also higher for SOM by 0.01.

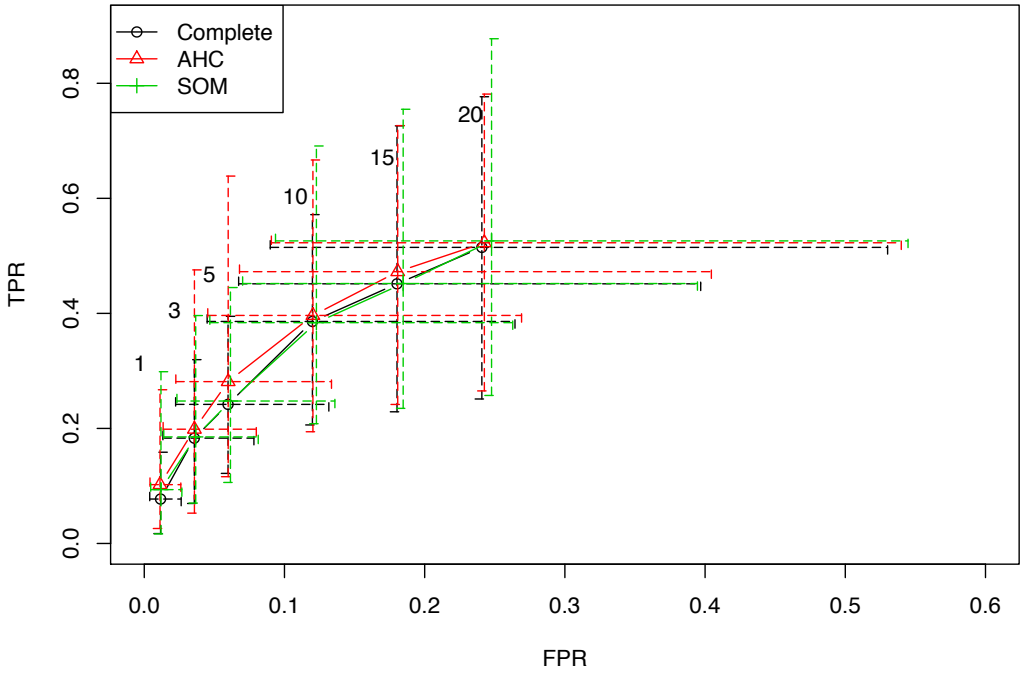

Figure A.6: Performance of the best algorithms for three datasets. Each data point represents mean FPR and TPR values for all 28 components for a given Top-N. Values of N are shown above the lines. Dotted lines represent 95% confidence intervals of FPR and TPR for each of the data points.

Appendix B

Data Extractions Scripts

A prototype tool to extract rediscovery data from Bugzilla-based defect report tracking systems.

B.1 Web Scraper

```
import urllib2
import pandas as pd
import sys
import time
import os
from WebScraperUtil import command_line_options
from bs4 import BeautifulSoup
from retrying import retry


class ExtractionParams:
    '''
    This class defines default extraction parameters.
    Use command_line_options to override the constructor.
    '''

    def __init__(self):
        self.attrs = ['id', 'product', 'component', 'reporter', 'bug_status
            ','resolution','priority','bug_severity','version','short_desc'
```

```
            ,'opendate','dup_id','dup_list']
19          self.bug_lookup_url_pre = 'https://bugs.eclipse.org/bugs/buglist.
                cgi?bug_id='
20          self.bug_lookup_url_post = '&columnlist=product%2Ccomponent%2
                Creporter%2Copendate%2Cpriority%2Cbug_id%2Cbug_severity%2
                Cversion%2Cresolution%2Cbug_status%2Cshort_desc&query_based_on
                =&query_format=advanced'
21          self.bug_profile_url = 'https://bugs.eclipse.org/bugs/show_bug.cgi?
                id='
22          self.out_filename = 'output/eclipse/'+str(time.strftime('%c')) + '
                _eclipse_defect_rediscovery.csv'
23          self.file_input_dir = 'input/eclipse'
24
25
26  def retry_if_exception(exception):
27      '''Retry http request if exception occurs.'''
28      print 'retrying: ' + str(exception)
29      return isinstance(exception, Exception)
30
31
32  @retry(retry_on_exception=retry_if_exception, wait_random_min=5000,
        wait_random_max=20000)
33  def start_http_request(url):
34      '''Send http request.'''
35      response = urllib2.urlopen(url)
36      return response
37
38
39  def get_bug_metadata(bug_id, list_of_attributes, params):
40      '''
41      Extracts bug metadata from Bugzilla.
42      :param bug_id: an int bug_id
43      :param list_of_attributes: a list of attributes from the available
            attributes
44      :param params: an object of the ExtractionParams class
45      :return: a dictionary containing attribute:value as key:value pairs
46      '''
47      bug_list_html = start_http_request(params.bug_lookup_url_pre + str(
            bug_id) + params.bug_lookup_url_post)
```

```
48      bug_list_soup = BeautifulSoup(bug_list_html, 'html.parser')
49
50      attributes = {}
51      attributes[list_of_attributes[0]] = bug_id
52
53      for i in range(1, len(list_of_attributes)):
54
55          if list_of_attributes[i] == 'dup_id':
56              attributes[list_of_attributes[i]] = bug_id
57
58          elif list_of_attributes[i] == 'dup_list':
59              attributes[list_of_attributes[i]] = None
60
61          elif list_of_attributes[i] == 'opendate':
62
63              try:
64                  attributes[list_of_attributes[i]] = str(bug_list_soup.find(
                        'td', {'class': 'bz_opendate_column'}).contents[0]).
                        rstrip()
65
66              except Exception, e:
67                  attributes[list_of_attributes[i]] = 'NA'
68
69          elif list_of_attributes[i] == 'short_desc':
70
71              try:
72                  attributes[list_of_attributes[i]] = str(
73                      bug_list_soup.find('td', {'class': '
                            bz_short_desc_column'}).find('a').contents[0]).
                            rstrip()
74
75              except Exception, e:
76                  attributes[list_of_attributes[i]] = 'NA'
77          else:
78              try:
79                  attributes[list_of_attributes[i]] = str(bug_list_soup.find(
                        'td', {'class': 'bz_'+list_of_attributes[i]+'_column'})
                        .find('span').contents[0]).rstrip()
80              except Exception, e:
```

```
81                  attributes[list_of_attributes[i]] = 'NA'
82
83      return attributes
84
85
86  def get_dup_ids(string_having_dups, dup_list):
87      '''
88      From the string having the dups form the Bugzilla web page, extract
            each dup as int, and updates the dup_list of the bug.
89      :param string_having_dups: a string containing all dups for a given bug
90      :param dup_list: a list int dup ids
91      :return: updated dup_list
92      '''
93      for dup in string_having_dups.findAll('a'):
94
95          val = (dup.contents[0])
96
97          try:
98              int(val)
99
100         except ValueError:
101             val = str(dup['href']).partition('=')[2]
102
103         dup_list.append(val)
104     return dup_list
105
106
107 def write_row(bug_metadata, tmp_lists, list_of_attributes):
108     '''
109     Writes a single entry/row in intermediate temporary lists for a given
            bug.
110     :param bug_metadata: a dictionary containing all extracteded attributes
            to be written
111     :param tmp_lists: a temporary list of lists
112     :param list_of_attributes: a list of attributes used in extraction
113     :return: updated list of lists (essentially added one item to each item
            in each list)
114     '''
115     for _ in xrange(len(list_of_attributes)):
```

```
116             tmp_lists[_].append(bug_metadata[list_of_attributes[_]])
117     return tmp_lists
118
119
120 def write_csv(list_of_attributes, tmp_lists, params):
121     '''
122     Writes the attributes to an actual output file.
123     :param list_of_attributes: a list of attributes used in extraction
124     :param tmp_lists:  a temporary list of lists
125     :param params: an object of the ExtractionParams class
126     :return: none
127     '''
128     bug_df = pd.DataFrame(columns=list_of_attributes)
129
130     for cols in xrange(len(list_of_attributes)):
131         bug_df[list_of_attributes[cols]] = tmp_lists[cols]
132
133     file_name_csv = params.out_filename
134     bug_df.to_csv(file_name_csv, encoding='utf-8')
135
136
137
138 def create(n, constructor = list):
139     '''Creates temporary list of lists.'''
140     for _ in xrange(n):
141         yield constructor()
142
143
144 def read_input_file(file_input_dir):
145     '''Read bug ids from input file. . See Github wiki for more
        instructions.'''
146     for file in os.listdir(file_input_dir):
147         if file.endswith('.htm'):
148             file_name = file_input_dir + '/' + file
149             return open(file_name, 'r')
150
151
152 def extract_data(params):
```

```
153     '''Intiates data extraction with extraction params when an input bug id
            file is supplied.'''
154     list_of_attributes = params.attrs
155     tmp_lists = list(create(len(list_of_attributes)))
156 
157     html_file = read_input_file(params.file_input_dir)
158     source_code = html_file.read()
159 
160     soup = BeautifulSoup(source_code, 'html.parser')
161     dupe_table = soup.find('table', {'id': 'duplicates_table'})
162 
163     for row in dupe_table.findAll('tr'):
164 
165         for col in row.findAll('td', {'class': 'id'}):
166             bug_id = int(col.find('a').contents[0])
167 
168             print '*************************\nExtracting Dups of:  ' + str
                (bug_id) + '\n*************************'
169             bug_metadata = get_bug_metadata(bug_id, list_of_attributes,
                params)
170 
171             bug_profile_html = start_http_request(params.bug_profile_url +
                str(bug_id))
172 
173             bug_profile_soup = BeautifulSoup(bug_profile_html, 'html.parser
                ')
174             dup_list = []
175 
176             if (bug_profile_soup.find('span', {'id': 'duplicates'})) is not
                 None:
177                 dup_list = get_dup_ids((bug_profile_soup.find('span', {'id'
                    : 'duplicates'})), dup_list=[])
178 
179             bug_metadata['dup_list'] = dup_list
180             tmp_lists = write_row(bug_metadata, tmp_lists,
                list_of_attributes)
181 
182             for i in dup_list:
183                 print 'Extracting Dup #:  ' + str(dup_list.index(i)+1)
```

```
184                 bug_metadata = get_bug_metadata(i, list_of_attributes,
                        params)
185                 bug_metadata['dup_id'] = bug_id
186                 bug_metadata['dup_list'] = None
187                 tmp_lists = write_row(bug_metadata, tmp_lists,
                        list_of_attributes)
188
189     write_csv(list_of_attributes, tmp_lists, params)
190
191
192 def create_list_of_bugs_to_be_extracted(starting_id, ending_id, params):
193     '''Defines the ranfe of bug ids to be extracted.'''
194     list_of_bugs = range(starting_id, ending_id+1)
195     return list_of_bugs
196
197
198 def extract_data_by_brute_force(params, args):
199     '''Intiates data extraction with extraction params for chronological
            extraction.'''
200     print 'extracting by brute-force'
201
202     starting_id = int(args[2])
203     ending_id = int(args[3])
204
205     bug_list = create_list_of_bugs_to_be_extracted(starting_id, ending_id,
            params)
206     list_of_attributes = params.attrs
207     tmp_lists = list(create(len(list_of_attributes)))
208
209     for bug_id in bug_list:
210         print 'Bug Id# ' + str(bug_id)
211
212         bug_profile_html = start_http_request(params.bug_profile_url + str(
                bug_id))
213         bug_profile_soup = BeautifulSoup(bug_profile_html, 'html.parser')
214         dup_list = []
215
216         if (bug_profile_soup.find('span', {'id': 'duplicates'})) is not
                None:
```

```
217             dup_list = get_dup_ids((bug_profile_soup.find('span', {'id': '
                    duplicates'})), dup_list=[])
218
219         if len(dup_list) > 0:
220
221             bug_metadata = get_bug_metadata(bug_id, list_of_attributes,
                    params)
222             bug_metadata['dup_list'] = dup_list
223             tmp_lists = write_row(bug_metadata, tmp_lists,
                    list_of_attributes)
224
225         else:
226             bug_metadata = get_bug_metadata(bug_id, list_of_attributes,
                    params)
227             bug_metadata['dup_id'] = None
228             tmp_lists = write_row(bug_metadata, tmp_lists,
                    list_of_attributes)
229
230     write_csv(list_of_attributes, tmp_lists, params)
231
232
233 def get_latest_bug_id(params):
234     '''Command line option: Checks the latest reported bug id in Bugzilla.
            '''
235     url = params.bug_lookup_url_pre.partition('bug_id=')[0] + 'chfield=%5
            BBug%20creation%5D&chfieldfrom=7d'
236     bug_list_html = start_http_request(url)
237
238     print url
239
240     try:
241         bug_list_soup = BeautifulSoup(bug_list_html, 'html.parser')
242         bug_id_list = bug_list_soup.findAll('td', {'class': 'first-child
                bz_id_column'})
243         latest_bugs = []
244
245         for id in bug_id_list:
246             latest_bugs.append(id.find('a').contents[0])
247
```

```
248         print(max(latest_bugs))
249
250     except Exception, e:
251         print e
252
253     return
254
255
256 def get_missing_count(params):
257     '''Command line option: Checks if bug id is inaccessible.'''
258     list_of_attributes = ['id', 'missing_status']
259     tmp_lists = list(create(len(list_of_attributes)))
260
261     file_name = ''
262
263     for file in os.listdir(params.file_input_dir):
264         if file.endswith('.csv'):
265             file_name = params.file_input_dir + '/' + file
266
267     print file_name
268
269     df = pd.read_csv(file_name)
270
271     for i in df['id']:
272
273         bug_profile_html = start_http_request(params.bug_profile_url + str(
                i))
274         bug_profile_soup = BeautifulSoup(bug_profile_html, 'html.parser')
275         missing_info_dict = {'id': str(i),
276                              'missing_status': bug_profile_soup.find('span'
                                 , {'id': 'title'}).contents[0]}
277         print missing_info_dict
278         tmp_lists = write_row(missing_info_dict, tmp_lists,
                list_of_attributes)
279
280     write_csv(list_of_attributes, tmp_lists, params)
281
282
283 if __name__ == '__main__':
```

```
284
285     params = ExtractionParams()
286     params = command_line_options(params, sys.argv)
287     if len(sys.argv) == 4:
288
289         a = int(sys.argv[2])
290         b = int(sys.argv[3])
291         r = (b - a) / 40
292
293         while a <= b:
294
295             sys.argv[2] = str(a)
296
297             if a + r < b:
298                 sys.argv[3] = str(a + r)
299
300             else:
301                 sys.argv[3] = str(b)
302                 params.out_filename = 'output/' + str(sys.argv[1]) + '/' +
                        str(sys.argv[1]) + '_' + '_' + str(
303                         sys.argv[2]) \
304                                         + '_' + str(sys.argv[3]) + '__' + str
                                            (time.strftime('%c')) + '.csv'
305                 extract_data_by_brute_force(params, sys.argv)
306                 break
307
308             params.out_filename = 'output/' + str(sys.argv[1]) + '/' + str(
                    sys.argv[1]) + '_' + '_' + str(sys.argv[2]) \
309                                     + '_' + str(sys.argv[3]) + '__' + str(
                                        time.strftime('%c')) + '.csv'
310             extract_data_by_brute_force(params, sys.argv)
311             a += r + 1
312
313     else:
314
315         if sys.argv[2] == 'last':
316             get_latest_bug_id(params)
317
318         elif sys.argv[2] == 'missing':
```

```
319             get_missing_count(params)
320
321         else:
322             extract_data(params)
```

B.2 Web Scraper Util

```
1  import time
2
3
4  def command_line_options(obj, args):
5      '''
6      Overrides the constructor with command line options.
7      :param obj: an instance of ExtractionParams class
8      :param args: command line args
9      :return: updated instance
10     '''
11
12     if len(args) >= 2:
13         print 'Extracting Bugs for: ' + str(args[1])
14         if args[1].lower() == 'mozilla':
15             obj.attrs = ['id', 'product', 'component', 'reporter', '
                   bug_status', 'resolution', 'priority',
16                          'bug_severity', 'version', 'short_desc', 'opendate
                               ', 'dup_id', 'dup_list']
17             obj.bug_lookup_url_pre = "https://bugzilla.mozilla.org/buglist.
                   cgi?bug_id="
18
19             obj.bug_lookup_url_post = "&query_format=advanced&
                   query_based_on=&columnlist=product%2Ccomponent%2Cbug_status
                   %2Cresolution%2Cshort_desc%2Cbug_id%2Copendate%2Cpriority%2
                   Creporter%2Cbug_severity%2Cversion"
20
21             obj.bug_profile_url = "https://bugzilla.mozilla.org/show_bug.
                   cgi?id="
22             obj.out_filename = 'output/mozilla/' + str(time.strftime("%c"))
                    + '_mozilla_defect_rediscovery.csv'
23             obj.file_input_dir = 'input/mozilla'
24
```

```
25          elif args[1].lower() == 'gnome':
26              obj.attrs = ['id', 'product', 'component', 'reporter', '
                    bug_status', 'resolution', 'priority',
27                           'bug_severity', 'version', 'short_desc', 'opendate
                                ', 'dup_id', 'dup_list']
28              obj.bug_lookup_url_pre = "https://bugzilla.gnome.org/buglist.
                    cgi?bug_id="
29
30              obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                    Cbug_status%2Cresolution%2Cshort_desc%2Cchangeddate%2
                    Cbug_id%2Copendate%2Cpriority%2Creporter%2Cbug_severity%2
                    Cversion&query_based_on=&query_format=advanced"
31
32              obj.bug_profile_url = "https://bugzilla.gnome.org/show_bug.cgi?
                    id="
33              obj.out_filename = 'output/gnome/' + str(time.strftime("%c")) +
                     '_gnome_defect_rediscovery.csv'
34              obj.file_input_dir = 'input/gnome'
35
36          elif args[1].lower() == 'kde':
37              obj.attrs = ['id', 'product', 'component', 'reporter', '
                    bug_status', 'resolution', 'priority',
38                           'bug_severity',
39                           'version', 'short_desc', 'opendate', 'dup_id', '
                                dup_list']
40              obj.bug_lookup_url_pre = "https://bugs.kde.org/buglist.cgi?
                    bug_id="
41
42              obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                    Cbug_status%2Cresolution%2Cshort_desc%2Cbug_id%2Copendate%2
                    Cpriority%2Creporter%2Cbug_severity%2Cversion&query_format=
                    advanced"
43
44              obj.bug_profile_url = "https://bugs.kde.org/show_bug.cgi?id="
45              obj.out_filename = 'output/kde/' + str(time.strftime("%c")) + '
                    _kde_defect_rediscovery.csv'
46              obj.file_input_dir = 'input/kde'
47
48          elif args[1].lower() == 'apache':
```

```
49              obj.attrs = ['id', 'product', 'component', 'reporter', '
                    bug_status', 'resolution', 'priority',
50                          'bug_severity',
51                          'version', 'short_desc', 'opendate', 'dup_id', '
                                dup_list']
52              obj.bug_lookup_url_pre = "https://bz.apache.org/bugzilla/
                    buglist.cgi?bug_id="
53
54              obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                    Cbug_status%2Cresolution%2Cshort_desc%2Cbug_id%2Copendate%2
                    Cpriority%2Creporter%2Cbug_severity%2Cversion&query_format=
                    advancedapache    "
55
56              obj.bug_profile_url = "https://bz.apache.org/bugzilla/show_bug.
                    cgi?id="
57              obj.out_filename = 'output/apache/' + str(time.strftime("%c"))
                    + '_apache_defect_rediscovery.csv'
58              obj.file_input_dir = 'input/apache'
59
60
61          elif args[1].lower() == 'documentfoundation':
62              obj.attrs = ['id', 'product', 'component', 'reporter', '
                    bug_status', 'resolution', 'priority',
63                          'bug_severity',
64                          'version', 'short_desc', 'opendate', 'dup_id', '
                                dup_list']
65              obj.bug_lookup_url_pre = "https://bugs.documentfoundation.org/
                    buglist.cgi?bug_id="
66
67              obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                    Cbug_status%2Cresolution%2Cshort_desc%2Cbug_id%2Copendate%2
                    Cpriority%2Creporter%2Cbug_severity%2Cversion&query_format=
                    advanced"
68
69              obj.bug_profile_url = "https://bugs.documentfoundation.org/
                    show_bug.cgi?id="
70              obj.out_filename = 'output/documentfoundation/' + str(
71                  time.strftime("%c")) + '
                        _documentfoundation_defect_rediscovery.csv'
```

```
72              obj.file_input_dir = 'input/documentfoundation'
73
74          elif args[1].lower() == 'ooo':
75              obj.attrs = ['id', 'product', 'component', 'reporter', '
                    bug_status', 'resolution', 'priority',
76                           'bug_severity',
77                           'version', 'short_desc', 'opendate', 'dup_id', '
                                dup_list']
78              obj.bug_lookup_url_pre = "https://bz.apache.org/ooo/buglist.cgi
                    ?bug_id="
79
80              obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                    Cbug_status%2Cresolution%2Cshort_desc%2Cbug_id%2Copendate%2
                    Cpriority%2Creporter%2Cbug_severity%2Cversion&query_format=
                    advancedapache     "
81
82              obj.bug_profile_url = "https://bz.apache.org/ooo/show_bug.cgi?
                    id="
83              obj.out_filename = 'output/ooo/' + str(time.strftime("%c")) + '
                    _apache_defect_rediscovery.csv'
84              obj.file_input_dir = 'input/ooo'
85
86          elif args[1].lower() == 'kernel':
87              obj.attrs = ['id', 'product', 'component', 'reporter', '
                    bug_status', 'resolution', 'priority',
88                           'bug_severity',
89                           'version', 'short_desc', 'opendate', 'dup_id', '
                                dup_list']
90              obj.bug_lookup_url_pre = "https://bugzilla.kernel.org/buglist.
                    cgi?bug_id="
91
92              obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                    Cbug_status%2Cresolution%2Cshort_desc%2Cchangeddate%2
                    Cbug_id%2Copendate%2Cpriority%2Creporter%2Cbug_severity%2
                    Cversion&query_based_on=&query_format=advanced"
93
94              obj.bug_profile_url = "https://bugzilla.kernel.org/show_bug.cgi
                    ?id="
95              obj.out_filename = 'output/kernel/' + str(time.strftime("%c"))
```

```
            + '_kernel_defect_rediscovery.csv'
96          obj.file_input_dir = 'input/kernel'
97
98      elif args[1].lower() == 'redhat':
99          obj.attrs = ['id', 'product', 'component', 'reporter', '
                bug_status', 'resolution', 'priority',
100                     'bug_severity',
101                     'version', 'short_desc', 'opendate', 'dup_id', '
                            dup_list']
102         obj.bug_lookup_url_pre = "https://bugzilla.redhat.com/buglist.
                cgi?bug_id="
103
104         obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                Cbug_status%2Cresolution%2Cshort_desc%2Cbug_id%2Copendate%2
                Cpriority%2Creporter%2Cbug_severity%2Cversion&
                query_based_on=&query_format=advanced"
105
106         obj.bug_profile_url = "https://bugzilla.redhat.com/show_bug.cgi
                ?id="
107         obj.out_filename = 'output/redhat/' + str(time.strftime("%c"))
                + '_redhat_defect_rediscovery.csv'
108         obj.file_input_dir = 'input/redhat'
109
110     elif args[1].lower() == 'novell':
111         obj.attrs = ['id', 'product', 'component', 'reporter', '
                bug_status', 'resolution', 'priority',
112                     'bug_severity',
113                     'version', 'short_desc', 'opendate', 'dup_id', '
                            dup_list']
114         obj.bug_lookup_url_pre = "https://bugzilla.novell.com/buglist.
                cgi?bug_id="
115
116         obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                Cbug_status%2Cresolution%2Cshort_desc%2Cbug_id%2Cpriority%2
                Creporter%2Cbug_severity%2Cversion%2Copendate&
                query_based_on=&query_format=advanced"
117
118         obj.bug_profile_url = "https://bugzilla.novell.com/show_bug.cgi
                ?id="
```

```
119             obj.out_filename = 'output/novell/' + str(time.strftime("%c"))
                    + '_novell_defect_rediscovery.csv'
120             obj.file_input_dir = 'input/novell'
121
122         elif args[1].lower() == 'gentoo':
123             obj.attrs = ['id', 'product', 'component', 'reporter', '
                    bug_status', 'resolution', 'priority',
124                          'bug_severity',
125                          'version', 'short_desc', 'opendate', 'dup_id', '
                             dup_list']
126             obj.bug_lookup_url_pre = "https://bugs.gentoo.org/buglist.cgi?
                    bug_id="
127
128             obj.bug_lookup_url_post = "&columnlist=product%2Ccomponent%2
                    Cbug_status%2Cresolution%2Cshort_desc%2Cbug_id%2Copendate%2
                    Cpriority%2Creporter%2Cbug_severity%2Cversion&query_format=
                    advanced"
129
130             obj.bug_profile_url = "https://bugs.gentoo.org/show_bug.cgi?id=
                    "
131             obj.out_filename = 'output/gentoo/' + str(time.strftime("%c"))
                    + '_gentoo_defect_rediscovery.csv'
132             obj.file_input_dir = 'input/gentoo'
133
134     return obj
```

Appendix C

Recommender Scripts

A prototype tool implementing the novel approach, core features are listed here[1].

C.1 Naive Bayes Implementation

```
1  ##############################################################################
2  # A parallellized implementaion of Naive Bayes based recommender.
3  # This script must be run to register the algorithm to recommenderlab pkg.
4  #
5  # :dependency: Must override the recommenderlab split_known_unkonwn.
6  #              split_known_unkonwn_origi is provided for this purpose.
7  ##############################################################################
8  require(recommenderlab)
9  require(parallel)
10 require(plyr)
11 require(hash)
12
13 no_cores <- detectCores() - 1
14 EvaluateModel <- function(u=NULL,d=NULL,d.table.one,d.table.zero,u.table.
       one,u.table.zero,p.one,p.zero,type){
15
16   d<- as.character(d)
17   u<- as.character(u)
18
```

[1]Full code base of the tool will be shared via GitHub.

```
19    if(is.null(u.table.one[[u]])==FALSE){
20      l.u.p.one <- u.table.one[[u]]
21      l.u.p.zero <- u.table.zero[[u]]
22    }else{
23      l.u.p.one = l.u.p.zero = 0
24    }
25
26    l.d.p.one <- d.table.one[[d]]
27    l.d.p.zero <- d.table.zero[[d]]
28
29    if((l.u.p.one*l.d.p.one*p.one)>(l.u.p.zero*l.d.p.zero*p.zero)){
30      predicted.class <- (l.u.p.one*l.d.p.one*p.one)
31    }
32    else{
33      if(type=='ratings'){
34        predicted.class <- NA
35      }else{predicted.class <- 0}
36    }
37
38    return(predicted.class)
39  }
40
41  ## A parallel binary Naive Bayes Based Recommender
42  BIN_NB <- function(data, parameter = NULL) {
43    ut.mat = as(data,'matrix')
44
45    total.count = dim(ut.mat)[1]*dim(ut.mat)[2]
46    one.count = sum(ut.mat)
47    zero.count =  total.count - one.count
48
49    u.table.one = as.data.frame(cbind(rownames(ut.mat),as.numeric((rowSums(ut
          .mat)+1)
50      / one.count)))
51    u.table.one[[2]] = as.numeric(as.character(u.table.one[[2]]))
52
53    u.table.zero = as.data.frame(cbind(rownames(ut.mat),as.numeric(((dim(ut.
          mat)[2] - rowSums(ut.mat))+1)
54     / zero.count)))
55    u.table.zero[[2]] = as.numeric(as.character(u.table.zero[[2]]))
```

```
56
57
58  d.table.one = as.data.frame(cbind(colnames(ut.mat),as.numeric((colSums(ut
        .mat)+1) / one.count)))
59  d.table.one[[2]] = as.numeric(as.character(d.table.one[[2]]))
60
61  d.table.zero = as.data.frame(cbind(colnames(ut.mat),as.numeric(((dim(ut.
        mat)[1] - colSums(ut.mat))+1)
62   / zero.count)))
63  d.table.zero[[2]] = as.numeric(as.character(d.table.zero[[2]]))
64
65  u.table.one <- hash( as.character(u.table.one[[1]]), u.table.one[[2]])
66  u.table.zero <- hash( as.character(u.table.zero[[1]]), u.table.zero[[2]])
67
68  d.table.one <- hash( as.character(d.table.one[[1]]), d.table.one[[2]])
69  d.table.zero <- hash( as.character(d.table.zero[[1]]), d.table.zero[[2]])
70
71  p.one <- one.count/total.count
72  p.zero <- zero.count/total.count
73
74  model <- c(list( d.table.one=d.table.one
75                  ,d.table.zero=d.table.zero
76                  ,u.table.one=u.table.one
77                  ,u.table.zero=u.table.zero
78                  ,p.one=p.one
79                  ,p.zero=p.zero
80                  ,data=data
81                  ))
82
83  predict <- function(model, newdata, n=10, data=NULL,type=c('topNList', '
        ratings'), ...) {
84    #print(as(newdata,'matrix'))
85    n.data <- newdata
86    m.test <- as(newdata,'matrix')
87    newdata <- as.data.frame(as.table(m.test))
88
89    names(newdata)[1] = 'usr'
90    names(newdata)[2] = 'dfct'
91    names(newdata)[3] = 'cls'
```

```
92
93      newdata$cls[newdata$cls == TRUE] <- 1
94      newdata$cls[newdata$cls == FALSE] <- 0
95
96      newdata$cls <- mapply(EvaluateModel,newdata$usr
97     ,newdata$dfct, MoreArgs = list(model$d.table.one
98     ,model$d.table.zero
99     ,model$u.table.one
100    ,model$u.table.zero
101    ,model$p.one
102    ,model$p.zero
103    ,type)
104    ,mc.cores = no_cores)
105
106   ratings <- as(newdata,'realRatingMatrix')
107
108      new_data=(n.data)
109      top_N= (getTopNLists(ratings, n=ncol(ratings)))
110
111
112     top_N <- removeKnownItems(top_N,new_data)
113
114     top_N <- bestN(top_N, n)
115
116     return(top_N)
117   }
118
119   ## construct and return the recommender object
120   new('Recommender', method = 'NB.2VAR', dataType = class(data)
121     ,ntrain = nrow(data), model = model, predict = predict)
122 }
123
124 ## Not implemented yet
125 REAL_NB <- function(data, parameter = NULL) {
126
127   model <- c(list(description='Naive Bayes for Real Ratings', data=data))
128
129   predict <- function(model, newdata, n=10,
```

```
130                         data=NULL, type=c('topNList', 'ratings', '
                                ratingMatrix'), ...) {
131
132     ###print('Naive Bayes hasn't been implemented for Real Ratings yet')
133   }
134
135   ## construct recommender object
136   new('Recommender', method = 'NB.2VAR', dataType = class(data),
137       ntrain = nrow(data), model = model, predict = predict)
138 }
139
140 ## register recommender
141 recommenderRegistry$set_entry(
142   method='NB.2VAR', dataType = 'binaryRatingMatrix', fun=BIN_NB,
143   description='A Naive Bayes Classifier (binary data).')
144
145 ## register recommender
146 recommenderRegistry$set_entry(
147   method='NB.2VAR', dataType = 'realRatingMatrix', fun=BIN_NB,
148   description='A Naive Bayes Classifier (real data).')
```

C.2 Temporal Splitting Implementation

```
1  ##############################################################################
2  # Temporal Splitting Schema
3  # :param schema_type: a string from the vector
4  #                     c('schema1','schema2','schema3','schema4')
5  # :param def.data: an input dataframe
6  # :param dt: an int time_interval_increment (in years)
7  # :return: two list of dataframes, each containing a dataframe for a
8  #                    temporal fold train and test based on imput params
9  ##############################################################################
10 do_time_split = function(schema_type, def.data, dt){
11   t_0 =  min(def.data$year)
12   t_f = max(def.data$year)
13   print(sprintf('t_f = %d    t_0 = %d    dt = %d',t_f, t_0, dt))
14   if(t_f - t_0 <= dt-1){
15     return(1)
16   }
17   train.data = list()
18   test.data= list()
19   for (i in seq((t_0 + dt), t_f, dt)){
20     switch(schema_type,
21            schema1={
22              train.data[[paste0('schema1',i)]] = def.data[def.data$year >=
                   t_0 & def.data$year < i, ]
23              test.data[[paste0('schema1',i)]] = def.data[def.data$year >= i
                    & def.data$year < i+dt, ]
24            },
25            schema2={
26              train.data[[paste0('schema2',i)]] = def.data[def.data$year >=
                   (i-dt)  & def.data$year < i, ]
27              test.data[[paste0('schema2',i)]] = def.data[def.data$year >= i
                    & def.data$year < i+dt, ]
28            },
29            schema3={
30              train.data[[paste0('schema3',i)]] = def.data[def.data$year >=
                   (i-dt) & def.data$year < i, ]
31              test.data[[paste0('schema3',i)]] = def.data[def.data$year >= i
                    & def.data$year <= t_f, ]
```

```
32              },
33              schema4={
34                train.data[[paste0('schema4',i)]] = def.data[def.data$year >=
                      t_0 & def.data$year < i, ]
35                test.data[[paste0('schema4',i)]] = def.data[def.data$year >= i
                       & def.data$year <= t_f, ]
36              }
37      )
38    }
39    return(list('train'=train.data,'test'=test.data))
40  }
```

C.3 Split Known Unknown

```
##############################################################################
# Modified version of split_KnownUnknown to leverage rownames for NB
# To check original, please visit the following link:
# https://github.com/mhahsler/recommenderlab/blob/master/R/evaluatio
# nScheme.R
#
# :param data: an input dataframe
# :param given: an int, specifying how many items to give to model
# :return: a list of two dataframes, each containing a dataframe for
#         a known and unknown splits need to leverage given-x protocol
##############################################################################
split_KnownUnknown_origi= function(data, given) {
  ## given might of length one or length(data)
  if(length(given)==1) given <- rep(given, nrow(data))
  nitems <- rowCounts(data)
  # print(nitems)
  allBut <- given < 0
  if(any(allBut)) {
    given[allBut] <- nitems[allBut] + given[allBut]
  }

  if(any(given>nitems)) {
    stop('Not enough ratings for user',
         paste(which(given>nitems), collapse = ', '))
  }

  l <- getList(data, decode=FALSE)
  known_index <- lapply(1:length(l),
                        FUN = function(i) sample(1:length(l[[i]]), given[i
                            ]))

  known <- encode(
    lapply(1:length(l), FUN = function(x)
      l[[x]][known_index[[x]]]),
    itemLabels = itemLabels(data@data))

  unknown <- encode(
```

```
37      lapply(1:length(l), FUN = function(x)
38        l[[x]][-known_index[[x]]]),
39      itemLabels = itemLabels(data@data))
40
41
42    known <- new('binaryRatingMatrix', data = known)
43    rownames(known) = rownames(data)
44    unknown <- new('binaryRatingMatrix', data = unknown)
45    rownames(unknown) = rownames(data)
46
47    return(list(
48      'known' = known,
49      'unknown' = unknown
50    ))
51  }
```

References

[1] Amazon, 2016. http://www.amazon.com.

[2] Code Coverage Tool — Intel Software, 2016. https://software.intel.com/en-us/node/522743.

[3] IMDb, 2016. http://www.imdb.com/.

[4] Jester joke recommender, 2016. http://eigentaste.berkeley.edu/.

[5] Netflix, 2016. http://www.netflix.com.

[6] Som toolbox / som_make, 2016. http://www.cis.hut.fi/somtoolbox/package/docs2/som_make.html.

[7] Spotify, 2016. http://www.spotify.com.

[8] Youtube, 2016. https://www.youtube.com/.

[9] Edward N Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[10] Anahita Alipour, Abram Hindle, and Eleni Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *Proc. of the 10th Working Conf. on Mining Softw. Rep.*, pages 183–192, 2013.

[11] Ethem Alpaydin. *Introduction to machine learning*. MIT Press, 2nd edition, 2010.

[12] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify

change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 23. ACM, 2008.

[13] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proc. of the 28th Int. Conference on Softw. Eng.*, pages 361–370, 2006.

[14] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 21–25. ACM, 2007.

[15] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate Bug Reports Considered Harmful... Really? In *Proc. Int. Conf. on Softw. Maintenance*, pages 337–345, 2008.

[16] Eric J Braude and Michael E. Bernstein. *Software Engineering: Modern Approaches*. John Wiley, 2nd edition, 2010.

[17] John S Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 43–52. Morgan Kaufmann Publishers Inc., 1998.

[18] Michael Buckley and Ram Chillarege. Discovering relationships between service and customer satisfaction. In *Software Maintenance, 1995. Proceedings., International Conference on*, pages 192–201. IEEE, 1995.

[19] George Candea, Stefan Bucur, and Cristian Zamfir. Automated software testing as a service. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 155–160. ACM, 2010.

[20] Iván Cantador and Pablo Castells. Semantic contextualisation in a news recommender system. In *Workshop on Context-Aware Recommender Systems (CARS 2009)*, pages 1–5, 2009.

[21] Gianfranco Chicco, Roberto Napoli, and Federico Piglione. Application of clustering algorithms and self organising maps to classify electricity customers. In *Power Tech Conference Proceedings, 2003 IEEE Bologna*, volume 1, pages 1–7. IEEE, 2003.

[22] Ram Chillarege, Shriram Biyani, and Jeanette Rosenthal. Measurement of failure rate in widely distributed software. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 424–433. IEEE, 1995.

[23] Dan Cosley, Shyong K Lam, Istvan Albert, Joseph A Konstan, and John Riedl. Is seeing believing?: how recommender system interfaces affect users' opinions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 585–592. ACM, 2003.

[24] Paolo Cremonesi, Roberto Turrin, Eugenio Lentini, and Matteo Matteucci. An evaluation methodology for collaborative recommender systems. In *Automated solutions for Cross Media Content and Multi-channel Distribution, 2008. AXMEDIS'08. International Conference on*, pages 224–231. IEEE, 2008.

[25] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1084–1093. IEEE Press, 2012.

[26] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[27] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[28] Robin Genuer, Jean-Michel Poggi, and Christine Tuleau-Malot. Variable selection using random forests. *Pattern Recognition Letters*, 31(14):2225–2236, 2010.

[29] Miha Grčar, Dunja Mladenič, Blaž Fortuna, and Marko Grobelnik. Data sparsity issues in the collaborative filtering framework. In *International Workshop on Knowledge Discovery on the Web*, pages 58–76. Springer, 2005.

[30] Paul Grey. How Many Products Does Amazon Sell? — ExportX, 2015. `https://export-x.com/2015/12/11/how-many-products-does-amazon-sell-2015/`.

[31] Michael Hahsler. Developing and testing top-n recommendation algorithms for 0-1 data using recommenderlab. *NSF Industry University Cooperative Research Center for Net-Centric Software and System*, 2011.

[32] Michael Hahsler. recommenderlab: A framework for developing and testing recommendation algorithms. *Nov*, 2011.

[33] Jon Herlocker, Joseph A Konstan, and John Riedl. An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms. *Information retrieval*, 5(4):287–310, 2002.

[34] Jonathan L Herlocker, Joseph A Konstan, Loren G Terveen, and John T Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53, 2004.

[35] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

[36] Lyndon Hiew. *Assisted detection of duplicate bug reports*. PhD thesis, University of British Columbia, 2006.

[37] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE International Conference on Data Mining*, pages 263–272. Ieee, 2008.

[38] G. Hughes. On the mean accuracy of statistical pattern recognizers. *IEEE Transactions on Information Theory*, 14(1):55–63, Jan 1968.

[39] Capers Jones. *Applied software measurement: global analysis of productivity and quality*. McGraw-Hill Education Group, 2008.

[40] Leon Wu Boyi Xie Gail Kaiser and Rebecca Passonneau. Bugminer: Software reliability analysis via data mining of bug reports. *delta*, 12(10):09–0500, 2011.

[41] Stephen H Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[42] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 247–254. ACM, 2001.

[43] Teuvo Kohonen. *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences*. Springer, 2001.

[44] Joseph A Konstan, Bradley N Miller, David Maltz, Jonathan L Herlocker, Lee R Gordon, and John Riedl. Grouplens: applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87, 1997.

[45] Yehuda Koren, Robert Bell, Chris Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[46] Gerald Kowalski. *Information Retrieval Systems: Theory and Implementation*. Kluwer Academic Publishers, 1st edition, 1997.

[47] Ahmed Lamkanfi and Serge Demeyer. Predicting reassignments of bug reports-an exploratory investigation. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 327–330. IEEE, 2013.

[48] Jong-Seok Lee, Chi-Hyuck Jun, Jaewook Lee, and Sooyoung Kim. Classification-based collaborative filtering using market basket data. *Expert systems with applications*, 29(3):700–704, 2005.

[49] Qing Li and Byeong Man Kim. Clustering approach for hybrid recommender system. In *Web Intelligence, 2003. WI 2003. Proceedings. IEEE/WIC International Conference on*, pages 33–38. IEEE, 2003.

[50] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002.

[51] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[52] Nathan N Liu, Evan W Xiang, Min Zhao, and Qiang Yang. Unifying explicit and implicit feedback for collaborative filtering. In *Proceedings of the 19th ACM inter-*

national conference on Information and knowledge management, pages 1445–1448. ACM, 2010.

[53] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26, October 2008.

[54] Victor Luckerson. The Number of Movies on Netflix Is Dropping Fast — TIME, 2016. `http://time.com/4272360/the-number-of-movies-on-netflix-is-dropping-fast/`.

[55] Martin Maechler, Peter Rousseeuw, Anja Struyf, Mia Hubert, and Kurt Hornik. *cluster: Cluster Analysis Basics and Extensions*, 2015. R package version 2.0.3.

[56] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[57] Charles E Metz. Basic principles of roc analysis. In *Seminars in nuclear medicine*, volume 8, pages 283–298. Elsevier, 1978.

[58] A. V. Miranskyy, E. Cialini, and D. Godwin. Selection of customers for operational and usage profiling. In *Proc. of the 2nd Int. Workshop on Testing Database Systems*, pages 7:1–7:6, 2009.

[59] A. V. Miranskyy, E. Cialini, and D. Godwin. Selection of Customers for Operational and Usage Profiling. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pages 7:1–7:6, 2009.

[60] Andriy V Miranskyy, Matthew Davison, and Mark Reesor. Metrics of risk associated with defects rediscovery. *arXiv preprint arXiv:1107.4016*, 2011.

[61] Audris Mockus and David Weiss. Interval quality: Relating customer-perceived quality to process quality. In *Proceedings of the 30th international conference on Software engineering*, pages 723–732. ACM, 2008.

[62] Robert E Mullen and Swapna S Gokhale. Software defect rediscoveries: a discrete lognormal model. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 1–10. IEEE, 2005.

[63] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. ACM, 2012.

[64] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016.

[65] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. of the 29th Int. Conf. on Softw. Eng.*, pages 499–510, 2007.

[66] Mefta Sadat, Ayse Basar Bener, and Andriy V. Miranskyy. Rediscovery Datasets: Connecting Duplicate Reports, 2017.

[67] Mefta Sadat, Ayse Basar Bener, and Andriy V. Miranskyy. Rediscovery datasets: Connecting duplicate reports. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017), (to appear)*, 2017.

[68] Mefta Sadat, Ayse Basar Bener, and Andriy V. Miranskyy. Rediscovery Datasets: Connecting Duplicate Reports of Apache, Eclipse, and KDE, March 2017. `https://doi.org/10.5281/zenodo.400614`.

[69] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 158–167. ACM, 2000.

[70] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Application of dimensionality reduction in recommender system-a case study. Technical report, DTIC Document, 2000.

[71] Badrul M Sarwar, George Karypis, Joseph Konstan, and John Riedl. Recommender systems for large-scale e-commerce: Scalable neighborhood formation using clustering. In *Proceedings of the fifth international conference on computer and information technology*, volume 1, pages 1–6, 2002.

[72] J Ben Schafer, Joseph Konstan, and John Riedl. Recommender systems in e-commerce. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166. ACM, 1999.

[73] Ian Sommerville. *Software engineering*. Addison-Wesley, 9th edition, 2011.

[74] Yoonki Song, Xiaoyin Wang, Tao Xie, Lu Zhang, and Hong Mei. Jdf: detecting duplicate bug reports in jazz. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 315–316. ACM, 2010.

[75] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 253–262. IEEE, 2011.

[76] Ashish Sureka and Pankaj Jalote. Detecting duplicate bug report using character n-gram-based features. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 366–374. IEEE, 2010.

[77] Yuan Tian, David Lo, and Chengnian Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *Proc. of Int. Conf. on Softw. Maintenance*, pages 200–209, 2013.

[78] Robert Tibshirani, Guenther Walther, and Trevor Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–423, 2001.

[79] Teemu Tunkelo, Ari-Pekka Hameri, and Yves Pigneur. Improving globally distributed software development and support processes a workflow view. *Journal of Software: Evolution and Process*, 25(12):1305–1324, 2013.

[80] Stefan Wagner and Helmut Fischer. A software reliability model based on a geometric sequence of failure rates. In *International Conference on Reliable Software Technologies*, pages 143–154. Springer, 2006.

[81] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information.

In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 461–470. IEEE, 2008.

[82] Ron Wehrens, Lutgarde MC Buydens, et al. Self-and super-organizing maps in r: the kohonen package. *J Stat Softw*, 21(5):1–19, 2007.

[83] Roel Wieringa and Maya Daneva. Six strategies for generalizing software engineering theories. *Science of computer programming*, 101:136–152, 2015.

[84] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[85] Alan P Wood. Software reliability from the customer view. *Computer*, 36(8):37–42, 2003.

[86] Jialiang Xie, Minghui Zhou, and Audris Mockus. Impact of triage: a study of mozilla and gnome. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 247–250. IEEE, 2013.

[87] Robert K Yin. *Case study research: Design and methods*. Sage publications, 5th edition, 2013.

[88] Jie Zhang, Xiaoyin Wang, Dan Hao, Bing Xie, Lu Zhang, and Hong Mei. A survey on bug-report analysis. *Science China Information Sciences*, 58(2):1–24, 2015.

Index