The Report Committee for Christopher John Arges
Certifies that this is the approved version of the following report:

Data-Mining The Ubuntu Linux Distribution for Bug Analysis and Resolution

APPROVED BY

SUPERVISING COMMITTEE:

Joydeep Ghosh, Supervisor

Kate Stewart, Co-supervisor

Data-Mining The Ubuntu Linux Distribution for Bug Analysis and Resolution

by

Christopher John Arges, B.S.C.E.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2012

This report is dedicated to my wife and best friend, Allie.

Acknowledgments

I would like to first thank my wife Allie for understanding the time commitment that this report took. I would like to give a big thank you to Dr. Ghosh for supervising this research and providing advice on scoping the project. A special thank you goes to Kate Stewart for her feedback that helped form the topic that I eventually chose, and her comments that shaped the final report. In addition, I would like to thank Ursula Junque for her help with some of the database work in collecting data from Launchpad. Another thank you goes to José Plans for originally getting me interested in bug data metrics. In addition, I thank all the other co-workers that provided feedback and input. Finally, to all my friends and family who had to constantly listen to me talk about this research paper, I am thankful and blessed to have your support.

Data-Mining The Ubuntu Linux Distribution for Bug Analysis and Resolution

Christopher John Arges, M.S.E.
The University of Texas at Austin, 2012

Supervisor: Joydeep Ghosh
Co-supervisor: Kate Stewart

The Ubuntu Linux Distribution represents a massive investment of time and human effort to produce a reliable computing experience for users. To accomplish these goals, software bugs must be tracked and fixed. However, as the number of users increase and bug reports grow advanced tools such as data mining must be used to increase the effectiveness of all contributors to the project. Thus, this report involved collecting a large amount of bug reports into a database and calculating relevant statistics. Because of the diversity and quantity of bug reports, contributors must find which bugs are most relevant and important to work on. One study in this report created an automatic way to determine who is best fit to solve a particular bug by using classification techniques. In addition, this report explores how to initially classify if a bug report will be eventually marked invalid or not.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the age of computers and the Internet, one finds themselves swimming in a sea of data. This sea can be unyielding and vast, yet there are some that dare to venture into the unknown armed only with a small fishing boat.

Cheap storage, robust databases, fast networks, and smart data collection lead to a great hoard of information that is waiting to be turned into knowledge. Specifically, in the world of software development there are vast repositories to track software defects, code revisions, and project progress. These resources are essential in modern development, yet they require extensive filtering to achieve the correct 'view' for the user. In addition, these limited views only provide a glimpse of the system and it becomes difficult to see the bigger picture. To address these issues, many software engineering tools now provide statistics, graphs, and other functions to see more of this data. This visualization is critical in understanding the patterns that exist in software development to improve the overall software engineering experience.

Software development practices and processes range wildly in different organizations. One measure of process maturity is the 'Capability Maturity Model' [30]. An 'Optimizing' level of maturity is defined by continually im-

proving process performance, while also implementing a repeatable, defined, and managed process. This is the goal of an organization to achieve maturity in the CMM model. Data mining the vast resources of bug information can help achieve this level of maturity by not only better understanding the current trends, but predicting future trends and becoming more productive and less reactive. Because most organizations have very limited resources they must continually determine how to properly allocate resources to achieve their goals.

Most software has bugs, and intuitively as the amount of code increases in a given system we expect the probability of bugs to increase. However, given there are limited resources, how does one find which bugs are the most critical to fix? In addition, how does one plan for the future and predict where more testing is required, or perhaps more attention to fixing bugs. How does one track the quality of a given product? Who is best suited for a particular bug? How can one predict a good bug report and an invalid bug report? These are the types of questions that need to be answered to improve the overall engineering experience and make software of a higher caliber.

Chapter 2

Background

While applying data mining to software engineering is fairly broad and evident for many projects, this paper will only examine an open source Linux distribution called Ubuntu. Therefore, a background in how these technologies work is critical to understanding the overall paper and motivation of the research. First, the definitions of open source software, and how it relates to the Ubuntu project will be explored. Next, an explanation of the challenges and processes involved in a Linux distribution will be discussed. In addition, bug tracking systems will be explained to give an overview of the type of data that is explored in this paper. Finally, the specific system called 'Launchpad' will be explained in detail to give the data meaning and structure.

2.1 Open Source Software and Development Model

While many definitions exist for 'Open', 'Free' or 'Libre' software, overall they share some common characteristics in how they are distributed and developed. Because Ubuntu is a distribution, it consists of many pieces of software which have different licenses. In addition, development philosophies which are embodied in a software license may vary from project to project,

but also share some common characteristics. It's important to understand a few of these philosophies to give a basis of these licenses that in turn guide the development of open source projects.

One organization called the Free Software Foundation (FSF) classifies Free Software as the freedom to run the program for any purpose, the freedom to study and modify a program, the freedom to redistribute copies, and the freedom to distribute modified copies [29]. In addition, the Open Source Definition (OSD) was created by the Open Source Initiative (OSI) organization to give some scope on what constitutes open-source software [27]. It defines 10 principles of open-source software: software must be free to redistribute, the program must include source code, the license must allow people to experiment and redistribute, users have a right to know who has made the program, there should be no discrimination against any person or group, the license cannot restrict anyone from making use of a program in a specific field, there should be no additional license to redistribute the program, the license must not be tied to a specific product, the license must not restrict other software, the license must be technology-neutral [27]. Finally, the Ubuntu project has its own specific beliefs which mirror many of these beliefs as well. This includes that every computer user should have the freedom to download, run, copy, distribute, study, share, change and improve the software for any purpose without paying additional licensing fees. In addition the software should be able to be used in the language of the users choice, the software should be accessible for all users, and that software included in the distribution meets this philosophy

[1]. While there are subtle differences between these definitions, it is clear that the importance is not on cost, but on freedom of use of the software. These philosophies are generally implemented in the more concrete concept of an Open Source or Free software license.

Software licenses can be either free, copyleft or closed. Free licenses allow for the source to be redistributed and modified at will. Closed licenses can be extremely varied and for the sake of this paper are considered not Open Source or Free licenses. Copyleft is a license that compels the user to redistribute the source code for any derivative works of that software [32]. Because of these types of licenses, Linux distributions are made possible and source code changes are freely shared and explored. Examples of Open Source licenses include the GNU Public License (GPL), the BSD license, and the Artistic License. The Linux kernel is licensed under the GPLv2 license and as such any hardware or software that is released using Linux must distribute any Linux source code modifications for that project. For example, the Ubuntu distribution must make any additional Linux software code patches available, and generally contribute these back to the original project. Another example is the Android operating system which runs using Linux. Any additional code used to enable those devices must also be made available. This model ensures that standard code is shared back into the Linux kernel instead of being fragmented and hoarded. Overall, these licenses and examples give a glimpse of the Open Source ecosystem and development model.

Open Source projects are diverse in their development models and thus

it's difficult to give a singular definition of all projects; however, because of the aforementioned philosophies there are certain patterns that typically exist. Developers can be located in a single office or across the world, providing additional diversity and increasing the need for clear and open communications. One analogy described in the book "The Cathedral and the Bazaar" [28] compares Closed Source development with a cathedral while Open Source development is like a bazaar or open marketplace. Essentially, the bazaar method presents a more "flat" model wherein anyone can contribute or suggest changes to the project. In practice, many projects start because a developer had "an itch to scratch" [28], meaning they were deeply interested in solving a particular problem and just started writing code. However, this is the critical phase in software development where good initial design and planning can result in a successful, long term project that the bazaar flocks to.

A paper written by G. Madey, V. Freeh, and R. Tynan, [25] gives a rough structure of the development process of a typical Open Source project. The average Open Source project will have a maintainer or a team of maintainers, which are responsible for the master code base and incorporating changes. Projects will typically have a mailing list, forum or an Internet Relay Chat (IRC) channel in which communication occurs between persons interested in the project. These people interact iteratively between a few phases. The initial "problem discovery" phase occurs when users and developers communicate issues on public forums and share ideas about a new problem. Next, volunteers that share an interest in the problem are found to help work to-

gether and plan. Together the volunteers discuss a solution and agree on an action. Next, code development and testing start. This can be done by any developer, but usually each developer will test their code before contributing it back on a mailing list or source code management system. The source code or patch is then reviewed by other developers in a common forum before its inclusion in the master code base. This patch can be accepted or denied with feedback until the community or maintainer of the project is satisfied with the quality of the code. Generally the attribution of the code changes goes to the originator, and most modern source code management systems include the author's name and email. Finally, a group of members of the project or the maintainer will periodically release a version of the code known to be "stable". This stable version can then be developed further and maintained [25]. While this provides a simplistic view, other contributions besides code can be made to a project. A person wanting to either join or start a project could also contribute documentation, translations, sounds, pictures, or bug reports which all can help that particular project [32].

Now that the basic idea of Open Source software is described, it is important to understand specifically how projects such as Linux and how distributions of Linux are built by a community of people.

2.2 The Linux Distribution

To understand the ecosystem of a Linux distribution, one must understand the components and processes involved. First, Linux must be explained

because it provides the base operating system. Then the other components and processes of a distribution can be explored giving context to the data-set used in the report.

Linux is an operating system started by Linus Torvalds in 1991. Today, it runs many of the world's web servers, smart-phones, embedded systems, and desktops. Linux is described as an operating system kernel, which provides the software interface between the computer hardware and application and library software. To motivate the size and effort of the Linux kernel, David Wheeler has written a few papers estimating the size and overall cost to develop the Linux kernel if it was a single proprietary project. In 2004, he calculated that the Linux kernel at that time had around 4 million lines of code and total development cost was $611 million dollars. This is assuming an average developer salary of $56,000 a year [31]. This result was revised in 2011 showing the Linux kernel had reached 14 million lines of code and is worth a total development cost of around $3 billion dollars, assuming a mean wage of $73,000 a year [5]. These are just estimates, but it is clear that this piece of software is very important for many businesses and people. The kernel provides the core of a distribution, but just as critical are the tens of thousands of packages that are also part of a typical Linux distribution.

A Linux distribution is a set of packages including the kernel that have been tested, integrated and assembled to provide the end-user with a base operating system and applications. Many distributions exist including: Ubuntu, Slackware, Debian, Red Hat, Fedora, SuSE, and Gentoo. In addition distri-

butions specifically for embedded devices such as OpenEmbeded, OpenWrt, MontaVista and WindRiver exist. Various designs and methodologies guide these distributions and give users a wide array of choices in their system [32]. In addition, Linux distributions typically include the (Gnu's not Unix) GNU tools which include critical user libraries as well as the GNU toolchain used to compile most software in a Linux distribution. Window managers, desktop environments, documentation, translations, and applications are also included to give the user a host of tools in order to be productive. To provide these programs or packages a distribution provides a system of installing these packages using a command-line or graphical tool. Many packages rely or depend on other programs or libraries. These programs are called "dependencies" and a package management tool for the distribution must work out which programs need to be installed to satisfy all dependencies for the requested package. Thus the distribution provides a host of packages on a remote server such that the user can install.

For each package a distribution provides, the code must be gathered, compiled, tested, and released. The code for a package might be gathered from an "upstream" or original project source. In addition it could be gathered from another distribution or intermediate source. Finally, some source may be completely original or distinct to that distribution. After the code is gathered, it can be compiled and tested. Occasionally compilations will fail, or other issues may arise. These can be fixed with source code patches that are introduced by the distribution. After a package has been tested and is work-

ing, it can be released such that users of that distribution can easily install the package on their system. This cycle of gather, compile, test and release occurs many times and may coincide with major distribution release cycles. If any bugs are found in the package after release, those fixes may be applied to the distribution's stable release version of the source or contributed back to the original project. Because the number of defects or bugs can become difficult to track through conventional communication channels alone, a bug tracker can be used to track issues and progress for any number of packages associated with a Linux distribution.

A bug tracker is essentially a database of bug information. Examples of Open Source bug trackers include: Launchpad, Bugzilla, Redmine, GitHub, Trac, Assembla and DebBugs. While each has its own specific design, a common feature of these trackers is the ability for a user to create a new bug, give it a description, attach files, assign a bug to a user to work on, show a bug status, and give a bug a priority. For this report all information was gathered from a bug tracking database as it provides a wealth of information about how defects are handled in a Linux distribution. While this section has discussed a general Linux distribution, now the specific distribution used in the report will be discussed.

2.3 Ubuntu and Launchpad

Ubuntu is a Linux distribution started by Mark Shuttleworth and a host of developers in 2004. It is a derivative of the Debian Linux distribution, and

as such uses many of the same tools [16]. Currently, it holds the place as one of the most popular Linux distributions [6][19]. Ubuntu is released every six months and has a version number that corresponds with the year and month of release. For example, 12.04 was released in April 2012. Every two years a Long Term Support (LTS) release is made which is supported for a full five years [17]. Ubuntu is contributed to by many volunteers in the community as well as employees of a company called Canonical. Ubuntu involves many people and roles and uses a project called Launchpad to help organize and provide a host of services for contributors.

Launchpad is a website and collection of programs that help in hosting and facilitating collaboration to develop software. It was started in 2004 by Canonical, and currently is released under the AGPL license [10]. It consists of a bug tracking component, a code hosting and review mechanism using Bazzar, software translation help, project release and management, Ubuntu packaging and building, specification and project planning, and forums [11]. The bug tracking features include methods of sharing bug reports across projects, linking to bug fixes to code hosted in Launchpad, sending emails to users, and a web interface for accessing bugs. In particular, the bug tracking aspects as well as the web interface for accessing bugs were used extensively in this project to gather and view relevant data. Next, Launchpad's bug tracking software will be explained in more detail to give meaning to the dataset.

The Launchpad bug tracker is accessed primarily through a web browser and provides a host of data relevant to those working on and tracking defects

in software projects. It can be used for open source projects as well as closed source projects alike. The Ubuntu distribution tracks all of its bugs using Launchpad, and the data gathered for this report has just concerned itself with these types of bugs that are publicly visible. Any registered Launchpad user can file a bug against any particular project. Because of this, much care has been taken in how to handle communications and how to follow up with users to help solve bugs. In addition to manually submitted bug reports, there are utilities that detect crashes in Ubuntu and opt to automatically send bug reports. These are only sent if the user opts in, but provide an automated way of detecting and filing bugs. Finally, the bugs that are automatically sent will include information, tags, and attachments in the bug to aid in the debugging of the issue. Next the data structures and fields employed by Launchpad will be explored.

2.4 Launchpad Bugs

To first understand how Launchpad bugs work, one must understand the structure and fields that are used by Launchpad. Part of the design rationale of Launchpad's bug tracker is that "when you share free software, you share its bugs" [4]. Launchpad bugs can be either private or public, and they can be associated with a project or multiple projects. They can be associated with a package within a distribution, an upstream bug, and even a project hosted on Launchpad. This is useful when an issue is related to multiple projects or packages and needs to be fixed for each of these projects. Instead

of creating one issue per project that it affects, a bug can contain a "bug task" that contains its own information about that particular task. Multiple "bug tasks" may be associated with a single "bug". A bug task contains an affected package, a targeted series (for a distribution), a status, an importance, an assignee and a milestone. This way the issue can be tracked across various projects. Each bug can also be tracked by associated it with a bug owner, being an assignee to the bug, being marked as affected by the bug, commenting on the bug, or even subscribing to the bug. This way the bug can be effectively tracked and the people working on it can be identified. Bug comments and attachments are also associated with the bug and show the date posted along with the user that posted the comment or attachment. Each bug also has the concept of a "tag" which is a collection of words that describe various bits of information relevant to that bug. These tags can be helpful for many bug-related tasks, including triaging bugs. Bugs also have the ability to be targeted to a series or release of a distribution. For example, if a bug affects a version of the Linux kernel in the Ubuntu Precise release then it can be targeted as such. Finally, because Launchpad is also a source code management system (using Bazzar), branches of code repositories can be linked to a bug to provide an easy way to obtain code fixes from other users. While understanding the structure of a bug is important, one must also understand the steps required to file a bug and how it is handled to completion.

Any Ubuntu user can file bugs in a few ways when an issue is detected. A bug can be reported by first registering with Launchpad. Next, the user

determines if the bug really is a bug; this means the user will typically search online forums, chat channels, or mailing lists to see if this problem can be addressed without filing a bug. If the bug is really a feature request, then there are other channels provided for that information. After this, the user could also try and search for existing bugs and see if any similar bugs exist. If they are affected by a similar bug, they can click a button that indicates the bug also affects that user. At this point, if the user decides that indeed they have a unique bug they begin to create a new bug report. To create a new bug, a user can either use a program called "ubuntu-bug", a menu in development releases to report bugs, a pop-up when a crash has been detected by the system, or using the web-page in launchpad associated with the project where the bug is, to report the bug against the specific project. The advantage of not using the web-page directly is that helper programs such as "Apport" can collect system logs and information about the bug to aid in debugging the issue. Either way, the user will eventually be presented with a web page in which to enter their information about this bug. Figure 2.1 shows an example of the webpage showing an Ubuntu bug. This includes a summary or title for the bug, a larger description, and the ability to attach various files, screenshots, or scripts to help with debugging.

Once the user has submitted the bug, it is given a unique number and information is uploaded to the Launchpad server [14]. At this point in the bug we have a title, description, attachments, an owner registered on launchpad, a creation date, and an affected package which has been identified by Apport or

Figure 2.1: Launchpad Webpage Showing an Ubuntu Bug. [2]

the user which exhibits the bug. At this point the bug has its first and only bug task which has a "status" and "importance" field. The default initial value for the status is "New", while the default value for importance is "Undecided". If the user decides this bug task is of a higher importance they can change this field. A summary of statuses and importances are given in tables 2.2 and 2.3 respectively. After filing the bug, is it now in the hands of the Ubuntu community to resolve it.

After the bug is created it first must be 'triaged', much like a responder

Status	Description
New	Not looked at yet.
Incomplete	Cannot be verified, the reporter needs to give more info.
Opinion	Doesn't fit with the project, but can be discussed.
Invalid	Not a bug. May be a support request or spam.
Won't Fix	Doesn't fit with the project plans, sorry.
Confirmed	Verified by someone other than the reporter.
Triaged	Verified by the bug supervisor.
In Progress	The assigned person is working on it.
Fix Committed	Fixed, but not available until next release.
Fix Released	The fix was released.

Figure 2.2: Launchpad Bug Statuses and Descriptions. [9]

Importance	Description
Undecided	Not decided yet. Maybe needs more discussion.
Critical	Fix now or as soon as possible.
High	Schedule to be fixed soon.
Medium	Fix when convenient, or schedule to fix later.
Low	Fix when convenient.
Wishlist	Not a bug. It's an enhancement/new feature.

Figure 2.3: Launchpad Bug Importances and Descriptions. [9]

at an emergency room. The goal is to provide an additional filter before a bug report is acted upon by developers, or potentially dismiss a case as not being an 'Invalid' bug, or being a duplicate of another bug. This is critical because most developers have limited time and seemingly unlimited supply of bugs. This triaging is accomplished by a specialized team with certain permissions that have a documented workflow for transitioning bugs to and from various states. Figure 2.4 shows a specific workflow for transitioning a bug from a

'New' state in great detail. Essentially, there are various ways to handle bug tasks originating in a new, incomplete, or confirmed state. These bug tasks can be invalidated, expired, marked as duplicates, or the triager can ask the owner for more information marking the bug task incomplete. Eventually, when the bug task is ready for further action it is assigned a person or team to work on the bug and marked 'triaged'. A bug task is considered to be fully triaged when the bug task is linked to a package, it has an importance, it has a state other than 'new' or 'incomplete' and a Launchpad user is assigned to the bug task. In addition, a triager will check for a proper title, description, and ensure the quality of the report. Finally, the triager will mark additional fields to indicate if the bug only affects Ubuntu, or if it is a bug with the upstream package. If the bug is an upstream bug, Launchpad can link to the external bug report as well. At this point the bug and its bugtasks are ready to be worked on by an Ubuntu developer.

Once the developer is assigned and starts to work, the bug task is marked 'In Progress'. All communication about the bug is stored in the bug itself, and attachments and patches can be added to the bug. In addition bugs can be given descriptive tags to help classify and sort various issues. Once a fix is identified, it is tested and confirmed by either the developer if they can reproduce the issue, or the bug owner. If this fix works the bug task can be marked 'Fix Committed' after it is uploaded to the affected packages codebase in Ubuntu. This then eventually gets built, tested, and verified by Ubuntu developers, the community and various Quality Assurance (QA)

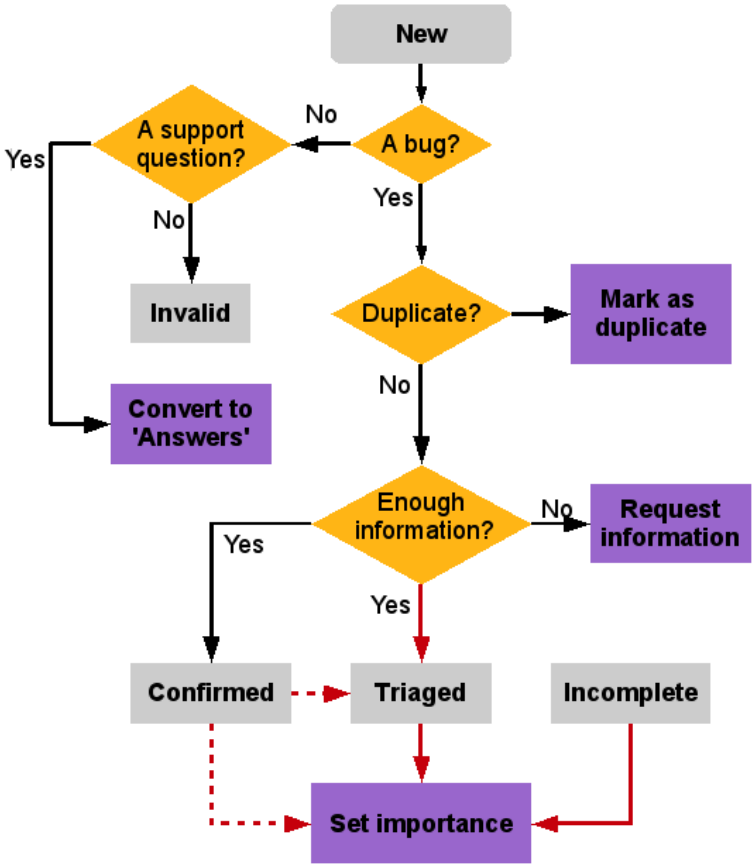

Figure 2.4: Triager Workflows for Incoming New Bugs. [3]

teams. Finally, when the package has passed this testing phase it is released into the main repository and considered 'Fix Released'. At this point the bug task is completed, and it is considered closed. However, there may be other bug tasks needed to be completed before the entire bug is resolved. Even then, a bug still exists in the Launchpad system and in the future could be reopened in case of regressions or other affected packages.

This description of Launchpad, bugs, bug tasks, and how open source works gives a glimpse into all the hard work done by the Ubuntu ecosystem. The design works well for highly distributed teams, and a highly diverse universe of packages. While many books have been written about these topics, the hope is that this overview gives us a sufficient arsenal of knowledge to be able to understand the data collected and the motivations behind it. Working smarter is preferred to working harder, and in the case of bugs, being able to work more effectively with a limited supply of developers makes the world of difference. Currently, there are already efforts to rank bugs in terms of 'heat' or 'gravity' in order to give the most important bugs visibility to be fixed. These efforts are determined by domain experts given existing observations and data gathered. It is the hope of this report that additional insights can be gained to assist with these efforts as well. The next section will explain how data was gathered and stored for this report.

Chapter 3

Gathering Data

This section will explain how the data was collected from Launchpad and stored for this project. While seeming to be the most straightforward task to describe, it proved to be the most time-consuming part of the project. First, how the data was gathered from Launchpad will be explained. Next, the project used to download and store the data will be explained. Finally, a description of the data fields and some examples of the data will be explored.

3.1 Downloading from Launchpad

One important feature of Launchpad is the ability to download bug information via a web interface. This interface was carefully designed by the Launchpad developers and provides a very robust interface for accessing information. In addition to simply providing a robust interface, extensive documentation is available [12] in order to assist accessing and discovering information. Finally, because Launchpad was released as an open source project some aspects of this project required one to "use the source" or access the source code.

In order to use the web interface, a number of different URLs can be

accessed that result in a search result, or a particular bug's information. For example, to access the information for bug number 1, one could issue an HTTP GET command to the URL: `https://api.launchpad.net/1.0/bugs/1`. This will return an XML representation of the bug's information as well as links to other aspects of the bug. In addition to accessing a specific bug's information one could search for active tasks against Ubuntu using the following URL: `https://api.launchpad.net/1.0/ubuntu?ws.op=searchTasks`. This search can be supplied with additional parameters, but defaults to returning all active tasks against the Ubuntu distribution. The response is in JSON format with links to retrieve the bug tasks in groups of 75. Using the web interface can allow for some very interactive and interesting programs; however for this project it was decided that using a python library called Launchpadlib to interface with the API was the best choice. Next, the project used to download and store data will be explained.

3.2 Shadow Database

In order to accomplish the task of downloading information from Launchpad and storing into a database a project called shadow-database was started to effectively 'shadow' much of the data already existing in Launchpad. The goal of this was to provide a way to collect information using the API in a reliable, robust, simple, and efficient way. In the spirit of open-source, the project is actually hosted on Launchpad at the following location: `https://code.launchpad.net/shadow-database`.

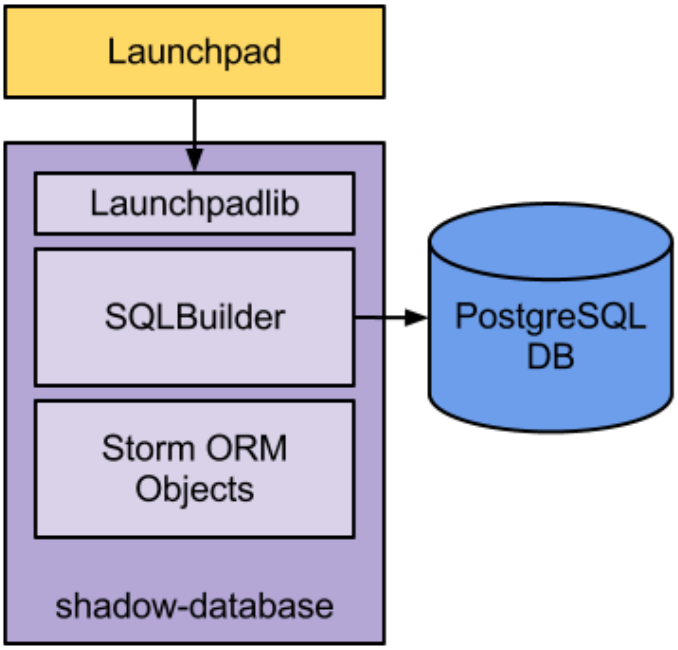

Figure 3.1: Shadow-database project architecture.

Figure 3.1 shows the overall architecture of the project. The shadow-database project is written entirely in Python with some bash and SQL scripting as well. The PostgreSQL database is created in advance using scripts provided in the project. After this, bugs are downloaded from Launchpad via Launchpadlib and parsed by the SQLBuilder Python class. Part of the work was in optimizing the download of a large volume of bugs. In order to accomplish this, some Launchpadlib interfaces were used along with a browser object that would return the raw JSON text returned by a web query. This

```
./scripts/shadow.py --limit 100 --retry 2 --bugs --verbose --save 10
```

Figure 3.2: Example ShadowDB invocation.

query response was then parsed by a Python JSON parser to populate an object much more quickly than iterating through each property of the object using Launchpadlib. After SQLBuilder obtains the data it then uses Storm, an Object Relational Manager (ORM) written in Python for SQL databases[15], to create the data. In fact, Launchpad itself uses Storm as well and because it is also open source it seemed to be a very natural choice. The SQLBuilder creates a Storm ORM object of the appropriate type depending on the data (bug, bugtask, person, bugmessage), and this code is meant to provide a mapping between Python and the SQL tables. This object then is easily inserted into the already created PostgreSQL database. Depending on the script invocation, one can download a limited number of bugs and commit at certain intervals. In addition, there are options to trap for server errors and retry a request for a specified number. Currently, this project is not complete in that it only downloads certain aspects of the data, the goal is to mirror as much as possible and is reasonable given the goals of data mining Launchpad data. An example of an invocation of the script is as follows:

The script in Figure 3.2 shows a command to download 100 bugs and retry twice if there are errors. In addition, this command commits changes to the database for every 10 new bugs, and is verbose in explaining exactly what is being downloaded. This script proved to be useful in collecting data as

the data collection took a few weeks of computer time to download the bugs. Once we have this data in the database it's important to understand what all is collected and what the data structures look like.

3.3 Data Structures

Because of the large amount of data and relationships between the data, an SQL database was required for storing the data. Initially the data was stored into a very large text file which contained the data in JSON format. This yielded very large files that took extremely large amounts of time to do any processing once the data became greater than a gigabyte. Part of the shadow-database project involved creating an SQL schema based on the tables already used in the Launchpad project. This was done with the help of a co-worker and the result was the set of tables in Figure 3.1.

milestone	bugactivity
person	distribution
bug	emailaddress
bugtarget	sourcepackage
bugtracker	workingteam
bugwatch	workingteampackages
bugmessage	distroseries
bugattachment	distrosourcepackage
bugstatus	bugsubscription
bugimportance	architecture
bugbranch	teammembership
bugtask	updates

Table 3.1: Full listing of all relevant tables.

person	bugstatus
bug	bugimportance
bugtarget	bugtask

Table 3.2: Listing of all tables actually used.

Part of the project was scoping the data to find the essential features to begin analysis. The selection in Figure 3.2 shows the tables actually used in the analysis. The bug, bugtask and person tables were the primary sources of information with the bugstatus, bugimportance tables essentially providing a mapping between an integer and the appropriate labels. Next the fields given by the main tables will be explored.

Column	**Type**
id	integer
display_name	text
owner	integer
description	text
name	text
date_created	timestamp without time zone
karma	integer
hide_email_addresses	boolean
preferred_mail_address	integer
is_valid	boolean
private	boolean
irc_nicknames	text
is_team	boolean

Table 3.3: Person table.

The person table contains an ID which is the primary key used by the bug and bugtask tables to identify owners and assignees of those compo-

Field	Value
id	1046
display_name	Chris J Arges
owner	
description	
name	christopherarges
date_created	2007-08-11 15:51:23.84256
karma	3419
hide_email_addresses	
preferred_mail_address	t
is_valid	t
private	f
irc_nicknames	arges,arges
is_team	f

Table 3.4: Example entry in person table.

nents. The display_name is the person's full name, while the name field is their Launchpad username. Because a 'person' object encapsulates both the concept of a team and of a actual person there are fields that are relevant for only teams. If is_team is true, then the owner corresponds to a person or another team which owns the team. The date of creation is when the person was registered with Launchpad, and is_valid is marked false if a person ever deactivated their account. A karma score is calculated based on the number of activities a user performs within Launchpad [13]. E-mail addresses can be registered with Launchpad and stored in preferred_mail_address which points to a table containing email addresses. For this study this table was not populated. Additionally e-mails can be hidden from the public if hide_email_address is set to true. A person can also be marked 'private' which means that the person is

not visible to public users. For this study only public data was used. Finally, irc_nicknames are stored as a string if registered by the Launchpad user. Next, the bug table is examined in detail.

Column	Type
id	integer
date_created	timestamp without time zone
date_last_message	timestamp without time zone
date_last_updated	timestamp without time zone
date_made_private	timestamp without time zone
description	text
duplicate_of	integer
heat	integer
gravity	integer
information_type	text
latest_patch_uploaded	timestamp without time zone
message_count	integer
number_of_duplicates	integer
other_users_affected_count_with_dupes	integer
owner	integer
private	boolean
security_related	boolean
tags	text
title	text
users_affected_count	integer
users_affected_count_with_dupes	integer
web_link	text
who_made_private	integer

Table 3.5: Bug table.

The bug table maps directly to an actual web-page associated with a bug. For example, bug 1 can be seen on `https://launchpad.net/bugs/1` which contains many persons and bugtasks. The id field corresponds directly

to the bug number also found in Launchpad. The date created, last message, last updated, and made private fields all show timestamps of these relevant events and when they occurred. The date created is required, however the other fields are not required as those events may have not happened. A description field corresponds to the long description given in the bug. This usually contains various information about how to reproduce, what is expected, what actually happens, and what versions are affected. A duplicate_of field indicates the bug number that this bug is a duplicate of. If this field is not null then we know this particular bug is a duplicate of another bug. A heat field is calculated based on if the bug is private, if it's a security issue, the number of duplicates, the number of affected users, and the number of subscribers for that particular bug [7]. The information_type field shows if a particular bug is public, public security, private security, or private. A description of these fields are shown in Table 3.7.

The latest_patch_uploaded shows the date the last patch was uploaded, this is useful in identifying how recently a patch was uploaded to the bug or if a patch even exists. A message_count field shows the number of bug comments associated with a single bug. The number_of_duplicates fields shows how many bugs are marked as a duplicate of this bug. The users_affected_count_with_dupes show the total affected users including those in bugs are are marked duplicates of the current bug. The other_users_affected_count_with_dupes field is similar but shows only the users affected in the bug duplicates. The owner field points to an entry in the person table and indicate who filed the bug initially. The

private and security_related fields concern the visibility and security concerns of a bug. For this report only public bugs were considered. The tags field contains a list of words that give the bug additional descriptions, in this table that list is flattened into a string with commas separating each word. Tags can be added by users, bug triagers, and even automated scripts. Table 3.8 shows some commonly used tags and their meanings.

The title identifies a summary of the issue and is displayed at the top of the bug's webpage. The users_affected_count just shows the number of users affected by the top level bug and not any duplicates. The web_link is a URL that points directly at the webpage for that particular bug. And finally the who_made_private shows which person made the bug private. Next, the bugtask table will be explained.

The bugtask table contains all information related to a task within a bug. It contains its own ID to identify it within the table, but the 'bug' field contains the mapping back to the id of the bug in the bug table. This means there can be a multiplicity of bugtasks that map back to a single bug. However, a single bugtask cannot map to multiple bugs. The assignee field contains an integer that maps to the ID of an entry in the person table. The many date fields show when various events occurred for that particular bug-task. The date_created is when the initial task was created, which can happen when a bug has been targeted to a new package or series. The date_assigned field shows when the bugtask was assigned to an actual person or team. The date_closed field indicates when a bug has reached a closed state which corre-

sponds to a state of 'Fix Released', 'Invalid', 'Expired', or 'Won't Fix'. The date_fix_committed, date_fix_released, date_incomplete, date_in_progress, and date_triaged show when the bug state has last changed to those particular bug states. Many of these fields will be NA in the dataset because the bug may have not ever been set in that particular state. The NA in this case indicates that the bugtask never had this particular state transition. The date_left_new and date_left_closed show when a bug has left the 'New' status or the date the bug was re-opened from a Closed state [12]. The milestone field is a mapping to a bugmilestone entry, but is not used in this report. The owner is the person who created the bugtask. The status entry corresponds to a bugstatus, and the importance entry corresponds to a bugimportance. The target corresponds to a bugtarget table and shows the package that the bugtask affects. Finally, a bug_watch mapping points to a bugwatch entry, but is not used in this report. While there are many other aspects of the data that could be explored, this gives the main fields used in this report. With this breadth of information, the next step is to visualize and explore this massive data set.

Column	Data
id	1
date_created	2004-08-20 00:00:00
date_last_message	2012-05-23 23:09:13.221333
date_last_updated	2012-06-30 11:33:03.967201
date_made_private	
description	Microsoft has a majority market share ...
duplicate_of	
heat	7038
gravity	0
information_type	Public
latest_patch_uploaded	
message_count	1696
number_of_duplicates	2
other_users_affected_count_with_dupes	1566
owner	1
private	f
security_related	f
tags	ubuntu
title	Microsoft has a majority market share
users_affected_count	1564
users_affected_count_with_dupes	1566
web_link	https://bugs.launchpad.net/bugs/1
who_made_private	

Table 3.6: Example entry in bug table.

Type	Description
Public	Everyone can see this information.
Unembargoed Security	Everyone can see this information pertaining to a resolved security related bug.
Embargoed Security	Visible only to users with whom the project has shared embargoed security information.
Private	Visible only to users with whom the project has shared private information.

Table 3.7: Launchpad Bug Information types.

Tag	Use case
apport-bug	A bug reported using "Report a Problem" in an application's Help menu contains lots of details!
apport-collected	A bug that has had apport-collect ran against it which will contain additional information
apport-crash	A crash reported by apport - Ubuntu's automated problem reporter
apport-package	A bug reported by apport when a package operation failed
ftbfs	Bugs describing build failures of packages.
hw-specific	A bug that requires a specific piece of hardware to duplicate
iso-testing	A bug found when performing iso testing and also tracked at `https://iso.qa.ubuntu.com/`
likely-dup	The bug is likely a duplicate of another bug but you can't find it. (Maybe an upstream bug too.)
packaging	The bug is likely to be a packaging mistake.
dist-upgrade	A bug that was encountered when upgrading between releases of Ubuntu
verification-done	A Stable Release Update bug with a package in -proposed that has been confirmed to fix the bug
verification-failed	A Stable Release Update bug with a package in -proposed that has been verified to not fix the bug
verification-needed	A Stable Release Update bug with a package in -proposed needing testing
patch	This bug has a patch attached to it

Table 3.8: Relevant Tags [18].

Column	Type
id	integer
assignee	integer
bug	integer
date_created	timestamp without time zone
date_assigned	timestamp without time zone
date_closed	timestamp without time zone
date_confirmed	timestamp without time zone
date_fix_committed	timestamp without time zone
date_fix_released	timestamp without time zone
date_incomplete	timestamp without time zone
date_in_progress	timestamp without time zone
date_left_closed	timestamp without time zone
date_left_new	timestamp without time zone
date_triaged	timestamp without time zone
milestone	integer
owner	integer
status	integer
importance	integer
target	integer
bug_watch	integer

Table 3.9: Bugtask table.

Column	Data
id	1
assignee	
bug	1
date_created	2008-08-06 01:03:40.536857
date_assigned	
date_closed	
date_confirmed	2010-07-26 09:06:34.737289
date_fix_committed	
date_fix_released	
date_incomplete	
date_in_progress	
date_left_closed	2010-07-26 09:06:34.737289
date_left_new	2008-08-06 01:07:17.769186
date_triaged	
milestone	
owner	2
status	7
importance	2
target	1
bug_watch	
target_name	clubdistro

Table 3.10: Example entry in bugtask table.

Chapter 4

Visualization

The database used in this report includes 430,832 bugs, 581,276 bug-tasks, 152,519 people and 963,370 bugmessages. This data was gathered over a few weeks during the beginning of July of 2012. Because the data was gathered over a long period of time, changes to Launchpad bugs in the meantime may not be properly reflected in the database used for the study. Regardless of this, the data is still useful for this report.

After the data was collected into a database, the next step was to choose an appropriate tool to perform visualization of the data and analysis. For this project, the R programming language was chosen for the large amount of statistical, data mining, and visualization libraries available for the project. In addition, R runs well on many platforms including Linux, and is a fully open source project. Finally, documentation was widely available and there was personal interest in learning about this programming language.

Now a tour of various visualizations created in R using the dataset will be explored. Some of these graphs were useful in seeing patterns that could be important for analysis, or perhaps interesting in future work.

Figure 4.1 shows histograms of features not related to time of the

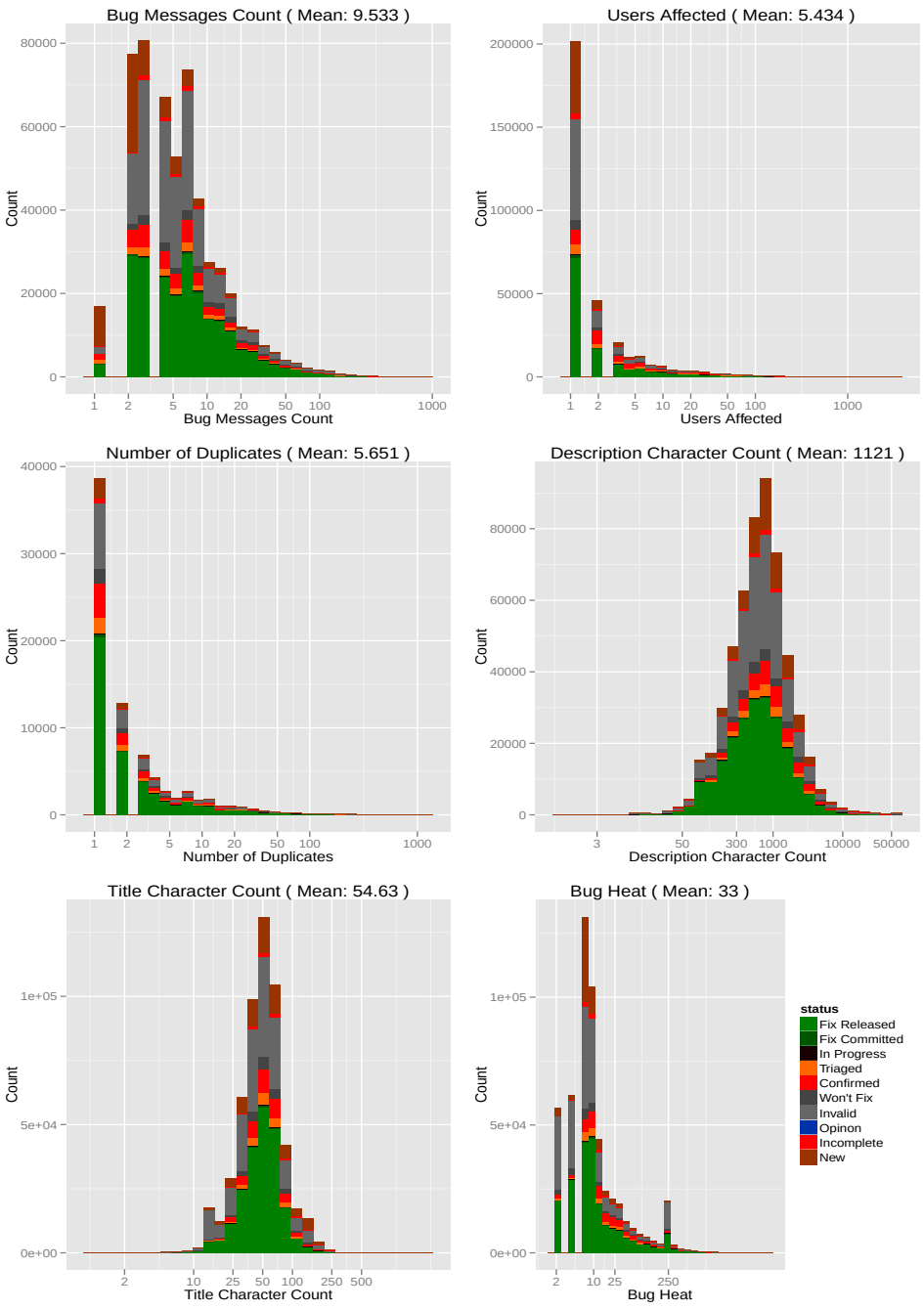

Figure 4.1: Histograms of Bug Dataset Including: Bug Messages Count, Users Affected, Number of Duplicates, Description Character Count, Title Character Count, and Bug Heat.

dataset of all bugs open and closed. Each of these plots were scaled logarithmically along the x-axis to better visualize the distribution of the data. In addition, the histogram shown is stacked with each color corresponding to the final status for the bugtask being examined. The colors were chosen to match those used on the Launchpad website. The number of bug messages per each bug shows that on average there are 9 messages per bug, and that most messages are less than 10 messages. The next graph on the upper right shows the mean number of users affected for each bug is 5.4. In addition we see that the number of affected users drops sharply even if the graph is scaled logarithmically. The next graph in the middle left shows the mean number of duplicates is 5.6 and looks similar to the number of users affected. The graph in the middle right shows the number of characters in the description. The mean number of characters is 1121, which could be due to many of the bug reports having a description automatically generated by bug bots and programs like 'Apport'. The graph in the bottom left shows the number of characters in the title has a mean of 54. The final graph on the bottom right shows the bug heat has a mean of 33, with a spike around 250. Because bug heat is a derived feature from other data in the bug, this could be explained by how the bug heat is calculated.

Figure 4.2 shows histograms of time-related relevant features in the dataset of closed bugs only. Each of these histograms are scaled logarithmically along the x-axis, and the status is shown as a stacked bar. Because the dataset only contains closed bugs, the only statuses shown are "Fix Released", "Won't

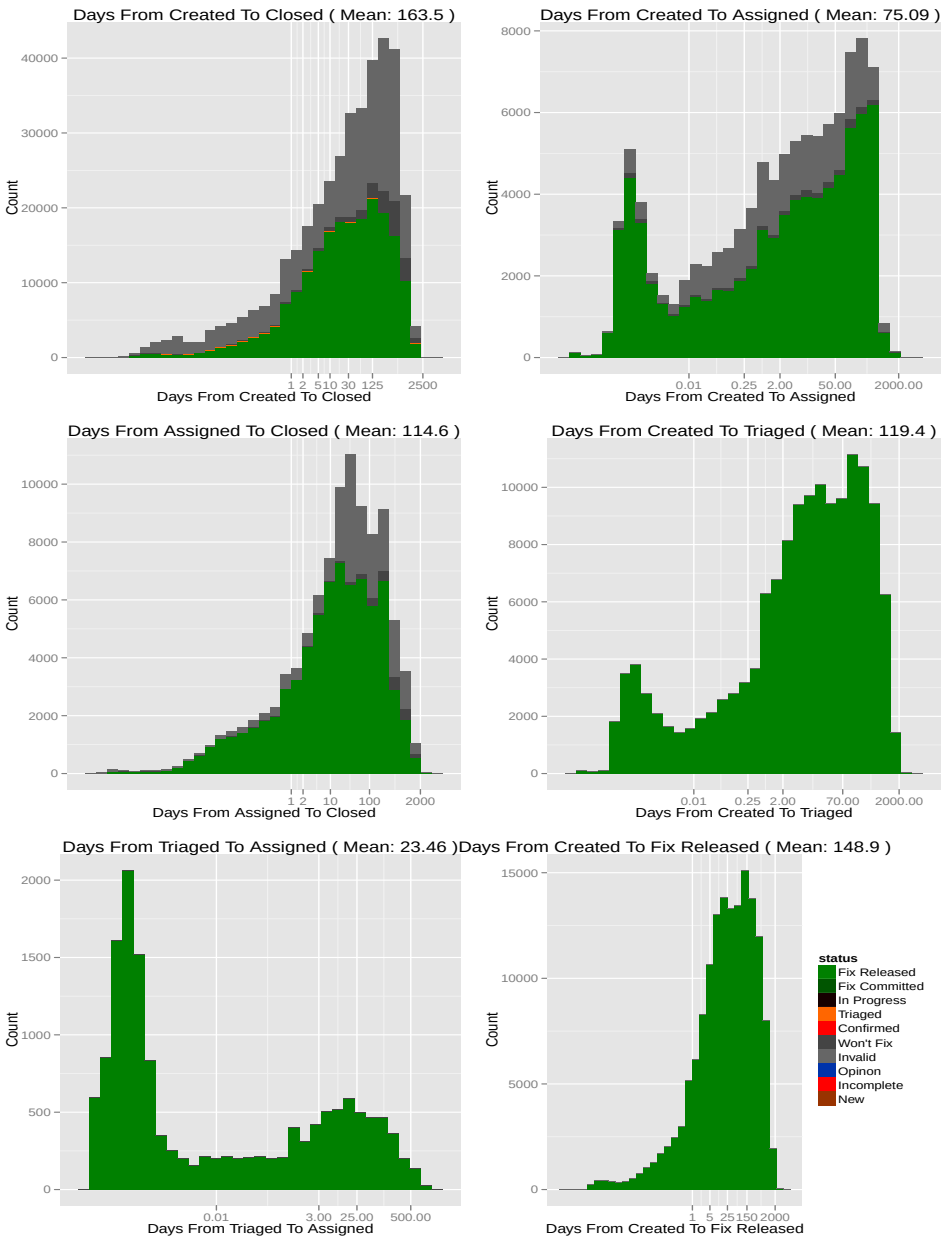

Figure 4.2: Histograms of Status Transition Related Bug Dataset Including: Days From Created to Closed, Days From Created to Assigned, Days From Assigned To Closed, Days From Created To Triaged, Days From Triaged to Assigned, and Days From Created To Fix Released.

Fix", and "Invalid". While a transition of created to closed exists for all closed bugs, the other state transitions don't always occur for the other bugs and those particular data points have been eliminated. The upper left graph shows that the days from created to closed has a mean of 163 days. This measures the days from the date a bugtask is created until it reaches a closed bug state. It can be observed that some bugs take an extremely long time to fix, and that a good portion of the bugs are actually resolved as 'Invalid'. This means that time spent on these 'Invalid' bugs did not result in a fix. Therefore a way to predict these types of bugs would be valuable. The graph in the upper right shows the mean days from created to assigned is 75 days. This shows a peak sometime between 0 and 0.01 days and another peak closer to the mean. The early peak could be caused by bugs that are created and immediately assigned to a person either by an automated script or manually. The middle left graph shows the days from assigned to closed having a mean of 114 days. This could be somewhat of an indicator of the real time it takes to solve a bug; however, some assignees prefer to assign themselves to a bugtask before actually spending time fixing it. The graph in the middle right shows the days from created to triaged has a mean of 119 days. This again shows a peak somewhere between 0 and 0.001 and another peak close to the actual mean. The early peak could be caused by automated script or manually changing the script right after creating the bug. The next graph in the lower left shows the days from triaged to assigned has a mean of 23 days. Again we see a peak between 0 and 0.01 days, and another peak around the mean. The final graph

in the lower right shows the days from created to fix released has a mean of 148 days. This is essentially the same as the days from created to closed with other final bugtask statuses eliminated. Overall, it can be seen that another value would be to reduce the time from created to assigned as it has a mean of 75 days. Thus the motivation for a way to automatically triage and assign a bug is established.

Next, a correlation matrix of various features is plotted in Figure 4.3 to show their inter-relationships. The dark blue squares show a strong positive correlation, while the dark red squares show a strong negative correlation. This graph shows that the number of messages in a bug is positively correlated with the number of users affected, the users affected including duplicate bugs, and the number of duplicates. This makes sense because as the number of people affected by a bug increases, more comments on the bug will be added by those affected users. In addition the number of users affected by a bug and the number of users affected by a bug with duplicates is very strongly correlated because they are directly related. Some bug states were also compared to see how they correlate with other variables. The bugtask state of 'Invalid' seems to be very negatively correlated with a bugtask state of 'Fix Released'. This makes sense, because essentially they are both closed states that are orthogonal.

Figure 4.4 shows two bar charts which explore the frequency of various factors in the dataset of all open bugs. The top bar chart shows the most frequent bugs per team type. The team type is a derived by taking the package

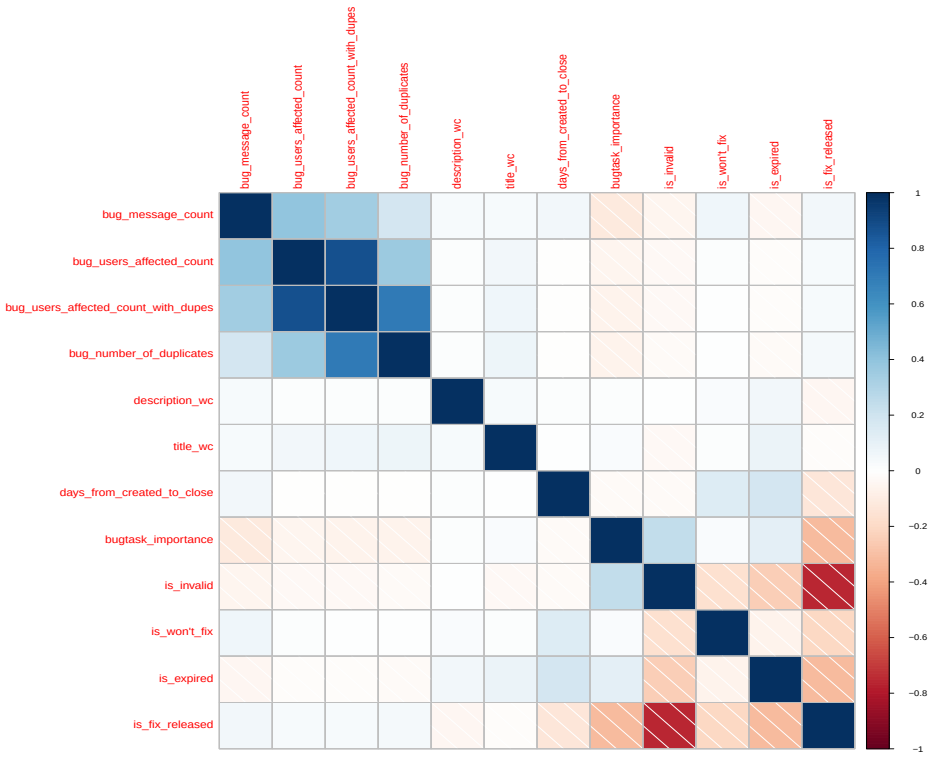

Figure 4.3: Correlation Matrix of Bug Dataset Features.

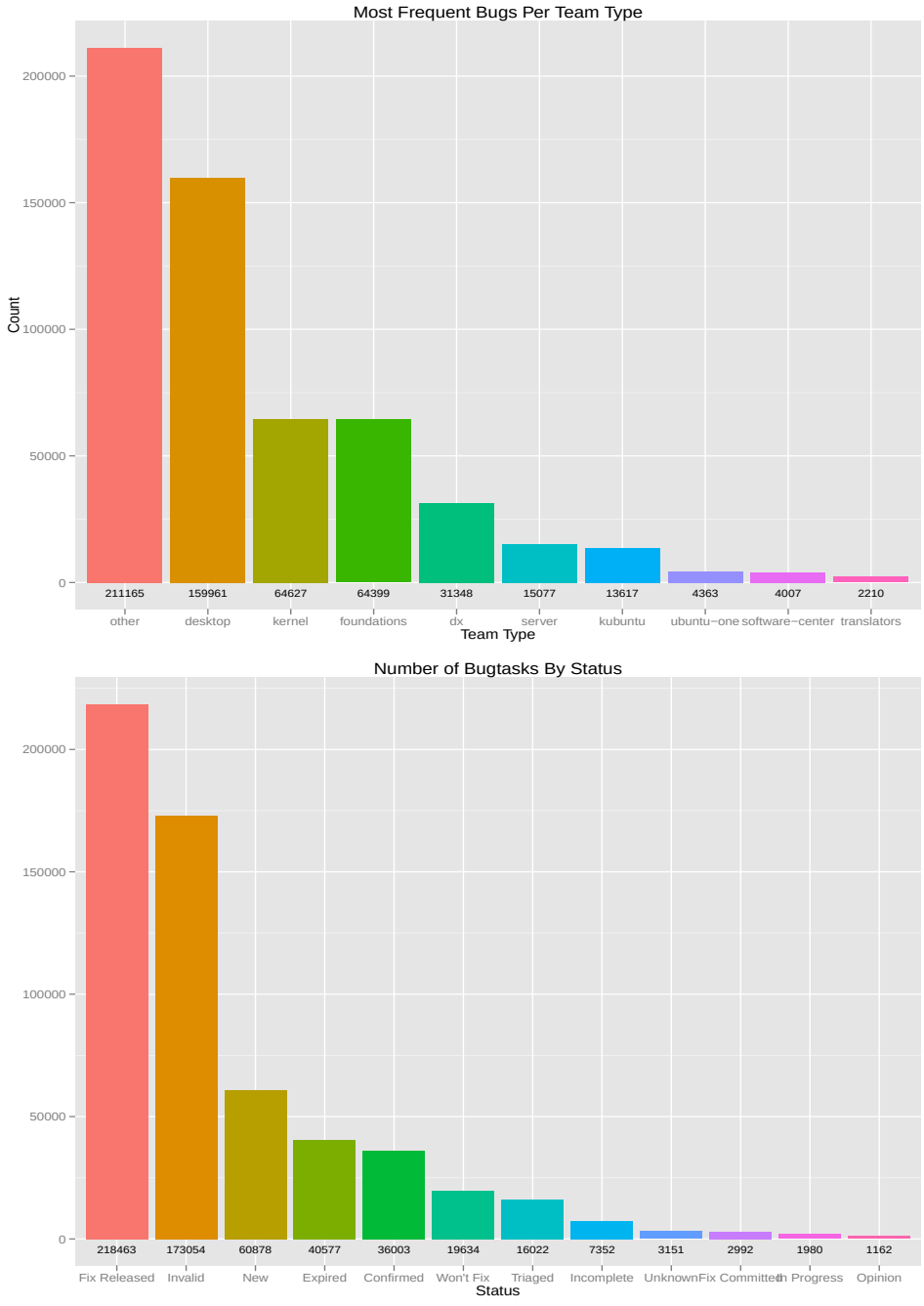

Figure 4.4: Barcharts of Number of Bugtasks by Importance and Most Frequently Used Tags.

name and looking up which 'team' in Ubuntu is generally responsible for care of that package in a manually populated table. Because this graph includes all bugtasks in the history of Ubuntu, there are some packages that were not mapped or no longer have a team associated with them. For that reason 211,165 of bugtasks' packages are marked as 'other' which indicates that a mapping could not be found. However, this chart is still useful in that it shows that 159,961 desktop bugtasks were counted which constitute a majority of the classified bugs. Next, kernel and foundations bugs are also a major source of the bugs at 64,627 and 64,399 bugtasks respectively. The next chart in this figure is the number of bugtasks by status. This includes all bugs historically both open and closed. It shows that 218,463 bugtasks were fixed, while 173,054 bugtasks were marked invalid. The large invalid count shows the value in predicting these particular bug tasks earlier rather than later.

Figure 4.5 shows two other bar charts with other important counts of bugtask fields. The top graph shows the number of bugtasks by importance. A total of 295,733 bugtasks were set to 'Undecided', 110,980 were marked 'Medium', and 65,532 were set to 'Low'. By default, Launchpad sets the importance to 'Undecided', so this indicates that a majority of bug users do not actually set the importance of the bugtask. This also shows that most users view their bugs as 'Medium' to 'Low' importance, and that 'High' or 'Critical' bugtasks are less likely. Finally, the bar chart on the bottom shows the most frequent tags in all bugtasks. The most frequent tags in descending order are: 'apport-bug', 'i386', 'amd64', apport-crash', and 'lucid'. After speaking

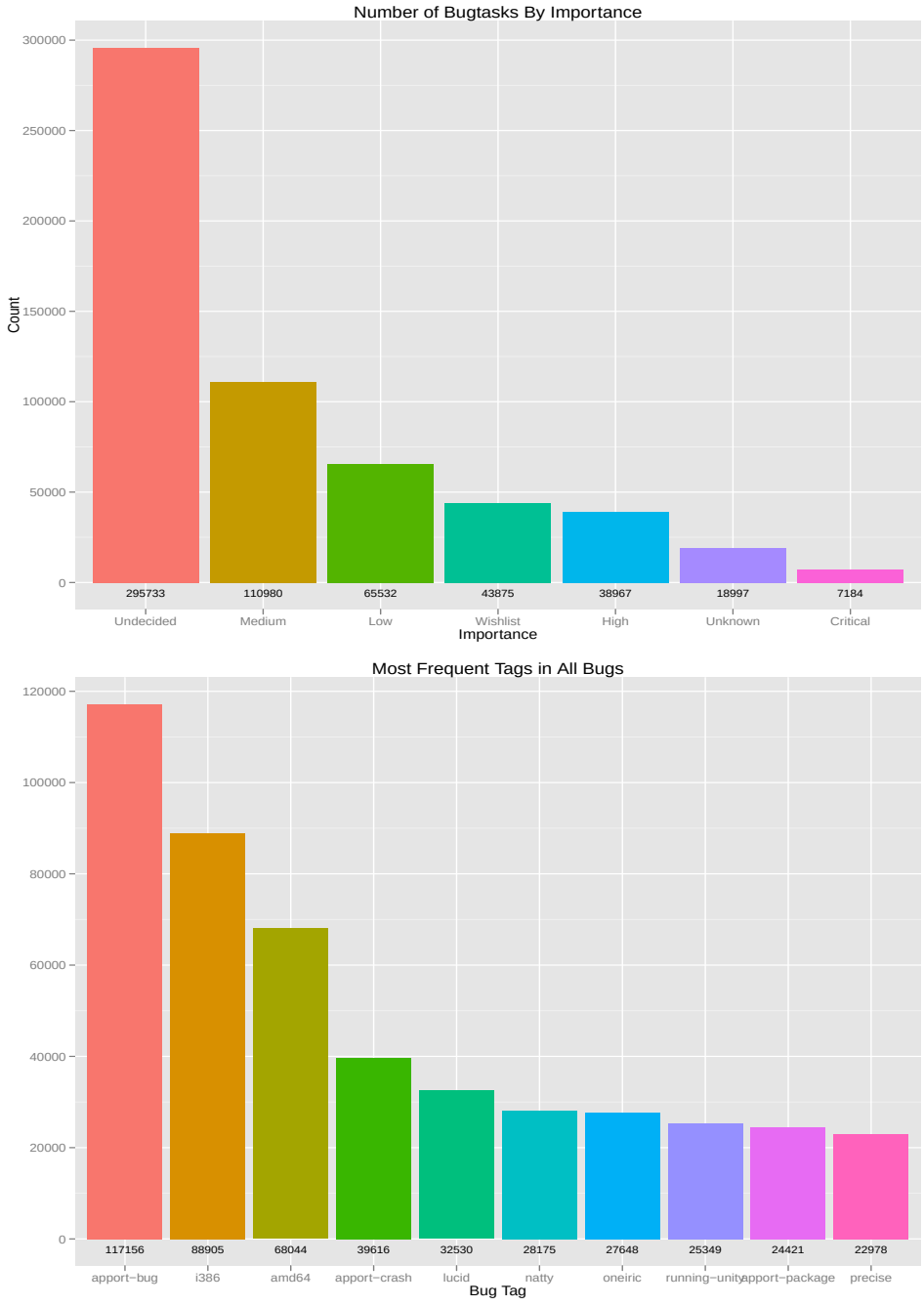

Figure 4.5: Barcharts of Number of Bugtasks by Status and Importance.

with defect analysts and subject experts, it was indicated that these tags are most likely set by automated programs and not manually by users. This indicates that these scripts are proving useful at tagging bugs, and in addition filing bugs. Because 'apport-bug' shows up as the most frequent tag, it is an indicator that the 'apport' program that automatically collects information and files a bug for a user is extremely useful. Next, an examination of historical trends in users and bugtasks will be examined.

A historical chart showing the number of bugs opened and the number of registered Launchpad users that are involved in the bug reports is shown in Figure 4.6. The x-axis shows the date, while the y-axis shows the number of bugtasks or the number of users. In addition, the Ubuntu release date has been plotted with dashed lines to see if there were any cyclical features that could be discerned from this graph. It is important to note that the number of Launchpad users only counts those users that have registered in Launchpad and have been involved in either filing, fixing, or commenting on a bug. In addition the registered Launchpad users also includes team accounts and bots registered as Launchpad users. This is in no way an indicator of the actual number of Ubuntu users, but rather is more of a related indicator. As it can be clearly seen, as the number of open bugtasks increases, so do the number of registered Launchpad users. In addition, effort in fixing bugs can be seen in between releases and right after a release; however, it may be more useful to be able to sort bugs by which release they actually fix to give a better idea of how release cycles affect the number of open bugtasks. This could be something to

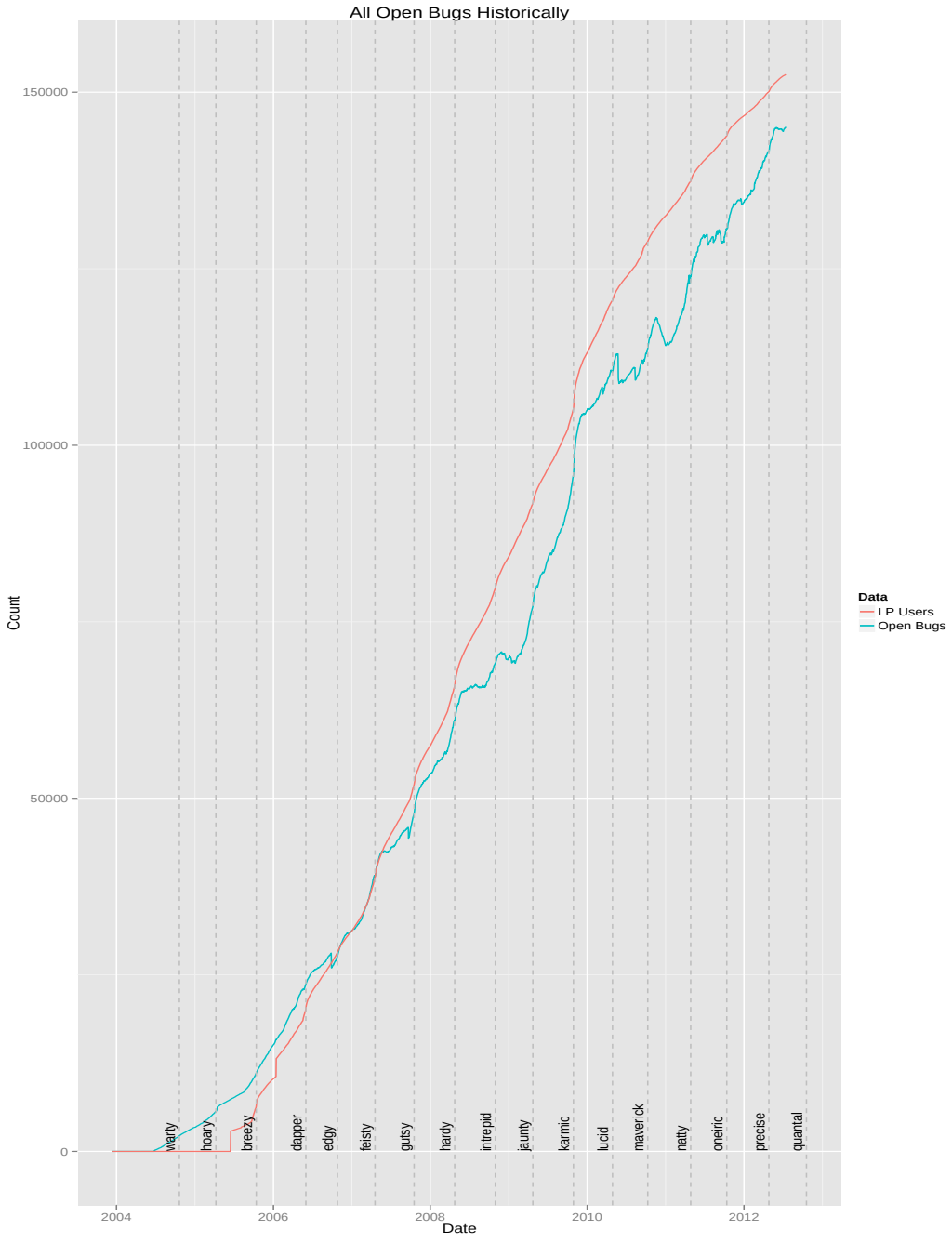

Figure 4.6: Line Chart With Historical Data Showing Bugs Open and Number of Registered Launchpad Users.

explore in future work.

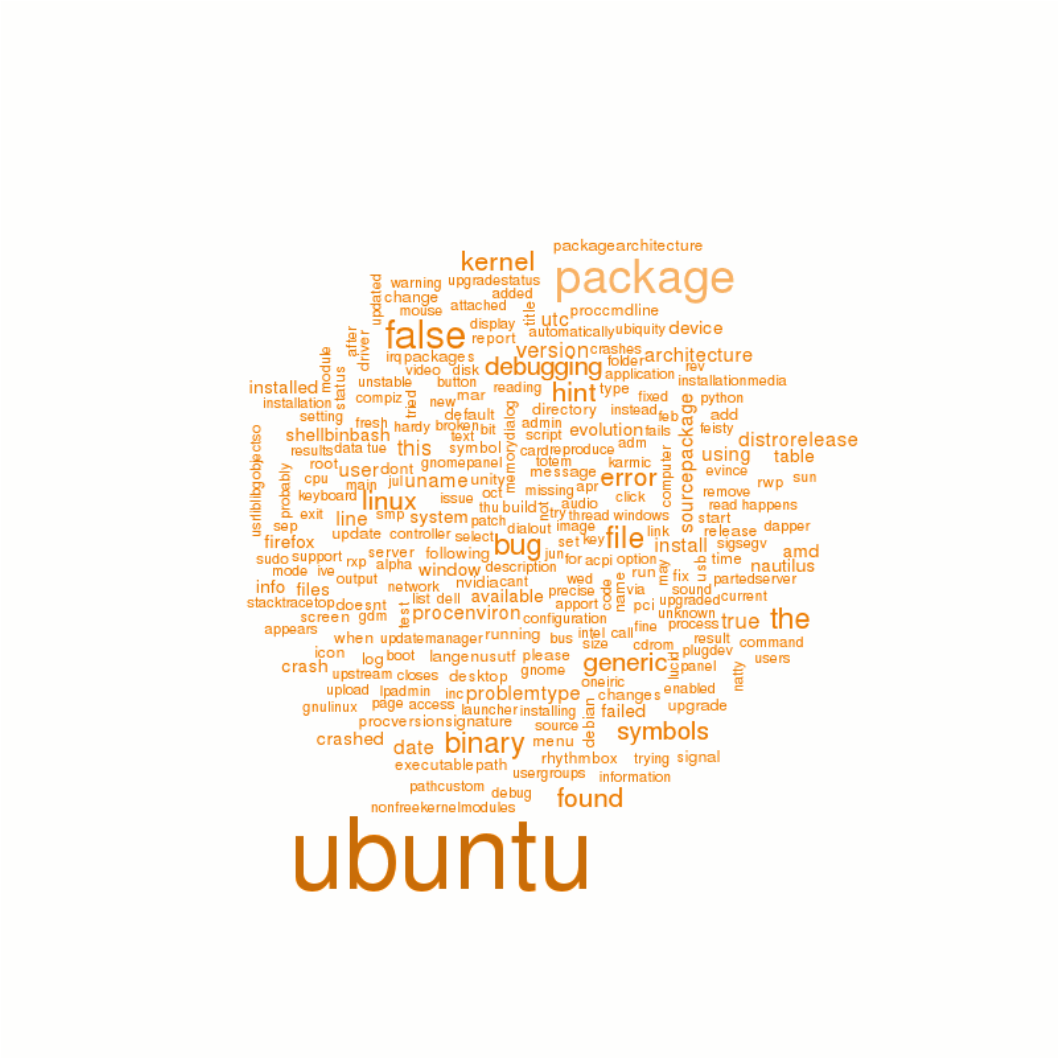

Figure 4.7: Word cloud with text from Ubuntu bugs.

Finally, a word cloud was rendered using R and shown in Figure 4.7. This includes all words from the titles, descriptions, and package names. All English stopwords, numbers, punctuation were removed from the list, and

only the 300 most frequent words were graphed. While the actual bugmessage contents could have been included in well, the amount of time required to compile all word frequencies for just the title, description and package names across all bugs was an extremely lengthy process on the order of 8 hours. As we can see a few words stick out such as: Ubuntu, package, false, kernel, debugging, bug, binary, found, and Linux.

While more visualizations could be produced, these few graphs give a glimpse into the rich dataset provided by Launchpad and Ubuntu. However, it is sufficient to motivate the analysis as the need to reduce the number of invalid bugtasks and time to assignment are seen. Next, these hypotheses will be explored and examined.

Chapter 5

Analysis

After collecting the information and visualizing the data, a few questions were asked. First, it was observed that within 436,128 closed bugtasks, that 24,335 were marked 'Invalid', and 79,477 marked 'Fix Released'. An 'Invalid' bugtask means that the bugtask was not actually a bug. Thus, predicting these bugtasks in advance would allow bug triagers to prioritize bug reports that are more likely to be fixed rather than ones that end up being marked 'Invalid'. In addition, it was observed that on average it takes 75 days to go from created to assigned. Part of the reason this takes time is that someone must manually assign themselves or another person to the bugtask. Thus, the concept of an 'autotriager' was explored that would predict the assignee based on historically closed bug reports. In addition, not only predicting a single assignee, but giving a list of potential assignees and load balancing assignee's tasks was considered important. Finally, this list of potential assignees was given a score that indicated the number of currently in-progress bug tasks to indicate their load, and thus rank the results in such a way that no one developer becomes overloaded with bug-tasks. Table 5.1 summarizes the hypothesis in this report. First, some research was done into existing papers that have explored similar problems.

H1	Predict Invalid Bugtasks
H2	Automatically Assign Incoming Bugtasks

Table 5.1: Hypotheses examined in this report.

5.1 Existing Research

One paper written by John Anvik, Lyndon Hiew and Gail C. Murphy entitled "Who Should Fix this Bug?" [20] used a Support Vector Machine (SVM) classification algorithm to identify who should solve a bug report. They used a supervised algorithm and were able to achieve precision of 50% to 64% on one bug report dataset. Another paper that also did automatic triaging was entitled "COSTRIAGE: A Cost-Aware Triage Algorithm for Bug Reporting Systems" , by Jin-woo Park, Mu-Woong Lee, Jinhan Kim, Seung-won Hwang Sunghun Kim [26]. This paper improved on the Content-Boosted Collaborative Filtering (CBCF) approach which augments the process with 'Developer Profiles' and additional bug categorization. Because there is a cost parameter the accuracy and time to fix can be adjusted based on a parameter. One result with the Apache bug data set resulted in an accuracy of 65.9% and a reduction in cost of 30%.

5.2 Techniques Used

This report uses classification algorithms to assign a 'class' based on predictors for both hypothesis. Part of the challenge in this report was correctly identifying which R package would provide the best implementation of

a particular algorithm. In addition, carefully sampling and reducing the size of the data set was necessary to allow for timely calculations on available computer systems. After some experimentation, two classification algorithms were chosen: Naive Bayes and Support Vector Machines (SVM). Random Forests were also tried unsuccessfully during analysis as another classification technique, but due to limitations with the R package it was unable to handle assigning more than 32 classes to a dataset. Both techniques used supervised learning techniques. Only the set of closed bugtasks were used as they contained proper mappings from input parameters to the desired output class. For this research, only closed bugtasks would contain the final assignee that solved the bug, and the final status that the bugtask contained when closed. The one drawback is that if data for a particular output class is not available in the training set, then the model will not have that particular class as an outcome. For predicting 'Invalid' and 'Fix Released' bug tasks, this was not an issue because all classes are already enumerated and known in the dataset, and it is trivial to sample enough to encompass each class. However, for the auto-triager, the training set must be carefully sampled to maximize the number of different bug task assignees in order to assign those particular assignees. Now each hypothesis and experiments will be explained in greater depth.

5.3 Predict Invalid Bugtasks

Because of the great number of invalid bugtasks, there is motivation to be able to predict the final status of a bug based on its initial attributes. This

way, an incoming bugtask could be prioritized based on if it is classified as a bugtask that will eventually be 'Fix Released'. For this analysis the bugtask importance, package name, owner, and package type are all known at the time the bug is filed. These variables formed the predictors, while the target class is the bugtask status. A few classification algorithms were attempted to be used on the predictors including: Random Forests, AdaBoost, Naive Bayes, and SVM. Random Forests and AdaBoost proved difficult to be able to use in R with this particular data set. In the end, only Naive Bayes and SVM were used, and their performance was compared using various sizes for the training set.

For this classifier, a loop was run varying the overall sample size to determine the performance. This number varied from 1,000 to 4,000 in increments of 500. For each size, the test was repeated 20 times to account for some of the variance and randomness of the sampling. For each test, R's 'sample' function was used to randomly sample the desired size from the set of closed bug tasks. Then the data was partitioned into a training and validation set using the R function 'createDataPartition'. The training and validation sets were sampled using this function. In addition this function tries to balance the outcome class distributions among the training and validation sets. Next each classifier was trained using the training set and the performance on predicting the validation set was calculated. The results for this classifier are graphed in Figure 5.1, and a summary is provided in Table 5.2. As it can be seen in Figure 5.1, the performance of the Naive Bayes classifier drops a bit in performance

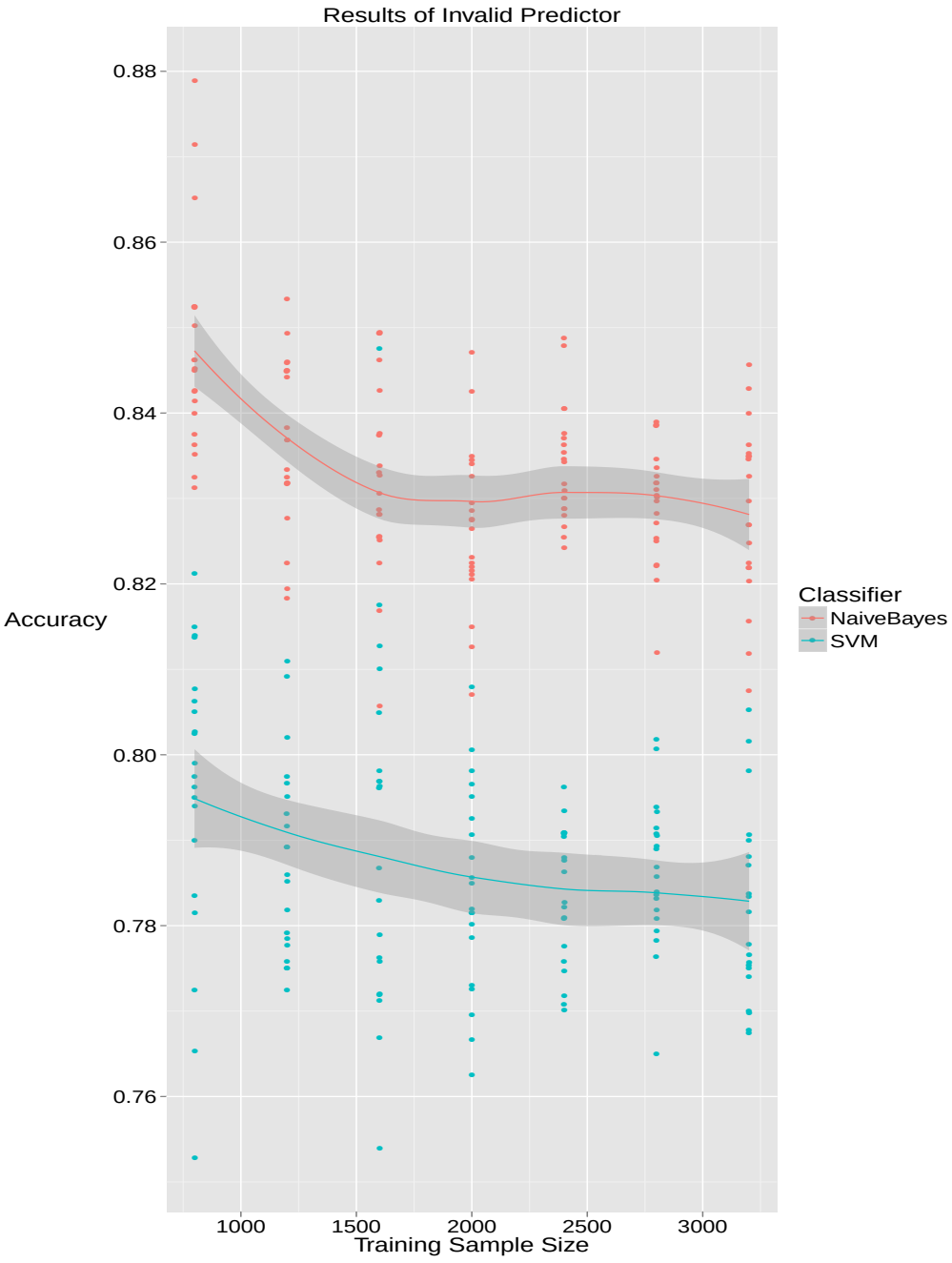

Figure 5.1: Results of Predict Invalid Bugtasks Classifier.

as training set size increases. However, it seems to stabilize around 83% and outperforms SVM by about 5%.

Classifier	**Minimum**	**Mean**	**Max**
Naive Bayes	0.8057	0.8335	0.8789
SVM	0.7528	0.7873	0.8475

Table 5.2: Summary of Results for the Invalid Bugtasks Classifier.

5.4 Automatically Assign Incoming Bugtasks

Because there is a significant time involved from when a bug is created until it is assigned an actual person to look at the case, automating the assignment would prove to be useful and potentially reduce the time it takes on average to complete a bug. For this analysis the bugtask importance, package name, package type, and bug owner were used as predictor variables for the classifier. The target class was the bugtask assignee. With this analysis a few classification algorithms were attempted including: Random Forests, AdaBoost, NaiveBayes and SVM. Because of limitations in AdaBoost and Random Forests not being able to handle larger number of predicted classes, only SVM and Naive Bayes were used.

For each classifier a loop was run varying the overall sample size to determine the performance. This number varied from 1,000 to 4,000 in increments of 500. The partitioning of the validation and training sets was identical to the invalid bug predictor. In addition, the test case was repeated 20 times each. The results for this classifier are graphed in Figure 5.2, and a summary

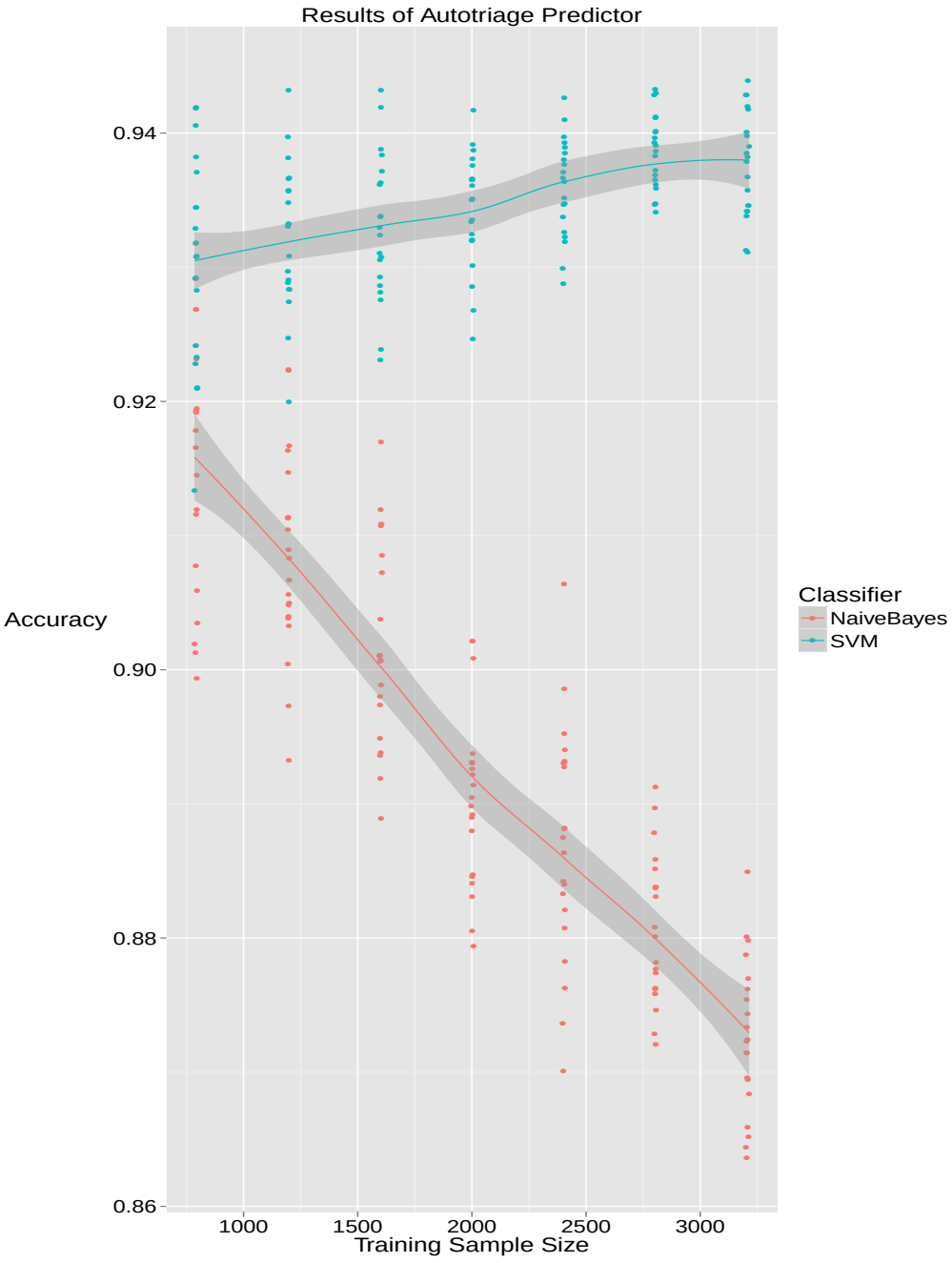

Figure 5.2: Results of the Autotriage Classifier.

is provided in Table 5.3. As it can be seen in 5.2, the SVM classifier accuracy settles around 93% when increasing the training sample size.

Classifier	**Minimum**	**Mean**	**Max**
Naive Bayes	0.8636	0.8937	0.9317
SVM	0.9134	0.9346	0.9439

Table 5.3: Summary of Results for the Autotriage Classifier.

In addition to just predicting the single assignee, the top five predicted classes could be determined by exposing the class prediction probabilities on a particular input. In addition, the number of current 'In Progress' bugtasks are calculated per users, and a number indicating the bug load can be determined for each user. Using this data one could determine how to weight the user whom is best suited for a bugtask and whom is the least busy in order to more effectively assign bugs. Table 5.4 shows an example of the output for one particular case.

original.id	**predict.id**	**predict.probabilty**	**predict.names**	**predict.busy**
219	219	0.47542552	atnan	7
219	33	0.06143801	steven3000	1
219	150	0.04774470	seb128	0
219	27	0.03599966	berni	5
219	114	0.03567299	janitor	8

Table 5.4: Autotriager Ranking.

5.5 Results

Overall, it can be seen that both the autotriager and status predictor classifiers were effective at their respective tasks. These could be useful in helping to more efficiently complete bug tasks and reduce the workload of those that are fixing bugs. The SVM functions were both tuned for each experiment, but there could be other parameters to be adjusted to perhaps improve performance. In addition, the Autotriager was very sensitive to how the data was sampled, and thus care must be taken in obtaining proper sampling to get enough data for the bugtask assignees. In conclusion, these two techniques could be adjusted and implemented to provide useful implementations for defect analysis and Launchpad users.

Chapter 6

Conclusion

Overall 430,832 bugs, 581,276 bugtasks, 152,519 people and 963,370 bugmessages were collected from Launchpad into a database to complete this report. A summary of how Ubuntu, Open Source software, Launchpad and bug reporting was given to sketch the overall domain that was studied in this report. Next, a project was created called 'shadow-database' that provided a means to collect information from the Launchpad web interface into a PostgreSQL database to be used for analysis. This data was then visualized using R and its graphs were discussed in this report. Next, analysis was performed to study if bug reports could be automatically assigned someone to fix it, and if an individual bugtask's final status could be predicting upon its initial creation. Overall these experiments proved successful, and it is hoped that they can be adapted and used.

6.1 Future Work

Because much of this research involved just collecting and assembling the data, there are many more experiments and hypotheses that could be explored. Throughout the course of this paper, Ubuntu defect analysis were

consulted and provided feedback and useful comments for additional data, visualizations, and experiments that would be useful in their work and for the Ubuntu community. Additional visualizations to better examine the release cycles would be very helpful. In addition, examining how many bugs are filed by bots, and examining the effectiveness of such systems. It would be beneficial to adapt the auto-triager to be well integrated into Launchpad itself, and in turn evaluate if it does improve the average days it takes to close a bug. Another experiment would be to more closely study the patterns in number of open bugs per release cycle and predict spikes in bug reports based on historical trends. Because many of the defect analysts rely on calculated scores such as 'heat' to indicate if a bug needs to be prioritized, it may be helpful to better tune this based on analysis. Another helpful function would be in predicting how many days a bugtask will take to complete and thus be able to plan an prioritize fixing the bug. Overall, there is much work that can be done with this dataset. It is hoped that the collection process can be improved and the visualizations and training can occur in such as way to be integrated into the overall process; hopefully, making it more efficient.

Appendix

Appendix 1

Code Listing

1.1 setup.R

```
# lpMiner.setup – Get data from SQL database.
#
# Copyright (c) 2012
# Written by Chris J Arges <christopherarges@gmail.com>
#

# Clean environment.
rm(list=ls())

# Limit the number of entries.
limit<-""

# GATHER data from database.
    ----------------------------------------------------------------------

rename = "bugtask.id AS id,
        bug.id AS bug_id,
        bugtask.assignee AS bugtask_assignee,
        bugtask.date_created AS bugtask_date_created,
        bugtask.date_assigned AS bugtask_date_assigned,
        bugtask.date_closed AS bugtask_date_closed,
        bugtask.date_confirmed AS bugtask_date_confirmed,
        bugtask.date_fix_committed AS bugtask_date_fix_committed,
        bugtask.date_fix_released AS bugtask_date_fix_released,
        bugtask.date_incomplete AS bugtask_date_incomplete,
        bugtask.date_in_progress AS bugtask_date_in_progress,
        bugtask.date_left_closed AS bugtask_date_left_closed,
        bugtask.date_left_new AS bugtask_date_left_new,
        bugtask.date_triaged AS bugtask_date_triaged,
```

```
28            bugtask.owner AS bugtask_owner,
29            bugtask.status AS bugtask_status,
30            bugtask.importance AS bugtask_importance,
31            bugtask.target AS bugtask_target,
32            bugtask.target_name AS bugtask_target_name,
33            bug.date_created AS bug_date_created,
34            bug.date_last_message AS bug_date_last_message,
35            bug.date_last_updated AS bug_date_last_updated,
36            bug.description AS bug_description,
37            bug.duplicate_of AS bug_duplicate_of,
38            bug.heat AS bug_heat,
39            bug.latest_patch_uploaded AS bug_latest_patch_uploaded,
40            bug.message_count AS bug_message_count,
41            bug.number_of_duplicates AS bug_number_of_duplicates,
42            bug.other_users_affected_count_with_dupes AS
                  bug_other_users_affected_count_with_dupes,
43            bug.owner AS bug_owner,
44            bug.tags AS bug_tags,
45            bug.title AS bug_title,
46            bug.users_affected_count AS bug_users_affected_count,
47            bug.users_affected_count_with_dupes AS bug_users_affected_count_with_dupes,
48            bugtarget.name as bugtarget_name"
49
50  sql_query = paste("SELECT * FROM (SELECT", rename, "FROM bugtask ",
51    "JOIN bug ON bugtask.bug = bug.id ",
52    "JOIN bugtarget ON bugtask.target = bugtarget.id ",
53    "ORDER BY bug.id) as bugtaskbug WHERE bug_id > 1",limit)
54
55  person_bugtask_assignee_totals_query = "
56    SELECT person.id as id, person.name as name, count(*) as total FROM person,
          bugtask WHERE bugtask.assignee = person.id GROUP BY person.id ORDER
          BY total DESC"
57
58  person_query = "SELECT * FROM person"
59
60  require(RPostgreSQL)
61  con <- dbConnect(dbDriver("PostgreSQL"), dbname = "shadowdb")
62
63  # Get all closed bugs, and all bugs.
```

```
64 bd_all <- dbGetQuery(con, sql_query)
65
66 # Get all bug statuses/importances.
67 bug_statuses <- dbGetQuery(con, "SELECT * FROM bugstatus")
68 bug_importances <- dbGetQuery(con, "SELECT * FROM bugimportance")
69
70 # Get totals of bugtask assignee totals per person
71 person_stats <- dbGetQuery(con, person_bugtask_assignee_totals_query)
72 people <- dbGetQuery(con, person_query)
73
74 # Get package mapping from .csv file.
75 pkg_mapping<-read.csv("~/Ubuntu One/School/report/project/package-team-
       mapping.csv")
76
77 # Helper Functions
       ------------------------------------------------------------
78
79 # Lookup package type from name.
80 lookup_package <- function(name) {
81   # This function is a bit funky. If we can lookup directly return this,
82   # Otherwise, strip version info and return that.
83   result <- pkg_mapping[which(pkg_mapping$package == name),]
84   if (length(result$team) == 0){
85     f_name = unlist(strsplit(name, "-"))[1] # Strip version stuff.
86     result <- pkg_mapping[which(pkg_mapping$package == f_name),]
87     ret <- as.character(result$team)
88   }
89   ret <- as.character(result$team)
90   if (length(ret) == 0) {
91     return("other") # Default value.
92   }
93   return(ret)
94 }
95
96 conv_to_numeric_days <- function(timediff) { return(as.numeric(trunc(timediff, "day
       "))/60/60/24) }
97 remove_negative_values <- function(x) { return(x[which(x>0)]) }
98 remove_values_below <- function(x, cutoff) { return(x[which(x>cutoff)]) }
```

```
99
100 # From the help page for ?toupper
101 simpleCap <- function(x) {
102       s <- strsplit(x, " ")[[1]]
103       paste(toupper(substring(s, 1,1)), substring(s, 2),
104             sep="", collapse=" ")
105 }
106
107 safeName <- function(x) {
108   s<-gsub(" ","_",tolower(x))
109   return(s)
110 }
111
112 calculate_person_stats_open <- function(person_id, field, value) {
113   all_assigned <- which(bd_open$bugtask_assignee == person_id)
114   return(length(which(bd_open[all_assigned,][[field]] == value)))
115 }
116
117 # CALCULATE additional data.
      ------------------------------------------------------------------------
118
119 calculate_extra_data <- function(bd) {
120   # What are some of the intervals between the bug states?
121   bd$days_from_created_to_close <- conv_to_numeric_days(bd$bugtask_date_closed -
          bd$bugtask_date_created)
122   bd$days_from_created_to_fix_released <- conv_to_numeric_days(
          bd$bugtask_date_fix_released - bd$bugtask_date_created)
123   bd$days_from_created_to_triaged <- conv_to_numeric_days(bd$bugtask_date_triaged
          - bd$bugtask_date_created)
124   bd$days_from_created_to_assigned <- conv_to_numeric_days(
          bd$bugtask_date_assigned - bd$bugtask_date_created)
125   bd$days_from_triaged_to_assigned <- conv_to_numeric_days(
          bd$bugtask_date_assigned - bd$bugtask_date_triaged)
126   bd$days_from_assigned_to_closed <- conv_to_numeric_days(bd$bugtask_date_closed
          - bd$bugtask_date_assigned)
127
128   # Convert into catagorical variables. (sapply would be nice here)
129   for (s in bug_statuses$id) {
```

```
130       bd[[paste("is_",safeName(bug_statuses[s,]$name),sep="")]] <-
              bd$bugtask_status == s
131     }
132
133     # Which days of the week are these transitions most likely?
134     bd$bugtask_weekday_created <- weekdays(bd$bugtask_date_created)
135     bd$bugtask_weekday_closed <- weekdays(bd$bugtask_date_closed)
136     bd$bugtask_weekday_assigned <- weekdays(bd$bugtask_date_assigned)
137
138     # Calculate the numbers of letters in the title and description.
139     bd$description_wc <- nchar(bd$bug_description)
140     bd$title_wc <- nchar(bd$bug_title)
141
142     # Create package mapping row
143     bd$package_type <- sapply(bd$bugtarget_name, lookup_package)
144
145     # return modified value
146     return(bd)
147 }
148
149 bd_all <- calculate_extra_data(bd_all)
150 bd_closed <- bd_all[which(!is.na(bd_all$bugtask_date_closed)),]
151 bd_open <- bd_all[which(is.na(bd_all$bugtask_date_closed)),]
152
153 # determine current load for person stats
154 person_stats[["total_open_cases"]] <- sapply(person_stats$id, y <- function(
        person_id) {length(which(bd_open$bugtask_assignee == person_id))})
155 person_stats[["total_in_progress_cases"]] <- sapply(person_stats$id,
        calculate_person_stats_open, field="bugtask_status", value=9)
```

1.2 explore.R

```
1 # Launchpad Data mining in R - Data Exploration
2 #
3 # Copyright (c) 2012
4 # Written by Chris J Arges <christopherarges@gmail.com>
5 #
6
```

```
7  # Parameters for visualizations.
8  log_cols <- c("bug_message_count", "bug_users_affected_count", "
       bug_users_affected_count_with_dupes",
9    "bug_number_of_duplicates", "description_wc", "title_wc", "
         days_from_created_to_close",
10   "days_from_created_to_triaged", "days_from_assigned_to_closed",
11   "days_from_triaged_to_assigned", "days_from_created_to_fix_released")
12 reg_cols <- c("bugtask_status", "bugtask_importance")
13 all_cols <- c(log_cols, reg_cols, "is_invalid", "is_fix_released")
14
15 # Colors
16 status_colors = list(
17   "Fix Released"="#008000", "Fix Committed"="#005500", "In Progress"="
         #160000",
18   "Triaged"="#FF6600", "Confirmed"="#FF0000", "Won't Fix"="#444444",
19   "Invalid"="#666666", "Opinon"="#0033AA", "Incomplete"="#FF0000", "New"
         ="#993300"
20 )
21
22 importance_colors = list(
23   "Unknown"="#999999", "Critical"="#FF0000", "High"="#FF660E",
24   "Medium"="#008007", "Low"="#270000", "Wishlist"="#0000FF", "Undecided
         "="#999999"
25 )
26
27 # WORD CLOUD
       ------------------------------------------------------------------

28 # Warning: This is REALLY slow for large datasets.
29 # Adapted from http://addictedtor.free.fr/graphiques/RGraphGallery.php?graph=162
30 # http://www.r-bloggers.com/using-text-mining-to-find-out-what-
       rdatamining-tweets-are-about/
31 do_wordcloud <- function(data, max_terms=300) {
32   require(tm)
33   require(wordcloud)
34   require(RColorBrewer)
35   str.corpus = Corpus(VectorSource(data))
36   str.corpus <- tm_map(str.corpus, removePunctuation)
37   str.corpus <- tm_map(str.corpus, removeNumbers)
```

```
38    str.corpus <- tm_map(str.corpus, function(x)removeWords(x,stopwords()))
39    str.corpus <- tm_map(str.corpus, tolower)
40    tdm <- TermDocumentMatrix(str.corpus)
41    m <- as.matrix(tdm)
42    v <- sort(rowSums(m),decreasing=TRUE)
43    v.r <- v[seq(1,max_terms)] # reduce the number of terms
44    d <- data.frame(word = names(v.r),freq=v.r)
45    wordcloud(d$word,d$freq,max.words=max_terms)
46  }
47
48  # HISTOGRAMS
        ------------------------------------------------------------------------

49  do_histogram <- function(bd, column, title) {
50    require(ggplot2)
51    hist_data <- na.omit(data.frame(data=bd[[column]], status=factor(bug_statuses[
          bd[["bugtask_status"]],]$name,names(status_colors)) ))
52    hist_data <- hist_data[which(hist_data$data > 0),]
53    s<-summary(hist_data$data)
54    return(
55    ggplot(hist_data, aes(x=data,fill=status)) +
56      geom_bar() + opts(title=paste(title, "( Mean:", s[["Mean"]], ")")) + labs(y="
            Count", x=title) +
57      scale_fill_manual(limits=names(status_colors), values=as.character(status_colors))
58    )
59  }
60
61  # HISTORICAL graph
62  do_historical <- function() {
63    require(stats)
64    require(lubridate)
65    require(gridExtra)
66    require(ggplot2)
67
68    # Find minimal/maximal date
69    # TODO: Ensure minimal date is : 2003-12-14 20:41:00, found bogus data.
70    start_date = trunc(min(bd_all$bugtask_date_created), "day")
71    end_date = trunc(max(bd_closed$bugtask_date_closed), "day")
72
```

```
73  # Create the historical data
74  y <- function(dt) {
75    z<- length(which(
76      (bd_all$bugtask_date_created <= dt & dt < bd_all$bugtask_date_closed) |
77      (bd_all$bugtask_date_created <= dt & is.na(bd_all$bugtask_date_closed))
78    ))
79  }
80
81  # Create bug history.
82  bd_history <- data.frame(date = seq(from = start_date, to = end_date, by = "
        days"))
83  bd_history$total <- lapply(bd_history$date, y)
84
85  # Get person data
86  people_history <- data.frame(date = seq(from = start_date, to = end_date, by =
        "days"))
87  people_history$total <- lapply(people_history$date, function(dt) { z<- length(
        which(people$date_created <= dt)) } )
88
89  # Create Labels for Release Dates
90  release_dates <- as.POSIXct(c("2004-10-20", "2005-04-08", "2005-10-13",
        "2006-06-01",
91    "2006-10-26", "2007-04-19", "2007-10-18", "2008-04-24", "
          2008-10-30","2009-04-23",
92    "2009-10-29","2010-04-29","2010-10-10","2011-04-28","2011-10-13",
          "2012-04-26","2012-10-18"))
93  release_names <- c("warty","hoary","breezy","dapper","edgy","feisty","gutsy","
        hardy","intrepid",
94    "jaunty","karmic","lucid","maverick","natty","oneiric","precise","quantal")
95  rls.df <- data.frame(date=release_dates, name=release_names)
96
97  # http://sape.inf.usi.ch/quick-reference/ggplot2/geom_vline
98  # Plot data
99  p1<-ggplot(bd_history, aes(x=date, label=date)) + geom_line(aes(y=as.numeric(
        total), col="Open Bugs")) +
100   geom_line(data=people_history, aes(y=as.numeric(total),col="LP Users")) +
101   geom_text(data=rls.df, mapping=aes(x=as.numeric(date), y=0, label=name),
          angle=90, size=4, vjust=-0.4, hjust=0) +
102   opts(title="All Open Bugs Historically") + labs(y="Count", x="Date") +
```

```
103       geom_vline(xintercept=as.numeric(release_dates), color="gray", linetype="dashed
              ") +
104       scale_colour_discrete(name="Data")
105     plot(p1)
106   }
107
108   do_correlation <- function() {
109     require(corrplot)
110     cols <- c("bug_message_count", "bug_users_affected_count", "
              bug_users_affected_count_with_dupes",
111       "bug_number_of_duplicates", "description_wc", "title_wc", "
                  days_from_created_to_close","bugtask_importance",
112       "is_invalid", "is_won't_fix", "is_expired","is_fix_released")
113     bd_c.corr <- cor(bd_closed[cols])
114     corrplot(bd_c.corr, method="shade")
115   }
116
117   do_find_most_frequent_tags <- function(N, bd) {
118     # Get non-empty tags
119     bd.non_empty <- which(bd$bug_tags != "")
120     bd.tags <- bd[bd.non_empty,]$bug_tags
121     tags_list <- unlist(strsplit(bd.tags,perl=TRUE, ","))
122     return (head(sort(table(tags_list),decreasing=TRUE),N))
123   }
124
125   panel_values <- function(...) {
126     # http://stackoverflow.com/questions/3220702/display-values-in-stacked-
              lattice-barchart
127     # Rather convoluted way to add value labels to a barchart.
128     panel.barchart(...)
129     tmp <- list(...)
130     tmp <- data.frame(x=tmp$x, y=tmp$y)
131     # calculate positions of text labels
132     df <- ddply(tmp, .(y),function(x) {
133       data.frame(x, pos=0)
134     })
135     panel.text(x=df$pos, y=df$y,label=df$x,cex=0.8,pos=2,side=4)
136   }
137
```

```
138 do_plot_barchart <- function(data,title,ylab,xlab) {
139   df<-data.frame(names=factor(names(data),names(data)),value=data)
140   p<-ggplot(df, aes(x=names,y=value,fill=names)) + geom_bar() +
141     geom_text(mapping=aes(x=names, y=0, label=value), size=3, vjust=2, hjust
          =0.5) +
142     opts(title=title) + labs(y=ylab, x=xlab) + guides(fill=FALSE)
143   return(p)
144  }
145
146 do_treemap <- function() {
147   require(portfolio)
148   data<-data.frame(bd_closed)
149   map.market(id=data[["id"]], area=data[["days_from_created_to_close"]]+1, group=
        as.character(data[["package_type"]]), color=data[["bugtask_importance"]], main
        ="Bugs")
150 }
151
152 do_plots <- function() {
153   require(reshape)
154   require(gridExtra)
155
156   # Histogram Plots
157   p1<-do_histogram(bd_all, "bug_message_count","Bug Messages Count") +
        scale_x_log10(breaks=c(0,1,2,5,10,20,50,100,1000)) + guides(fill=FALSE)
158   p2<-do_histogram(bd_all, "bug_users_affected_count_with_dupes", "Users Affected"
        ) + scale_x_log10(breaks=c(0,1,2,5,10,20,50,100,1000)) + guides(fill=FALSE)
159   p3<-do_histogram(bd_all, "bug_number_of_duplicates", "Number of Duplicates")
        + scale_x_log10(breaks=c(0,1,2,5,10,20,50,100,1000)) + guides(fill=FALSE)
160   p4<-do_histogram(bd_all, "description_wc", "Description Character Count") +
        scale_x_log10(breaks=c(3,50,300,1000,10000,50000)) + guides(fill=FALSE)
161   p5<-do_histogram(bd_all, "title_wc", "Title Character Count") + scale_x_log10(
        breaks=c(2,10,25,50,100,250,500)) + guides(fill=FALSE)
162   p6<-do_histogram(bd_all, "bug_heat", "Bug Heat") + scale_x_log10(breaks=c
        (2,10,25,250))
163   grid.arrange(p1,p2,p3,p4,p5,p6,nrow=3,ncol=2)
164
165   p7<-(do_histogram(bd_closed, "days_from_created_to_close", "Days From Created
        To Closed")+scale_x_log10(breaks=c(0,1,2,5,10,30,125,2500))) + guides(fill=
        FALSE)
```

```
166 p8<-(do_histogram(bd_closed, "days_from_created_to_assigned", "Days From
        Created To Assigned")+scale_x_log10(breaks=c(0,0.01,0.25,2,50,2000))) +
        guides(fill=FALSE)
167 p9<-(do_histogram(bd_closed, "days_from_assigned_to_closed", "Days From
        Assigned To Closed")+scale_x_log10(breaks=c(0,1,2,10,100,2000))) + guides(
        fill=FALSE)
168 p10<-(do_histogram(bd_closed, "days_from_created_to_triaged", "Days From
        Created To Triaged")+scale_x_log10(breaks=c(0,0.01,0.25,2,70,2000))) +
        guides(fill=FALSE)
169 p11<-(do_histogram(bd_closed, "days_from_triaged_to_assigned", "Days From
        Triaged To Assigned")+scale_x_log10(breaks=c(0,0.01,3,25,500))) + guides(fill
        =FALSE)
170 p12<-(do_histogram(bd_closed, "days_from_created_to_fix_released", "Days From
        Created To Fix Released") + scale_x_log10(breaks=c(0,1,5,25,150,2000)))
171 grid.arrange(p7,p8,p9,p10,p11,p12,nrow=3,ncol=2)
172
173 do_correlation()
174
175 weekdays <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "
        Saturday", "Sunday")
176 clt<-table(bd_closed$bugtask_weekday_created)[weekdays]
177 crt<-table(bd_closed$bugtask_weekday_closed)[weekdays]
178
179 a<-data.frame(Weekday=factor(weekdays, weekdays), Closed=as.vector(clt),
        Created=as.vector(crt))
180 mx<-melt(a, id.vars=1)
181 ggplot(mx, aes(x=Weekday, y=value, fill=variable)) + geom_bar(stat="identity",
        position=position_dodge()) + scale_colour_discrete(name="Bug Activity") +
182     opts(title="Closed Bug Activity Per Day of Week") + labs(y="Count", x="
            Weekday")
183
184 # bar plots
185 p1<-do_plot_barchart(head(sort(table(as.factor(as.character(bd_all[["package_type"
        ]])))),decreasing=TRUE),10),"Most Frequent Bugs Per Team Type","Count","
        Team Type")
186 p2<-do_plot_barchart(sort(table(bug_statuses[bd_all[["bugtask_status"]],]$name),
        decreasing=TRUE),"Number of Bugtasks By Status","","Status")
187 p3<-do_plot_barchart(sort(table(bug_importances[bd_all[["bugtask_importance"]],]
        $name),decreasing=TRUE),"Number of Bugtasks By Importance","Count","
```

```
        Importance")
188   vals_all<-do_find_most_frequent_tags(10, bd_all)
189   p4<-do_plot_barchart(vals_all,"Most Frequent Tags in All Bugs","","Bug Tag")
190   grid.arrange(p1,p2,nrow=2)
191   grid.arrange(p3,p4,nrow=2)
192 }
193
194 # Do visualizations here!
      ----------------------------------------
195
196 # single or multiple plots.
197 pdf(file="plot%03d.pdf",onefile=FALSE,paper="special",pointsize=9,width=10,
      height=16)
198 #pdf(file="plot.pdf", pointsize=9, width=16,height=12)
199
200 do_plots()
201 do_historical()
202
203 # the following plots take a long time to produce (these CAN lock up your machine)
204 #do_treemap()
205 #do_wordcloud(bd_all)
206
207 dev.off()
```

1.3 analysis.R

```
1  # lpMiner.Analysis - Launchpad Data mining in R
2  #
3  # Copyright (c) 2012
4  # Written by Chris J Arges <christopherarges@gmail.com>
5  #
6  require(caret)
7  require(klaR)
8  require(e1071)
9  require(ggplot2)
10 require(tm)
11 require(gridExtra)
12
```

```
13 fieldsUsed<-c("bugtask_assignee", "bugtask_importance", "bugtask_status", "
       bugtarget_name",
14   "package_type","bug_owner","bug_id","bug_description","bug_title")
15
16 # Autotriager
17 a.X=c("bugtask_importance", "bugtarget_name","package_type", "bug_owner")
18 a.Y=c("bugtask_assignee")
19 a.formula=bugtask_assignee ~ bugtask_importance + bugtarget_name + package_type
       + bug_owner
20
21 # Status classification
22 c.X=c("bugtask_importance", "bugtarget_name", "package_type", "bug_owner")
23 c.Y=c("bugtask_status")
24 c.formula=bugtask_status ~ bugtask_importance + bugtarget_name + package_type +
       bug_owner
25
26 get_person <- function(id,field) {
27   return(person_stats[which(person_stats$id == id),][[field]])
28 }
29 get_person_name <- function(id) {
30   return(person_stats[which(person_stats$id == id),]$name)
31 }
32
33 #
34 # parse_data - format dataset for analysis
35 #
36 parse_data <- function(input, sampleSize=5000) {
37   # cut down to the sample size
38   data<- input[sample(1:nrow(input), sampleSize, replace=FALSE),]
39
40   # Format data so it's ready.
41   dataAll <- data[fieldsUsed] # filter it down to only the used fields
42   dataAll$bugtask_importance <- as.factor(dataAll$bugtask_importance)
43   dataAll$bug_owner <- as.factor(dataAll$bug_owner)
44   dataAll$bugtask_assignee <- as.factor(dataAll$bugtask_assignee)
45   dataAll$bugtask_status <- as.factor(dataAll$bugtask_status)
46   dataAll$bugtarget_name <- as.factor(dataAll$bugtarget_name)
47   dataAll$package_type <- as.factor(as.character(dataAll$package_type))
48
```

```
49    # Return the parsed data, plus the partitions.
50    return (dataAll)
51  }
52
53  #
54  # analyze.nb - Generalized naive bayes analysis function
55  #
56  analyze.nb <- function(train, validation, formula, X, Y, laplace)
57  {
58    # Create NB model
59    model <- naiveBayes(formula, train, laplace=laplace)
60
61    # Measure performance of model against validation data.
62    results <- predict(model, validation[,-match(Y,names(validation))], class="raw")
63
64    # Determine results.
65    confusion<-confusionMatrix(results, validation[[Y]])
66
67    # return a list with the model and confusion matrix.
68    return(list(model=model,results=results,confusion=confusion))
69  }
70
71
72  #
73  # analyze.svm - Generalized SVM analysis function
74  #
75  analyze.svm <- function(train, validation, formula, X, Y, gamma, cost)
76  {
77    #http://stackoverflow.com/questions/7782501/how-to-interpret-predict-result-
          of-svm-in-r
78    # Create SVM model. gamma,cost determined by train.svm
79    model <- svm(formula, train, gamma=gamma, cost=cost)
80
81    # Get probabilities of predictions.
82    results <- predict(model, validation[,-match(Y,names(validation))])
83
84    # Determine results.
85    confusion<-confusionMatrix(results, validation[[Y]])
86
```

```
87    # return a list with the model and confusion matrix.
88    return(list(model=model,results=results,confusion=confusion))
89  }
90
91  # autotriage.svm - autotriage using svm
92  autotriage.svm <- function(train, validation) {
93    return (analyze.svm(train, validation, formula=a.formula,X=a.X,Y=a.Y, gamma
          =0.1, cost=10))
94  }
95
96  # autotriage.nb - autotriage using nb
97  autotriage.nb <- function(train, validation) {
98    return (analyze.nb(train, validation, formula=a.formula, X=a.X, Y=a.Y, laplace=0)
          )
99  }
100
101 # classifystatus.svm - predict ending status using svm
102 classifystatus.svm <- function(train, validation) {
103   return (analyze.svm(train, validation, formula=c.formula, X=c.X, Y=c.Y, gamma
          =0.01, cost=10))
104 }
105
106 # classifystatus.nb - predict ending status using nb
107 classifystatus.nb <- function(train, validation) {
108   return (analyze.nb(train, validation, formula=c.formula, X=c.X, Y=c.Y, laplace=1))
109 }
110
111 # testing functions
      ------------------------------------------------------------


112
113 do_tests <- function(testFunction, function_name, dataTrain, dataValidation, s) {
114     print(paste("test", s, function_name, dim(dataTrain)[1], dim(dataValidation)[1] ))
115     e_time <- system.time(results<-testFunction(dataTrain,dataValidation)
            $confusion$overall)
116     accuracy_total <- as.numeric(results["Accuracy"])
117     time_total <- as.numeric(e_time[1])
118     ret<-c(dim(dataTrain)[1], as.numeric(accuracy_total),as.numeric(time_total))
119     return(ret)
```

```
120 }
121
122 do_sample_comparison_a <- function(sampleSizes=seq(1000,8000,by=1000)) {
123
124   df.a.nb<-data.frame(row.names=c("sample_size","accuracy"))
125   df.a.svm<-data.frame(row.names=c("sample_size","accuracy"))
126
127   for (s in sampleSizes) {
128     print(paste("sampleSize",s))
129
130     dataAll <- parse_data(bd_closed, sampleSize=s)
131     data.parts <- createDataPartition(dataAll[[a.Y]],2, p=0.8)
132     dataTrain <- dataAll[data.parts$Resample2,]
133     dataValidation <- dataAll[data.parts$Resample1,]
134     dataTrain <- dataTrain[c(a.Y,a.X)]
135     dataValidation <- dataValidation[c(a.Y,a.X)]
136
137     df.a.svm <- rbind(df.a.svm, c(do_tests(autotriage.svm,"autotriage.svm",
          dataTrain, dataValidation, s)))
138     df.a.nb <- rbind(df.a.nb, c(do_tests(autotriage.nb,"autotriage.nb", dataTrain,
          dataValidation, s)))
139
140   }
141
142   return(list(a.nb=df.a.nb, a.svm=df.a.svm))
143 }
144
145 do_sample_comparison_c <- function(sampleSizes=seq(1000,9000,by=1000)) {
146
147   df.c.nb<-data.frame(row.names=c("sample_size","accuracy"))
148   df.c.svm<-data.frame(row.names=c("sample_size","accuracy"))
149
150   for (s in sampleSizes) {
151     print(paste("sampleSize",s))
152
153     dataAll <- parse_data(bd_closed, sampleSize=s)
154     data.parts <- createDataPartition(dataAll[[c.Y]],2, p=0.8)
155     dataTrain <- dataAll[data.parts$Resample2,]
156     dataValidation <- dataAll[data.parts$Resample1,]
```

```
157       dataTrain <- dataTrain[c(c.Y,c.X)]
158       dataValidation <- dataValidation[c(c.Y,c.X)]
159
160       df.c.nb <- rbind(df.c.nb, c(do_tests(classifystatus.nb,"classifystatus.nb",
              dataTrain, dataValidation,s)))
161       df.c.svm <- rbind(df.c.svm, c(do_tests(classifystatus.svm,"classifystatus.svm",
              dataTrain, dataValidation,s)))
162     }
163
164     return(list(c.nb=df.c.nb, c.svm=df.c.svm))
165   }
166
167   do_visualize <- function(results, title="Results") {
168     res<-results
169     colnames(res) <- c("sample","accuracy","time")
170     p<-ggplot(res, aes(x=sample,y=accuracy)) + geom_point() + geom_smooth(
            method=lm) +
171       scale_x_continuous(name="Training Sample Size") + scale_y_continuous(name="
              Accuracy") +
172       opts(title=title)
173     return(p)
174   }
175
176   #
177   # get_top_results - used for parsing the output from the model and returning the
178   # top probabilities for the bugtask_assignees
179   #
180   get_top_results <- function(x, validation, N=5) {
181     probs<-attr(x$results,"probabilities")
182     valid<-validation
183
184     myret<-data.frame()
185     df <- data.frame(probs)
186     for (r in row.names(probs)) {
187       row <- probs[which(row.names(probs) == r),]
188       top<-head(sort(row,decreasing=TRUE),N)
189
190       ret<-data.frame(
191         id=r,
```

```
192         original.id=valid[r,]$bugtask_assignee,
193         predict.id=names(top),
194         predict.probabilty=as.numeric(top),
195         predict.names=unlist(lapply(names(top), get_person_name)),
196         predict.busy=unlist(lapply(names(top), get_person,field="
                total_in_progress_cases")))
197       myret<-rbind(myret,ret)
198     }
199     return(myret)
200   }
201
202   # to get the visualizations and perform the experiments just run this.
203   generate_graphs <- function() {
204     pdf(file="results%03d.pdf",onefile=FALSE,paper="special",pointsize=9,width=10,
          height=16)
205     xc<-do_sample_comparison_c(rep(seq(1000,10000,by=500),each=20))
206     xa<-do_sample_comparison_a(rep(seq(1000,10000,by=500),each=20))
207     p1<-do_visualize(xc$c.svm, "Results of SVM Status Classifier")
208     p2<-do_visualize(xc$c.nb, "Results of Naive Bayes Status Classifier")
209     grid.arrange(p1,p2,nrow=2)
210     dev.off()
211   }
```

Bibliography

[1] About ubuntu our philosophy.
http://www.ubuntu.com/project/about-ubuntu/our-philosophy.

[2] Bug #1004707 - launchpad bug entry.
https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1004707.

[3] Bugs - how to triage - charts - ubuntu wiki.
https://wiki.ubuntu.com/Bugs/HowToTriage/Charts.

[4] Bugs - your project - launchpad help.
https://help.launchpad.net/Bugs/YourProject.

[5] Cost of linux.
http://linuxcost.blogspot.com/2011/03/cost-of-linux.html.

[6] Distrowatch.com. http://distrowatch.com/.

[7] Launchpad - bug heat.
https://bugs.launchpad.net/+help-bugs/bug-heat.html.

[8] Launchpad blog - reimagining the nature of privacy in launchpad (part 1). http://blog.launchpad.net/general/reimagining-the-nature-of-privacy-in-launchpad-part-1.

[9] Launchpad bugs. https://bugs.launchpad.net/.

[10] Launchpad help - legal. https://help.launchpad.net/Legal.

[11] Launchpad tour. https://launchpad.net/+tour/index.

[12] Launchpad web service api documentation. https://launchpad.net/+apidoc/.

[13] Launchpad, your account, karma. https://help.launchpad.net/YourAccount/Karma/.

[14] Reportingbugs - community ubuntu documentation. https://help.ubuntu.com/community/ReportingBugs.

[15] Storm in launchpad. https://launchpad.net/storm.

[16] Ubuntu for you. http://www.ubuntu.com/ubuntu.

[17] Ubuntu wiki - lts. https://wiki.ubuntu.com/LTS/.

[18] Ubuntu wiki - tags. https://wiki.ubuntu.com/Bugs/Tags.

[19] Usage statistics and market share of linux for websites. http://w3techs.com/technologies/details/os-linux/all/all.

[20] J. Anvik, L. Hiew, and G.C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[21] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful really? In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 337–345. IEEE, 2008.

[22] G. Bougie, C. Treude, D.M. German, and M.A. Storey. A comparative exploration of freebsd bug lifetimes. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 106–109. IEEE, 2010.

[23] M. Conklin. Beyond low-hanging fruit: Seeking the next generation in floss data mining. *Open Source Systems*, pages 47–56, 2006.

[24] J. Howison, M. Conklin, and K. Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering (IJITWE)*, 1(3):17–26, 2006.

[25] G. Madey, V. Freeh, and R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Americas conf. on Information Systems (AMCIS2002)*, pages 1806–1813, 2002.

[26] J. Park, M.W. Lee, J. Kim, S. Hwang, and S. Kim. Costriage: A cost-aware triage algorithm for bug reporting systems. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[27] B. Perens et al. The open source definition. *Open sources: voices from the open source revolution*, pages 171–188, 1999.

[28] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[29] R. Stallman et al. The free software definition. *Free Software Free Society: Selected Essays of Richard M. Stallman*, pages 41–44, 1996.

[30] C.P. Team. Capability maturity model® integration (cmmi sm), version 1.1. *Software Engineering Institute, Carnegie Mellon University/SEI-2002-TR-012. Pittsburg, PA*, 2002.

[31] D.A. Wheeler. More than a gigabuck: Estimating gnu/linuxs size, 2001.

[32] M.W. Wu and Y.D. Lin. Open source software development: an overview. *Computer*, 34(6):33–38, 2001.

[33] T. Xie, J. Pei, and A.E. Hassan. Mining software engineering data. In *Software Engineering-Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 172–173. IEEE, 2007.