2012 IEEE 23rd International Symposium on Software Reliability Engineering

An Empirical Study of Bugs in Machine Learning Systems

Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang
School of Information Systems
Singapore Management University, Singapore
{*ferdianthung,shaoweiwang.2010,davidlo,lxjiang*}*@smu.edu.sg*

***Abstract*—Many machine learning systems that include various data mining, information retrieval, and natural language processing code and libraries are used in real world applications. Search engines, internet advertising systems, product recommendation systems are sample users of these algorithm-intensive code and libraries. Machine learning code and toolkits have also been used in many recent studies on software mining and analytics that aim to automate various software engineering tasks. With the increasing number of important applications of machine learning systems, the reliability of such systems is also becoming increasingly important. A necessary step for ensuring reliability of such systems is to understand the features and characteristics of bugs occurred in the systems. A number of studies have investigated bugs and fixes in various software systems, but none focuses on machine learning systems. Machine learning systems are unique due to their algorithm-intensive nature and applications to potentially large-scale data, and thus deserve a special consideration.**

In this study, we fill the research gap by performing an empirical study on the bugs in machine learning systems. We analyze three systems, Apache Mahout, Lucene, and OpenNLP, which are data mining, information retrieval, and natural language processing tools respectively. We look into their bug databases and code repositories, analyze a sample set of bugs and corresponding fixes, and label the bugs into various categories. Our study finds that 22.6% of the bugs belong to the algorithm/method category, 15.6% of the bugs belong to the non-functional category, and 13% of the bugs belong to the assignment/initialization category. We also report the relationship between bug categories and bug severities, the time and effort needed to fix the bugs, and bug impacts. We highlight several bug categories that deserve attention in future research.

I. Introduction

Many real systems, such as search engines, social networks, and online library services, employ data mining, information retrieval, natural language processing, and many other algorithm-intensive solutions. In software engineering research, many machine learning solutions are also used to find bugs [22], localize bugs [7], recommend fixes [16], [30], detect duplicate bug reports [15], [36], mine specifications [29], and many more. Analyzing and improving machine learning systems has potentially much impact.

Bugs are prevalent in software systems. Machine learning systems are no exceptions. Bugs could adversely affect these solutions; they could give wrong output, crash without running to completion, or runs too slow for it to be usable by end users or systems. Preventing, detecting, and resolving bugs in algorithm-intensive solutions are thus important.

An important step to reduce bugs is to analyze historical bugs that have occurred in a system. Post-mortem analysis of bugs for understanding various characteristics of various bugs [8], [9], [24], [26], [27], [42] can have many benefits. It can provide knowledge for guiding the design of bug detection tools, triaging bug reports, locating likely bug locations, suggesting possible fixes, gauging testing and debugging costs, measuring software quality, and helping to monitor and manage development processes.

To better understand the nature of bugs in machine learning systems and to potentially help to prevent, resolve, mitigate, or manage such bugs, this paper performs an exploratory study of *real* bugs from such systems. There have been a number of studies that investigate defects[1] in other kinds of software systems [9], [21], [24], [31], [32], [34], [42]. However, none so far focuses on machine learning systems. Many software systems studied in prior studies, such as email clients (e.g., Columba), text editors (JEdit), integrated development environments (e.g., Eclipse), web browsers (e.g., Mozilla), are primarily designed to render user interfaces and handle user interactions. The user interface-intensive systems are inherently different from data mining, information retrieval, and natural language processing systems that are algorithm-intensive, often repeatedly run for a long period of time, and deal with large-scale datasets.

In this study, we analyze three algorithm-intensive machine learning systems/libraries: (1) Apache Mahout, a library of data mining solutions to analyze large scale data by employing parallelization, (2) Apache Lucene, a library of information retrieval solutions that support various ways to retrieve relevant documents from a textual corpora given a query efficiently, and (3) Apache OpenNLP, a toolkit of various natural language processing (NLP) solutions.

To investigate bugs that appear in the three systems, we analyze their bug repositories. The three libraries developed under Apache foundation use JIRA—a commercial bug/issue tracking system—to store the reported bugs. We choose libraries from Apache as their JIRA bug repositories typically contain links from bug reports to a list of changes in the corresponding source control repositories that fix those bugs. Our initial manual investigation finds that most of the links are correct and most of the closed bug reports

[1]In this paper, we use the term defects and bugs interchangeably.

1071-9458/12 $26.00 © 2012 IEEE
DOI 10.1109/ISSRE.2012.22

are linked to all their corresponding changes. We manually analyze a randomly selected sample of bugs to investigate the nature of bugs in these systems.

Our study aims to answer a number of research questions including: How often bugs appear in machine learning systems? What kinds of bugs appear in such systems? How severe are various categories of bugs? How long does it take to fix various categories of bugs? How many files are impacted by various category of bugs. We answer the above questions by performing both manual and automated analysis on a randomly sampled set of closed bug reports and their corresponding bug fixing commits.

Our contributions are as follows:

1) We are the first to perform a large scale, semi-automated analysis of bugs in machine learning systems which are algorithm-intensive and should be considered differently from user interface-intensive systems.
2) We manually categorize 500 bug reports and their fixes from three machine learning systems into various categories by using the bug categorization in [34].
3) We investigate the relationships between bug categories and bug severities, bug fixing time and effort, and bug impact to the system. To the best of our knowledge, we are the first to investigate such relationships for a variety of bug categories. We believe our analysis of the bugs would provide guidance for preventing, detecting, mitigating, resolving, and managing bugs in similarly algorithm-intensive machine learning systems.

The rest of this paper is structured as follows. In Section II we describe our dataset and methodology. Section III presents our empirical study which describes our research questions and their answers. Section IV describes related work. We conclude and mentions future work in Section V.

II. Dataset & Methodology

In this section, we describe the three machine learning systems used in this study. We then present our methodology.

A. Dataset

Apache Mahout is a library that provides various data mining algorithms [2]. Various algorithms are implemented in the library including clustering, classification, collaborative filtering, and frequent pattern mining. The goal of the project is to provide scalable data mining solution. Mahout realizes this by building the algorithms on top of Apache Hadoop which is an open source implementation of the Map-Reduce framework. It has been developed since Jan 2008 based on the first entry in the source code repository. Its latest release at the end of May 2012 is version 0.6. It consists of 175,295 lines of code and 1,251 Java files (on May 23, 2012). To analyze bugs that are reported for Mahout, we analyze its JIRA issue repository located at:

https://issues.apache.org/jira/browse/MAHOUT

For our manual analysis, we randomly pick 200 bug reports out of more than 1000 issues of various types (e.g., bugs, improvements, new features, tasks, tests, wishes, brainstorming, etc.) contained in the JIRA repository of Mahout.

Apache Lucene is an information retrieval library written in Java [1]. It supports full-featured high performance retrieval of documents from textual corpora. It has various features including scalable indexing and powerful search algorithms. It has been developed since Sep 2001 and the latest release at the end of May 2012 is version 3.6. It consists of 554,036 lines of code and 2,564 Java files (on May 23, 2012). To analyze bugs that are reported for Lucene, we analyze its JIRA issue repository located at:

https://issues.apache.org/jira/browse/LUCENE

For our manual analysis, we randomly pick 200 bug reports out of more than 4000 issues contained in the JIRA repository of Lucene.

Apache OpenNLP is a "machine learning based toolkit for the processing of natural language text" [3]. It supports various natural language processing tasks including tokenization, segmentation, chunking, parsing, part-of-speech tagging, named entity resolution, and many more. It has been developed since Dec 2010 and the latest release at the end of May 2012 is 1.5.2. It consists of 78,224 lines of code and 697 Java files. To analyze bugs that are reported for Mahout, we analyze its JIRA issue repository located at:

https://issues.apache.org/jira/browse/OPENNLP

For our manual analysis, we randomly pick 100 bug reports out of more than 500 issues contained in the JIRA repository of OpenNLP.

We analyze snapshots of JIRA repositories of Mahout, Lucene, and OpenNLP dated up to 11 May 2012. We focus on only *closed* bugs from the JIRA repositories, as bug reports that are not closed may not be bugs or have no fixes or enough information for our analysis yet. Table I shows the numbers of closed bugs for Mahout, Lucene, and OpenNLP. We also show the durations (in years) between the first and last bugs in Column "Duration."

Table I
Numbers of Closed Bugs in Machine Learning Systems

Application	Size – Lines of Code	Bug Count	Duration
Mahout	175,295 (Version 0.6)	314	4.28 years
Lucene	554,036 (Version 3.6)	1,533	10.59 years
OpenNLP	78,224 (Version 1.5.2)	113	1.39 years

B. Methodology

Our empirical study consists of several steps:

1) *Data Collection.* We download the bug repositories and source control repositories of the three software systems. For many software systems, these two repositories are often maintained separately and thus are not properly linked (c.f. [41]). Fortunately, for many

Table II
BUG CATEGORIES BASED ON CODE DEFECT TYPES IN [34]

Category	Definition
algorithm/method	The implementation of an algorithm/method does not follow the expected behavior.
assignment/initialization	Error in assigning variable values.
checking	Missing necessary checks that lead to an error or a wrong error message.
data	Wrong use of data structure.
external interface	Error in interfacing with other systems or users, such as using deprecated methods from other systems, required updates to own external interfaces for ease of usage, etc.
internal interface	Error in interfacing with another component of the same system, such as violating the contract of inheritance, wrong use of operations from other classes, etc.
logic	Incorrect expressions in conditional statements (e.g., if, while, etc.)
non-functional	Violations in non-functional requirements, such as improper variable or method names, wrong documentation to the implementation of a method, etc.
timing/optimization	Error that causes concurrency or performance issues, such as deadlock, high memory usage, etc.
configuration	Error in non-code (e.g., configuration files) that affects functionality.
others	Other bugs that do not fall into one of the above categories.

programs in Apache Software Foundation, the JIRA repositories store the linkages between reported bugs and code commits in source control repositories that fix them. Our initial analysis on a number of randomly sampled links and bug reports finds that most links are correct and most bug reports are properly linked to all the commits that fix them. Thus, we base our analysis on reported bugs, commits, and links between them.

2) *Manual Bug Report Categorization.* We then pick a sample set of 500 bugs as described in Section II-A for our analysis. We manually look into the description of each bug report to decide its labels. We use the set of categories proposed by Seaman et al. in [34] and add one new category: configuration, to capture bugs in configuration files. The categories and their meaning are shown in Table II. The description of a bug report is often short and ambiguous; for these cases, we look not only at the textual description but also at the code changes that fix the bug. This helps us to find more about the nature of the bug and decide a better category of a bug report. For example, Figure 1 shows a sample bug report. The description of the report talks about "nightly builds" but is unclear what the error actually is. The logs from the code commits that fix the bug is clearer that it tells us the bug is about producing invalid final offset for input containing "`</br>`" in HTMLStripCharFilter, and we can confirm this by looking into the files changed. Thus, we can categorize this bug into the algorithm/method category. Each of the first two authors manually labeled half of the bug sample set, and they met for many hours and verified each other's labels to ensure their correctness.

Figure 1. Example of ambiguous description

3) *Statistics Computation.* We compute various statistics for every bug category and for all bugs that we investigate. These statistics are used to answer the various research questions that we have on the number of bugs appearing in machine learning systems, the numbers of different categories of bugs, and the relationships between bug categories and bug severities, fixing time and effort, and bug impact on the overall systems.

III. EMPIRICAL STUDY

This section presents our research questions and answers, and discusses additional analysis and threats to validity.

A. Research Questions

We are interested in analyzing these research questions:

RQ1: How often bugs appear in algorithm-intensive machine learning systems?
RQ2: What kinds of bugs appear in algorithm-intensive machine learning systems?
RQ3: How severe are various kinds of bugs?
RQ4: How long does it take to fix various bugs?
RQ5: How much effort is needed to fix various bugs?
RQ6: How many files need to be fixed for various bugs?

The first question investigates the density and speed of bugs appearing in the three machine learning systems. The second question analyzes occurring frequencies of each kind of bugs. We investigate bug severities and the time durations taken to fix bugs in the third and fourth questions respectively. We analyze the bug-fixing effort by counting the number of revisions needed to fix a bug in the fifth question. We analyze the impact of the bugs in terms of the number of files that need to be changed to fix the bugs in the sixth question. We describe the answers to the questions in the following subsections.

B. RQ1: Bug Frequencies

Based on bug counts shown in Table I in Section II, we divide the number of historically closed bugs in a system by the size of a snapshot of the system we find that Lucene has

the highest number of bugs (2.77 bugs/kLOC) followed by Mahout (1.79 bugs/kLOC) and OpenNLP (1.45 bugs/kLOC) (shown in Table III). This means that for every line of code, developers of Lucene need to fix more bugs than those of Mahout and OpenNLP. These numbers also seem to be higher than those in operating systems as shown in previous studies [9], [27], [31]. The last column of Table III shows the average numbers of bugs occurred for the systems per year. Lucene is the oldest system, while OpenNLP is the youngest system. Lucene has the highest number of bugs per year, followed by OpenNLP, followed by Mahout.

Table III
BUG DENSITIES IN MACHINE LEARNING SYSTEMS

Application	Bug Count Per kLOC	Bug Count Per Year
Mahout	1.79 bugs/kLOC	73.36 bugs/year
Lucene	2.77 bugs/kLOC	144.76 bugs/year
OpenNLP	1.45 bugs/kLOC	95.68 bugs/year

Note that high bug numbers or densities do not necessarily imply low software quality as previous studies have shown (e.g., [13]). Many factors, such as the popularity of a project (e.g., the number of developers contributing to the project, the number of users reporting bugs), the release cycles of the project, and the size and complexity of the project, may affect bug counts and densities. It is interesting future work to explore and understand the factors that affect bug counts in machine learning systems.

> Lucene is more than 10 years old and has a higher bug density (2.77 bugs per kLOC) than Mahout (about 5 years old, 1.79 bugs per kLOC) and OpenNLP (about 2 years old, 1.45 bugs per kLOC).

C. RQ2: Bug Types

We manually extract 500 randomly chosen bugs from the three machine learning software systems. These bugs are then manually labeled. The distribution of bugs based on the 11 categories is shown in Table IV. We find that most bugs are categorized as algorithm/method (22.60%), followed by non-functional (15.60%), and followed by assignment/initialization (13.00%). Only 1% of the bugs falls under the category others. This indicates that the other 10 categories are sufficient to cover most of the bugs.

Table IV
BUG TYPES

Type	Count	Percentage
algorithm/method	113	22.60%
non-functional	78	15.60%
assignment/initialization	65	13.00%
checking	57	11.40%
external interface	38	7.60%
internal interface	38	7.60%
data	28	5.60%
logic	27	5.40%
configuration	27	5.40%
timing/optimization	24	4.80%
others	5	1%

Algorithm/method bugs correspond to incorrect implementation of a defined algorithm. An example of such bugs is shown in Figure 2. It describes a bug on the fuzzy search algorithm that does not perform appropriate boosting.

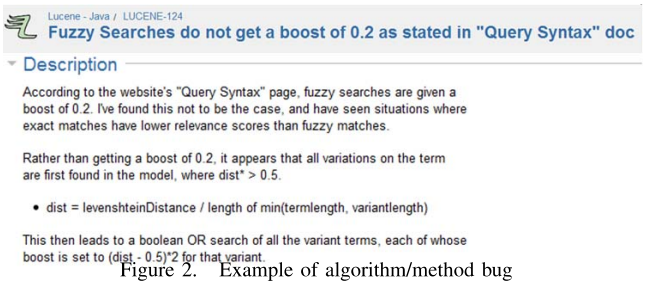

Figure 2. Example of algorithm/method bug

Non-functional bugs correspond to bugs that do not affect the actual functionality of the system. An example of such bugs is shown in Figure 3. It describes an inconsistency between the documentation and code.

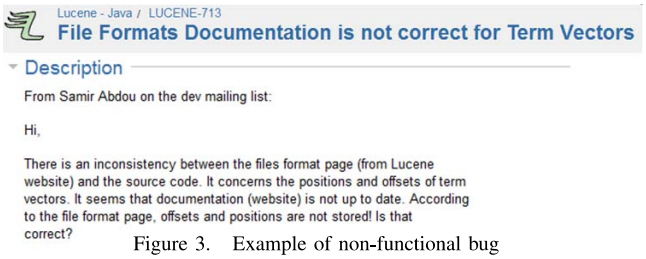

Figure 3. Example of non-functional bug

Assignment/initialization bugs correspond to unassigned or wrongly assigned variables. An example of such bugs is shown in Figure 4. The QueryParser.getDefaultQuery method does not set a default value for "slop."

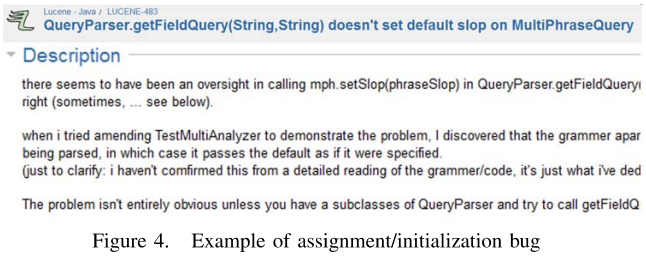

Figure 4. Example of assignment/initialization bug

Timing/optimization bugs have the least occurrences (aside from others). This might be the case as the machine learning systems are often implemented based on well-known and scalable algorithms and thus they are generally scalable. It might be the case also because most performance bugs are detected early and do not appear in the bug repositories. It might also be possible that performance bugs experienced by the users of the systems are hard to reproduce or debug without the specific settings and data inputs used by the users, and thus are not fixed or closed.

> The most common categories of bugs in Mahout, Lucene, and OpenNLP are: algorithm/method (22.6%), non-functional (15.6%), and assignment/initialization (13.0%).

Table V
BUG COUNTS FOR VARIOUS BUG SEVERITIES

Type	Severity	Count	Proportion	Type	Severity	Count	Proportion
algorithm/method	Blocker	1	0.88%	internal interface	Blocker	1	2.63%
	Critical	3	2.65%		Critical	0	0.00%
	Major	70	61.95%		Major	22	57.89%
	Minor	33	29.20%		Minor	11	28.95%
	Trivial	6	5.31%		Trivial	4	10.53%
assignment/initialization	Blocker	2	3.08%	logic	Blocker	0	0.00%
	Critical	1	1.54%		Critical	0	0.00%
	Major	34	52.31%		Major	15	55.56%
	Minor	24	36.92%		Minor	8	29.63%
	Trivial	4	6.15%		Trivial	4	14.81%
checking	Blocker	2	3.51%	non-functional	Blocker	0	0.00%
	Critical	1	1.75%		Critical	0	0.00%
	Major	32	56.14%		Major	38	48.72%
	Minor	17	29.82%		Minor	29	37.18%
	Trivial	5	8.77%		Trivial	11	14.10%
configuration	Blocker	0	0.00%	timing/optimization	Blocker	0	0.00%
	Critical	0	0.00%		Critical	0	0.00%
	Major	19	70.37%		Major	19	79.17%
	Minor	6	22.22%		Minor	5	20.83%
	Trivial	2	7.41%		Trivial	0	0.00%
data	Blocker	0	0.00%	others	Blocker	0	0.00%
	Critical	1	3.57%		Critical	0	0.00%
	Major	15	53.57%		Major	3	60.00%
	Minor	12	42.86%		Minor	0	0.00%
	Trivial	0	0.00%		Trivial	2	40.00%
external interface	Blocker	1	2.63%				
	Critical	1	2.63%				
	Major	21	55.26%				
	Minor	12	31.58%				
	Trivial	3	7.89%				

D. RQ3: Bug Severity

Next, we investigate the relationship between various bug types and their reported severities. There are five severity levels[2] that a bug reporter can assign: Blocker, Critical, Major, Minor, and Trivial. Blocker is the most severe category while trivial is the least severe category. Table V shows the numbers and proportions of bugs at various severity levels for various bug categories.

It is notable that the default severity level is *major* when a user creates a new bug report in the JIRA issue tracking systems. This may be a reason why *major* bugs dominate every bug type since users are often unable to distinguish well among the severity levels and simply leave the reported bug at the default severity level [12]. To reduce biases, it would be interesting future work to incorporate automated techniques (e.g., [18], [28]) to infer more accurate severity levels for bugs before carrying out more detailed analysis.

Nevertheless, from the table, we notice that no one bug category predominates the blocker and critical bugs. Six out of the eleven bug categories contain a bug with either the blocker or critical severity label: algorithm/method, assignment/initialization, checking, data, external interface, and internal interface. Most serious bugs (labeled as blocker and critical) are categorized as algorithm/method, assignment/initialization, checking, and external interface. Each of the data and internal interface categories has only one bug with the critical or blocker severity label. Five out of the eleven bug categories do not contain a bug with either the blocker or critical severity label though: configuration, logic, non-functional, timing/optimization, and other.

Looking at the proportion of trivial bugs, category others has the highest proportion of trivial bugs, followed by the logic and the non-functional categories. This is intuitive as logic bugs often capture corner cases that rarely happen and non-functional bugs are relatively unimportant to the correct working of the systems—many of them are minor issues related to poor variable and method naming.

> Most severe bugs (blocker and critical) are categorized as: algorithm/method (4/14), assignment/initialization (3/14), checking (3/14), and external interface (2/14). Among the 11 categories, others has the highest proportion of trivial bugs (40%).

E. RQ4: Bug-Fixing Duration

Next, we investigate the relationship between bug categories and bug-fixing time. We measure bug-fixing time by the number of days that have passed until a bug report is closed and not re-opened (at least until the time we crawled the repositories in May 2012). Table VI shows the minimum, maximum, mean, and median numbers of days that have elapsed before bugs of various categories are fixed.

We notice that the minimum period between the time when a bug is reported and the time when the bug is fixed for all bug categories is just a few seconds. Most of such

[2]In JIRA bug reports, the word priority is used instead of severity.

Table VI
BUG-FIXING DURATIONS IN TERMS OF DAYS

Type	Min	Max	Mean	Median
algorithm/method	0.0022	2433.7033	91.7238	3.8740
assignment/initialization	0.0003	160.9271	9.9160	0.5000
checking	0.0017	195.7766	17.1335	1.1175
configuration	0.0016	195.9011	22.3583	2.8032
data	0.0014	676.3825	40.8279	2.2666
external interface	0.0006	1700.5871	69.2463	0.4275
internal interface	0.0029	1688.5275	93.3543	2.4852
logic	0.0016	59.8305	6.8892	1.2537
non-functional	0.0006	1330.4142	47.8057	0.6949
timing/optimization	0.0017	569.7309	71.6649	3.3596
others	0.0005	1.3128	0.3344	0.0594

bugs are reported and fixed by the developers themselves: The developers may have noticed the bug much earlier; they just report the bug to the bug repository for future post-mortem analysis after they find a solution for the bug and right before they commit the necessary code changes to the source control repository. This fact follows the observation made by Lamkanfi and Demeyer [20] and such cases may need to be filtered before further analysis on bug-fixing time.

The maximum bug-fixing time, however, can take a few months or years. Three bug categories with the highest maximum bug-fixing time are algorithm/method, external interface, and internal interface, which could take more than 1,500 days. In terms of mean bug-fixing time, internal interface, algorithm/method, and timing/optimization bugs take the longest to be closed (more than 70 days). Table VII gives further breakdowns of durations into a month, a year, and more than a year for various bug categories, and we see that most bugs are fixed within a month.

We look deeper into the average days for fixing these types of bugs by splitting them into high severity bugs (Blocker, Critical), medium severity bugs (Major, the default severity level when a user creates a new bug report), and low severity bugs (Minor, Trivial) as shown in Table VIII. We notice that, in terms of mean and median, low severity algorithm/method and internal interface bugs take longer to be fixed than high severity ones. This may be the case as developers are generally more concerned with high severity bugs than low severity ones. For external interface bugs, the reverse happens. We manually investigate the bugs to understand what the cause may be. We find some bugs that were only fixed after many years. An sample bug is a compatibility issue with older versions of JDK and an external library; the developers might have ignored the bug as users could simply upgrade their JDK or external library to avoid the issues. Due to the small sample sizes of bugs of various categories, our results may not be statistically significant. It would be future work to collect more relevant bug samples for further analysis.

Note that there can be many other factors besides bug category and severity that may affect bug-fixing durations, such as the release cycles of a project, special organized bug-fixing activities, popularity of a project, etc. It would be interesting to investigate what could be done to shorten bug-fixing durations.

> In terms of median bug-fixing duration, bugs in algorithm/method (3.87 days), timing/optimization (3.36 days), and configuration (2.80 days) take the longest to be fixed.

F. RQ5: Bug-Fixing Effort

Next, we measure the bug-fixing effort each bug category. We define bug-fixing effort as the number of revisions it takes to fix a bug such that the corresponding bug report is closed and not re-opened again (until the last time when we collected the data in May 2012). Table IX shows the minimum, maximum, mean, and median numbers of revisions needed to fix bugs in various categories.

We notice that a bug in the checking category requires 11 revisions to be fixed. It involves changes to code in the main trunk and in the branches. The developer performed one commit to the trunk. After some months, the same fix was propagated to other branches. Another bug in the category of algorithm/method needs 9 revisions to be fixed. For this bug, the developer performed an initial fix and then several subsequent fixes to address several sub-problems. The developer did the changes over multiple days, and made separate code commits for different parts of the problem.

Considering the average numbers of revisions needed to correct various kinds of bugs, we find that timing/optimization bugs require the most number of revisions, although we find that timing/optimization bugs occur the least often as shown in Table IV (besides "others").

Note that the number of revisions needed to fix a bug may only be a weak proxy of bug-fixing efforts since revision check-in patterns may vary widely across developers and projects, and do not reflect the actual amount of time spent by developers in fixing bugs[3]. There is prior work studying bug-fixing efforts (in terms of actual time spent). For example, Weiß et al. [40] have studied 125 bugs from the JBOSS project where effort data is available; they find that the average effort for fixing a bug is about 4.8 ± 6.3 hours. Future work would be to improve our study by collecting effort data for machine learning systems, and adapting techniques to more accurately estimate bug-fixing effort.

In this paper, we further investigate the numbers of bugs in each category that require 1 revision, 2 revisions, and more than 2 revisions to be fixed. Table X provides this information. We note that 81.84% of the logic bugs (i.e., bugs in a conditional expression) were fixed in one revision. Also, 78.46% of the assignment/initialization bugs were fixed in one revision. On the other hand, only 25% of the timing/optimization bugs were fixed in one revision; most

[3]Such *actual* bug-fixing time is different from bug-fixing duration which measures the whole lifespan of a bug.

Table VII
NUMBERS OF BUGS FIXED WITHIN VARIOUS DURATIONS

Type	Duration	Count	Proportion	Type	Duration	Count	Proportion
algorithm/method	within a month	80	70.80%	internal interface	within a month	28	73.68%
	within a year	30	26.55%		within a year	7	18.42%
	more than a year	3	2.65%		more than a year	3	7.89%
assignment/initialization	within a month	61	93.85%	logic	within a month	25	92.59%
	within a year	4	6.15%		within a year	2	7.41%
	more than a year	0	0.00%		more than a year	0	0.00%
checking	within a month	50	87.72%	non-functional	within a month	65	83.33%
	within a year	7	12.28%		within a year	10	12.82%
	more than a year	0	0.00%		more than a year	3	3.85%
configuration	within a month	22	81.48%	timing/optimization	within a month	16	66.67%
	within a year	5	18.52%		within a year	6	25.00%
	more than a year	0	0.00%		more than a year	2	8.33%
data	within a month	23	82.14%	others	within a month	5	100.00%
	within a year	4	14.29%		within a year	0	0.00%
	more than a year	1	3.57%		more than a year	0	0.00%
external interface	within a month	28	73.68%				
	within a year	9	23.68%				
	more than a year	1	2.63%				

Table VIII
AVERAGE BUG-FIXING DURATION PER SEVERITY GROUP FOR TOP-3 CATEGORIES (BASED ON MEAN BUG-FIXING DURATION)

Bug Type	High		Medium		Low	
	Mean	Median	Mean	Median	Mean	Median
algorithm/method	62.60	0.73	51.88	4.90	166.21	3.21
internal interface	1.08	1.08	110.42	2.49	74.46	3.09
external interface	57.21	57.21	104.29	0.23	21.79	0.34

Table IX
NUMBERS OF REVISIONS NEEDED TO FIX A BUG

Type	Min	Max	Mean	Median
algorithm/method	1	9	1.9646	1
assignment/initialization	1	4	1.2923	1
checking	1	11	1.7544	1
configuation	1	5	1.7037	1
data	1	5	1.6429	1
external interface	1	6	1.7368	1
internal interface	1	8	1.8947	1
logic	1	4	1.2593	1
non-functional	1	6	1.5641	1
timing/optimization	1	7	2.4167	2
others	1	1	1.0000	1

of the timing/optimization bugs need multiple revisions to be completely resolved.

> In terms of mean revisions needed to fix bugs, timing/optimization bugs require the most effort (2 revisions) to fix. Most bugs in other categories can be fixed in one revision.

G. RQ6: Bug Impact

Finally, we analyze the relationships between bug categories and their impact. We measure the impact of a bug as the number of files that need to be changed to fix the bug. Table XI shows the minimum, maximum, and average numbers of files impacted by various categories of bugs.

Non-functional bugs impact the largest maximum number of files. There is a bug that impacts 676 files. The fix to this bug requires fixing an inconsistent line ending style and has to be done in many files. There is a data bug that affects 124 files since many classes need to be changed from using one data structure to another.

In terms of median number of files impacted, timing/optimization bugs are ranked first again. In terms of the mean numbers of files impacted, non-functional bugs are ranked first. Many non-functional bugs are formatting issues and need to be applied to many files.

We investigate further the numbers of bugs in each category that impact 1-2 files, 3-5 files, and more than 5 files. Table XII provides the statistics. We note that 81.84% of the logic bugs (i.e., bugs in a conditional expression) only impact one file. Also, 70.37% of the configuration bugs only impact one file. On the other hand, only 29.17% of the timing/optimization bugs impact one file; most of the timing/optimization bugs require changes to more than 2 files to be fixed. Again, this may imply that timing/optimization bugs may be hard to fix.

> In terms of median number of files needed to be fixed, timing/optimization bugs have the most impact. In terms of maximum number of files needed to be fixed, non-functional bugs may have the most impact.

H. Additional Analysis

The algorithm/method category deserves special attention: this category has the most number of bugs; many severe bugs belong to this category, which may be related to the algorithm-intensive nature of machine learning systems; its bug-fixing time is long; and it is among the bug categories that require the most effort to fix. Another bug category that deserves attention is timing/optimization bugs. Although this category has the least number of bugs, it requires the most effort to fix, and it is among the categories with longest bug-fixing time. Thus, there may be a need for techniques that could help to reduce algorithm/method and timing/optimization bugs in machine learning systems.

It may also be interesting to investigate techniques that can help to propagate changes needed to fix non-functional, external interface, and data bugs that affect many files.

Table X
BUG-FIXING EFFORT (DETAILED)

Type	Effort	Count	Proportion	Type	Effort	Count	Proportion
algorithm/method	1	71	62.83%	internal interface	1	25	65.79%
	2	17	15.04%		2	3	7.89%
	>2	25	22.12%		>2	10	26.32%
assignment/initialization	1	51	78.46%	logic	1	22	81.84%
	2	11	16.92%		2	4	14.81%
	>2	3	4.62%		>2	1	3.70%
checking	1	32	56.14%	non-functional	1	54	69.23%
	2	16	28.07%		2	14	17.95%
	>2	9	15.79%		>2	10	12.82%
configuration	1	16	59.26%	timing/optimization	1	6	25.00%
	2	7	25.93%		2	10	41.67%
	>2	4	14.81%		>2	8	33.33%
data	1	16	57.14%	others	1	5	100.00%
	2	8	28.57%		2	0	0.00%
	>2	4	14.29%		>2	0	0.00%
external interface	1	26	68.42%				
	2	5	13.16%				
	>2	7	18.42%				

Table XI
NUMBERS OF IMPACTED FILES PER BUG

Type	Min	Max	Mean	Median
algorithm/method	1	117	11.7434	3
assignment/initialization	1	74	5.2459	2
checking	1	115	7.2982	2
configuration	1	40	5.2692	1
data	1	124	12.3571	3
external interface	1	92	13.3514	3
internal interface	1	52	7.6842	4.5
logic	1	15	2.7778	2
non-functional	1	676	16.7143	2
others	1	47	10.2000	1
timing/optimization	1	69	11.5833	5

Interestingly, logic bugs (i.e., errors on conditional statements) which are often studied in fault localization studies (e.g., [23], [25]) do not appear frequently. This category is also among the categories of bugs with the highest proportion of trivial bugs. It is also among the categories of bugs that require the lowest bug-fixing effort. Similarly, assignment/initialization bugs, which are also often studied in fault localization studies, are among those that require the lowest bug-fixing effort. Thus there is a need to investigate the effectiveness of existing fault localization tools or develop new methods for algorithm / method bugs and other bug categories which are considered less in existing studies.

I. Threats to Validity

There are further threats that may potentially affect the validity of our findings. Threats to internal validity relate to experimenter bias and errors. Our study involves manual inspection of bugs. This process is potentially error-prone. To reduce this threat, each bug report is labeled by one person and is checked by at least another. Any discrepancy is discussed until a consensus is reached.

Threats to external validity relate to the generalizability of our findings. We have analyzed three machine learning systems: Mahout, Lucene, and OpenNLP. To improve the generalizability of our findings we intentionally pick three systems from different branches of machine learning techniques that are widely used in software engineering research, including data mining, information retrieval, and natural language processing. However, they are all Apache Software Foundation projects which use the JIRA repositories and the characteristics of the bugs we find in this paper may be due to the Apache development and bug-fixing processes and/or the capabilities of JIRA. There are many other machine learning libraries and systems that we have not analyzed. Also, we only manually analyze 500 randomly sampled bug reports. Although it is not a very large number, we believe it is a good sample size as past studies, such as [4], [6], [37], [41], investigate similar numbers of manually labeled data. We plan to reduce this threat to external validity in the future by analyzing more systems and bug reports.

There are possible threats to construct validity. Our study aggregates bugs from different projects together; it may conceal specificities of bugs from each project. We will investigate whether bug statistics would be different when the bugs from different projects are analyzed separately.

IV. RELATED WORK

A. Empirical Study on Bugs

Seaman et al. investigate bugs in various projects with NASA and come up with various categories of bugs depending on where these defects occur: requirement documents, code, and test plans [34]. Their paper mostly describes how they come up with the categorization and how they label bugs following the categorization. It is unclear what types of projects are used in their study, possibly because the data comes from NASA and may not be publicly shared. In this study, we make use of their categorization and perform study on bugs from algorithm-intensive machine learning systems.

Pan et al. investigate bug fixes in a number of systems written in Java [32]. They categorize the types of bug fixes based on syntax of the code changes, such as addition of precondition check (IF-APC), addition of post-condition

Table XII
BUGS WITH VARIOUS NUMBERS OF IMPACTED FILES

Type	Impact	Count	Proportion	Type	Impact	Count	Proportion
algorithm / method	1-2	38	33.63%	internal interface	1-2	14	36.84%
	3-5	33	29.20%		3-5	7	18.42%
	>5	42	37.17%		>5	17	44.74%
assignment / initialization	1-2	43	66.15%	logic	1-2	22	81.48%
	3-5	10	15.38%		3-5	2	7.41%
	>5	8	12.31%		>5	3	11.11%
checking	1-2	29	50.88%	non-functional	1-2	40	51.28%
	3-5	16	28.07%		3-5	13	16.67%
	>5	12	21.05%		>5	24	30.77%
configuration	1-2	19	70.37%	others	1-2	4	80.00%
	3-5	2	7.41%		3-5	0	0.00%
	>5	5	18.52%		>5	1	20.00%
data	1-2	13	46.43%	timing / optimization	1-2	7	29.17%
	3-5	8	28.57%		3-5	6	25.00%
	>5	7	25.00%		>5	11	45.83%
external interface	1-2	17	44.74%				
	3-5	8	21.05%				
	>5	12	31.58%				

check (IF-APTC), and others. Different from their work, we focus on bugs in algorithm-intensive systems, and our categorization, adapted from [34], is more semantic-aware.

Zaman et al. study security and performance bugs in Firefox [42]. They investigate questions such as how fast bugs are fixed, who fix bugs, and what characteristics the bug fixes have. Lu et al. study concurrency bugs in MySQL, Apache Web Server, Mozilla, and OpenOffice [24]. They study questions such as what the types of concurrency bugs are, how many threads and variables are involved in the bugs, etc. Li et al. analyze bugs in Mozilla and Apache Web Server [21]. They categorize bugs in three dimensions: root cause, impact, and software component. Chou et al. investigate bugs in operating systems which investigate bugs in the Linux and OpenBSD kernels [9]. They analyze questions such as where the bugs are, how the bugs are distributed, how long bugs live, how bugs are clustered, and how bugs in different operating systems differ. A similar study was performed a decade later by Palix et al. [31]. Maji et al. study defects in mobile operating systems, including Android and Symbian [27]. Different from these studies, we consider a wider variety of bugs following the categorization in [34] and analyze how they affect machine learning systems.

B. Bug Categorization

Many studies have proposed automated approaches to categorize bugs. Huang et al. propose a semi-automated technique that can automatically categorize bugs into various categories, such as reliability, capability, integrity/security, usability, and requirements [14].

Some other studies automatically assign severity labels to bug reports. Menzies and Marcus predict the severities of bug reports from NASA [28]. Extending the work of Menzies and Marcus, Lamkanfi et al. investigate bug reports in open source projects and develop a technique that predicts coarse-grained severity labels (i.e., severe and non-severe) [18]. After that, Lamkanfi et al. explore various classification algorithms and investigate their effectiveness in predicting the severities of bugs [19].

There are also a number of studies that predict if a reported bug is a duplicate or not. Runeson et al. [33], Wang et al. [39], Jalbert and Weimer [15], Sun et al. [35], [36], Kaushik et al. [17], and Tian et al. [38] propose various approaches to identify if a bug report is a duplicate or not. The approach by Wang et al. [39] also use additional information, such as execution traces, aside from the bug reports. The approaches by Jalbert and Weimer [15] and Tian et al. [38] are fully automated but have lower accuracies. The other approaches are semi-automated and eventually needs a developer to decide if a bug report is a duplicate or not.

C. Bug Fixing

Giger et al. use decision trees to predict whether a bug may be fixed fast (i.e., takeing less time than the median fix time) [10]. Weiß et al. build a machine learning model to predict how long it takes for a bug to be fixed [40]. As mentioned in Section III-F, their study focuses on the actual amount of time spent by developers on a bug, instead of the bug-fixing duration that measures the whole life of a bug. Guo et al. build a model to predict whether a bug would be fixed for Windows bugs [11]. Bhattacharya and Neamtiu study hundreds of thousands of bug reports and suggest that better models may be needed for predicting bug fix-times in different kinds of software projects [5].

Jeffrey et al. propose an automated approach to fix bugs using association rule mining [16]. Tien et al. propose a graph mining based approach to fix bugs based on bug fixing histories [30]. In this study, we perform an empirical study that analyzes bug fixing time and effort for machine learning systems and relate them to different categories of bugs.

V. CONCLUSION AND FUTURE WORK

With the increasing amount of data accumulated from various domains, such as digital social networks, financial

markets, and space exploration, machine learning solutions are being deployed and used in more and more situations. Due to the high value of such systems, it is important to analyze historical bugs occurred in the systems and learn to prevent them from reoccurring.

In this study, we propose to investigate the characteristics of bugs found in algorithm-intensive machine learning systems that may be used with large-scale data. We focus on three machine learning systems: Apache Mahout, Apache Lucene, and Apache OpenNLP.

We investigate 500 fixed bugs randomly picked from the three systems, and manually categorize the bugs. We note that bugs in these systems may be more dense than open source operating systems (e.g., Linux, Android), and bugs in the algorithm/method category have the highest proportion (22.6%). We also study various statistics of bug categories in relation to bug severities, fixing durations, fixing efforts, and impact. We find that the algorithm/method category contains the highest number of most severe bugs and takes the longest median time to be fixed. The timing/optimization bugs take the second longest median time to be fixed and may require the most effort to fix and have the largest median impact on the source files in a machine learning system.

As future work, we plan to analyze more machine learning systems and bugs. We also plan to develop a machine learning technique to automatically categorize bugs into one of the bug categories that we have. We also plan to investigate the effectiveness of current bug finding and localization solutions on machine learning systems and in particular address algorithm/method and optimization/timing bugs. We would also like to investigate approaches that could help developers to propagate fixes for non-functional, external interface, and data bugs.

ACKNOWLEDGEMENTS

We thank Foyzur Rahman and Prem Devanbu for pointing us to the Apache JIRA repository that contains reasonably accurate links between bug reports and corresponding code changes. We also thank anonymous reviewers for providing valuable comments to improve this paper.

REFERENCES

[1] "Apache Lucene," http://lucene.apache.org/core/.
[2] "Apache Mahout," http://mahout.apache.org/.
[3] "Apache OpenNLP," http://opennlp.apache.org/.
[4] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *CASCON*, 2008.
[5] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *MSR*, 2011, pp. 207–210.
[6] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein, "LINKSTER: Enabling efficient manual inspection and annotation of mined data," in *SIGSOFT FSE*, 2010, pp. 369–370.
[7] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *ISSTA*, 2009.
[8] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong, "Orthogonal defect classification—a concept for in-process measurements," *TSE*, vol. 18, no. 11, pp. 943–956, 1992.
[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *SOSP*, 2001.
[10] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *RSSE*, 2010, pp. 52–56.
[11] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows," in *ICSE*, 2010, pp. 495–504.
[12] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "Towards a simplification of the bug report form in eclipse," in *MSR*, 2008, pp. 145–148.
[13] I. Herraiz, E. Shihab, T. H. D. Nguyen, and A. E. Hassan, "Impact of installation counts on perceived quality: A case study on Debian," in *WCRE*, 2011, pp. 219–228.
[14] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "AutoODC: Automated generation of orthogonal defect classifications," in *ASE*, 2011, pp. 412–415.
[15] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *DSN*, 2008.
[16] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "BugFix: A learning-based tool to assist developers in fixing bugs," in *ICPC*, 2009.
[17] N. Kaushik and L. Tahvildari, "A comparative study of the performance of IR models on duplicate bug detection," in *CSMR*, 2012.
[18] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *MSR*, 2010.
[19] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *CSMR*, 2011.
[20] A. Lamkanfi and S. Demeyer, "Filtering bug reports for fix-time analysis," in *CSMR*, 2012, pp. 379–384.
[21] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? an empirical study of bug characteristics in modern open source software," in *ASID*, 2006, pp. 25–33.
[22] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/SIGSOFT FSE*, 2005, pp. 306–315.
[23] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *ESEC/FSE*, 2005.
[24] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008, pp. 329–339.
[25] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.
[26] L. Ma and J. Tian, "Web error classification and analysis for reliability improvement," *JSS*, vol. 80, no. 6, pp. 795–804, 2007.
[27] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile OSes: A case study with Android and Symbian," in *ISSRE*, 2010, pp. 249–258.
[28] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008.
[29] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC/SIGSOFT FSE*, 2009, pp. 383–392.
[30] ——, "Recurring bug fixes in object-oriented programs," in *ICSE (1)*, 2010, pp. 315–324.
[31] N. Palix, G. Thomas, S. Saha, C. Calvès, J. L. Lawall, and G. Muller, "Faults in Linux: Ten years later," in *ASPLOS*, 2011, pp. 305–318.
[32] K. Pan, S. Kim, and E. J. W. Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, 2009.
[33] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007, pp. 499–510.
[34] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: making use of a decade of widely varying historical data," in *ESEM*, 2008.
[35] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.
[36] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010.
[37] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *ICSE*, 2012.
[38] Y. Tian, D. Lo, and C. Sun, "Improved duplicate bug report identification," in *CSMR*, 2012.
[39] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE*, 2008, pp. 461–470.
[40] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *MSR*, 2007.
[41] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: Recovering links between bugs and changes," in *FSE*, 2011, pp. 15–25.
[42] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *MSR*, 2011, pp. 93–102.