Automated Approaches for Bug Triaging

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Intelligent Systems

presented by
Igor Kovacevic

under the supervision of
Prof. Dr. Michele Lanza

June 2013

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Igor Kovacevic
Lugano, Yesterday June 2013

The real voyage of discovery consists not in seeking new landscapes, but having new eyes

Marcel Proust

Abstract

In software engineering, when bugs are reported through bug tracking systems, human intervention is needed to assign to developers the bugs to be fixed. This activity is known as bug *triaging*, which also includes the detection of duplicate or incomplete bug reports.

Bug triaging is still largely a manual and therefore error-prone process: When a bug is wrongfully assigned to a developer, this leads to the phenomenon of bug *tossing*, the re-assignment of a bug to another developer, which in turn leads to a loss of time and wasted resources.

We present an approach, based on an extended model of bug tossing and a set of machine learning techniques, to automatically suggest the developers most competent to tackle the fixing of given bugs, thus reducing the bug tossing phenomenon.

We implemented a toolset which mines bug tracking system repositories, models and analyses the mined data, and recommends developers best suited to fix specific bugs. We validated our approach and toolset on a custom made extensive and publicly available dataset.

Acknowledgements

First of all, I would like to thank my advisor Prof. Dr. Michele Lanza for his support and his kind and patient supervision. I'm very grateful for giving me the opportunity to pursuit a master thesis in an elegant and structured manner.

I also want to thank all the members from the REVEAL research group that were there when I was wandering in their office in search of inspiration. Thanks to all the people that I have the opportunity to met at USI, specially to my colleagues, I learned a lot from you. To my family, dad and mom, thanks for the support, and for the unconditional love, thanks for everything, I'm the way I am thank to you.

Thanks to my girlfriend Nadja, for his love and patience and to have not thrown at me any dangerous object when I was spending all of my time in front of my desk, I know that was not easy, Thank you darling!

Last but not least, thanks to all my friends that continued to ask for me despite the fact that in the last month the probability of seeing me was nearly the same as seeing an UFO.

Contents

Figures

Tables

Chapter 1

Introduction

In World War I doctors had to face tough questions treating the wounded, who to help first, which patients could wait and which could not. Time and resources were limited and good decisions saved people. The process of prioritizing intervention based on the gravity of the injuries is called "Triaging" and happens everyday in medical emergencies and disasters. The term Triaging originates from the French verb 'trier', meaning 'to sort'.

Likewise, in software engineering, resources are limited and the triaging decisions are also important. They are present in the context of the management and tracking of issue reports or feature requests.

Large projects often have facilities to track and manage the work to be done. Most of the time this is done trough a bug-tracking system. Users and Developers can use those systems to report a faulty behavior of the software product they use or submit a request for a new feature. However, there must be someone that does the *triaging*. The person who triages the reports or bugs have two primary goals:

1. Reduce the number of reports to the most complete ones. To achieve this goal there are two subproblems that must be resolved. Dealing with duplicate reports and filtering reports that are badly described. In large software projects several people may submit a report describing the same bug. These duplicate reports must be gathered together so that the developers effort is not wasted by having several people fixing the same problem. Bugs that have too little information need to be excluded or reprocessed to include important information, so that developers can focus on solving the actual problem, instead of wasting time trying to reproduce a badly described bug.

2. Find the person that will tackle the problem. This activity is called *expert finding*. This is not trivial: good triagers must have a knowledge of the entire system (modules, components,et cetera) and also of the people that are involved. This is difficult in open source projects since the assessment of the activities and expertise of people that are globally distributed is difficult.

Sometimes the triager is not able to analyze the report, and the help of experienced developers or triagers is needed. In this case the triager assigns the report to an expert, giving it the role of a triager for that issue, and that person is responsible for the decision on how to deal with the problem.

Jeong et al. [2009] have shown that the assignment of a bug in Eclipse is made in 40 days on average and that if the developer cannot find a solution the re-assignment is made in 100 days. These numbers suggest that a wrong triaging decision can have important consequences and can lead to a waste of resources and time.

The re-assignment of a bug to another developer is known as *Bug Tossing*. Various studies have shown that in open source projects between 30% and 60% of the bugs are tossed. This means that often wrong decisions are made.

Bug Tracking Systems were introduced to improve the development process but the rise of global software development combined with the ubiquitousness of the Internet brought an increment of the triaging cost that is not negligible. Every day in the Mozilla Project Foundation the number of submitted bugs are around 30 and if we suppose that for every bug the time to evaluate it and assign it to a developer is 5 minutes, they spend 2.5man/hour on the triaging activity that could be spent on actual coding and product improvement.

Bug Triaging is largely dominated by manual processes, and we know that human involvement is error prone. To assist bug triagers we need to find an automatic approach to boost the productivity and reduce errors of a single triager. Several studies indicate an approach to automate the bug triaging, from finding duplicates, to recommenders that produce a list of potential developers using several machine learning techniques. The ultimate goal behind the research in this area is to lower the cost of triaging activities by automating most of the processes and reducing human involvement.

In this thesis we perform an exhaustive review of the research field. We investigate the preprocessing of the data, mining two different bug-tracking systems, Bugzilla and Jira. We analyze different machine learning algorithms in detail and augment the classifiers with *tossing graphs*. A tossing graph represents the bug tossing event for every developer in the project. This can lead to insights on the behavior of single developers and reveal internal team structure. We propose different meta-models to deal with bug repositories and bug tossing events and we discuss the limitations of the approach and possible solutions.

The outcome of this work is a tool that mines Bugzilla and Jira, produces a recommender to assist bug triagers and permits to investigate the use of tossing graphs and ranking algorithms on bug triaging tasks. Our approach achieves an accuracy of up to 70% in 5 open source project that use Bugzilla and 5 that use Jira.

1.1 Contributions

In this thesis we make the following contributions:

- **A generalized bug report meta-model:** To investigate different techniques on a general dataset, we must fetch the data from two different bug-tracking systems, Jira and Bugzilla. Since they represent bug reports in different ways, we propose a meta-model that generalizes the information need and allows to cope with both systems.
- **A novel model for bug tossing:** Until now, nobody has proposed a model that can be used in practice to establish effectively if a bug is tossed or not. Present models illustrate the phenomena of bug tossing only from a theoretical point of view without any practical implications. To fill the gap between theory and practice we present a model for bug tossing that is directly applicable to real bug-tracking system (Jira and Bugzilla).
- **An exhaustive review of the techniques used in research to automate bug triaging:** We illustrates the main techniques used in the research area to deal with bug triaging. We provide the results of those methods and discuss their limitations.

1.2 Structure of the Document

The rest of the document is structured in different chapters, described below.

- **Chapter 2** describes the work related to our research. A brief history of software engineering from the first software configuration management systems to the introduction of a research field called Mining Software Repositories (MSR). Then we describe the state of the art of bug triaging and we give a brief overview of the two bug-tracking systems, Jira and Bugzilla.
- **Chapter 3** introduces machine learning discussing some basic algorithms and the main evaluation methods.
- **Chapter 4** illustrates the theoretical foundation for the application of all techniques in the context of bug triaging. Moreover, we present the dataset and our research framework that we developed and used in this thesis.
- **Chapter 5** presents the evaluation of our methods.
- **Chapter 6** concludes this thesis and discusses future work.
- **Appendix A** shows how to deal with technical issues regarding the mining of Jira and Bugzilla.

Chapter 2

Related Work

2.1 Software Evolution and Bug Tracking Systems

The first software configuration management (SCM) system, SCCS, was developed by Rochkind [1975]. This marked the moment when software systems started to be perceived as evolving constructs. The term *software evolution* was coined by Lehmann (Lehman [1980]) who established a set of laws that govern this phenomenon. Lehman remarked that "software must change to adapt to a changing world".

One of the laws introduced by Lehman says that the functional content of a software system must be continually increased to maintain user satisfaction. This mean that software systems are in continuous growth, thus leading to more errors in the code. An error in a software system is called *bug*. For each software system there is an ad-hoc approach to deal with errors in the system.

During the nineties the first systematic approach to do management of bugs was created, a bug-tracking system (BTS). The first bug-tracking systems, GNATS [1] and Debuggs[2] were very simple from a user interface point of view and they were superseded by Mozilla's Bugzilla[3] in 1998. Bugzilla was the first web-based bug-tracking system that incorporated numerous functionalities such as test planning, documentation integration/generation, multiple projects support, and so on. After that SourceForge[4] offered a sorce code repository service with bug managing capabilities and many others proposed their own solution, including Jira[5] in 2002 and Google[6] with their own custom bug-tracker in 2007.

In the same years in the International Conference on Software Engineering a workshop shed some light on a promising new field, Mining Software Repositories. MSR focuses on two key points:

[1] GNATS: http://www.gnu.org/software/gnats/
[2] Debuggs: http://www.debian.org/Bugs/
[3] Bugzilla: http://www.bugzilla.org/
[4] SourceForge: http://sourceforge.net/
[5] Jira: http://www.atlassian.com/software/jira
[6] GoogleCode: https://code.google.com/

1. Discovery of novel approaches to mine information from software repositories
2. Techniques of extraction of information from these repositories

Typical mined software repositories are source control repositories, bug trackers, archived communications (mailing lists, IRC chats, instant messaging), code repositories, and so on.
In the last years many studies have been made using mined informations from repositories. Mockus et al. [2002] compared open source development with closed source development mining emails, CVS and bug-trackers. Canfora and Cerulo [2006] came up with the notion that unstructured communication between developers like emails can be a valuable source of information to help understanding the code. Tichelaar et al. [2000] proposed the Release History Database that combines bug and version report data into a relational database. D'Ambros and Lanza [2006] used the concept of RHDB in their tool, BugCrawler, that visualize the relationship between the evolution of software artifacts and how they are affected by bugs. A meta-model called FAMIX is proposed to model source code and is extended to support several metrics. Another research done by D'Ambros et al. [2012] is to compare several approaches for predicting software defects combining source code model FAMIX, source code history data, bug-tracker and source code metrics.
As we can see, software repositories are mined to extract useful information for various purposes. In this thesis we mine bug-tracking systems to assist one of the principal activities of bug triaging, expert finding.

2.2 State of the Art

Anvik et al. [2005] analyzed open bug repositories and described their relative difficulty to mine them. The main reason is that in open source projects bug repositories do not have access restrictions and almost anyone can create or update bug reports. In these conditions, many bugs need to be triaged requiring a vast amount of resources and time. The main goals of bug triaging are:

Finding duplicates. Often, the same bug is reported by different people. This leads to several similar reports describing the same bug. Anvik et al. [2006] showed that the number of duplicate bug reports is substantial, ranging from 20% to 30% in systems such as Mozilla and Eclipse. Very often a common technique for finding duplicates is to use natural language text procedures to find similarities between bugs. Once the duplicates are found, triagers have two ways to proceed, exclude duplicates from the reports, or use duplicates to enrich the information about the main "sibling issue" [7]. Bettenburg et al. [2008b] suggested that the second alternative is preferable. They have shown that incorporating duplicate bugs in the training of a classifier for assisting expert finding improves the performance compared to a classifier trained without duplicates.

Finding experts. When a bug is submitted, the triager needs to assign it to a suitable person that can solve the issue. In this context Cubranic and Murphy [2004] proposed to use machine learning and text categorization to assist the search of experts using the informations from the bug description. The accuracy achieved with this approach is around 30% on a collection of 15,859 bugs from a large open source project. Anvik et al. [2006]

[7] the main sibling issue is the duplicate report considered valid, the only report that will contain all the information about the issue gathering all the data from all the duplicates in respect to the same issue

outlined some difficulties in tracing information between bug and source code repositories and expanded the work of Cubranic proposing a semi-automatic approach that suggests the most top three suitable developers reaching accuracy values of 57% to 64%. Bettenburg et al. [2008a] made a survey on how bug reports are used and propose a tool, CUEZILLA, that measures the quality of bug reports. Jeong et al. [2009] proposed a way to improving the accuracy of expert predictors based on tossing graphs, a Markov model that outline the bug tossing event for each bug and each developer. Bhattacharya and Neamtiu [2010] extended the work of Jeong et al. augmenting the classifier with tossing graphs and incremental learning, reaching high accuracy values.

All of these approaches use unstructured information to train the ML classifiers such as bug description and bug title combined with text categorization methods, In our approach we use only structured information and show that the training time spent on training the classifiers is too large compared to the gain in accuracy obtained using unstructured informations.

2.3 Bug-tracking Systems

In this section we present two of the most common bug-tracking systems, Bugzilla and Jira.

2.3.1 Bugzilla

Figure 2.1. Mozilla Foundation Bugzilla main page

Born in 1996, Bugzilla is a free bug-tracking system developed by the Mozilla Foundation.It is under used in thousand of organization worldwide. It is written in Perl and uses MySQL or PostgreSQL as relational database. It is flexible enough to support multi projects environment. For each project (in Bugzilla called product) one can create several components to distribuite the reports into categories and ease the interaction and the management of each project.

Since Bugzilla can contain an enormous amount of bugs, it provide simple and advanced search features.

Some interesting features are:

1. Communicate with teammates
2. Manage quality assurance
3. Submit and review patches

When we open Bugzilla at the main page (see Figure 2.1) we can create an account and submit an issue or search for a related problem. To submit a bug we need to set the project and fill the Bug form. We can see in Figure 2.2 that the required fields are product, component, version, and the summary. Most of the times when a user sets a wrong field such as component, other project members change this field to the correct one, but this delays the process of fixing the issue. Bugzilla offers an XML-RPC or a JSON-RPC webservice to interact with it.

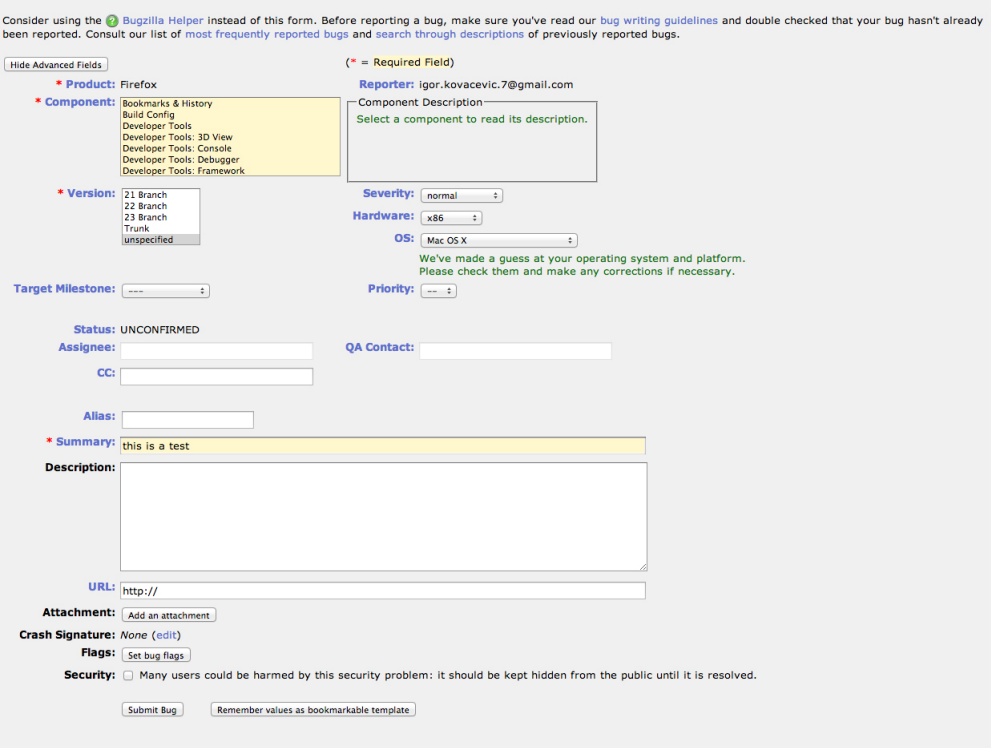


Figure 2.2. Bugzilla bug submission form

The problem is that a lot of methods are instable or experimental. When we had to mine it, we decide to crawl the web page using the advanced search features that fetches the results in csv format. In the table 2.1 we outline some features of the API[8] of bugzilla. We can see in the table 2.1 that most of the needed features are INSTABLE or EXPERIMENTAL.

[8]API : Application Programming Interface

Package	Type	Operations	Description	Stability
Bug	Utility Functions	fields	Get informations about valid bug fields, including the lists of legal values for each field.	UNSTABLE
Bug	Bug Information	search	Allows you to search for bugs based on particular criteria.	UNSTABLE
Bug	Bug Information	attachments	It allows you to get data about attachments, given a list of bugs and/or attachment ids.	EXPERIMENTAL
Bug	Bug Information	comments	Get data about comments, given a list of bugs and/or comment ids.	STABLE
Bug	Bug Information	get	Gets informations about particular bugs in the database.	STABLE
Bug	Bug Information	history	Gets the history of changes for particular bugs in the database.	EXPERIMENTAL
Product	List Products	get	Returns a list of informations about the products passed to it.	EXPERIMENTAL
User	User Information	get	Gets informations about user accounts in Bugzilla.	

Table 2.1. Bugzilla webservice operation support

Bugzilla is free, supports large projects and its features are sufficient for most organizations. Moreover it is tested and maintained since a long time now. The downside is that it is old and the user interface it is not modern and user friendly.

2.3.2 Jira

Jira is a popular issue tracker launched in 2003. Developed by Atlassian, Jira is a commercial product. It is used for issue, feature, task and project management.It offers a better user interface than Bugzilla and it is highly expansible and customizable. One of the main reasons for its diffusion is that Atlassian offers free licenses to open source projects.

It is implemented in Java and support several databases: PostgreSQL, MySQL, Oracle, DB2, and so on. Jira is provided with the Tomcat web server in the "installation bundle package" that contains all software necessary to install and use the product.

We can see in Figure 2.3 that the simple user interface gives all the necessary information about the product:

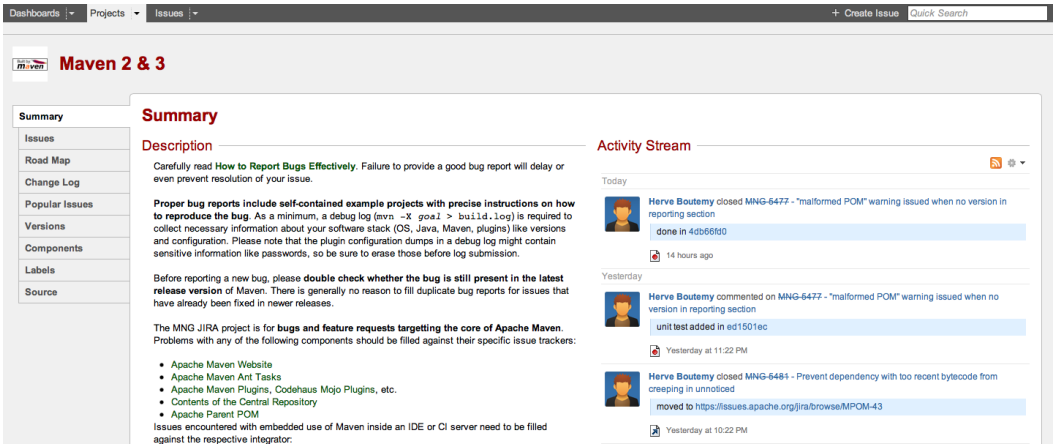


Figure 2.3. Jira main page of the product Apache Maven

- Project status
- Issues list
- Issues assigned to you
- In-progress issues
- Open issues and so on

Jira is well suited for large projects. It also supports multiple projects and categories like Bugzilla. The search features are powerful and provide a language similar to SQL[9], the Jira Query Language (JQL), to manage data in the repository. Moreover, it offers a RESTful[10] API webservice for querying the repository. The results are fetched in JSON [11] format. In Figure 2.4 we can see a JSON response for a query to fetch bug information for the issue LUCENE-1. In figure 2.5 we see the web form offered by Jira to create a bug report. The overall support for common operations on repository are a lot better compared to Bugzilla webservice.

[9]SQL: Structured Query Language is a special-purpose programming language designed for managing data held in a relational database management system

[10]RESTful API: is a web API implemented using HTTP and REST principles

[11]JSON : JavaScript Object Notation, is a text-based open standard designed for human-readable data interchange.

```
{
   expand: "renderedFields,names,schema,transitions,operations,editmeta,changelog",
   id: "12314151",
   self: "https://issues.apache.org/jira/rest/api/2/issue/12314151",
   key: "LUCENE-1",
 - fields: {
    - progress: {
         progress: 0,
         total: 0
      },
      summary: "lock files don't work in JDK 1.1",
      timetracking: { },
    - issuetype: {
         self: "https://issues.apache.org/jira/rest/api/2/issuetype/1",
         id: "1",
         description: "A problem which impairs or prevents the functions of the product.",
         iconUrl: "https://issues.apache.org/jira/images/icons/issuetypes/bug.png",
         name: "Bug",
         subtask: false
      },
    - votes: {
         self: "https://issues.apache.org/jira/rest/api/2/issue/LUCENE-1/votes",
         votes: 0,
         hasVoted: false
      },
      fixVersions: [ ],
    - resolution: {
         self: "https://issues.apache.org/jira/rest/api/2/resolution/1",
         id: "1",
         description: "A fix for this issue is checked into the tree and tested.",
         name: "Fixed"
      },
```


Figure 2.4. Example of an issue detail request on the Jira RESTful webservice

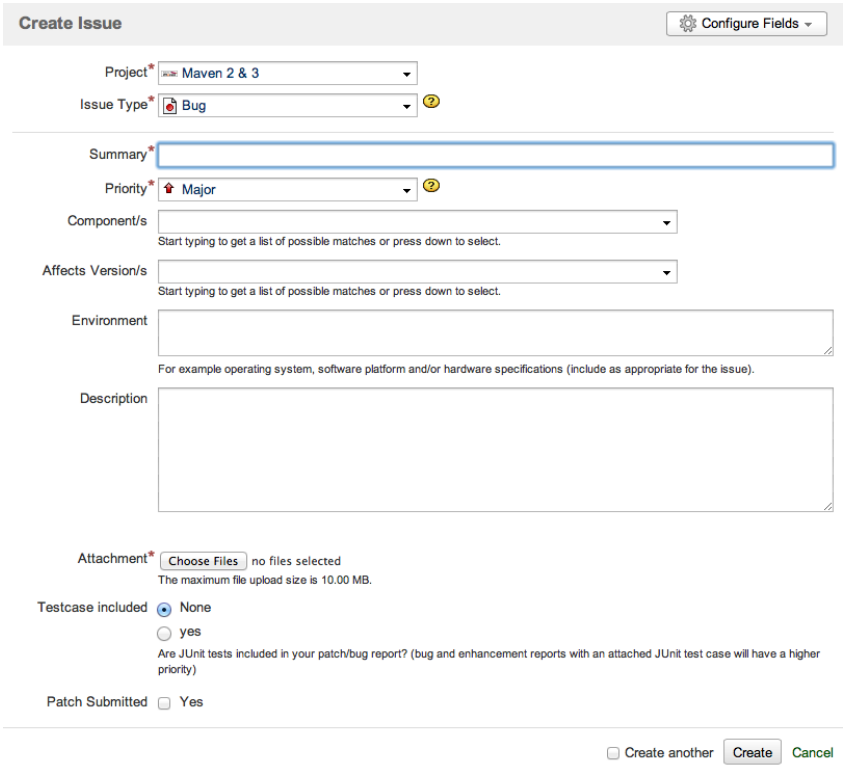


Figure 2.5. Jira issue creation page.

2.3.3 Problems and improvements in bug-tracking systems

Bug-tracking systems were created with the goal of managing bugs in a project. Looking at Bugzilla and Jira it is clear that these systems were developed with developers in mind as the only clients. Many operations to fill a good bug report need expertise on the project. This is a clear sign that the "naive user" was not considered as a potential client, but nowadays this is the case. To list one clear sign of this vision we can think of the field component. This field represent the software component which probably contains the defect, and setting it correctly requires knowledge of the system. To improve bug-tracking systems, this kind of fields must be automated and suggested to the user by the system. Another aspect that is missing and that can be improved, is the validation procedure. Both systems we analyzed do not provide a validation of the bug reports. There must be a validation procedure that avoids bad descriptions or missing fields using some Information Retrieval techniques to assess the quality of the reports.

2.4 Summary

We have shown a brief history of software engineering from the birth of the first SCM to MSR. We have given an overview of bug triaging and the related issues. We analyzed the state of the art procedures to assist people in bug triaging and to introduce bug-tracking systems, we discussed Bugzilla and Jira, two common solutions for bug-tracking. In the next chapter we lay the theoretical foundations of our work, from machine learning to bug report information extraction.

Chapter 3

Intermezzo:Machine Learning

Machine Learning (ML), a branch of Artificial intelligence, was defined in 1959 by Arthur Samuel as the "Field of study that gives computers the ability to learn without being explicity programmed". Tom M. Mitchel gives a more formal definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". Since then many things have changed, and we encounter, without knowing, many machine learning applications, from search engine *ranking systems* that find relevant document in order by relevance given a specific query, to *collaborative filtering system* that recommend similar books to buy given our buying history or preferences. In security applications we can encounter *face recognition* that given a photo or video can establish who the person is. In medical treatment there are software systems that predict diseases given the symptoms. In other words, machine learning is ubiquitous.

Machine learning can be organized in algorithms types [1]:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

To understand better these types we illustrates the problem of ML learning with Figure 3.1.

We see that the learning problem is concerned with finding the right model that outputs desired data, given a certain input. In general, the model is found by having a dataset called training data (often a subset of the real data) and some computation by one of the many machine learning algorithms with that data. In general we say that the *Training Set* is used to train the model or *classifier* to perform the task we seek. The training data is made of *instances* that represent some situations or cases. The single instance is a case of the phenomena we want to model, for example if we try to predict weather today, a single instance can be the yesterdays pressure, temperature and other weather conditions.
The *features* are properties of the system we want to model, for example if we want to model the face of a person, features may be the distance between the eyes, the color of the eyes, the

[1] There exist more types, but in this thesis we focus only on the main one.

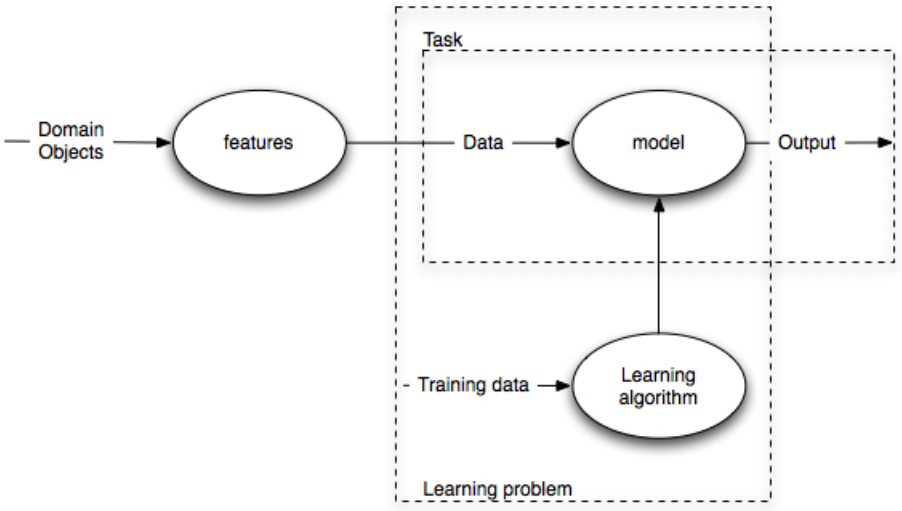


Figure 3.1. The ML Learning problem

geometric proprieties of the nose, and so forth. An important aspect that is not present in this picture is the *validation dataset*, is used to validate the model or classifier with some input data. The validation is important since it is the only way we can measure how our model performs with new data. To summarize, we have a *classifier* that learns a *model* using a *training set* made of *instances* that contain *features* representing the phenomena.

Now that we have an idea of the main purpose of machine learning we can discuss tree algorithm types. We begin with **supervised learning** (SL). SL is a set of algorithms and techniques that produce a model or a classifier using training data that are *labeled*. This means that for each *instance* we know the output (label).

The **unsupervised learning** approach instead tries to model the data without knowing the output of the provided instances. One example of this classification can be clustering of similar news, where the goal is to gather all similar stories (news aggregator) without knowing in advance the category of each article.

Reinforcement learning is based on finding the right actions for an agent in a environment that maximize some reward function. We can think of a practical example, an autonomous Helicopter flight. In this application we have the helicopter (the agent) that tries to find the best rotor actions (the actions) to remain balanced and fly correctly. The reward function in this case is a function which represents the goodness of actions with respect to the helicopter balancing and flying.

In this work we focus on *supervised learning*. Algorithms that belong to the class of supervised learning are subdivided into more categories: linear models, decision tree and probabilistic models.

3.1 Linear Models: Support Vector Machines

A linear model or classifier is used to identify a class of an object with certain characteristics. This classification is made using the values of a linear combination of the different characteristics of object (or instance). The characteristics are called features, and are represented as a feature vector $\vec{X}$. The classification is made by mapping all values above a certain threshold to a class and the others to the second class. In this case the classifier is said to be binary.

In a more formal fashion the linear model tries to find a hyperplane H : **Hyperplane H.** *where* $\langle , \rangle$ *is the dot product, and* $w \in H$ *and* $b \in R$. *Such a hyperplane naturally divides H into two half-spaces, and therefore can be used as the decision boundary of a binary classifier.*

$$\{x \in H | \langle w, x \rangle + b = 0\} \tag{3.1}$$

In Support Vector Machines the algorithms try to maximize the distance between the closest point $\vec{X}$ to the hyperplane H.

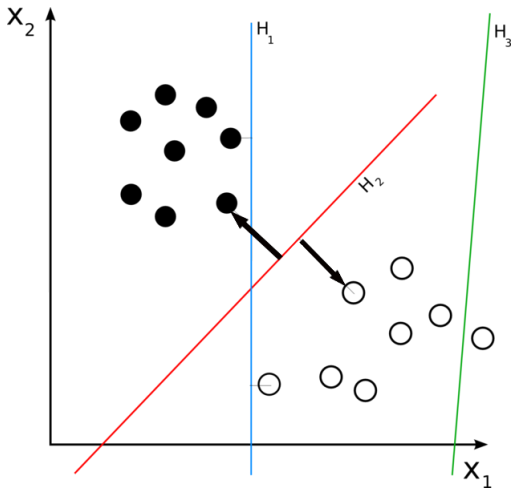


Figure 3.2. Example illustrating support vector machine models

The name support vector machine is derived from the vector that "supports" the plane as a pillar.

Multi-class classification To learn models that are able to classify into more than two classes, like in the case of expert recommender, where we want to find the expert in a population of n people, we need a multi-class classification algorithm. In SVM this is done by finding multiple Hyperplanes. In the case depicted in Figure 3.2 we see 3 different hyperplanes. In this configuration we are able to identify 7 different classes. This is achieved with a classification called one-vs-all. With this technique the algorithm tries to identify one class among the rest, then the same for the following classes, one at the time.

3.2 Decision Trees: C4.5

C4.5 is an algorithm belonging to the *decision trees* family developed by Ross Quintlan. A *decision tree* is a tree-structured classification model known for its simple interpretation, even

by nonexpert users. In Figure 3.3 we see a decision tree induced from a sample dataset that represent the possibility of playing golf for each possible weather scenarios. The Classification start from the top node until we reach a leaf, passing through the node that has the attribute corresponding to the instance we want to classify.

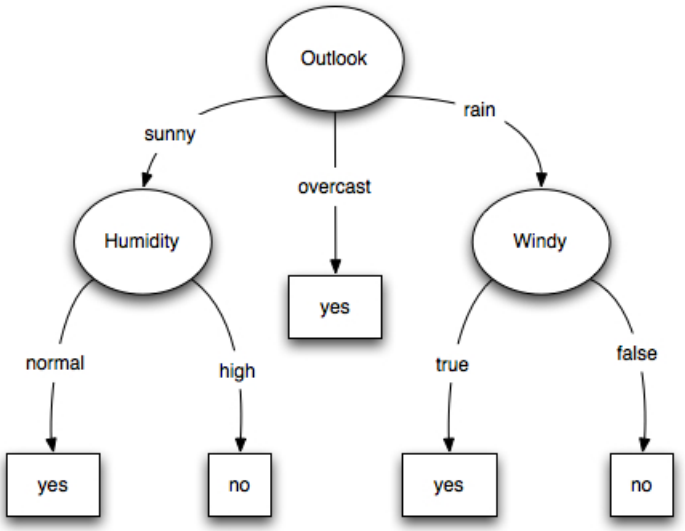


Figure 3.3. Decision Tree that tries to predict if is appropriate to play golf with the actual weather conditions

The algorithm used for building trees works in a top-down fashion, the algorithm begins with the root node, then splits the root node into different disjoint sets selecting the best attribute for the split. There are different splitting criteria, but the one used in the C4.5 is the entropy measure :

Entropy H(s). *where S is a set of training example for which entropy is being calculated, and S_i is the set of training examples that belong to class c_i*

$$H(s) = -\sum_{i=1}^{c} \frac{|S|}{|S_i|} \log_2 \frac{|S|}{|S_i|}. \tag{3.2}$$

This measure is highest at the point where the classes are equally distributed and lowest where one S_i contains all examples ($S_i = S$) and all other $S_j, j \neq i$ are empty. Then the best split is found by searching the Highest information Gain among the splits.

Information Gain IG(S,A), *where $IG(S,A)$ measures the difference in entropy from before the split $H(S)$ and after the split $H(S_t)$.*

$$Gain(S,A) = H(S) - \sum_{t} \frac{|S_t|}{|S|} H(S_t) \tag{3.3}$$

3.3 Probabilistic models: Naive Bayes

A probabilistic classifier try to determine the probability that an instance belongs to certain class. The Naive Bayes classifier estimate this probability applying Baye's theorem and making a strong (naive) independence assumption. In simple words a Naive Bayes classifier assumes that features does not affect each others. The posterior probability for a given class is given by:

$$P(C|F_1,..,F_n) = \frac{P(C)P(F_1,..,F_n|C)}{P(F_1,...,F_n)} \tag{3.4}$$

This probability express the intuition that if something occurred frequently in the past, then it is more likely to occur in the future. The equation 3.4 states that the probability that an instance is of class C given that the instance has the features values $F_1,..,F_n$, is equal to the probability that the instance is of class C times the probability that the features values will have these values if the class is C, divided by the probability that the features will have the values $F_1,..,F_n$.
Taking into account the strong independent assumption the formula refeq:posterior probability reduces to:

$$P(C|F_1,..,F_n) = \frac{1}{Z}P(C)\prod_{i=1}^{n}P(F_i|C) \tag{3.5}$$

Where Z is a scaling factor dependent on $F_1,...F_n$. Despite the fact that this assumption is false in a lot of real-world applications, Domingos and Pazzani [1996] found that in practice classifiers based on Naive Bayes perform surprisingly well and it has been found to be a mathematically reasonable assumption.

3.4 Evaluation in machine learning systems

In this section we illustrate concepts and techniques to evaluate machine learning classifiers.

3.4.1 Accuracy, recall and F-score

To evaluate classification performance, a general approach is to use accuracy computed as:

$$Accuracy = \frac{Correct\ Prediction}{Number\ of\ instances} \tag{3.6}$$

The problem of using accuracy is when the class distribution if skewed. For example if we are trying to predict if a produced item will fail after 10 years and the probability of failing is 5%. A bad classifier can predict all the instances as non failing and the accuracy will be 95%, but this information will be misleading.

A better measure to assess the quality of a classification is to use precision and recall. To discuss easely the computations of the following formulas we assume that we are evaluating a binary classifier. An example could be a classifier that receive as input a picture and predict if the photo contains cats or dogs.
Precision and recall are defined as follows:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{3.7}$$

Precision measures how many instances that were predicted as dogs were actually dogs.

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{3.8}$$

Recall measures of all the instances that actually are dogs, what fraction did we correctly predict as dogs.
To understand the computation for precision and recall we need to understand the confusion matrix depicted in the table 3.1.

		Actual class	
		DOG	CAT
Predicted	DOG	true positive	false positive
	CAT	false negative	true negative

Table 3.1. Confusion Matrix

In this table we have the values of the classification outcome (predicted) and the real value of the instance classified (actual class). The true positive and true negative are when we classify correctly an instance while the false positive and the false negative are when we made an error in the classification.

With those measures we can establish the quality of a classifier, but now instead of having a row number to judge the classifier we have two measures. Ideally we want to have a classifier that has high precision and high recall, but the measure are inversely proportional, if we have a very high precision classifier we will have a low recall. The balance between the two is the goal of a good classifier. To combine the two measures we can use the *F-score* measure. This measure incorporates both types of information and gives us a number to judge the quality of the classification. Higher the F-score, the better the classifier will perform.F-score is defined in the formula 3.9 where P and R are precision and recall

$$F_{score} = 2\frac{PR}{P+R} \tag{3.9}$$

3.4.2 Classifier comparison

Using F-score we have a way of establish the quality of a classification. If we want to compare two algorithms or two classifiers we need a test to assess which is the best. To address this problem there are numerous statistical tests. We decided to use the Z-statistic.

Z-statistic

$$Z = \frac{p_a - p_b}{\sqrt{\frac{2p(1-p)}{N}}} \tag{3.10}$$

Where p_a and p_b are the quality measure of each classifier (Accuracy,F-score,and so on), p is the difference between p_a and p_b and N is size of the dataset used in the classification

The classifier A is assumed to be better than classifier B if Z > $Z_\alpha/2$ where $Z_\alpha/2$ is the upper/lower bound obtained from a standard normal distribution at confidence level 1-α.

Here we have some values for $Z_\alpha/2$ with different confidence level

$1-\alpha$	0.99	0.98	0.95	0.9	0.8	0.7	0.5
$Z_\alpha/2$	2.58	2.33	1.96	1.65	1.28	1.04	0.67

Table 3.2. $Z_\alpha/2$ at different confidence levels $1-\alpha$

3.5 Summary

In this chapter we have introduced machine learning and presented the necessary theoretical foundations for applying ML to a general problem. We showed various types of learning paradigms and the principal algorithms. We discussed how to evaluate classifiers and how to compare classification algorithms.

In the next chapter we show how to extract the necessary information from bug-trackers in order to apply the machine learning techniques we discussed to bug triaging and build an expert recommender.

Chapter 4

Automated Approaches for Bug Triaging

4.1 Bug Tossing

In software development when a bug is assigned to a developer and subsequently reassigned to another developer the phenomenon is called *bug tossing*. Numerous studies present discussions and statistics on bug tossing but nobody has presented a formal way to mine this phenomenon from a bug-tracking system. In this section we model and formalize this concept.

In Bugzilla, to establish if a bug is tossed or not, if we simply count the changes in the assignment field, sometimes we make a mistake. Unfortunately, most bug-tracking systems treat the "assigned to" field in different ways. Sometimes a bug is assigned to a team of developers, then reassigned to the right developer inside those teams. From time to time the same field is used to assign the bug to a default value (in Bugzilla nobody@bugzilla.org) that corresponds to setting a label equal to "to be assigned". Some times the same field remains unchanged despite the fact that several people are trying to fix the same bug proposing solutions and submitting patches. To overcome these unstructured ways of handling the assignment we propose a model or heuristic to establish if bug is tossed or not.

Two important pieces of information are the assignee-to field and the names or identification numbers of the patch submitters. They can reveal the true fixer of a bug and the tossed status. The typical cases that we encounter in bug-trackers are:

1. Bug is assigned to someone and the same person fixes the bug.
2. Bug is assigned to someone, then the bug is re-assigned to another developer who actually fixes the bug.
3. Bug is assigned to someone, but another person provides a patch that leads to the resolution of the issue.
4. Bug is assigned to someone, the same person provides several patches before actually fixing the bug.

5. Bug is assigned to someone, the same person decides to reassign the bug to the value nobody@bugzilla.org and then after a while regains access to the same bug to fix it.

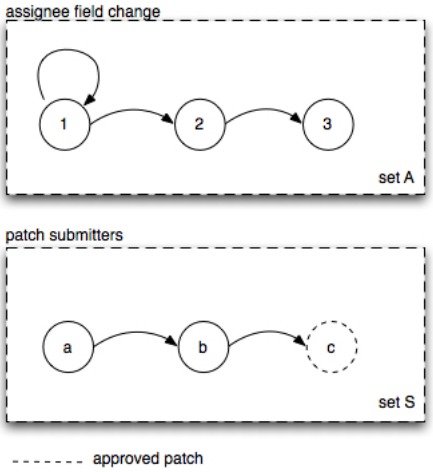


Figure 4.1. State machine that models bug tossing events

Figure 4.1 represent our tossing model; The set {A} is a set containing all the developers that were assigned to the bug. A self loop represent a developer that decides to drop the responsibility for the bug and then regains it after a while. If the size of the set {A} is more than 1, the bug is considered as tossed. In the case of project that uses the value nobody: if the last node of the graph in the set {A} is a nobody value the size is considered as the $size - 1$. This is due to the fact that the node nobody is sometimes used to close the bug without giving any indication about which quality assurance user needs to do the last piece of work. If the size of the set {A} is less than 2, but there are several different submitters in the set {S}, the bug is also considered tossed, since the actual contribution is done by some of the submitters, ideally the last one.

4.2 Labeling bug reports

To do supervised learning it is necessary to have a dataset that has instances with known class labels. For bug triaging recommenders this means that every bug report needs to be labeled to the name or id of the developer who fixed the bug. It seems a trivial task: we can use just the assigned-to field or the fixed field. In reality those fields are used in different ways for every project. Sometimes the fixed field is set to the last person who worked on the report, but not necessarily the real solver of the issue. In some cases the assigned-to field does not change despite the fact that the bug has been solved from someone different from the first assignee. To address this problem, Anvik and Murphy [2011] and others have proposed a set of project-specific heuristics to label the reports. Those heuristics derive from manual inspections of the bug reports for each project. Some example could be:

- If a report is resolved as FIXED, label it with whoever submitted the last approved patch.
- If a report is resolved as FIXED, label it with whoever marked the report as resolved.

4.2.1 Heuristics

In our approach we also used *heuristics* to label the bug reports. In this section we outline the rules we used:

- Label the report with whoever submitted the last approved patch.
- If there is not any patch submission in the activity history, label the report using the assigned-to field:
  - If the last assigned-to field does not correspond to nobody (i.e null in jira or nobody@mozilla.org), label the report with the last assigned-to field
  - If in the history of the assignments is empty, the report is considered invalid.

4.2.2 Email aliasing

In open source bug-tracking systems it is allowed to have aliases for an account. This means that the same person can have multiple names in the activity history, thus leading to confusion in the interpretation of an assignment.

We have tried to tackle this problem with similarity metrics between aliases like *Levenshtein distance* [1] and more complicated ones but without success.

4.3 Knowledge discovery in bug reports

In this chapter we tell a story of a Jira bug to explain the problem of establishing the tossing status and extracting the real bug fixer. This is done by means of a graph generated from our tool.

4.3.1 Case 1: a bug's life

We begin by illustrating a story of the bug CAMEL-3240, see figure 4.2.

This issue, is in fact a feature request that says "*Graceful shutdown will force shutting down routes if timeout triggeres. We should add option to let end user control this. So Camel instead just gives up. Then end user can take action, such as trying to shutdown again or whatever.*".

The first action is the triager that assigns the bug to njiang on 14 jannuary 2011 at time 01:06. The same day , approximatevely one hour later, njiang set the report as Resolved (2011-01-14T02:29).
After 15hours circa, the issue is then reopened and assigned to boday by himself (2011-01-14T15:50). Boday then dispatches the issue to nobody (unassigned) at time 2011-01-14T17:52.
After a couple of minutes the issue passes to hadrian who works on the issue for two days until davsclaus, an external person, changes the status to Resolved, and assigns it again to boday. Dkulp (another external person) in the end closes the bug marking it as Closed.

[1] Levenshtein distance is a string metric for measuring the difference between two sequences.

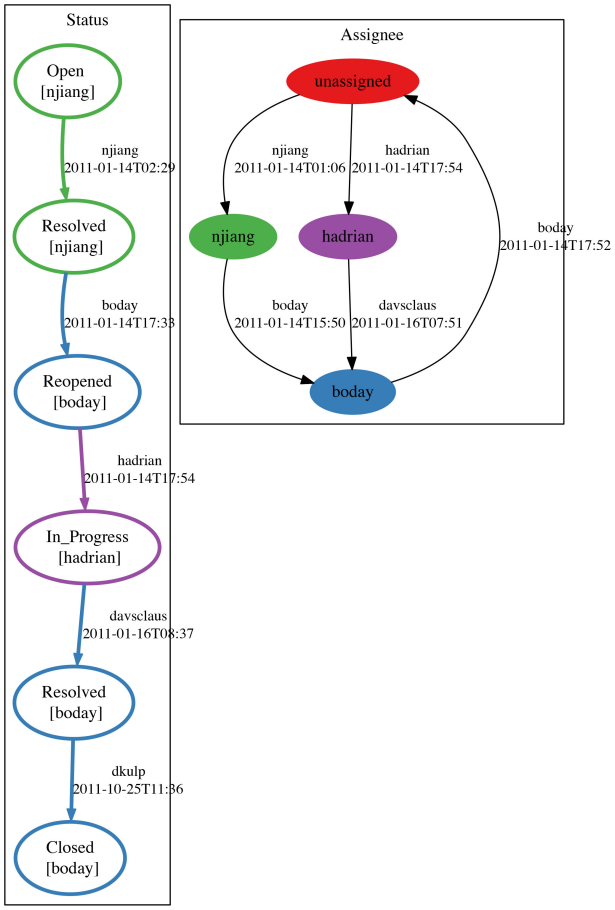


Figure 4.2. Jira Issue history of the project CAMEL,issue 3240

Looking at this situation the real fixer could be hadrian or boday. In the end boday submitted the last patch when the bug was assigned to hadrian. This is an example of submitters who work on the issue despite the issue not being assigned to them.

4.3.2 Case 2: misbehaviors in a bug's report

The second example, figure 4.3, include two misbehaviors or difficulties to extract informations.

1. External submitter who fixes the bug
2. Username/email aliasing

Without explaining all the story we can see that the issue was resolved or treated by two people, njiang and hzbarcea. A naive approach is to assign the role of fixer to hadrian, since he is the last assignee. With a more detailed inspection we discover that the real fixer is not present in the history of the assignee-to field.

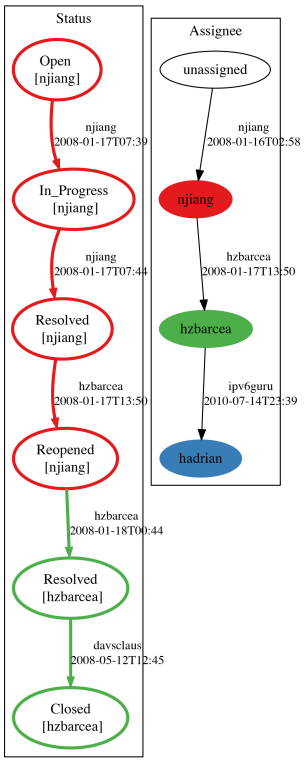


Figure 4.3. Jira Issue history of the project CAMEL, issue 276

In fact the issue was resolved by an external developer who submitted a patch, solving the problem. Moreover the last change in assignment is an evidence of *username aliasing*, in fact hzbarcea and hadrian are the same person.


```
    created: "2010-07-14T23:39:36.299+0000",
  - items: [
      - {
            field: "assignee",
            fieldtype: "jira",
            from: "hzbarcea",
            fromString: "Hadrian Zbarcea",
            to: "hadrian",
            toString: "Hadrian Zbarcea"
        }
    ]
},
```


Figure 4.4. JSON response that indicates email aliasing in issue report CAMEL-276

Figure 4.4 shows that the values fromString and toString are the same, in fact represent the name of the user, but the from and to are different, which indicates the usernames. In Jira it is easy to find aliases using this kind of data, but for Bugzilla this is not possible since it has only one field related to the developer, a this field correspond to from and to field of Jira described earlier.

4.4 Summing up

In the previous sections we showd how the unstructured practices of bug-tracking systems make difficult the process of data extraction. To overcome some problems we proposed a bug tossing model and a labeling heuristic. The problem of email aliasyng remain and can lead to wrong results in our work, however we think that the number of users that uses aliases are not high and we decided to ignore this issue.

4.5 Projects selection

To generalize our approach and to expect statistically significant results we decided to select open source projects that meet certain criteria. The requirements are composed of three criteria. The first one is that we want to discuss two bug-tracking systems, Jira and Bugzilla, therefore our projects must use one of the two **bug-tracker**. The second criteria is the **years of development**. To have enough history to do machine learning and to be able to use the approach on a test set that is big enough we decided to set the threshold to ($\geq$5). The last requirement is the **number of report that have the status set to fixed**. This requirement is a cause of our approach. Since we measures bug tossing using only fixed reports it is critical to have a dataset big enough, therefore we setted this value to >4000.

To find projects that meet those criteria we chose to search the Mozilla Foundation and Apache Software Foundation code base. These are two big non-profit corporations that produce software that satisfies our requirements. Mozilla uses Bugzilla as bug-tracking system, while Apache uses both Jira and Bugzilla. In table 4.1 we show the projects that we have chosen to build our dataset.

Project name	Bug Tracking system	First bug in the dataset	Fixed bugs
Thunderbird	Bugzilla	2000	5,298
Toolkit	Bugzilla	1999	7,457
Bugzilla	Bugzilla	1994	7,800
MailNews Core	Bugzilla	1997	8,254
Firefox	Bugzilla	2001	15,586
SeaMonkey	Bugzilla	1998	18,479
CXF	Jira	2006	3,997
CAMEL	Jira	2007	5,162
HADOOP	Jira	2005	4,793
HBASE	Jira	2007	5,020
LUCENE	Jira	2001	3,209

Table 4.1. Project selection

In the following lines we describe the two groups of projects we have chosen.

Bugzilla

1. Thunderbird : Email client
2. Mozilla Toolkit : The Mozilla Toolkit is a set of APIs, built on top of Gecko, which provide advanced services to XUL applications. These services include Profile Management, Chrome Registration, Browsing History, Extension and Theme Management, Application Update Service, and Safe Mode.
3. Bugzilla : Bug-tracking system
4. MailNews Core : Mail and news components common to Thunderbird and SeaMonkey
5. Firefox Desktop: Mozilla's Web browser.
6. SeaMonkey : An all-in-one internet application suite, including web browser, e-mail and newsgroup client, and HTML composer.

Jira

1. CFX: Services framework. CXF helps you build and develop services using front-end programming APIs, like JAX-WS and JAX-RS. These services can speak a variety of protocols such as SOAP, XML/HTTP, RESTful HTTP, or CORBA and work over a variety of transports such as HTTP, JMS or JBI.
2. Camel: Integration framework based on known Enterprise Integration Patterns.
3. Hadoop (common): Software for reliable, scalable, distributed computing. The project includes three modules and we have used the Hadoop Common modules that contains common utilities that support the other Hadoop modules.
4. HBase: is the Hadoop database, a distributed, scalable, big data store.
5. Lucene: is a high-performance, full-featured text search engine library written entirely in Java.

4.6 Research Framework

4.6.1 Overview

To perform our study on how to deal with Bug triaging we employed a research framework. This framework is used to conduct the statistical analysis on the dataset and build and evaluate a recommender for bug assignments. The high-level architecture is shown in Figure 4.5. First we describe the data extraction then we discuss the data preprocessing and the machine learning toolset. The datasets consists of two group of projects where each group uses a different system for managing and tracking issue reports. The Mozilla Foundation projects use Bugzilla, while the projects we selected from the Apache Foundation use Jira.
To get the data from the first group we asked the Mozilla Foundation to give us a sanitized[2] copy of the entire history of the Mozilla projects. This copy is stored as a SQL Dump, this means that we needed a tool for extracting the data from a SQL Database and storing it in our format.

[2]A sanitized copy means a copy of the DB that does not contain sensitive information to avoid security risks

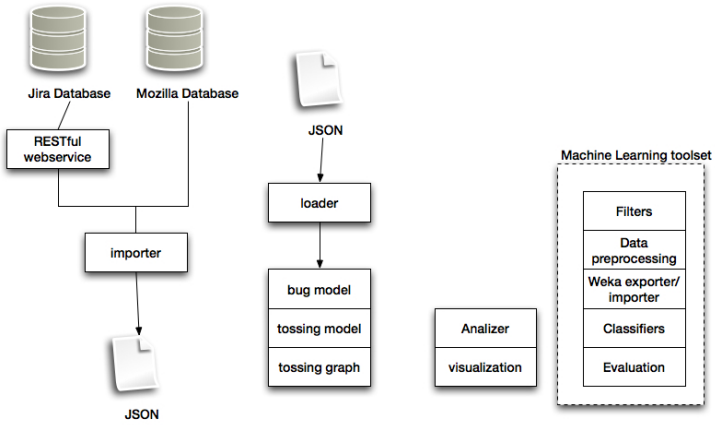


Figure 4.5. Research framework overview

In the case of the Apache Foundation we decided to extract the informations getting the data from the web service provided by Jira. This service delivers the information of the bug reports in JSON format.

4.6.2 Bug report Meta-Model

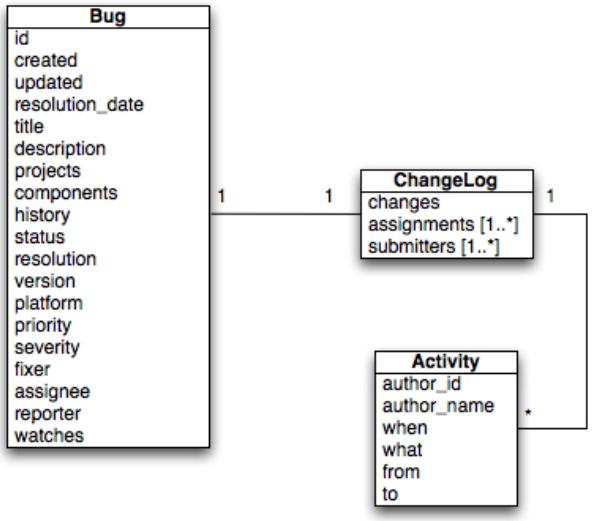


Figure 4.6. Bug report meta-model that enable us to treat Jira Issues and Bugzilla bugs equally

The Bug meta-model depicted in Figure 4.6 illustrates the structure of an issue or feature request in a Bug Tracking System. It contains five types of informations:

1. **Description of the problem** contains a unique identifier, the date of creation, resolution and last update, a short description (generally represented by the title) and a long description, the project and the component that belongs to and the history (changelog).

2. **Status** is composed of two fields, status and resolution, where each can have multiple values. The status field indicates the state in which the bug is currently in. Thus can be NEW, UNCONFIRMED, ASSIGNED, RESOLVED, CLOSED, VERIFIED. The resolution indicates how the problem was solved. It can be FIXED, DUPLICATE, INVALID, WORKS FOR ME, WON'T FIX. Those values clearly depend on the type of convention used in the bug tracking system, but generally are similar to each other.

3. **Condition** represent the specific conditions in which the issue came up, version of the software and platform.

4. **Criticality** is indicated by severity and priority field, where the values are system dependent.

5. **People involved in** are all the people that have worked on the issue. The reporter is the person who announces the bug, the assignee is the person who is assigned to the bug, the fixer is the person who fixes the bug.

ChangeLog

The changelog represents the activity history of the issue. Those are all activities related to the issue during its life. The activity can be of several types:

- Attachment (submit a patch or information about the bug)
- Status change, both status and resolution
- Assignee change
- General Field change (component, project, and so on)

The changelog is an important piece of information for mining a bug tracking system. In our approach we extract:

1. Patch submitters
2. Bug fixer
3. Tossing path

Analyzing manually the data from Bugzilla and Jira we realized that sometimes the field are erroneously changed and do not represent the truth about the issue. One of the fields that can be misused is the assignee field. This field is used to indicate the person who will tackle the problem but sometimes it is assigned to the name of a team, or sometimes assigned to a value that represent nobody like "nobody@bugzilla.org". In this setting it is difficult to extract the real bug fixer and the people that have worked on the problem so we used a heuristic based approach.

To perform all statistical analyses we implemented all statistical methods in a class named

Analyzer and different issue report graph methods.
The descriptive statistics methods contains developers statistics, triagers statistics and bug statistics. The graph methods provide a way to investigate visually the bug activity history with two kind of graphs.

1. Bug Life graph: the graph in Figure 4.7 illustrates all the percentages of bug status transitions used in the history of the project.

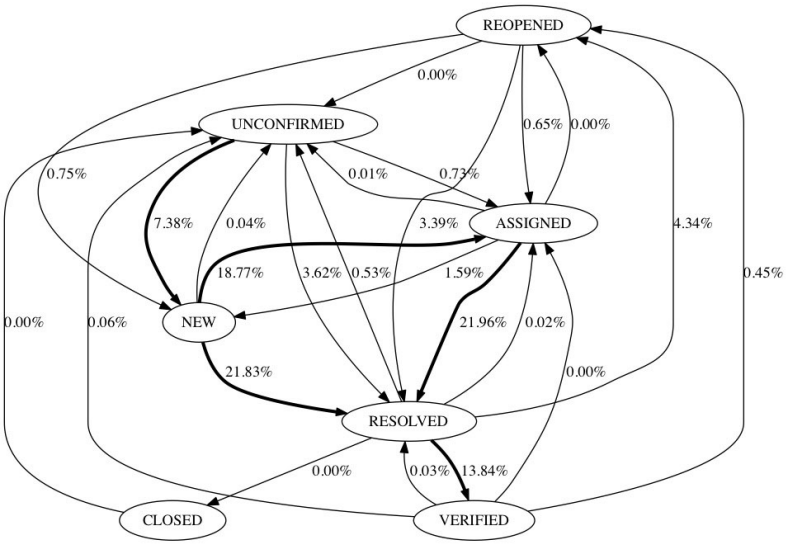


Figure 4.7. Bug Life graph: All transition probabilities between bug states

2. Bug status graph: the graph in Figure 4.8 can reveal the bug tossing event and all the transitions between the creation and closing of a single bug. With this graph it is easy to see if a bug has been tossed, who are the people involved in the bug fixing process, and the time between one transition and the other.

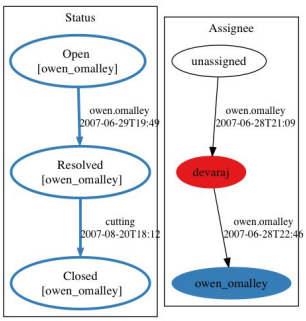


Figure 4.8. Bug status graph: this kind of graph ease the interpretation of the report history

4.6.3 Machine Learning toolset

The machine learning toolset is composed of two parts:

1. Data handling and preprocessing
2. Classification and evaluation algorithms

Data handling and data preprocessing

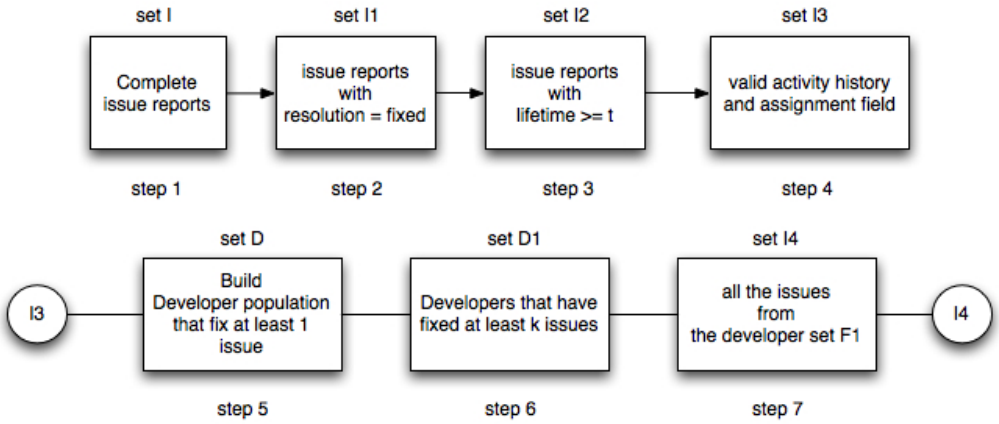


Figure 4.9. Data preprocessing: Steps in the dataset filtering, from the raw dataset to the final filtered dataset used to do machine learning classification

In every project, to do machine learning effectively, there is a need for a phase where the data is being preprocessed and filtered to meet certain criteria. Without that phase the classification part can lead to bad results.

In this part dedicated to data preprocessing we perform the filtering in more steps as depicted in Figure 4.9. To explain more accurately each step we separated each part in a separate subset of the dataset.

1. The first set is the complete issue report dataset extracted from the Database or from the web service. It contains all the activity history and data for each issues in the history of the project.
2. The second set (Set {I1}) contains all the issues that has the resolution set to FIXED. This is due to the fact that for our purpose, building a recommender, the issues that have the resolution set to a value different from FIXED are only contributing to generate noise in the dataset. In fact Jeong et al. [2009] have shown that considering the issues with the resolution set to VERIFIED generate noisy tossing graph and worse result.
3. In the set {I2} there are all the bugs that have a lifespan more than a certain threshold k. This filtering is to prevent having fake bugs in the dataset. The fake bugs are those that were introduced by mistake from a person and closed after a couple of minutes.

4. {I3} contains all the issues that have the activity history not corrupted, e.g missing assignee field, or ambiguous activity history.

5. To do the next steps of the filtering we generate a developer population from the set {I3}. This dataset contains people extracted from the set {I3} considering all the persons that have fixed at least one issues. This set is different from all the set I* in content type, the sets D* contains list of developers and the respective fixed issues idendification numbers, while the sets I* contains only issue reports.

6. {I4}: we sort the developer population with respect to the count of fixed issue per developer and filter out all the developers that does not reach a threshold of k bugs.

7. The final step is to gather all the issues that are present in the activity of the developers in the set {I4}, simply by removing all the issues that are not present in the set {I4} from the set {I3}.

Classification and evaluation algorithms

The classification algorithms are implemented using the Weka Data Mining libraries. There are all the algorithms mentioned in the machine learning introduction, Naive Bayes classifier, C4.5, and Support Vector Machine. In our toolset we implemented also:

- Top-k classification
- cross validation techniques
  - Inter-folding technique
  - incremental technique (Intra-folding)
- Tossing graphs augmented Classifier

Top-k classification

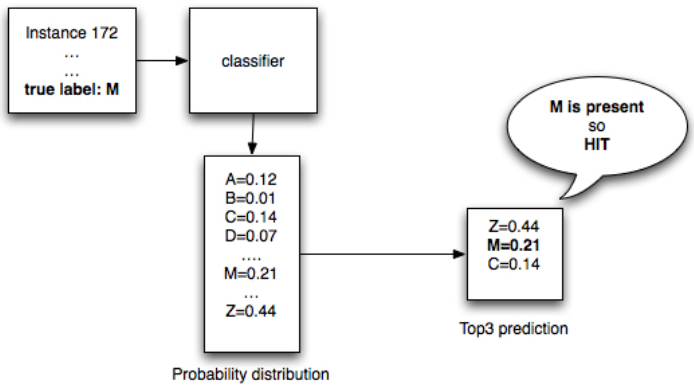


Figure 4.10. Example of a Top3 classification

To compare our results with previous studies we needed to implement top-k classification. Top-k classification is extending the prediction of the classifier with the k most probable outcomes of the model. Instead of considering just one possible outcome we extract the classes with the highest probability. Since all classifiers used in this work produce a probability distribution it is straightforward to compute the top-k classification.

Cross validation techniques

Cross validation is a model validation method for estimating the quality of a statistical model. Generally the procedure is divided in n rounds. In each round the dataset is divided into into complementary subsets, one subset is used for training and the others for testing.

K-fold cross-validation

The most popular cross validation technique is the K-fold cross-validation shown in Figure

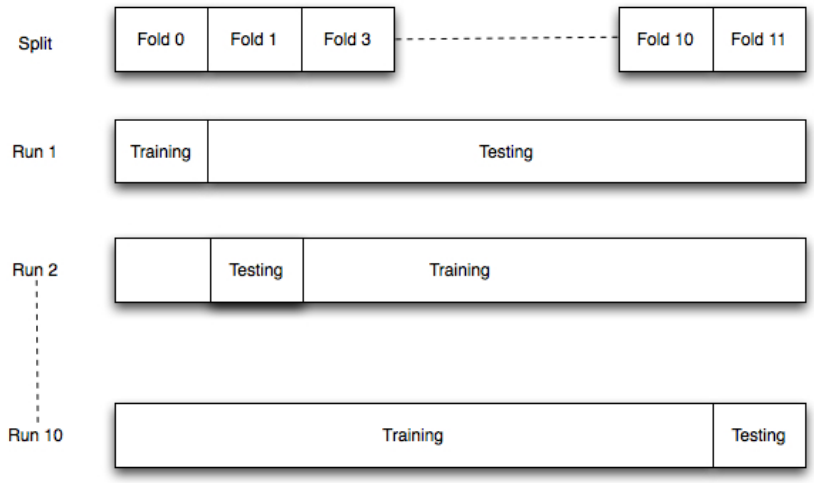


Figure 4.11. K-Fold cross validation, in this example K=11

4.11. In this type of validation the dataset is splited into k equal size subsets performing K rounds. In each round the subset K is used for validation and the rest of $k-1$ subsets are used for training. In this manner each instance in the original dataset is used once in validations and multiple times in the trainings.

Interfolding

In this technique depicted in Figure 4.12 the dataset is ordered in chronological order and is divided into k subsets. There are k rounds and in each rounds the subset of previous round is added to the training dataset and the testing subset is the kth subset.

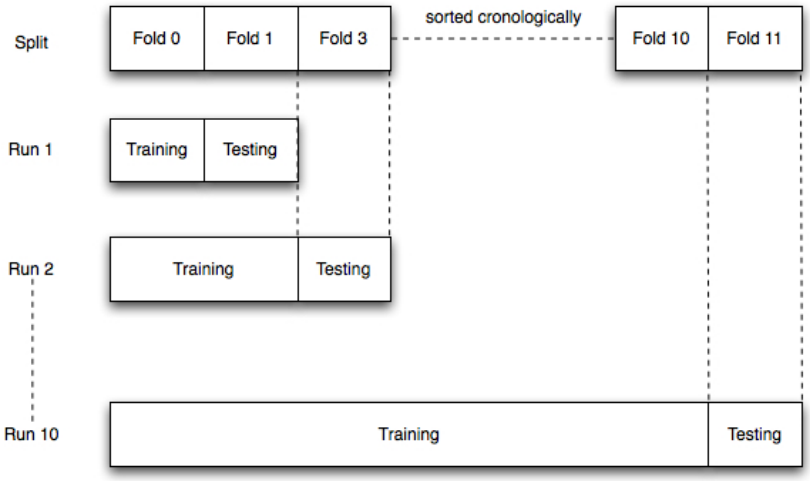


Figure 4.12. Inter folding cross validation

Incremental learning

One way to prevent the classifier to be outdated is the incremental learning or intra-folding technique shown in Figure 4.13. Like in the inter-folding cross validation, the dataset is partitioned into k subsets, but for each instance test, the classifier is updated with all the previous instances of the kth fold (e.g when the evaluation of the k instance from the fold 3 is performed, all the instances from 0 to k-1 from the fold 3 are present in the training dataset together with all the previous folds, fold 1, fold 2).

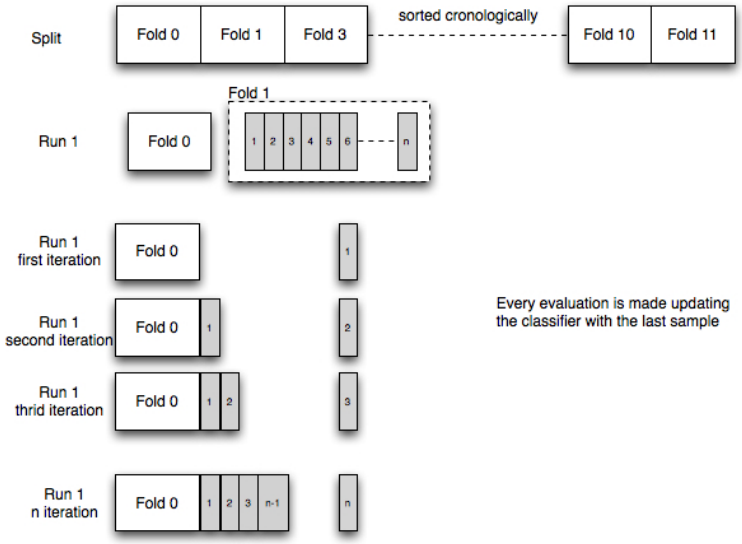


Figure 4.13. Intra folding cross validation, aka incremental learning

Tossing graphs augmented Classifier

Jeong et al. [2009] introduced the concept of Tossing graphs to improve the knowledge about the system. With this information they were able to extract development team structure and improve a bit the assignment recommendation. In few words they have implemented a Markov Model that incorporate the tossing events of bugs for each developer. Consider the scenario depicted in the table 4.2.

Tossing paths
C ⟶ B ⟶ A
B ⟶ E ⟶ F ⟶ A
D ⟶ C ⟶ B ⟶ A
A ⟶ D ⟶ C ⟶ E
E ⟶ C ⟶ A ⟶ B
F ⟶ D ⟶ B ⟶ A

Table 4.2. Bug tossing events

We have 6 bugs that has been tossed several times. Since we are trying to find the most suitable developer, we are interested in the most probable fixer of the bug, in this case developer A, shown in Figure 4.14. To find that developer, we need to compute the probability of tossing the bug to the set of fixers (in this case the developers that have fixed the bugs are A, B and E) using the formula 4.1, where $D \longrightarrow D_i$ is a tossing event from D that reaches the fixer D_i.

$$Pr(D \longrightarrow D_j) = \frac{\#(D \longrightarrow D_i)}{\sum_{i=1}^{n} D \longrightarrow D_i} \tag{4.1}$$

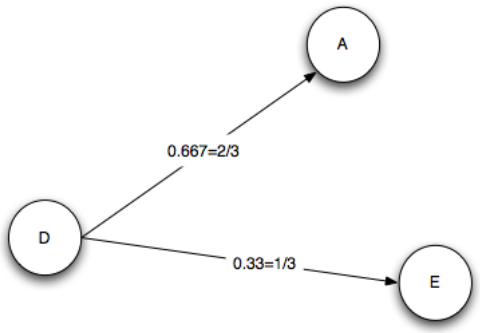


Figure 4.14. Tossing graph for developer D

Multi-feature tossing graph

Beside tossing probability, Bhattacharya and Neamtiu [2010] have tried to incorporate more information in the graph. Product, component and last know activity of the developers were added to the tossing graph creating a Multi-feature tossing graph.

They have shown that augmenting the classification set (from a machine learning classifier) using a multi-feature tossing graph and a ranking metric they were able to increase prediction accuracy by an average of 10%.

Bhattacharya and Neamtiu [2010] use a machine learning classifier (NB,SVM) to predict the set of possible developers removing all the people that were inactive in the past 100 days obtaining a set CP $D_1, D_2, D_3, ..., D_j$. This set then is augmented computing for each D_i the most probable developer T_i based on the ranking metrics 4.2. In the end they have a modified set CP $D_1, T_1, D_2, T_2, ..., D-j, T_j$ that are used for predicting the real fixer.

$$\begin{aligned} R(D_k) = & Pr(D_i \longrightarrow D_k) \\ & + MatchedProduct(D_k) \\ & + MatchedComponent(D_k) \\ & + LastActivity(D_k) \end{aligned} \tag{4.2}$$

$Pr(D_i \longrightarrow D_k)$ is the tossing probability as we computed in 4.1, MatchedProduct and MatchedComponent are equal to 1, if the developer have received a toss from more than one bugs with the same component or product, otherwise 0. Last Activity is equal to 1 only if the developer D_k has been active in the last 100 days, otherwise 0.

4.7 Summary

In this chapter we have shown a practical model for bug tossing to explain the information extraction process for our research. We discussed the main issues related to data preparation showing that in deed bug-tracking system practices make the process of mining informations more difficult.
The last part was dedicated to our research framework, we gave a high level architecture view and we presented the dataset used in the thesis. Now we discuss the results of those approaches on our dataset beginning with project insights that gave us a more detailed view on the projects we have chosen.

Chapter 5

Results

In this chapter we report the results of each phase of the approach on each group of projects, Jira and Bugzilla projects. During the discussion of the results, the reader should gain more detailed knowledge on each project, and on each of the steps of the approach.

5.1 Descriptive statistics

5.1.1 Bugzilla and Jira workflow

Bugs move through numerous states in their lifetime, sometimes states are similar in different bug-tracking systems. The commonality is that the ideal workflow model is almost never respected. In those model a bug is suppose to step through a series of states where only certain transitions are permitted. For example ideally a bug that reach the status "verified" shouldn't move to "new" but in reality this happens.

In figure 5.1 and in figure 5.2 we can see the ideal workflow for both systems, Jira and Bugzilla, while in figure 5.3 and in figure 5.4 the real workflow extracted from the repositories of the project Firefox that uses Bugzilla and Hadoop that uses Jira.

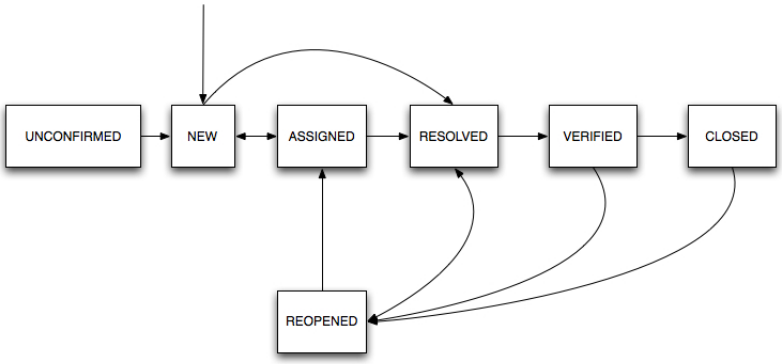


Figure 5.1. Bugzilla workflow

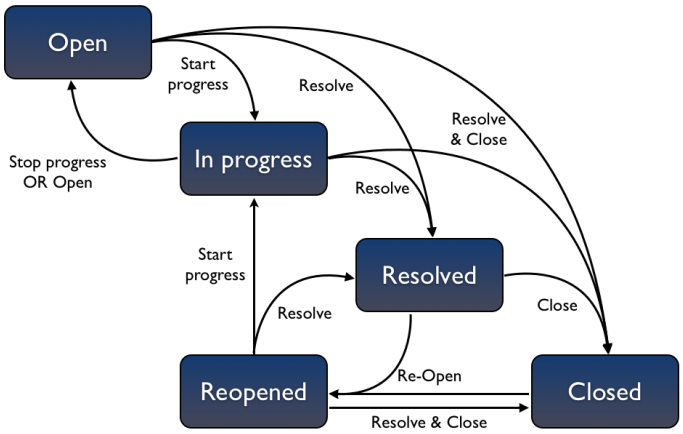


Figure 5.2. Jira workflow

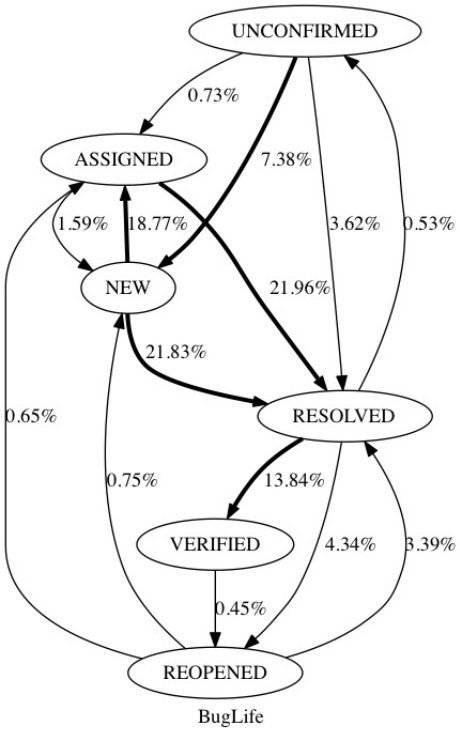


Figure 5.3. Firefox workflow, state transition probabilities

As we can see in the workflows extracted from Firefox (Fig 5.3) and Hadoop (Fig 5.4) the ideal path is not always respected.

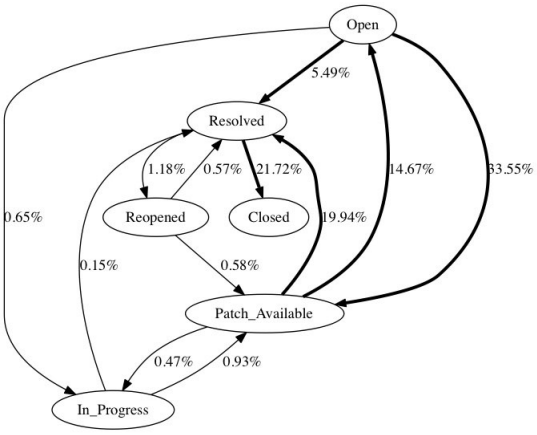


Figure 5.4. Hadoop workflow, state transition probabilities

Bugzilla and Jira Workflow statuses

The main status values used in Bugzilla and Jira are defined as follows:

- Bugzilla
  - unconfirmed: This is the starting point when a Bug is created and the user who submit the bug have not enough privilege.
  - new (confirmed): Is the next step, when a bug is confirmed to be a real issue. If the submitter is a empowered user, the bug start at new.
  - assigned: When the triager assign the issue to a developer the bug is set to assigned.
  - resolved: The bug reaches the state resolved when there is a possible solution.
  - verified: The QA verified the solution, if is good the bug have reach the end, if the solution is not satisfying the bug proceed to the status reopened.
  - reopened: The solution doesn't fix the problem therefore the bug is reopened, and will probably reach again the status new.
- Jira
  - open: Is the first status of a bug report, correspond to the new status of Bugzilla. In Jira there is no unconfirmed state therefore all the issues starts directly in the new state.
  - in progress: this is the status when the bug is assigned to someone and the work is in progress.
  - patch available: there is a patch available and need to be checked.
  - resolved: The bug reaches the state resolved when there is a possible solution.
  - closed: The bug has been processed and fixed.
  - reopened: The solution doesn't fix the problem therefore the bug is reopened, and will probably reach again the status new.

5.1.2 Bug status

The proportion of fixed bugs and duplicate bugs vary a lot between the project with Jira and the Projects with Bugzilla.

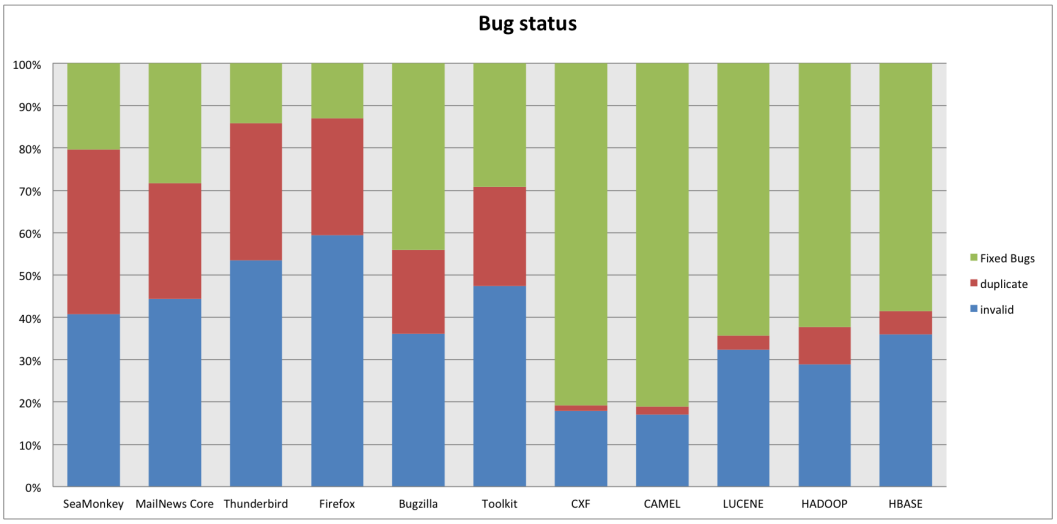


Figure 5.5. Projects bug status statistics

Project	Bugs	invalid	duplicate	Fixed Bugs
SeaMonkey	90,877	37,012	35,386	18,479
MailNews Core	29,092	12,916	7,922	8,254
CXF	4,948	888	63	3,997
Thunderbird	37,253	19,926	12,029	5,298
Firefox	120,468	71,547	33,335	15,586
Bugzilla	17,666	6,390	3,476	7,800
Toolkit	25,588	12,118	6,013	7,457
CAMEL	6,361	1,080	119	5,162
LUCENE	4,991	1,612	170	3,209
HADOOP	7,693	2,223	677	4,793
HBASE	8,565	3,084	461	5,020

Table 5.1. Projects bug status

At a first look at the figure 5.5 or in the table 5.1 seems that with Jira it is easier manage the work and to avoid duplicate bugs.

- The percentage of fixed bugs for Bugzilla is 24.8% while for Jira is 69.4%.
- The percentage of duplicate bugs for Bugzilla is 28.2% while for Jira is 4.1%.

A reason could be that the projects chosen with Jira are too specific and that the contributors have more expertise than the projects with Bugzilla.
If we take Hadoop for example, is a framework for distributed computing and the users of this

project are more likely to be developers than the users of Firefox.
In fact if we compare only the Bugzilla projects with each other we can see that the proportion of bug fixes decrease with the product specificity (i.e Thunderbird vs Toolkit).

An interesting fact is that the proportion of fixed bugs decrease with the increment of people. The more people are involved in the project, the less bugs are fixed.

5.1.3 Bug tossing

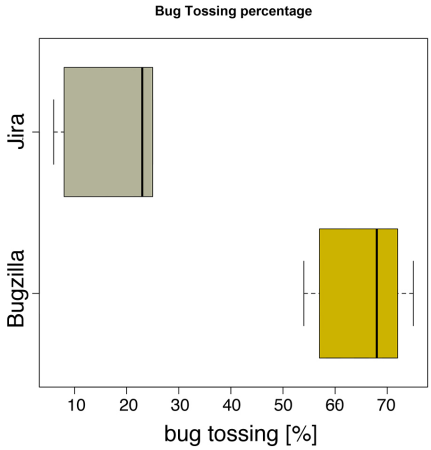


Figure 5.6. Bug-tracking system bug tossing statistics

In figure 5.6 we can see that around 65% of the Bugs in Mozilla projects are being tossed and 17% for Jira projects.
This number is quite high, compared to 44% found by Jeong et al. [2009]. For both Jira and Bugzilla we used the same Tossing model, and this could be the reason for this difference. Jeong et al. [2009] used a model of tossing different from ours. They consider a tossing event when a developer passes a bug to another developer. In our model we increment the tossing count when a developer not assigned to the bug submits a correct patch.
To explain the difference between the two systems shown in figure 5.6 or in table 5.1 we have tried to find a correlation between bug tossing and some other phenomena.

- Expertise measured as fixer/reporter ratio: Using the ratio of how many bugs were fixed by the same person who report the bug as a measure for contributors population expertise we haven't found any correlation with the bug tossing.
- Project size: There is some correlation with the project size, intuitively big project are more difficult to manage than smaller one.
- Number of contributors: Also in this case there is a correlation with the number of contributors, fewer people work better. In a large open source project more people come and leave. The knowledge on the entire project population is difficult and thus more prone to bad decisions that can lead to bug tossing events.

Projects	valid bugs	tossed bugs	tossed bugs [%]
SeaMonkey	18,479	9,889	54%
MailNews Core	7,767	4,407	57%
Thunderbird	4,704	2,993	64%
Firefox	12,796	9,634	75%
Bugzilla	7,228	5,205	72%
Toolkit	6,600	4,732	72%
CXF	3,132	175	6%
CAMEL	3,955	329	8%
LUCENE	2,446	614	25%
HADOOP	4,355	990	23%
HBASE	3,761	952	25%

Table 5.2. Projects bug tossing

We have not found a measurable evidence to explain the difference between the two systems. Our conjecture is that the better user interface and the simpler workflow of Jira decrease error possibilities thus leading to a decrease of the phenomena of bug tossing.

5.1.4 Developers activity

Project	developers	fixers	reporters	Fixer/Reporter proportions
Thunderbird	367	175	1,455	0.356
Toolkit	679	338	1,363	0.490
Bugzilla	432	178	1,198	0.446
MailNews Core	581	260	1,769	0.319
Firefox	907	482	2,950	0.387
SeaMonkey	969	461	3,274	0.267
CXF	467	37	962	0.328
CAMEL	419	40	677	0.508
HADOOP	356	208	372	0.630
HBASE	330	169	368	0.703
LUCENE	290	40	462	0.531

Table 5.3. Contributors statistics

As shown in figure 5.8 In Jira projects only 30% of contributors have actually fixed at least one bug. The situation in Bugzilla projects is worse, only 11% (figure 5.7). Those numbers indicate that a major fraction of people does not contribute directly to the project improvement. The large percentage of people are reporters in both groups, Jira and Bugzilla.
This measure can show the expertise of the contributors population. This ratio express how many people that submits and fix a bug report. In fact this practice is a sign that indicate that the person who have submitted the bug knows and how to fix it have expertise. In the table 5.3 we can see the projects specialization in more detail represented by the ratio Fixer/Reporter.
In fact we can see a significant difference between Jira projects and Bugzilla. The mean of the ratio for Bugzilla is 0.37 while the mean in Jira is 0.54.

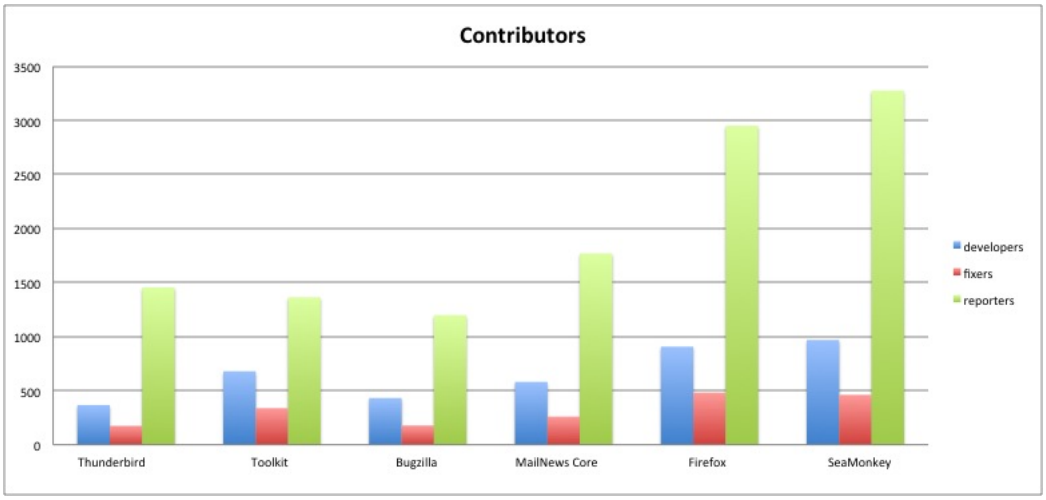


Figure 5.7. Contributors in Bugzilla projects

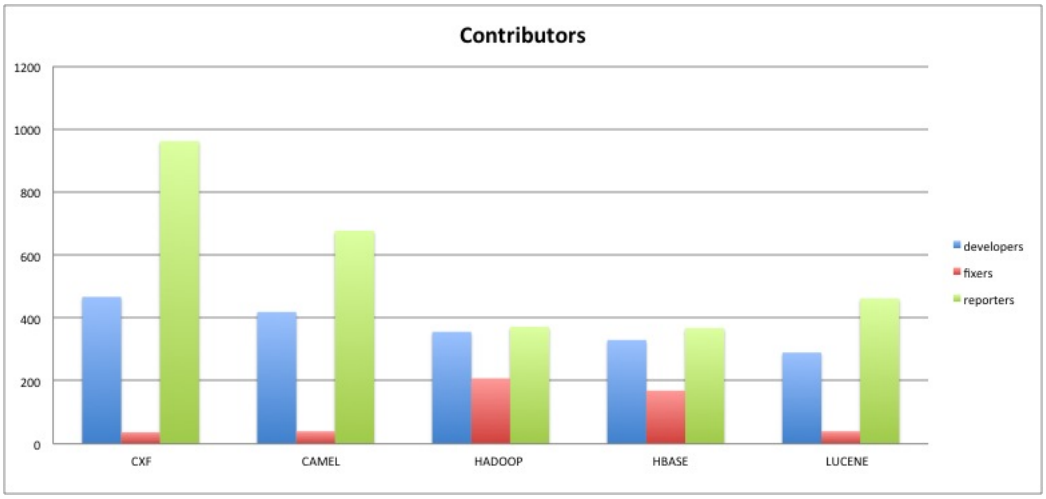


Figure 5.8. Contributors in Jira projects

5.2 Expert Recommender

In this section we discuss all the steps taken to build the expert recommender. We present the result and discuss the differences. In detail we discuss the pros and cons of using structured informations versus unstructured informations. We consider which is the best algorithm for classification and what filtering parameters and cross-validation perform best. In the last part of the chapter we show the tossing graphs application in our dataset and we show how they perform.

	Unstructured info		Structured info	
Projects	Accuracy	Time [s]	Accuracy	Time [s]
Thunderbird	42.58%	0.9	27.05%	138.4
Toolkit	36.14%	3.4	23.23%	746.8
Bugzilla	39.12%	1.7	33.83%	404.7
MailNews Core	40.67%	3.5	30.69%	604.5
Firefox	37.38%	24.1	22.78%	3,777.7
SeaMonkey	51.88%	44.5	35.41%	6,399.9
CXF	63.52%	0.5	62.00%	55.7
CAMEL	60.59%	0.5	49.62%	65.9
HADOOP	43.16%	1.5	25.46%	198.9
HBASE	58.28%	1.2	22.45%	188.5
LUCENE	65.83%	0.1	40.21%	50.0

Table 5.4. Structured information vs unstructured information comparizion with respect to traing time and accuracy of the classifier

5.2.1 Feature vs text categorization

In all the approaches we discussed in the state of the art section there is a common pattern. The use of text categorization or some Information Retrieval technique to incorporate bug description or title in the feature vector of the classifiers. In our experiment we show that using this information does not increment the quality of the classification and results in a large increment in classifier training time.

To compare the two techniques we used the following settings:

1. Text Categorization with unstructured information
    - Features: Component, Bug Creation Date, Priority, Severity, Platform and title and description converted with String to Vector.
    - String to Vector (WEKA) settings: IDF transform and Stop list enabled
    - Classifier Naive Bayes
2. Only structured information
    - Features: Component, Bug Creation Date, Priority, Severity, Platform
    - Classifier Naive Bayes

All the comparisons were made using 9/10 dataset split, 90% of the data for training and 10% for testing.

In figure 5.9 we see that there is no gain in accuracy using unstructured information with respect to the classifier with only structured information.
If we observe the training time column in table 5.4 we see that the time increases up to 18 times. This suggests that the approach of using only unstructured information for our dataset is better.

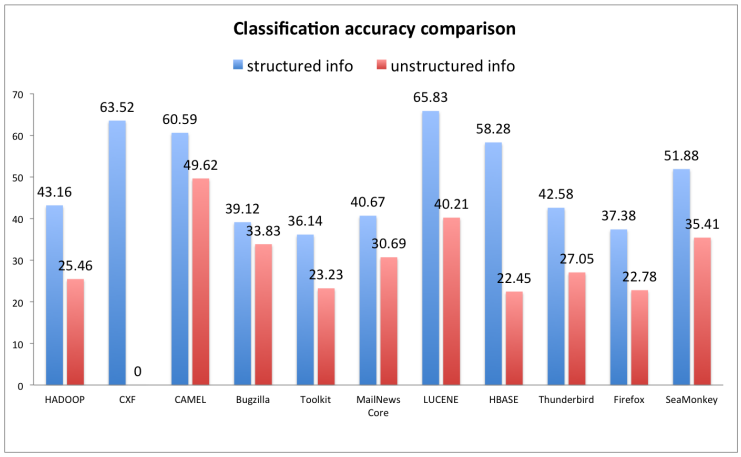


Figure 5.9. Comparison using unstructured information or structured information

5.2.2 Classifier algorithms

In our experiment we used three classifiers, Naive Bayes, C4.5 and Support Vector Machines. The difference reported from others Bhattacharya and Neamtiu [2010] were not significant. In our case (table 5.5) Naive Bayes and C4.5 outperformed SVM. This probably due to the features selected. In any case the Support Vector Machines were slow to train compared to the other so we discarded them.

To compare the algorithms we used the following settings:

1. Features: Component, Bug Creation Date, Priority, Severity, Platform
2. Cross-validation: inter-folding

		Accuracy		
Project	Dataset size	Naive Bayes	C4.5	SVM(RBF)
Thunderbird	4,202	71.66%	67.69%	58.89%
Toolkit	5,751	60.69%	56.22%	23.50%
Bugzilla	6,738	69.07%	58.32%	47.82%
MailNews Core	7,011	67.05%	61.35%	46.61%
Firefox	11,470	49.99%	41.23%	16.37%
SeaMonkey	16,393	54.90%	47.77%	16.32%
CXF	3,074	87.28%	77.98%	71.23%
CAMEL	3,920	87.81%	85.98%	63.06%
HADOOP	3,811	54.28%	60.75%	21.68%
HBASE	3,308	60.93%	69.21%	34.98%
LUCENE	2,371	81.48%	73.95%	53.32%

Table 5.5. Classification algorithm comparison

Looking at the table 5.6 the difference between C4.5 and Naive Bayes is significant. Overall Naive Bayes performs better than C4.5 and the overhead in training time is not too much.

		C.45		Naive Bayes	
Project	Dataset size	F-score	training time	F-score	training time
Thunderbird	4,202	0.63	3.5	0.66	7.7
Toolkit	5,751	0.54	4.2	0.53	54.3
Bugzilla	6,738	0.56	4.5	0.63	23.7
MailNews Core	7,011	0.59	6.5	0.60	39.3
Firefox	11,470	0.42	31.7	0.43	276.4
SeaMonkey	16,393	0.46	52.3	0.49	492.7
CXF	3,074	0.75	0.7	0.85	1.5
CAMEL	3,920	0.83	1.1	0.84	2.9
HADOOP	3,811	0.59	1.5	0.48	16.5
HBASE	3,308	0.65	1.0	0.53	8.9
LUCENE	2,371	0.72	0.4	0.78	1.0

Table 5.6. F-score with J48 and Naive Bayes

5.2.3 Developer activity filter

The first refinement in our steps to the final recommender is the Developer Activity filter. We filter all the bugs that does not belongs to the set of active developers. Active developers are those who have fixed at least k bugs. In this experiment we measured the accuracy of the classifier C4.5 with different k (1,5,10,15).In this dataset the right balance between number of developers(that have fixed bugs) and threshold is 10, where we have enough developers to train the classifier and good accuracy.

To compare the filtering impact we used the following settings:

1. Features: Component, Bug Creation Date, Priority, Severity, Platform
2. Cross-validation: inter-folding

Project	no filter	5 fixes	10 fixes	15 fixes
Thunderbird	60.760%	65.254%	67.795%	69.930%
Toolkit	51.207%	56.076%	56.994%	60.965%
Bugzilla	56.884%	58.758%	59.567%	60.707%
MailNews Core	55.091%	59.028%	60.923%	63.352%
Firefox	38.420%	41.229%	42.784%	42.598%
SeaMonkey	44.978%	47.054%	48.010%	49.510%
CXF	77.395%	78.041%	78.847%	78.642%
CAMEL	85.707%	86.131%	86.307%	87.363%
HADOOP	55.762%	59.821%	62.221%	62.287%
HBASE	63.536%	66.817%	71.082%	73.533%
LUCENE	72.933%	74.305%	75.119%	76.358%

Table 5.7. Accuracy with bug fix filtering

As we see in the table 5.7, increasing the threshold of fixed bugs per developer, increases the accuracy of the classifier. This is due to the fact that increasing this number we remove

Project	no filter	5 fixes	10 fixes	15 fixes
Thunderbird	174	76	46	35
Toolkit	332	147	107	77
Bugzilla	178	91	60	51
MailNews Core	257	131	80	57
Firefox	475	238	161	131
SeaMonkey	458	246	179	150
CXF	32	25	21	18
CAMEL	38	30	28	22
HADOOP	206	115	78	54
HBASE	169	85	58	49
LUCENE	36	27	21	19

Table 5.8. Developers population size with bug fix filtering

all the developers that does not have sufficient data to be a valid candidate in the prediction of the recommender. The downside of filtering is that we reduce the population of developers that we use to train the classifier, see table 5.8.

5.2.4 Top-k accuracy

As presented in Anvik et al. [2006], it is possible to propose a list of developers to assist the triager in the decision. The triager insted of having to look all the population of developers can focus on the list provided by the recommender. In this way the time to process a bug it is reduced a lot. Comparing the results in the table 5.9 or looking at the figure 5.10 we can see that the accuracy of the classifiers increase by around 8% from the top-1 to the top-10 classification technique.

Those results are consistent with the previous research papers that indicate an increment of approximatively 10%.

Project	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10
Thunderbird	41.78%	54.92%	60.52%	63.63%	67.69%	69.17%	70.00%	71.43%	72.46%	73.29%
Toolkit	37.30%	48.65%	52.36%	54.18%	56.22%	57.58%	58.81%	59.25%	60.19%	60.62%
Bugzilla	44.64%	51.77%	54.30%	56.49%	58.32%	59.92%	60.92%	61.58%	62.38%	62.66%
MailNews Core	39.66%	48.25%	53.40%	59.04%	61.35%	63.73%	65.71%	67.10%	68.72%	69.22%
Firefox	33.09%	36.93%	39.21%	40.45%	41.23%	42.49%	43.24%	43.52%	44.38%	44.83%
SeaMonkey	30.54%	38.46%	43.39%	45.26%	47.77%	49.46%	51.10%	52.01%	53.02%	54.02%
CXF	61.31%	66.07%	70.99%	73.57%	77.98%	78.53%	79.11%	80.24%	81.26%	82.12%
CAMEL	67.58%	80.01%	83.96%	85.18%	85.98%	86.78%	87.08%	87.57%	87.69%	88.15%
HADOO	52.41%	55.87%	57.88%	59.30%	60.75%	61.75%	62.51%	62.98%	63.73%	64.55%
HBASE	59.75%	64.75%	65.51%	67.51%	69.21%	70.51%	70.91%	71.20%	71.56%	73.05%
LUCENE	55.08%	64.08%	68.53%	71.78%	73.95%	76.43%	78.50%	79.15%	80.37%	80.72%

Table 5.9. Accuracy with J48 with different k-classification

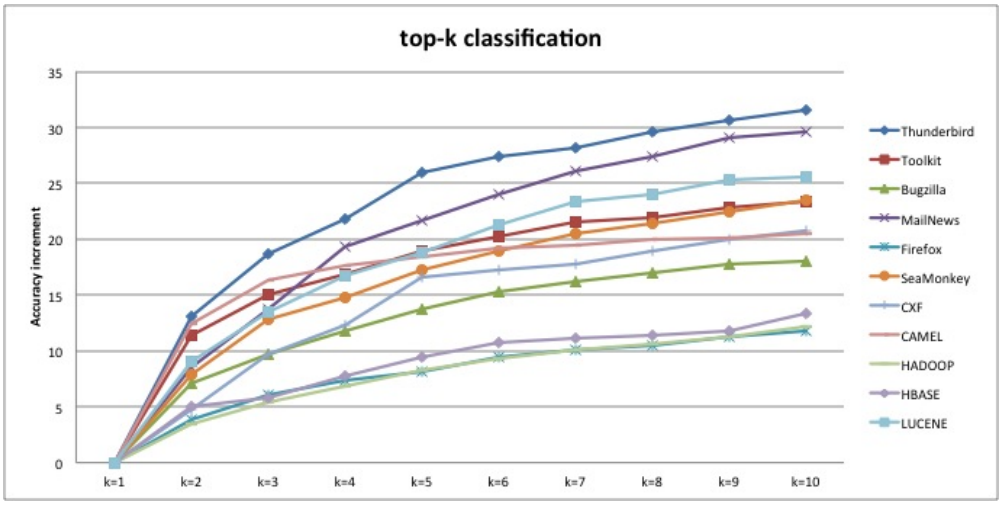


Figure 5.10. Accuracy increment with J48 with different k-classification

5.2.5 Cross validation techniques

In the table 5.10 or in figure 5.11 we present the results of the classification with two different folding techniques, inter-folding and intra-folding. Looking at the data the increment of accuracy with the intra-folding approach is significant. Around 8% of increment but the downside of this approach is the increment of training time that grows enormously, shown in figure 5.12. If the time is not the issue it is the way to proceed, since with intra-folding the situation it is close to reality where whenever a new bugs comes in, the classifier is updated with all the previously resolved bugs.

Project	Accuracy with inter-folding [%]	Training time with inter-folding [s]	Accuracy with intra-folding [%]	Training time with intra-folding [s]
Thunderbird	67.69%	31.00	72.55%	297.54
Toolkit	56.22%	4.12	68.22%	2089.44
Bugzilla	58.32%	4.34	68.77%	1280.05
MailNews Core	61.35%	6.01	69.65%	3173.20
Firefox	41.23%	31.03	54.41%	11217.11
SeaMonkey	47.77%	47.06	57.87%	5372.22
CXF	77.98%	2.08	81.89%	109.56
CAMEL	85.98%	1.08	86.47%	177.24
HADOOP	60.75%	1.58	70.22%	204.34
HBASE	69.21%	1.03	78.75%	101.87
LUCENE	73.95%	0.40	81.54%	37.94

Table 5.10. Accuracy with J48 with different folding techniques

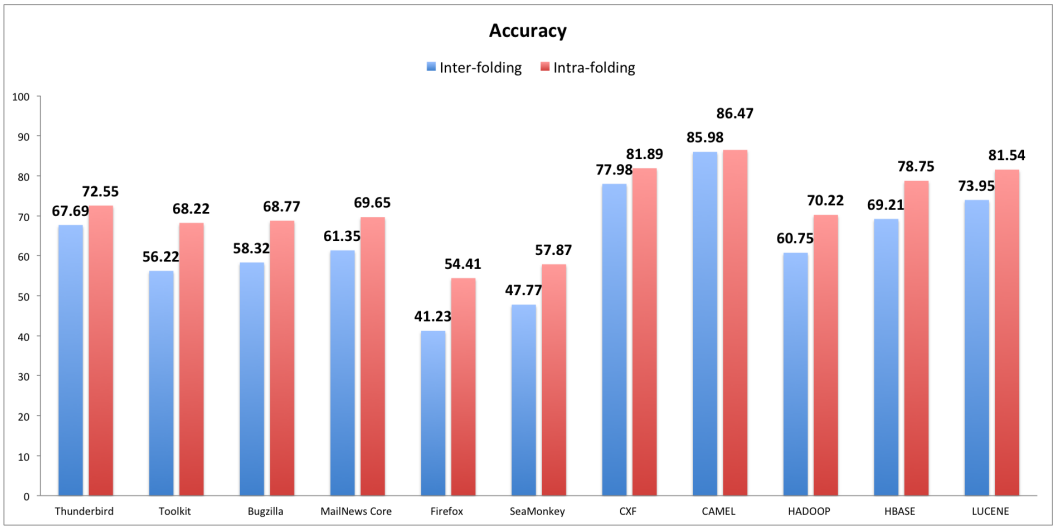


Figure 5.11. Accuracy with different folding techniques

Figure 5.12. Training time with different folding techniques

5.2.6 Tossing graphs

As indicated in the studies of Bhattacharya and Neamtiu [2010] and Jeong et al. [2009] the use of tossing graph can improve the accuracy of the recommender. In our experiment this was not the case. The use of tossing graph have not added any extra information to our recommender.

The possible reasons are:

- Filtered dataset: With a filtered dataset the tossing graph are to small and too sparse to give extra informations. Reducing the dataset to achieve high accuracy with the classifier have brought a problem in the extraction of tossing graphs
- The ranking function presented in Bhattacharya and Neamtiu [2010] could be unbalanced in our dataset.

To investigate better we developed a tool to explore the tossing graph and to fine tune the ranking function. In this tool it is possible to filter and classify bug instances and see all the tossing graphs for each developers in the predicted set.

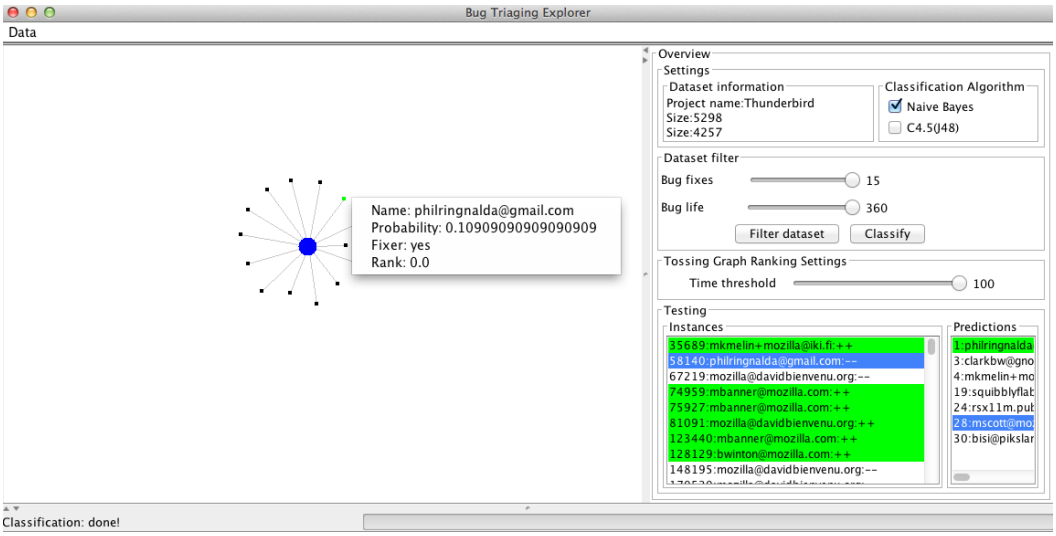


Figure 5.13. Screenshot of the Bug Tossing Explorer tool

In figure 5.13 is shown the main view of the application. In the upper left corner is possible to load a list of bugs in JSON format or crawl a Jira repository saving it to a JSON file.

The application is dived into two parts:

1. A graph viewer where are represented the tossing graphs of the developers selected
2. A list of settings to filter and classify the dataset loaded.

The main settings (Figure 5.14) present some informations about the data and two checkboxes to select the machine learning algorithm used in the classification step. The two algorithm are Naive Bayes, a probabilistic algorithm and C4.5 a decision tree. The second section is the filtering settings. In this part is possible to filter the data and classify the dataset (Figure 5.15).

The last settings present in the tool is the Tossing Graph Ranking Settings (Figure 5.16). In this are is possible to set the threshold of the formula 5.2.

$$R(D_k) = Pr(D_i \longrightarrow D_k) + newMatchedComponent(D_k) \tag{5.1}$$

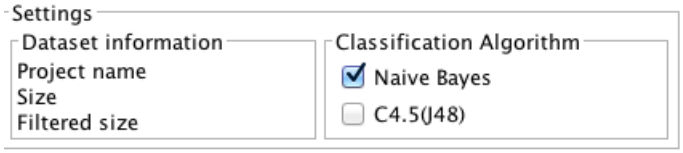


Figure 5.14. Screenshot of the Bug Tossing Explorer main settings

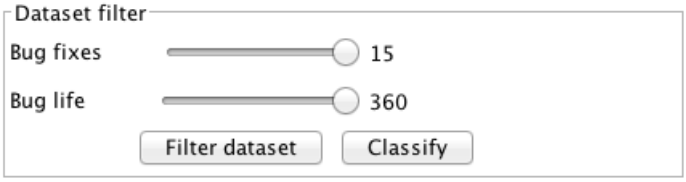


Figure 5.15. Screenshot of the Bug Tossing Explorer dataset filtering settings

Figure 5.16. Screenshot of the Bug Tossing Explorer tool tossing graph settings

$$newMatchedComponent(D_k) = MatchedComponent(D_k) * \frac{log(lastActivity)}{log(k)+1} \tag{5.2}$$

The last part is composed by two list. The first list on the left, are the instances of the test dataset. The right part are the prediction on the instance selected. This prediction shows all the developers that have a tossing event, and when a developer of that list is selected, the application present his tossing graph.

The tossing graph is shown as a connected graph with the nodes in three color:

- blue: The origin developer, who own the tossing graph
- black: all the developers who are the target of a tossing event from the origin
- green: the developer who is the target of the tossing event and that is the fixer of the selected issue (left list).

Every node is selectable and shows a popup window with the main informations; name, tossing probability, is the fixer (yes/no) and the ranking value.

5.3 Summary

In this chapter we saw how we can build an expert recommender. We have shown all the intermediate steps to achieve a good prediction accuracy. The part regarding the tossing graph unfortunately was not as we expected. Other studies have shown that augmenting the recommender with tossing graph can improve the accuracy, but this was not the case. We think that if we can fine tune the parameters of the tossing graph we can improve the results so we

developed the tool presented earlier to explore those kind of abstractions.

We reached an accuracy of 63% to 70% for a top five recommendation, and we think that these values are high enough to lower the effort of a triager. Without the assistance of a recommender the triager should look all the developers in the project to find the actual expert. With this approach, the triager have a restricted list that contains most of the time the real bug fixer.

Looking at the bug tossing data presented earlier, we see that in the projects we analysed the error rate is high, in fact the 65% of bug tossing in Mozilla projects means that 6 of 10 times the trigger make a mistake. If we use a recommender with an accuracy of 63%, this means that the bug tossing rate should reach approximatively a value of 37% and this traduce to a reduction of tossed bugs by more than 50%. We see the bug tossing using our approach in figure 5.17. For projects that uses Jira our approach does not lead to a bug tossing reduction. This is due to a low bug tossing rate of the projects and to reduce this number the expert recommender need to reach an accuracy of approximatively 90%.

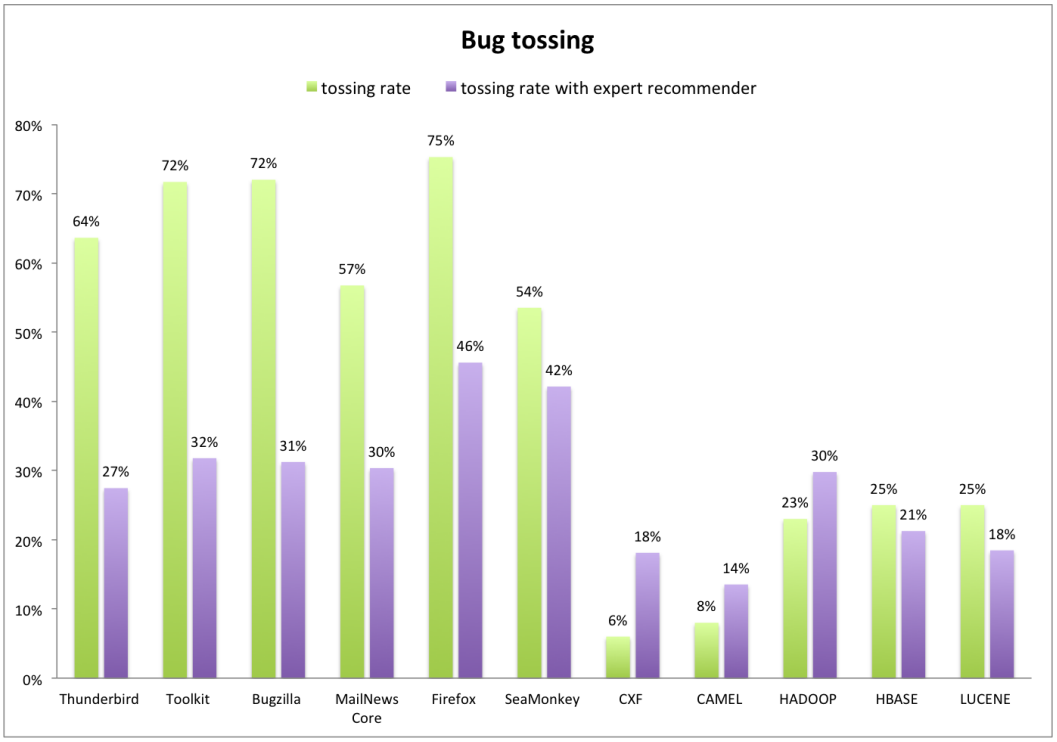


Figure 5.17. Bug tossing with our expert recommender system

Chapter 6

Conclusions

In this chapter we conclude our work summarising the overall thesis and indicating some pointers for future work.

6.1 Summary

Resource and time are limited in Software Engineering. When an issue is found in a software product *Bug Triaging* is used to assess the work and address the most suitable person to resolve the bug. Nowadays bug are being submitted at a fast pace and human and manual triaging become more and more difficult. Every time a bug is submitted the work to analyze it and find a valid developer require a lot of time. The research effort to solve this problem is to find a way to automatize the process of bug triaging.

Our goal was to explore the possible solutions available and try to improve it. To achieve our intent we developed a framework. Whit this framework we were able to test all the algorithms and techniques proposed by different researchers. In our framework we have developed three parts : A Bugzilla/Jira repository mining tool, a machine learning toolset and a bug tossing graph explorer.

During our analysis we constructed a dataset that is publicly available and that we investigate using descriptive statistics and visualization. This dataset is comprised of two groups of projects one group for each bug-tracking system, Jira and Bugzilla. We wanted to generalize the techniques without depending on a single bug-traking system.

To summarize our contributions:

- Bug meta-model : We described a Bug meta-model used to mine more generally Bug-Tracking system in the chapter 3. This model is used in our framework to mine software repositories (Jira and Bugzilla).
- Bug Tossing formal model: We created a more formal model to measure Bug Tossing events, since in the research field wasn't present any model to extract Bug Tossing effectively and precisely.

- An exhaustive review of the techniques used in research field to automatize bug triaging: In this thesis we explored all the possibility proposed in different research paper trying to achieve the same results implementing all the algorithm and techniques in our research framework.
- An extension of the state of the art technique: Bug Tossing graph are quite new to the research field, they where proposed by Jeong et al. [2009] in 2009. We believe that those graphs can improve the knowledge on the system, and since in our dataset we failed to replicate the results of Bhattacharya and Neamtiu [2010] we developed a tool to investigate and explore all the possibility we can achieve with those kind of abstractions. In this tool it is possible to "navigate" through the tossing graph of the developers tuning it is parameters to rank better in the case of developer assignment recommender.

6.2 Threats to validity

We identify the following threats to validity

- **Labelling the reports.** In bug reports, not always it is possible automatically to detect the actual fixer. To do that, we employ some heuristics similar to Anvik et al. [2006], that could be inaccurate for some projects. Moreover open source system like Bugzilla allow user to have aliases for the same account thus making more difficult the identification of a single developer.
- **Systems examined might not be representative.** Bug reports of two bug-tracking system were examined in this thesis. Since we wanted to compare open source project, we were able to find more general projects with one system (Bugzilla). Jira is a commercial product thus most open source project chose to use Bugzilla for that reason. Moreover, using Apache projects, where most of the product are frameworks or product where the client is a developer maybe we have introduced a project selection bias.

6.3 Future Work

Bug Triaging it is a rich process that have a lot of delicate parts that could be improved. We showed how to deal with labeled reports to construct a recommender. But we do not have explored deeply the process of extracting the data from repositories to label effectively and accurately the reports. There is email aliasing and wrong use of activity field that worth investigating more.

Another possible improvement of our work is to develop a more featured web application based on the Bug Tossing Explorer we developed in our framework. With this kind of tool will be possible to really explore more graphically and manually all the activity that are present in a bug tracker and extending Jira or Bugzilla with this kind of visualization could improve the triage activity.

Appendix A

Mining Bug-tracking systems: Jira and Bugzilla

In this chapter we show some problems and solutions for mining two bug-tracking systems, Bugzilla and Jira.

A.1 Bugzilla

Bugzilla is an old bug-tracker and does not provide good API or webservice that can be used to mine the repository. To tackle this problem is required to crawl the Bugzilla webpages using the advanced search feature and get the results in CSV format.
In this way we are able to collect all the major information about each issue automatically but we needed to crawl multiple pages since Bugzilla limit the number of search results.

To overcome this problem we had to filter the search result with bug creation date and crawl the complete project history by parts. Where each parts was 1 or 2 months of developing. To extract the list of bug and some major information the url to use is:

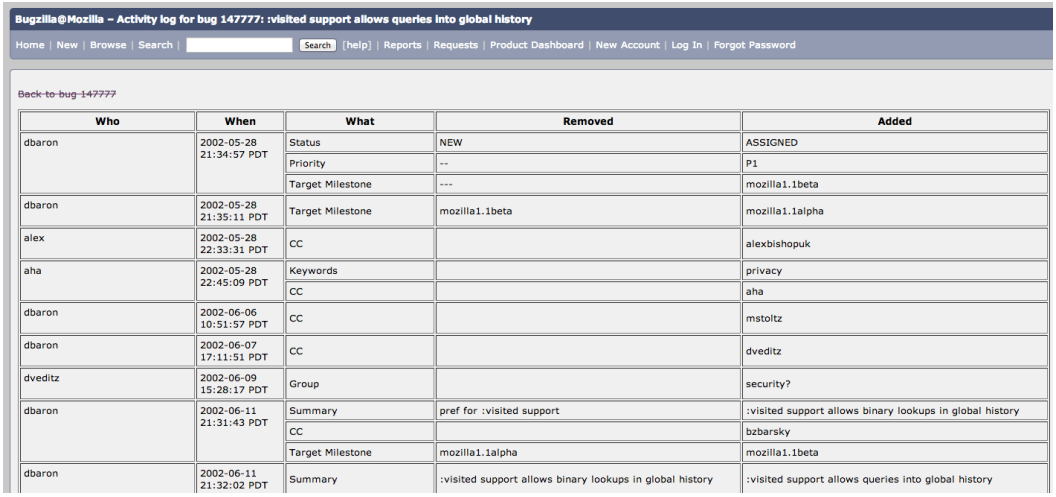
```
https://bugzilla.mozilla.org/buglist.cgi?chfield=%5BBug%20creation%5D&chfieldfrom=2013-03-01&chfieldto=
2013-05-01&product=SeaMonkey&query_format=advanced&resolution=---&ctype=csv
```

This url is composed of a couple of variables:

- product: Indicate the name of the software product
- component: Is the name of the component, if obmitted the results fetch all bugs
- chfield: This represent the name of the field used for filtering the results, in our case it was always "Bug Creation" that is the date in which the bug was created.
- chfieldfrom and chfieldto: Those are the values of the filtering field that represent in our case the starting and ending date.

Unfortunately the information collected was not enough. For our work we needed also the activity history or change-log for each bug.

To mine this kind of information we needed to parse a Bugzilla webpage result reachable



Bugzilla@Mozilla – Activity log for bug 147777: :visited support allows queries into global history

Home | New | Browse | Search | Search [help] | Reports | Requests | Product Dashboard | New Account | Log In | Forgot Password

Back to bug 147777

Who	When	What	Removed	Added
dbaron	2002-05-28 21:34:57 PDT	Status	NEW	ASSIGNED
		Priority	--	P1
		Target Milestone	---	mozilla1.1beta
dbaron	2002-05-28 21:35:11 PDT	Target Milestone	mozilla1.1beta	mozilla1.1alpha
alex	2002-05-28 22:33:31 PDT	CC		alexbishopuk
aha	2002-05-28 22:45:09 PDT	Keywords		privacy
		CC		aha
dbaron	2002-06-06 10:51:57 PDT	CC		mstoltz
dbaron	2002-06-07 17:11:51 PDT	CC		dveditz
dveditz	2002-06-09 15:28:17 PDT	Group		security?
dbaron	2002-06-11 21:31:43 PDT	Summary	pref for :visited support	:visited support allows binary lookups in global history
		CC		bzbarsky
		Target Milestone	mozilla1.1alpha	mozilla1.1beta
dbaron	2002-06-11 21:32:02 PDT	Summary	:visited support allows binary lookups in global history	:visited support allows queries into global history

Figure A.1. Bugzilla changelog wepage

at:

```
https://bugzilla.mozilla.org/show_activity.cgi?id=147777
```

Where the number after the id is the identification number of the bug.

A.2 Jira

Since Jira is much newer product than Bugzilla, it offers a comfortable way of mining data the jira query language and can fetch JSON results.

A.2.1 Jira jql language

The Jira query language, similar to sql, is a language that permit operation on the Jira Bugtracker. For our purposes we use only two query:

1. Get list of bugs:
   `https://issues.apache.org/jira/rest/api/2/search?jql=project=LUCENE+order+by+created+asc&fields=id,key,resolution,created&startAt=0`

   The only information we needed to change were the name of the project (e.g LUCENE) and startAt that is the offset of the results since also jira limit the search result size, but indicate the max result size at the response.

2. Get a bug extended information :
   `https://issues.apache.org/jira/rest/api/2/issue/LUCENE-9219/?expand=changelog`

A.3 Parsing JSON data with GSON

Gson is a Java library under Apache Lincense that allow to convert Java Object into JSON representation and in the other way around. The goal of this library are :

- Provide simple toJson() and fromJson() methods to convert Java objects to JSON and vice-versa
- Allow pre-existing unmodifiable objects to be converted to and from JSON
- Extensive support of Java Generics
- Allow custom representations for objects
- Support arbitrarily complex objects (with deep inheritance hierarchies and extensive use of generic types)

In our case we were faced with the need to interpret JSON results from the Jira RESTful webservice. To reach our goal we decided to use GSON. In the following portion of code we outline the way we can read a JSON string into a Java object.

```
String jsonContent=json.toString();
//Reading a json file to a defined class (i.e IssueList)
GsonBuilder gson = new GsonBuilder();
list=gson.create().fromJson(jsonContent, IssueList.class);
```

When we are able to model the JSON into a class object we can use the code illustrated above. But if we do not know the format of the data we can read anyway the data using a generic Map.

```
String jsonContent=json.toString();
//Reading a JSON file with undefined class
Map<String, Object> map = new Gson().fromJson(jsonContent,
        new TypeToken<Map<String, Object>>() {}.getType());
```

To save the Java Object into a JSON format follow the same reasoning. We need to give a Type object to the method toJson in order to transform correctly the object into a JSON string.

```
Type IssueListType = new TypeToken<IssueList>() {}.getType();
String json = new Gson().toJson(object,IssueListType);
```

A.3.1 Common issues

Reading and saving JSON data is relatively simple with the GSON library, but if we want to do some generic programming there are some issues. Since Java implement generic programming with type erasure, GSON does not know the type of the object during the conversion. To tackle this problem there are some ways.

The common ways is to write a custom serializer and deserializer and a type adapter. In this way GSON know how to proceed in the case encounter a special type of object and does not know how to pars it.

More information about this kind of practice on the GSON page:

`https://sites.google.com/site/gson/gson-user-guide#TOC-Collections-Examples`

Bibliography

John Anvik and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol.*, 20(3):10:1–10:35, August 2011.

John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, eclipse '05, pages 35–39, New York, NY, USA, 2005. ACM. ISBN 1-59593-342-5.

John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1.

Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008a. ACM.

Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? 2008b.

Pamela Bhattacharya and Iulian Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-8630-4.

Gerardo Canfora and Luigi Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 105–111, New York, NY, USA, 2006. ACM. ISBN 1-59593-397-2.

Davor Cubranic and Gail C. Murphy. Automatic bug triage using text categorization. In *SEKE*, pages 92–97, 2004.

Marco D'Ambros and Michele Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 229–238, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2536-9.

Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, August 2012. ISSN 1382-3256.

Pedro Domingos and Michael Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *Machine Learning*, pages 105–112. Morgan Kaufmann, 1996.

Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.

M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002. ISSN 1049-331X.

Marc J. Rochkind. The source code control system. *IEEE Trans. Softw. Eng.*, 1(1):364–370, March 1975. ISSN 0098-5589.

Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. Famix and xmi. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 296–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0881-2.