CodeFault:
Analyzing Human Dimensions of Software Engineering Processes

Final Technical Report

Sponsored by

Defense Advanced Research Agency (DARPA)

Performer:	InferLink Corporation
Award number:	140D6319C0016
Project title:	CodeFault: Analyzing Human Dimensions of Software Engineering Processes
Period of performance:	02/11/2019 – 04/10/2022
Estimated award value:	$1,500,000 (including Option)

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*

1 Introduction

Flawed code is costly in multiple ways. First, there is the burden of software maintenance, the cost of which can range from 40 to 80 percent of development (Glass 2001). However, a second, arguably more important issue is that the resulting software may be vulnerable to attack and thus a candidate for exploitation. This can have significant implications beyond the cost to fix the code itself. For example, lack of bounds checking can lead to buffer overflows, and an attacker that identifies such a weakness can easily exploit it to insert malicious code and take over remote systems or entire networks. While the code may work fine for most typical uses, intentional exploitation of the lack of bounds checking can have devastating effects beyond that associated with just software maintenance. This is most visibly demonstrated by the number of high profile breaches and ransomware scenarios that have played out in recent memory, such as the Equifax breach and WannaCry ransomware attack (which affected an estimated number of 300,000 machines), with several of these tied to the exploitation of vulnerabilities. In fact, exploit toolkits such as Angler have targeted multiple vulnerabilities as part of their ransomware injection process[1].

This SBIR topic called for innovative methods to identify faulty or insecure code, based on the human aspects of software development. Traditional approaches, such as static and dynamic analysis, continue to be helpful at identifying faulty code, but they do not focus on the *behavior by software developers* as the basis for risk. Thus, the core challenge of this topic is the need to understand what human behavioral patterns correlate with the authorship of flawed code. In essence, by understanding the failures of the past, we can potentially mitigate future risks and ultimately create a more reliable software engineering process. Our definition of behavior is broad: we consider not only how developers do development, but also how they interact with others, their communications with others, and so forth. There are two key capabilities needed to address this challenge. One is the ability to gather data about historical software development and vulnerabilities from a wide variety of sources related to software development, vulnerabilities, and the interactions of software engineers. The aggregated set of this data represents a rich history of examples to mine. The second capability is to leverage analytical approaches to unearth patterns in the historical data. More specifically, one needs measures to detect possible behavioral indicators and tools to build models of risk. These resulting models can be used to analyze new code and predict whether that code is likely to be flawed and/or insecure.

In Phase I of this SBIR, we designed the key elements of such a system, named **CodeFault**, designed to address this challenge. In particular, we designed CodeFault to gather and integrate data from multiple heterogeneous sources, including those related to social coding, social media, and vulnerability aggregation. We also investigated a variety of statistical and machine learning methods that could help identify potentially useful signals of risk and learn models that would leverage such historical data to predict risk. Finally, we prototyped selected capabilities, and demonstrated the feasibility of our approach. This involved acquiring IRB approval and gathering data from multiple sources, forming a range of possible hypotheses, running a series of descriptive statistics to look for interesting phenomena, and then demonstrating the capability to learn models from the data. Our Phase I work sets the stage for our proposed Phase II execution, which is to fully implement the CodeFault system we designed, and evaluate it end-to-end with a broad spectrum of data.

Figure 1 shows a high level summary of the CodeFault. In particular, the figure shows that CodeFault can gather data about software engineering practices from sources like GitHub, social communication from developers from sites like StackOverflow and Twitter, and data on vulnerabilities from sources like NVD and ICS-CERT. We can then use this historical data as the basis for identifying patterns of human behavior that are associated with faults. By identifying such patterns, we can then predict when there are more likely to be faults in previously unseen code.

[1] https://www.recordedfuture.com/recent-ransomware-vulnerabilities/

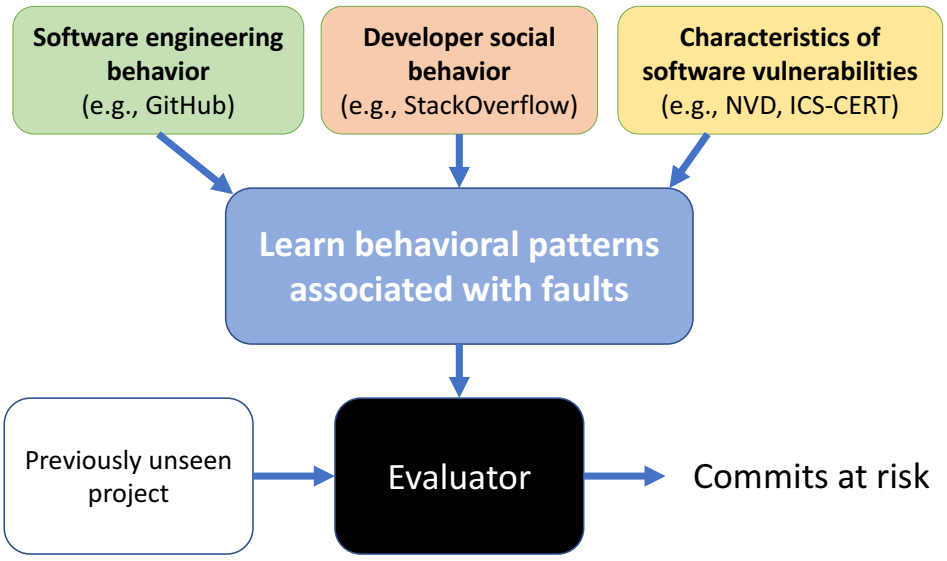


Figure 1: High-level dataflow of CodeFault

The types of patterns we can learn span both the human dimensions of engineering as well as social behavior. For example, one could envision learning that:

- Checking in many files at once is highly correlated with flaws
- Developers who have relatively fewer indicators of testing/QA expertise (e.g., do not have any such tags in their StackOverflow profile) tend to be associated with flawed commits
- Code with filenames that contains certain terms (like "File" or "Network") tends to be more frequently associated with CVEs
- Certain third-party libraries (e.g., such as those that involve network or database integration) are inherently riskier than others

In this final report, we focus what we learned and produced as part of the CodeFault Phase I and Phase II efforts. We also cover our findings from the Phase II Option, which extended this work to look beyond engineering behavior per se, and to focus on more complex social behavior as an indicator of risk. We discuss our work with Margin Research towards that end in later sections.

2 Phase I results

The goal of Phase I was to design the major components of the CodeFault system and ensure the general feasibility of our approach, so that we are well-positioned to build fully functioning prototype in Phase II. Our specific Phase I objectives were:

Objective 1: Gathering data on the human dimensions of software development, Our first objective was to design a strategy for gathering large amounts of software development data, including information on faults (bugs and vulnerabilities), as well as the human-oriented process data associated with those faults. In particular, we wanted to identify both a broad set of sources and a variety of methods for harvesting data so that, in Phase II, we would be able to generate rich collection of information to mine. As part of this effort, we wanted to identify a large set of attributes and metadata related to software development processes that we can gather (or derive) from the types of software projects we will be aggregating. These attributes/metadata can span from code-related attributes (e.g., time of day of check-in, etc.) to project/contributor related attributes (e.g., social profile of the developers involved, chat dialogue associated with software faults, etc.).

Objective 2: Learning to predict code faults. Our second objective was to design a process for learning to predict code faults. As part of this, we wanted to develop a methodology for understanding how to identify and judge faults from data collected. With faults identified and a rich set of associated data and metadata to explore, the goal was then to design how we could go from higher level (descriptive-style) analysis to deeper predictive models. These models would be built using existing statistical and machine learning methods. As part of this process, we wanted to design a methodology that would allow us to generate high quality predictions of risk for code that has been or will be added to a project.

Objective 3: Feasibility evaluation. The final objective was ensure that our approach would be valid for Phase II execution. In particular, we aimed to demonstrate selected aspects of our data gathering and learning approach so that we could be well-positioned for a successful, larger Phase II implementation. Our goal in this objective was thus not to build an entire system, but rather to demonstrate the potential for such an end-to-end system, using techniques described in Objectives 1 and 2. We aimed to identify a set of data to use as the basis for ensuring that our gathering and learning processes would be successful.

We next discuss how we met these objectives through three corresponding tasks we completed in the course of the Phase I.

2.1.1 Key Phase I Accomplishments

During Phase I, we accomplished the following:

- **Gathering data on human dimensions of software engineering**
  - Identified public (online) sources that contain valuable data related to the human dimensions of software engineering
  - Identified data gathering methods to automatically retrieve and extract data from these sources
  - Designed a methodology for identifying and prioritizing faults
  - Designed a methodology for linking users between social networks, which will allow us to leverage more historical software engineering behavioral data
  - Designed an aggregation architecture for gathering, linking, and enriching data

- **Learning to predict code faults**
  - Identified how advanced feature engineering methods could be integrated into the system
  - Identified statistical and machine learning methods that can be used to help predict risk
  - Designed an approach for combining models in order to reduce false positives
  - Identified a Bayesian style approach that combined local and global models so that evaluation of a new codebase took into account historical trends, but was sensitive to the differences that a particular repository might have (i.e., took into account local history)

- **Feasibility evaluation**
  - Applied for and was granted IRB exemption related to data gathering
  - Gathered public data on:
    - 5000+ projects spanning JavaScript, Ruby, Python, and PHP
    - 3000+ users on StackOverflow who were developers on these projects
    - 3000+ users on Twitter who were developers on these projects
    - 100+ JIRA based projects
  - Demonstrated the ability to identify GitHub and StackOverflow accounts belonging the same individual based on similarity metrics
  - Conducted a range of descriptive statistics on the combination of this data, in order to explore possible correlations and trends (from which we could potentially learn predictive models)

- Completed end-to-end example of gathering a sample data for an example project (Apache Tomcat), to which we engineered additional features (e.g., "part of day"), built independent models of prediction, and then combined models to explore how false positives can be reduced.
- Identified multiple "interesting findings" with the limited data/time available, e.g.,:
  - Apache Tomcat faults seem to committed less often in the morning than during other times of day
  - StackOverflow users that had Google in their bio somewhere tended to have a higher StackOverflow reputation than those who did not have Google listed
  - Those that contributed to PHP repositories in GitHub tended to have lower StackOverflow reputations than others who did not

2.1.2 Phase I, Task 1: Gathering data on the human dimensions of software development

The key goals of the first task were to focus on the data model, and in particular which sources and their attributes we would harvest. These, in turn, would be combined with an additional set of enriched attributes to further enhance the data. For example, although we will gather the time of day that code was committed, we want to enrich that with another attribute that applies a semantic label (e.g., "Late night") to help us discretize and categorize the data in human terms. This also allows us to leverage analysis and machine learning methods that support categorical labels, not just continuous data.

Sources and attributes

In terms of data sources, we defined a set of source types to harvest, many of which contain attributes related to the human element of software development. These included:

- **Social coding sites**: e.g., GitHub and BitBucket
- **Developer discussion sites**: e.g., StackOverflow
- **Bug databases**: e.g., JIRA and GitHub issues
- **Social media sources**: e.g., Twitter
- **Vulnerability databases**: e.g., NVD, ICS-CERT, etc.

Next, we defined the set of attributes we will gather from each source. For example, with GitHub, these attributes included:

- Project metadata, including
  - Project name and description
  - Whether the project has a wiki, documentation, etc.
- Commit data, such as
  - Dates and times
  - Amount of code added/removed
  - Contributor(s)
  - Log messages
- etc.

For a source like StackOverflow, which focuses more on developer knowledge, the attributes included:

- Developer reputation
- Developer profile information (presence of bio, current position, etc.)
- Developer questions and answers
- Developer medals (gold, silver, bronze)
- Developer tags
- Generally popular tags

- etc.

Data collection and integration

Next, we designed multiple methods for accessing data from various sources. These methods included:

- **API access**: GitHub and StackOverflow, for example, have extensive APIs that will allow us to easily retrieve several aspects of metadata we are intending to collect. A key constraint, however, is that we obey API quotas; in short, we must often harvest data incrementally.
- **Web crawling and extraction:** We have significant previous experience using AI techniques to automatically build web extractors for sites that have been crawled with off-the-shelf crawling software. By doing this, we can store the website as a local database. For example, we can scrape sources like Apache JIRA to gather public information about Apache faults that are made public.
- **Other custom methods:** We have other ways of extracting relevant information. One important method is to write a series of source control scripts that allow us to interrogate key details about the software development process. These include the range of git commands (clone, log, blame, etc.) that provide access to different aspects of the source control metadata, such as when files were checked in, by whom, what was changed, and so forth. We could use the GitHub API for some of this, but the capabilities there are not as expansive as what is available through git-style interrogation, plus it would require many API calls.

As part of our Phase I investigation, we tested the above methods to ensure that they would work as planned (see Feasibility Study, below).

One important issue we encountered as part of Phase I design was the need to link data *between* sources. For example, it is very helpful to understand that user *G* on GitHub is also user *S* on StackOverflow and user *T* on Twitter, since doing so at scale would allow us to look at engineering behavior (e.g., activities on GitHub) as a function of social behavior (e.g., activities on StackOverflow and Twitter). In short, this type of linkage greatly expands the range of behavioral attributes to consider as part of our analysis.

To link users between sources, we can sometimes leverage the explicit linkage by users. For example, as shown in Figure 2, it is not uncommon for GitHub users to specify their Twitter profile in their bio. Unfortunately, users do not do this for all venues that they frequent. For example, unlike with Twitter, developers *do not* tend to list their StackOverflow profile (a sample of which is shown in Figure 3) on GitHub. This is unfortunate because StackOverflow information could help us glean evidence of not only developer knowledge, but also community perception of their knowledge (i.e., is the developer an expert and, if so, at what?). In phase I we used a simple version (for expediency) of the statitsical entity resolution process described in Section 2.1

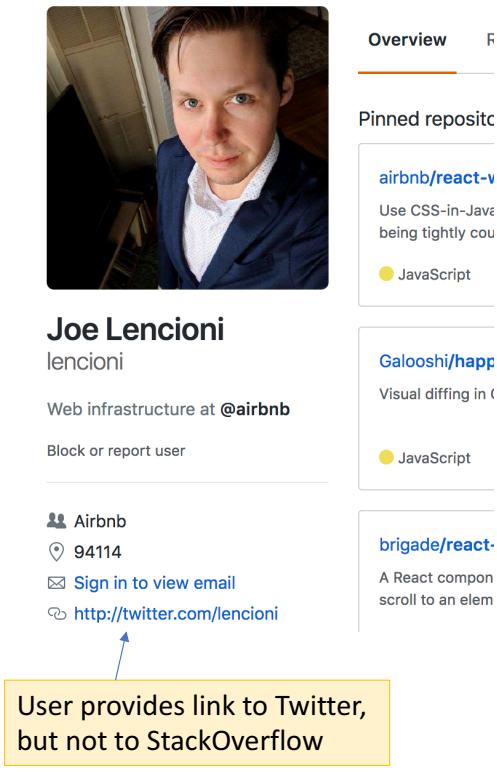


Figure 2: GitHub profile page

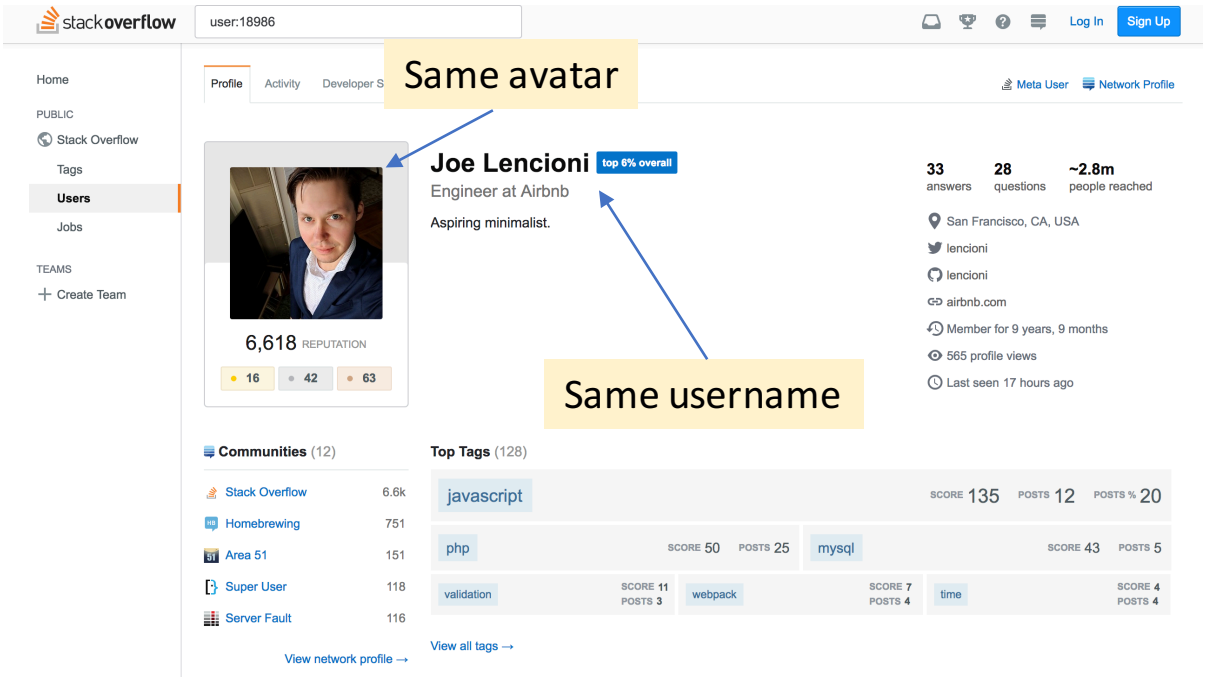


Figure 3: Sample StackOverflow profile page for same user

The simple approach we used in our feasibility study involved two steps: (a) fetch user(s) on source S_1 (e.g., StackOverflow) based on GitHub user name, and (b) verify that the user on source S_2 is the same by evaluating a user similarity function. For (b), our similarity function was simply to compare avatars (profile images) on both systems using computer vision-style techniques, such as histogram correlation.

Both parts (a) and (b) can be made more general and thus expand historical knowledge further, but we felt for Phase I it was more important to focus on designing the process and demonstrating basic feasibility. As we will discuss further in our Phase II proposal, advanced techniques such as entity resolution can consider a variety of additional attributes upon which to help judge similarity. We have previously done extensive work towards building out a probabilistic model for resolving entities between sources and will be proposing to leverage this in Phase II.

Fault identification

Fault identification is important because the models we ultimately wish to investigate are about fault prediction. Two key aspects of our approach involve: (a) how we will identify faults and (b) how we will identify the artifacts associated with their introduction.

In terms of the former, we found in Phase I that it was most practical to identify faults as either commits that specifically reference a fault (e.g., the log says "*Fixes…*") or it was a commit that closes an identified bug. An example of the types of log messages we see, some of which include "fixes", is shown in Figure 4 below. Note that the second example ("fixes encoding bug…") is the type of log message that we wish to use as the jumping off point for fault introduction.

```
[maven-release-plugin] prepare release 2.2.2
fixes encoding bug where core annotations not written in json_v1
exposes call.base and makes tostring pretty
bumps versions, notably kafka and spring-boot
backfills missing tests where duration query applies to annotationquery
makes getdependencies_notimestamps not date sensitive
```

Figure 4: Sample Git log messages

There are a number of ways to filter the log messages, and we need to be careful to reduce noise while retaining as much useful data as possible. For example, we do not wish to use a log message like "Added new feature which fixes the position at 0.0", since the "fixes" in that log is not about repair of a bug. At the same time, we do not wish to be over-restrictive in which log messages we key off of, since that will reduce the training set and possibly lead to more fragile models.

To address this challenge during Phase I, we iteratively crafted a text classifier so that we could better understand the challenge, for instance to better understand the types of examples to use for training. During Phase II, we plan to leverage existing text classification frameworks as a more robust and scalable solution to the problem.

Once a "fix" has been identified, we would then look at the specific lines of code removed by this fix, and this clues us into raw features “at the time of the commit.” For instance, we can see who the fixer was, how many lines were removed, their content, the log messages associated with removing the lines, etc. This is shown in Figure 5:

```
-         int bufLength = headerBufferSize + wrapper.g
+         int bufLength = headerBufferSize +
+                 wrapper.getSocket().getBufHandler().
```

Figure 5: Code removed and added

Finally, we go back through an entire commit history until we find which commit(s) introduced the lines of code that were removed and identified as a fault. This gives us information about the “time of commit” for the fault. So we can, for instance, identify which programmer added the faulty lines, and other commit-level features (e.g., was it part of a large commit, LOC, etc.). Fortunately, Git commands like "blame" are helpful in identifying when such code is introduced. For example, Figure 6 shows the type of "responsible party" commit information we are interested in:

```
commit 9718447a0342866373fec40498e960040d58b738
Author:                    @apache.org>
Date:   Thu Jan 15 09:21:17 2015 +0000

    InputBuffer refactoring. All compiles but not yet te

    git-svn-id: https://svn.apache.org/repos/asf/tomcat/

 10 files changed, 284 insertions(+), 207 deletions(-)
```

Figure 6: Git log entry for party associated with above flaw

There are a few important takeaways from this process to highlight:

- We only look at source code changes as part of fixes or resolutions (e.g., we do not look at documentation changes)
- We only look at lines removed; while we could also look at lines added, we feel that lines removed is a better approximator for the "problem". Furthermore, it is not practical to look historically at which commit forgot to add code (that was added later). Instead, it makes more sense to look for lines removed and then trace back to which commits added them.
- We include all commits that played a role in the lines being added. Thus, it could have been that two different commits, at different times, added code that ultimately was removed.

Architecture for data aggregation

The final task was to design an overall architecture for the data aggregation from multiple sources. A summary of our architecture is shown in Figure 7 below.

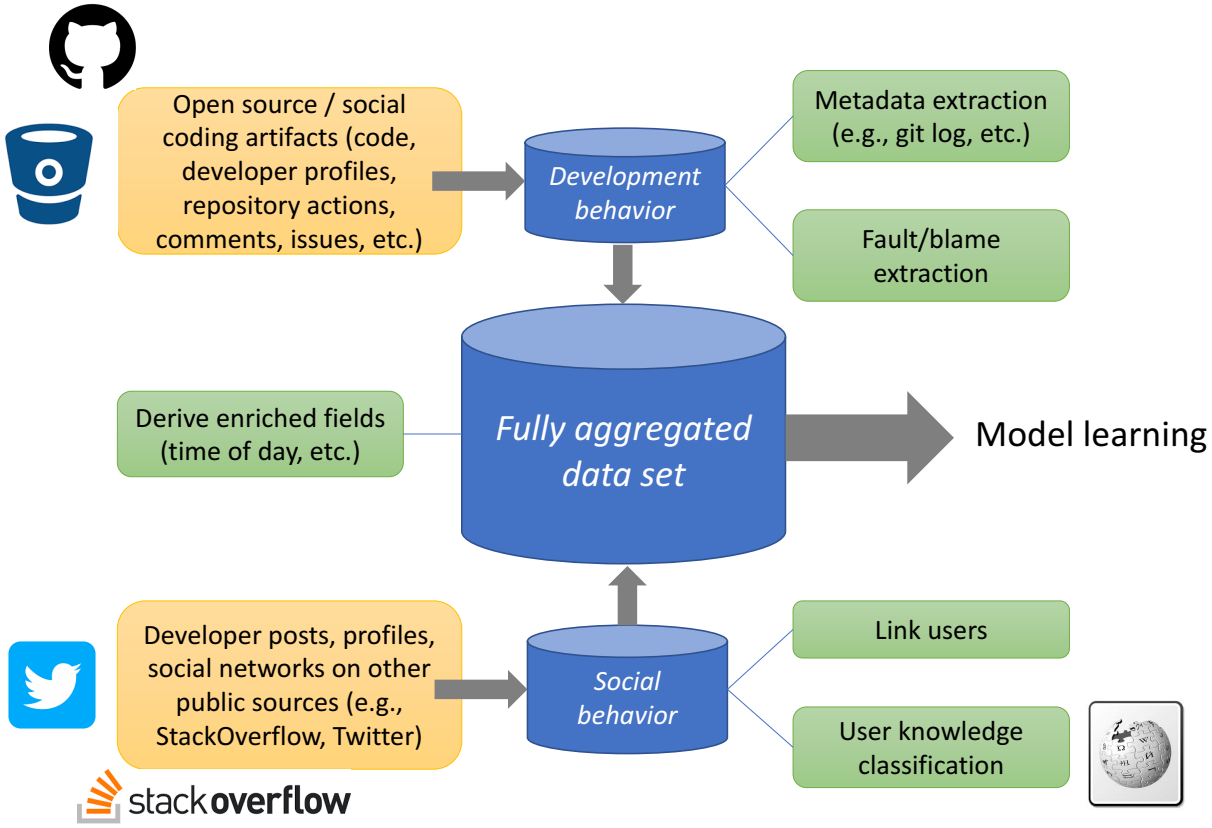


Figure 7: Key data and functions in CodeFault aggregation system

As the figure shows, there are five key phases of the data aggregation process: the retrieval and extraction from originating sources, the user linkage process, the enrichment process, and then fault identification. The result of this architecture is a set of behavioral data, both raw and derived, associated with fault information. The details about which enrichments to build relate more to model building, and thus we covered them as part of our work on Task 2, below.

2.1.3 Phase I, Task 2: Learning to predict code faults

Our goal in Task 2 of Phase I was to design key components of a subsystem to predict code faults. This in turn was broken into two main tasks: feature engineering and model generation. The former focused on

enrichments to the data. Meanwhile, model generation focuses on designs for how to predict code faults and, in doing so, to reduce the likelihood of false positives.

As part of our feature engineering work, we started with the raw features we could extract from different sources using various APIs, code-level gathering (e.g., git clone), and Web scraping (for non-API, code-related data such as bug databases). Thus, we started with a model of the very attributes we identified in Task 1 of our work.

Next, we designed how enrichments on the raw data would be implemented. To do this, we took an example project – Apache Tomcat – and applied a few simple enrichments related to the date of the commit. The goal was to label commits with semantic classifications related to the time of the commit and to do that on a per-row (commit) level. For example, post-enrichment, some of the data we gathered and applied this enrichment to looked as shown below:

COMMIT	HASH	DAY	PART OF WEEK	HOUR OF DAY	PART OF DAY	COMMIT LOG LINES
good	ff0129fcd36f391f019ab7e3c3f1233564110138	Thu	WEEKDAY	17	EVENING	7
bad	f89d5ae7aa33bd8c505d3f9f1ead0012b6aef2c1	Mon	WEEKDAY	18	EVENING	0
bad	f89d5ae7aa33bd8c505d3f9f1ead0012b6aef2c1	Mon	WEEKDAY	18	EVENING	6
bad	f3f0d4dd0cb26bc0fdf2f2eb92d93bac26316762	Sat	WEEKEND	20	EVENING	5
good	f2d19a9f06f4a3f16436dc3d8e034c62f8873970	Fri	WEEKDAY	22	LATE	6
good	f2930beee175ce20a81bd64f2aa9ca0ac3782982	Fri	WEEKDAY	4	LATE	6
bad	eae54419c6e196933998f63358367040edaa4a8c	Thu	WEEKDAY	12	AFTERNOON	43
bad	ea6f489a8f66179eac32abb91440ecf37fcbcaed	Wed	WEEKDAY	0	LATE	5
good	e8fff386d2650158c1aa8fdf572c407537e30762	Sat	WEEKEND	18	EVENING	7
bad	e6d99af12f6ed23d0044d19b66457266dffb97cb	Wed	WEEKDAY	19	EVENING	6
bad	e6c17c7b2df497a3ea3960642a562f0e095b2d91	Mon	WEEKDAY	18	EVENING	6
bad	e424fabd6c3c771c6ece1d712cb86d9de0e1d7a7	Sat	WEEKEND	19	EVENING	6
bad	dc55eb0f0456c7914617be27e9fe3e273e216abb	Fri	WEEKDAY	20	EVENING	6

The figure shows a list of some of the commits made to the Tomcat project. Each row corresponds to a particular commit. For each of these, we enriched the date/time of the commit to include the semantic classification related to the "part of the day" and "part of the week", as shown in the figure. These are potentially useful enrichments because such information is semantically meaningful to human routines (e.g., we typically do not work on weekends, working close to and after midnight is considered "late"), and it would not be surprising if certain routines were more conducive to higher quality development. We should note that it is possible to "layer" enrichments, in the sense that new enrichments may depend on earlier ones. For example, "*weekday*" and "*afternoon*" could be combined to emit "*work time*" enrichment for many types of developers.

Some of these engineered features are based upon both raw and derived features, and are meant to allow us to investigate very specific hypotheses about human behavior and its potential to lead to faulty code. For example, "*rushing*" may involve checking in code late at night or in rapid succession; quantifying "*rushing*" depends on the lower level raw and derived attributes. Our human-factor features are tied to behavioral hypotheses. As part of our Phase I work, we enumerated potential hypotheses of human-factor based features that might lead to faults. The purpose of this exercise was to test the methodology of going from intuition to hypothesis, to actual feature engineering work.

For each hypothesis, we identified the high level concept and how we might design programmatically derived features that represent them. Some of these hypotheses included:

- Programmer language, during communication, might indicate fault. For instance, programmers that use language indicative of lack of experience might introduce more faults
- The level of developer communication might imply more or less code fault. For instance, a team that displays more robust and timely communication may show more teamwork, peer review, concern for the code, and thus have less faults.

- Programmers knowledge about specific vulnerabilities. For instance, if user is aware of cross-site scripting (e.g., has XSS tag in StackOverflow profile), they are less likely to introduce that type of security issue
- Concepts labeled by humans might indirectly suggest areas of fault. For instance, code that includes variables with the word "password" or "pwd" might contain more faults (humans indirectly indicating authentication will occur).

The key is that each of these features can be derived from some combination of the derived and raw features. As an example, consider an enrichment that focuses on the types of words that a developer uses in communication. The focus of this feature is that the actual language of communication could suggest faulty code. We can describe this as the following hypothesis:

> *Certain n-grams, words or phrases in discussions inherently suggest more risk, and by identifying them we can predict future faulty code.*

To test this feature programmatically, we can follow these steps:

1. Start with raw features of communications (e.g., log messages, comments, etc.), associated with faults.
2. Compute the n-grams, terms, phrases (e.g. more than one term) from these communications (in step 1).
3. Sample communications regardless of type (e.g., enhancements, etc.)
4. Compute the n-grams, terms, phrases (e.g. more than one term) from these communications (in step 3).
5. Compare the probabilities from #2 and #4 to see if there are highly indicative terms
6. Measure how well these terms predict faults in a test-set (e.g., using leave-one-out cross validation style approaches)

This is just one example, and there are many other such enrichments we could design. The key observation is that much of our work on feature engineering relates to first establishing high-level hypotheses. These, in turn, naturally suggest which enrichments to generate. Deriving additional knowledge, through selective enrichments, is fundamentally part of the exploration process. Hypothesis iteration, refinement, and evaluation is a process that continues to occur throughout the exploration process.

Model generation

The second aspect of Task 2 was to design a process for going from historical data to a model for prediction about code risk with respect to new/unseen code. Our approach recognizes that there are three important steps in this process:

- **Hypothesis generation**, suggesting which independent variables might be responsible for a certain phenomena (e.g., faulty commits)
- **Computation of descriptive statistics** about these variables, to minimally evaluate the hypotheses and understand whether there is sufficient precedence for a deeper evaluation
- **Predictive model generation** by applying machine learning to a training set of such data, to see if unseen data can be forecasted accurately

These phases (especially the first) each need review from humans, but most of them (especially the latter two) can be driven primarily by automation. This is both desirable and feasible. While we may eventually wish for the machine to just "figure it out" from the data, our experience is that with new systems in new domains, human guidance and review of the automation is fundamentally necessary. That should not prevent us from leveraging unexpected findings, but because data from multiple sources can be noisy, inconsistent, and semi-structured, review of machine-driven deductions is typically very helpful.

As part of our Phase I, we developed several hypotheses (as described above) and will develop more under Phase II. Use of descriptive statistics is straightforward, as such statistics (mean, variance, etc.) are well known. However, some filtering may be necessary to look at sub-populations. For example, we may wish to cluster all StackOverflow security-related tags together (which StackOverflow does not do) in order to evaluate the population of developers who have that tag vs developers who do not. As we discuss in the feasibility evaluation below, we found multiple interesting statistics about some of the data we had gathered using such filtering techniques.

In terms of model generation, we looked at learning predictors for various types of data we gathered. Certain predictors were more relevant for some types of data than others. For example, Bayesian text classifiers are more appropriate for classification of log messages and other types of communication, whereas decision trees can be useful for categorical data. Furthermore, statistical modeling techniques like regression analysis (such as logistic regression) can also be applied to fit an equation based on multiple independent variables to commit quality (as the dependent variable). Much depends on the type of data (e.g., text, categorical, continuous, etc.). As discussed in our feasibility evaluation, we leveraged both Bayesian classification and regression to develop predictors for commit risk, for a particular example project.

The final aspect of our design was to investigate a methodology for combining models to reduce the likelihood of false positives. To this end, we looked at constructing an ensemble, such as a simple voting scheme as a basis for generating a prediction. As shown in Figure 8, the idea is to take independent models (i.e., models based on independent variables) and use their individual predictive capabilities in concert to generate a higher quality prediction. As the figure shows, terms used in filenames (which reflect human expressions of system development) may be one predictor, while the commit "part of day" (morning, evening, afternoon, etc.) might be another predictor. While each of these models may slightly predict quality of a commit, their combination may result in greater precision.

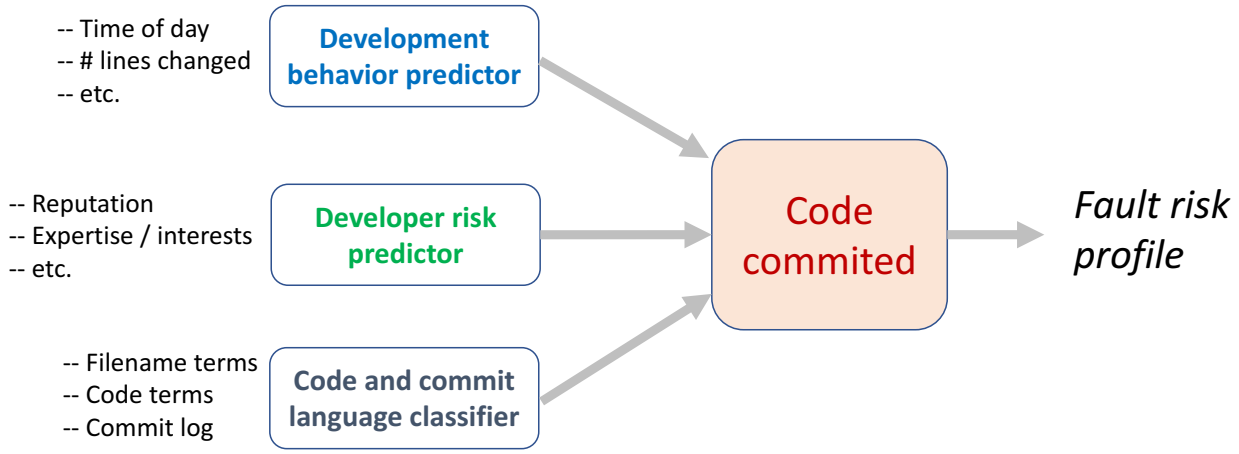


Figure 8: Combining models of risk to analyze committed code

For example, one could imagine a case where commits done in the evening were only somewhat likely (to be risky. Meanwhile, filenames that contained the term "Network" were a bit more likely. We could construct a more conservative virtual predictor that relied on both independent predictors flagging the commit as risky before deciding to predict risk. This approach, better known as ensembling, can often lead to a reduction of false positives, although recall is typically impacted. This can be mitigated by using several models and choosing a majority / threshold of predictors that agree on risk.

2.1.4 Phase I, Task 3: Feasibility evaluation

The final task of our Phase I effort was to conduct a feasibility evaluation. The key goals were to ensure that we could demonstrate selected aspects of our overall approach, as a testament to end-to-end viability. In particular, we had three subtasks to address:

1) Gaining approval to gather the data we are interested in
2) Prototyping selected aspects of the aggregation architecture we designed
3) Using selected samples of the gathered data for analysis and model generation

Shortly after project kickoff, we began work to have an IRB evaluate our protocol, and decide if it qualified as "exempt". To that end, we wrote and submitted an IRB protocol to Western IRB (WIRB) in Oct 2017 and they granted an exemption in Nov 2017. Next, we were informed by our CTR (Mr. Jeffrey Wright) that we should have HRPO review this judgment. We completed HRPO paperwork in December and submitted our material to HRPO via Mr. Wright. In Jan 2018, we were informed by HRPO that the exemption was upheld and that they did not have any remaining concerns. In short, we gained the necessary approvals in place for data gathering.

With the IRB exemption granted, the next steps was to gather data across a variety of sources, source types, and attributes corresponding to the data model we designed in Task 1. We originally proposed working with a small set of projects (25), but later felt that we had enough time and resources to gather data on significantly more, with the hope that the increased amount and range of data would better prepare us for Phase II. By the end of Phase I, we had gathering data on:

- 5000+ open source projects, via GitHub API and git-clone/log style methods
- 3000+ users on StackOverflow (API), using a novel user-linkage mechanism
- 3000+ users and their posts on Twitter (via Twitter API), using a direct link mechanism
- Bugs/issues on 100+ Apache projects via web scraping

In addition, we completed one notable data linking exercise between two of the above data sources. Specifically, we took users from GitHub and attempted to link them to StackOverflow profiles. We did this based on (a) username and (b) profile picture (avatar) similarity. This exercise was important because it allowed us to leverage vast amounts of additional behavioral data. As described in our earlier results, we completed (b) using a computer vision matching technique known as histogram correlation (Stricker & Orengo, 1995). This similarity metric takes two images and essentially looks at how clusters of RGB values align. Histogram correlation is generally regarded as a simple proxy for image similarity in many cases. To our surprise, we found that nearly 40% of the GitHub users we tried with the technique had corresponding profiles on StackOverflow that used both the same username and profile picture. Histogram correlation allowed us to ignore differences in pixelation, size, encoding (PNG vs GIF, etc.). An example of this, for the username "bobince" on GitHub and StackOverflow is shown in Figure 9. These two avatar images have a histogram correlation of 0.97, indicating very significant correlation.

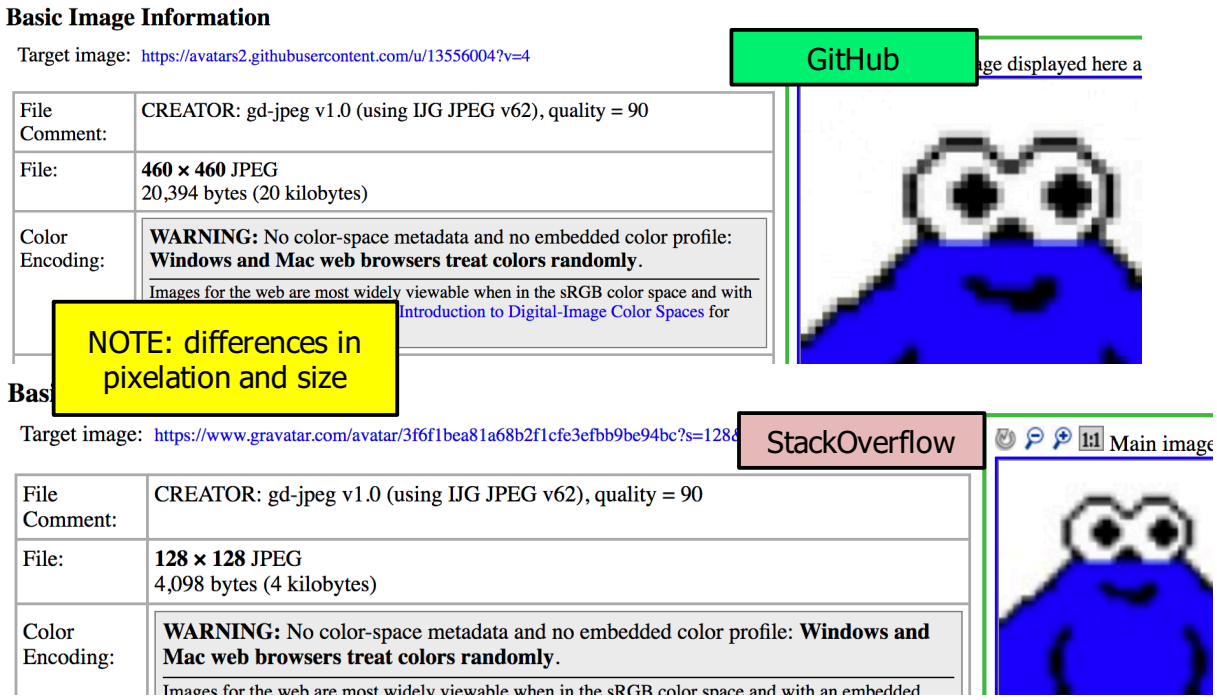


Figure 9: Using histogram correlation to link users

We also explored an advanced enrichment technique related to linking users. Specifically, we wanted to derive "interests" for users, based on their Twitter posts. More generally, we wanted to summarize their posting behavior, so that we could potentially draw conclusions about how interests mapped to development behavior/quality. For example, “do developers who post more about technology or security do more reliable (fault tolerant) development?”. In exploring the feasibility of looking at such derived variables, we focused on those developers that had explicitly listed a Twitter profile. We harvested a recent collection of their posts and then ran through a process that we developed in previous work (Macskassy & Michelson, 2010) to classify those posts in terms of their subject matter.

The first step in this process of deriving interests from a set of social media posts is to aggregate a number of "posts" (e.g., social messages) for that particular account. We then extract entities from the text, including proper nouns and hashtag words. The next step involves entity matching - namely figuring out which entity in the real-world is being referenced in the text. As shown in Figure 10 below, *Arsenal* could be a soccer team, a location that contains arms, or something else. To disambiguate the entities, we pass the entity (e.g., Arsenal) and the context (all of the words in the post, without that entity) and compare it to the text from a knowledge base (Wikipedia in this case). So, the text from the Wikipedia page associated with Arsenal is compared to the text ("winger, walcott, becks, England, ..."), as is the text from Arsenal the armory, etc. We then choose the entity from Wikipedia that maximizes the overlap according to a language model. Once each entity is disambiguated, we have subsets of the ontology centered around those entities, and associated weights (e.g, the more one mentions Arsenal, the more weight is given to that part of the ontology). We then find the most commonly occurring categories of the ontology (e.g., Arsenal is categorized as "English Football") and the most frequently occurring categories become those associated with the social user.

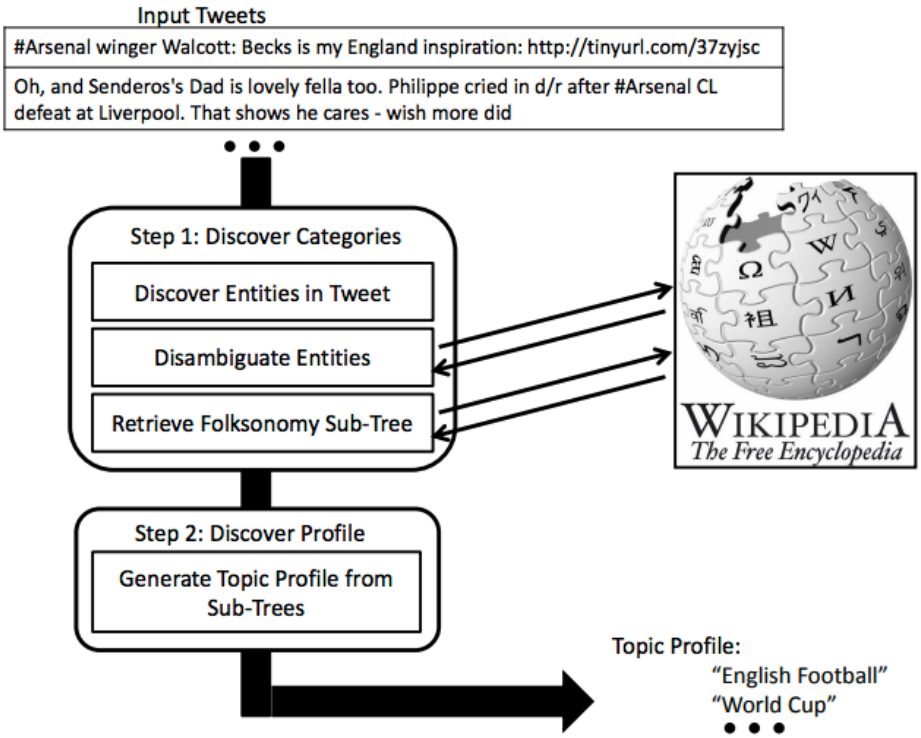


Figure 10: Topic classification of user social media posts

Next, we experimented with applying a variety of descriptive statistics to the data we collected. We were able to establish a number of findings about the data, including several related to how certain development project metadata correlated with number of faults reported, how self-reported developer bios correlated with StackOverflow reputations, and so forth.

Finally, we tested our ability to generate models from the data. To ensure the feasibility of our capability to model different combinations of variables, we looked at the challenge of predicting phenomena involving data collected from multiple sources. We chose StackOverflow reputation as the dependent variable, and then attempted to learn a predictor for that using a range of GitHub and StackOverflow variables. After a descriptive statistics review, we chose a set of variables that we deemed independent of each other, which we felt might be candidates at predicting reputation. These included:

- # of questions asked on StackOverflow
- # of answers provided on StackOverflow
- Whether or not user contributed to PHP, JavaScript, Ruby, or Python repositories on GitHub
- Having published GitHub Gists
- Having StackOverflow biography defined, and length of that bio
- Number of followers on GitHub

We ran an ordinary least squares (OLS) multiple regression against these set of variables to yield the results shown in Figure 11.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:          so_reputation   R-squared:                       0.937
Model:                            OLS   Adj. R-squared:                  0.937
Method:                 Least Squares   F-statistic:                     5369.
Date:                Wed, 04 Jul 2018   Prob (F-statistic):               0.00
Time:                        09:23:35   Log-Likelihood:                -31738.
No. Observations:                3269   AIC:                         6.349e+04
Df Residuals:                    3260   BIC:                         6.355e+04
Df Model:                           9
Covariance Type:            nonrobust
================================================================================================
                                   coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------------------------
contributes_to_php_repo       -619.0033    134.582     -4.599      0.000    -882.877    -355.130
contributes_to_javascript_repo   79.7548    146.912      0.543      0.587    -208.295     367.805
followers                         0.8770      0.153      5.717      0.000       0.576       1.178
public_gists                      9.3640      1.645      5.693      0.000       6.139      12.589
so_answers                       44.7332      0.230    194.590      0.000      44.283      45.184
so_questions                     43.3558      2.618     16.561      0.000      38.223      48.489
has_stackoverflow_bio           -49.9999    145.751     -0.343      0.732    -335.772     235.772
so_ratio_answer_questions        -8.4814      2.452     -3.459      0.001     -13.289      -3.674
length_so_bio                     0.5790      0.577      1.003      0.316      -0.553       1.711
==============================================================================
```

Figure 11: OLS regression of selected variables between StackOverflow and GitHub

Considering only significant variables (where abs(t) >= 3), we were able to conclude that:

- the model fits the data well (i.e., $R^2 = 0.937$)
- providing answers and/or questions are the most positively associated with reputation (this is predictable, since one does not get a high reputation without participating)
- having GitHub followers is not very indicative of reputation
- contributing to PHP repositories (via GitHub) was negatively correlated with reputation

We then looked at what the predictive quality of this logistic regression model and, using 5-fold cross validation, found that the F-measures averaged about 0.85, which is very encouraging about our predictive ability (i.e., tradeoff between recall and precision). We compared this to models generated on the same data via Gaussian Naïve Bayes and decision trees, and found that logistic regression outperformed both. While the phenomena we were predicting were not exactly the ultimate goal of this SBIR (i.e., to predict faulty commits/code), this exercise nevertheless demonstrates our ability to go from descriptive statistics, to model generation, to analysis of model optimization, which is the same process we will follow in Phase II.

3 Phase II Objectives

Having designed the key software components in Phase I, our focus tuned to building a fully-functional prototype that combines data aggregation from multiple heterogeneous sources of software engineering data with an analytical component that will learn and evaluate models of code risk. In particular, our objectives are:

- **Objective 1: Multi-source aggregation of software engineering behavioral data.** Our first objective was to build an aggregation system, based on our Phase I design, to collect and link software engineering data from multiple heterogeneous sources.
- **Objective 2: Analysis and generation of machine learning models to predict code risk**. The second objective of this Phase II was to build a system, based on our Phase I design, for analyzing historical software engineering data so that models of risk can be produced. These models can then be used to predict risk on new repositories and commits not seen previously, based on coding and social-related trends from the historical data, in conjunction with any local history/metadata from the repository being analyzed.

- **Objective 3**: **Testing and evaluation**. The third and final objective we have in this Phase II is to evaluate the functioning prototype we built.

4 Phase II Work

In this section, we discuss the key findings and advancements as part of Phase II. To achieve the objectives outlined above, this work focused on three corresponding activities:

Multi-source aggregation

We combined social coding sources like GitHub, with other social/engineering sources, such as Stack Overflow, to obtain richer insight into the basis for software engineering faults. In short, allowed the analysis component to leverage coding behavioral variables and social variables as the basis for predicting code risk. The aggregation system included multiple mechanisms for collecting data, including API-based gathering, extraction from HTML, and source control automation.

Model learning

We leveraged our Phase I work on a methodology for developing and evaluating hypotheses. This corresponded to the development of several models, automatically generated using a variety of machine learning techniques. To further broaden the applicability of our models, built a set of automated "experts", where each expert will specialize in determining risk based on particular variable or set of variables that are independent from other experts. We then combined evaluations from these experts as a way to increase accuracy and reduce false positives. Our resulting evaluator can thus examine a new repository and produce a risk determination based on such data.

Evaluation and testing

To do the evaluation, aggregated data from open source projects, software vulnerabilities, and developer social media (e.g., GitHub, StackOverflow) profiles. With this data gathered, we built models based on historical commits to these projects, and then used these as the basis to predict risk in future commits or in new open source projects. To evaluate our work, conducted a cross-validation style test in which we trained on selected portions of our corpus while testing on unseen portions; we then rotated that sampling across different combinations of such samples, so that we could measure the robustness of our models.

4.1 Summary of accomplishments

We will describe the detail associated with our work below, but here we highlight some of the key accomplishments of this work:

Objective 1 (Data aggregation):

- Built tools/software to
  - Gather and instrument GitHub-based software repositories
  - Identify faults based on "fix commits" in Git-based projects
  - Sample data from the GitHub archive (GHArchive)
  - Gather + extracted GitHub commit info from CVEs for training
  - Computed term frequencies in code gathered from GitHub archive tool
  - Match developer profiles between GitHub and Stack Overflow, using computer vision-based techniques
- Gathered and instrumented detailed data for 60 well known open source tools/systems (written in a variety of languages, including C++, PHP, Python), including: curl, wget, nginx, etc.
- Gathered social media data from Stack Overflow related to software engineering knowledge

Objective 2 (Model Learning):

- Built machine learning "code risk" pipeline which:
  - Uses data generated by Objective 1 tools to model faulty commits
  - Includes several types of ML classifiers
  - Combines outputs from ML classifiers into an ensemble-style meta predictor (using a simple voting scheme as well as XGBoost)
  - Leverages multiple risk ranking/scoring schemes to judge risk
  - Includes history-based derived features
  - Combined word embedding (Fasttext) with traditional ML methods (e.g., decision trees) in ensembles
- Learned NLP-style word embeddings for source code, for predicting risk based on code terms
  - Built "memory leak" Fasttext classifiers based on large GitHub samples
  - Built additional Fasttext classifier models based on local repositories
- We have experimented with alternative, deep learning approach
  - Using Google AutoML
  - Compared our work with AutoML and our own analysis pipeline, based on engineered data.

Objective 3 (Evaluation):

- Completed a detailed evaluation of our 60+ software repositories
- Built heatmap and graphical visualizations to demonstrate our results

4.2 Data aggregation and engineering

The first step in our work was to aggregate data from multiple sources. These included:

- **GitHub**: where we obtained all of the software repositories we analyzed for this work
- **NVD**: source of vulnerability data, many of which point to GitHub commits
- **StackOverflow**: developer Q&A forum; we tie GitHub contributors to StackOverflow users
- **GitHub Archive** (GHA): Historical GitHub activity aggregation data

For this work, extending Phase I activities, we cloned 50+ well-known open source repositories and explored a subset of these in detail for our analysis. In general, our process consisted of the following:

(i) Clone repository
(ii) Extract activity data (e.g., git log) and store in a relational database so that we can write ad-hoc queries against this set of information
(iii) Identify faults based on 3 different fault classification schemes and store "blame" information in the database
(iv) Gather GitHub metadata (e.g., developer names, bios, etc)
(v) Gather StackOverflow data for developers, based on the type of user similarity techniques discussed in Phase 1 of this work
(vi) Gather GitHub archive data based on fault classifications defined in (iii); this was known as "global" data (vs "local" repo data)

To accomplish (iii) and (vi) above, we used three different fault scenarios:

- Memory leaks (fault class #1)
- SQL injection (fault class #3)
- Program crash (fault class #5)

We label each of the above with a fault class ID (e.g., 1, 3, or 5) for simplified labeling. In later parts of this report, we will refer to these classes as shorthand.

For each of these scenarios in the repositories we examined, we looked for evidence of fixes. To do this, we searched commit log messages; for example, to identify fixes to memory leaks, we searched log messages that contained the phrases "fixed leaks" or "fixed memleaks", etc. Once these commits were identified, we looked at what lines were deleted as part of the fix; these were assumed to be part of the issue. Admittedly, this is not always the case; sometimes, a fix involves polishing code which is already correct. Similarly, commits that solely consist of "fix memleaks" might actually do more than that. Both critiques are fair to apply to our work. That said, there are flaws like this with the alternative approaches we considered and the impact of these flaws, if any, would be minor (because they are not the common case).

To help facilitate information extraction and fault finding, we built two tools: Commit Xray and Fault Finder. The former converted commit and developer data from each repo into the database. Fault Finder, in contrast, automatically identified relevant commits (such as "fix memleaks") and then identified the commits which introduced code that was ultimately deleted by the fix. Armed with this training data, we then set out to learn models of risk.

4.3 Model learning and evaluation

Here we describe our work towards developing an approach for the automated assessment of the risk of faults in a code repository. Specifically, our aim is to predict code fault risk at the level of individual code commits. For any individual code commit we want a probabilistic assessment of the risk that it is *faulty*. Our work explores our starting hypothesis that multiple factors such as properties of the code repository itself, context of individual code commits, as well as characteristics and also the behavior of developers all influence the fault risk. Our approach is based on exploring and uncovering the association between these potentially influencing factors and the occurrence of code faults. We have done this based on real data from (open source) software code repositories, that includes details about code commits and changes and also information about contributing developers.

The data mining over this data has two closely related objectives. One is to uncover explicit correlations (if any) between any of the many potential influencing factors, and code faults. The other is to *predict* the risk of fault, for any given code commit when new code is checked in. The algorithms for such data mining are thus based on 1) statistical analysis and 2) machine learning. We have explored a number of different statistical analysis and machine learning methodologies for fault risk prediction. And experimentally determined the best performing ones. Also identified certain approaches that do not seem to predict well.

The report is organized as follows. First, we describe the approaches we evaluated for risk assessment, to determine the most effective techniques to apply to this problem. Next, we describe statistical risk analysis. Finally, we describe the learning curve analysis of classification.

An Appendix section, at the end of this report, provides more enumerated detail on some of the evaluations.

4.3.1 Approach

Fundamentally our approach is based on applying statistical analysis on the data and experimentally evaluating a wide variety of machine learning classification algorithms to determine effective predictors for fault risk. In this section we first scope the potential *features* i.e,, code, developer or process based factors that may be predictors of fault risk. We then outline the different algorithms, based on statistical analysis or machine learning, that we evaluated for fault risk prediction.

4.3.2 Classification features

We considered the various aspects that can potentially influence code faultiness, and determined three distinct categories of potentially influencing features. The categories relate to properties of particular, those of developers, and those of the code.

Commit related features Individual commits have interesting aspects and we derive commit related features from the following properties:

(i) Commit activity. There are quantitative measures of commit activity, specifically the modifications, additions, deletions counts associated with each commit. Each of the counts becomes a commit related feature.
(ii) Commit time. We derive features from the (UTC) timestamp of a commit, specifically the hour of day, portion of the day (early morning, morning, afternoon etc.), and the day of the week.
(iii) Descriptive text. Each commit has a commit message associated with it, where the code committer pit's in a (free text) message about that commit. The (entire) commit message text becomes a feature (we will discuss shortly the form we represent this text feature in).
(iv) Commit moving history. Certain features are derived from the moving window history of commits. As an example, for any commit, the number of commits in that code repository in the past week/month/year etc. becomes a moving history based feature. The intuition behind such features is that a pattern in the *recent* history of a particular commit may bear upon the faultiness of that commit.

Developer related features These are features related to software developers associated with a code repository or associated with particular commits in the code. These include:

(i) Developer identity. Identifying fields such as the login ID of a developer and their name.
(ii) Developer reputation Features related to developer reputation metrics, such as their number of "likes", or "star ratings" etc.
(iii) Developer location.
(iv) Developer knowledge and skills. Features derived from the set of programming languages.

Code related features These are aspects of the code itself and include:

(i) File types. The types of code (or documentation) files associated with the commit (a ".c" file vs a ".exe" vs a ".html" file etc)
(ii) Folder paths. The folder names and path to the files associated with the commit.

Category	Features
Commit features	**Activity based:** *modifications count commit, additions count commit, deletions count commit* **Temporal based:** *committer date hour, committer date weekday, author date hour, author date weekday* **Descriptive text based:** *commit_message_text, commit_code_text, file_path_tokens_text* **Moving history based:** *number_commits, number_commits, number_faulty_commits, days_since_last_faulty_commit*

Developer features	**Identity:** *committer login, author login, committer email, author email* **Reputation:** *following count, followers count, public repos count* **Location:** *location* **Identity:** *number_of_languages, if_language known*
Code Features	**File type based:** *fileext* **File organization based:** *path1, path2*

4.3.3 Fault Risk Prediction

The fundamental problem is that of predicting whether a commit will result in a fault or not. Rather than a binary Y/N of whether the commit will be faulty, we want a probabilistic *likelihood* that a commit will be faulty. Supervised learning ,employing machine learning based classification lends itself well to such a task. The classification problem is formulated at a commit level. For each commit, the problem is to take the various different features (above) and provide a classification of fault risk. The classification target is simply the probability, a number 0-1 (by definition), that the commit will be faulty. As part of supervised learning, any classifier employed is trained on data about commits in a repository where the faulty commits are known.

We evaluated three different kinds of classifier types or frameworks, namely:

1) **Hand-crafted feature driven classifiers** These are classifiers that work off hand crafted features. For instance classifiers such as Decision Tree, SVM, Random Forest and many others that are built into most commonly used machine learning toolkits.
2) **Deep learning classifiers, especially ones within automated machine learning frameworks** These are classifiers in the unsupervised feature learning category where the system itself learns the features to be employed. The deep learning classification is based on a different paradigm of employing multiple layers of neural networks in the classifier. Also, rather than developing custom deep learning models ourselves we use cloud based automated machine learning frameworks where the (deep learning based) models are synthesized automatically. Further, the cloud platform offering the automated machine learning service provides the (typically) heavy computational resources deep learning modeling needs.
3) **Ensembles** In simple terms a classifier ensemble is akin to a committee of experts, where the opinion of multiple experts is taken into account. In this case we assembled and evaluated ensembles of *multiple* different classifiers and under different configurations for (best) combining their inputs.

Type	Classifiers	Features
Discrete	Decision Tree, Random Forest, KNN	*committer login, author login, committer email, author email, modifications count commit, additions count commit, deletions count commit, committer date hour, committer date weekday, author date hour, author date weekday, following count, followers count, public repos count, location, path1, path2, fileext, number of commits (previous day/week/month), days since last commit, number of faulty commits* (previous day/week/month), days since last faulty commit**
Numeric	AdaBoost, SVM, Gaussian Naïve Bayes (GNB), Quadratic Discriminant Analysis (QDA)	*modifications count commit, additions count commit, deletions count commit, committer date hour, committer date weekday, author date hour, author date weekday, following count, followers count, public repos count, number of commits (previous day/week/month), days since last commit, number of faulty commits* (previous day/week/month), days since last faulty commit**
Text	SGDC (Stochastic Gradient Descent) TEXT classifier	*Over (i) commit_message_text (ii) commit_code_text (iii) file_path_tokens_text* Note: Three separate classifiers

4.3.4 Hand-crafted feature driven classifiers

In Table 3 below we provide the results of evaluating different hand crafted classifiers. Some characteristics of the evaluation and also the table format are as follows:

1. We evaluated ten different classifiers, namely Random Forest (RF), Decision Tree (DT), AdaBoost (ABV), Support Vector Machine (SVM), Gaussian Naïve Bayes (GNB), Quadratic Discriminant Analysis (QDA), K-Nearest Neighbor (KNN), TCM, Gradient Descent (SGDC) classifiers for text fields which are commit message text (we call that text classifier TCM), for commit file path tokens TCF), and for deleted code snippet treated as text (TCC).
2. The results are provided by individual repositories, and for each repository for different fault types
    a. For each repository and fault type, we provide the accuracy of commit fault prediction, We report the accuracy for each of the 10 classifiers listed above, the accuracy itself is reported as <precision>, <recall> (<f-measure>)
    b. For each dataset (repository and fault) we also provide the distribution of faulty and non-faulty commits. We also provide the "baseline" accuracy, which measures the precision of identifying faulty commits if one were identifying such commits purely randomly. Which is the fraction of faulty commits in the entire set.
        i. A prefix of "Fault 1;Y:N=129:6426; BL=0.02" thus means that the ratio of faulty to non-faulty commits in this dataset is 129:6326. BL (baseline) is 129/(129+6326) = 0.02
3. For brevity, Table 3 contains a subset of results, for 6 out of the 40 evaluated repositories. The complete results (all repositories) are in the Appendix.

INDIVIDUAL CLASSIFIERS

Table 3. Faulty commit identification accuracy: comparison of multiple individual classifiers.

	RF	DT	ABV	GNB	SVM	QDA	KNN	TCM	TCF	TCC
Repo: nginx										
Fault 1;Y:N=129:6426; BL=0.02|	0.11,0.79(0.19)	0.59,0.74(0.66)	0.09,0.71(0.16)	0.20,0.36(0.26)	0.00,0.0(NA)	0.19,0.37(0.25)	0.10,0.2(0.13)	0.12,0.4(0.18)	0.05,0.84(0.09)	0.08,0.14(0.1)
Fault 5;Y:N=471:6084; BL=0.07|	0.41,0.94(0.57)	0.75,0.84(0.79)	0.37,0.86(0.52)	0.60,0.32(0.42)	0.07,0.03(0.04)	0.53,0.4(0.46)	0.42,0.55(0.48)	0.63,0.82(0.71)	0.40,0.78(0.53)	0.25,0.72(0.37)
Repo: apache										
Fault 1;Y:N=190:31074; BL=0.01|	0.05,0.93(0.09)	0.30,0.48(0.37)	0.02,0.57(0.04)	0.06,0.15(0.09)	0.00,0.01(0.0)	0.05,0.12(0.07)	0.01,0.06(0.02)	0.03,0.29(0.05)	0.02,0.82(0.04)	0.02,0.53(0.04)
Fault 3;Y:N=127:31137; BL=0.0|	0.03,0.74(0.06)	0.34,0.46(0.39)	0.01,0.56(0.02)	0.03,0.07(0.04)	0.00,0.0(NA)	0.03,0.09(0.04)	0.00,0.0(NA)	0.07,0.19(0.1)	0.02,0.74(0.04)	0.02,0.05(0.03)
Fault 5;Y:N=559:30705; BL=0.02|	0.12,0.91(0.21)	0.33,0.47(0.39)	0.04,0.74(0.08)	0.08,0.07(0.07)	0.01,0.03(0.01)	0.07,0.08(0.07)	0.02,0.09(0.03)	0.06,0.47(0.11)	0.04,0.92(0.08)	0.07,0.56(0.12)
Repo: wget										
Fault 1;Y:N=64:3955; BL=0.02|	0.22,0.78(0.34)	0.50,0.68(0.58)	0.04,0.54(0.07)	0.00,0.0(NA)	0.01,0.03(0.01)	0.00,0.0(NA)	0.03,0.08(0.04)	0.12,0.11(0.11)	0.03,0.43(0.06)	0.29,0.05(0.09)
Fault 3;Y:N=20:3999; BL=0.0|	0.00,0.0(NA)	0.00,0.0(NA)	0.00,0.0(NA)	0.00,0.0(NA)	0.00,0.0(NA)	0.00,0.0(NA)	0.00,0.0(NA)	0.00,0.0(NA)	0.01,0.11(0.02)	0.00,0.0(NA)
Fault 5;Y:N=130:3889; BL=0.03|	0.13,0.7(0.22)	0.43,0.43(0.43)	0.09,0.61(0.16)	0.00,0.0(NA)	0.05,0.02(0.03)	0.00,0.0(NA)	0.08,0.12(0.1)	0.14,0.2(0.16)	0.10,0.82(0.18)	0.06,0.06(0.06)
Repo: videolan/vlc										
Fault 1;Y:N=2527:78906; BL=0.03|	0.09,0.75(0.16)	0.23,0.4(0.29)	0.08,0.69(0.14)	0.03,0.99(0.06)	0.03,0.09(0.04)	0.03,0.99(0.06)	0.04,0.2(0.07)	0.07,0.59(0.13)	0.05,0.82(0.09)	0.06,0.49(0.11)
Fault 3;Y:N=1126:80307; BL=0.01|	0.07,0.85(0.13)	0.18,0.39(0.25)	0.03,0.75(0.06)	0.01,0.98(0.02)	0.01,0.04(0.02)	0.01,0.99(0.02)	0.01,0.11(0.02)	0.04,0.52(0.07)	0.03,0.7(0.06)	0.03,0.52(0.06)
Fault 5;Y:N=2653:78780; BL=0.03|	0.10,0.8(0.18)	0.31,0.46(0.37)	0.08,0.77(0.14)	0.03,0.98(0.06)	0.03,0.09(0.04)	0.03,0.98(0.06)	0.05,0.23(0.08)	0.10,0.66(0.17)	0.09,0.71(0.16)	0.07,0.56(0.12)

We have several interesting observations to make from the results and also conclusions that we can draw. Our eventual application goal is to find faulty commits more efficiently, compared to randomly considering commits for examination. That makes precision a more important factor for us. Which is what we will primarily consider, but under a recall of at least 0.1 (10%). With some classifiers we are able to achieve a high precision relative to the baseline (the fault density). For instance for wget Fault 1 we have a fault density defined baseline of 0.02 and are able to achieve precision of 0.22 and 0.50 with the Random Forest and Decision Tree classifiers respectively. In the case of the Decision Tree the fault finding improvement is by a factor 0.50/0.02 = 25, which is significant. Between Random Forest and Decision Trees the latter seem to be performing better. Next best seem to be some of the text classifier (TCN, TCF, TCC) which provide a precision of 0.1 or higher for many of the datasets. The weakest

performers here seem to be the quantitative attribute driven classifiers such as SVM, QDA etc. where the classification accuracy has no improvement over the baseline in most cases.

4.3.5 Ensembles

Next, we constructed ensembles of the (above) ten classifiers. Table 4(a) provides the evaluation for a simple voting based ensemble, for a representative subset of datasets (Appendix_Table 4(a) provides the complete results).

Table 4(a): Faulty commit identification accuracy: ensemble. *Below provides faulty commit identification accuracy in terms of precision and recall for ensemble of 10 classifiers.*

Votes:	2	4	6	8	9	10
nginx - Fault 1;Y:N=129:6426; BL=0.02| :	[0.09, 0.81(0.16)]	[0.2, 0.71(0.31)]	[0.31, 0.37(0.34)]	[0.75, 0.09(0.16)]	[1.0, 0.01(0.02)]	[0.0, 0.0(NA)]
nginx - Fault 5;Y:N=471:6084; BL=0.07| :	[0.32, 0.97(0.48)]	[0.61, 0.88(0.72)]	[0.82, 0.69(0.75)]	[0.98, 0.33(0.49)]	[0.98, 0.19(0.32)]	[0.0, 0.0(NA)]
apache - Fault 1;Y:N=190:31074; BL=0.01| :	[0.03, 0.93(0.06)]	[0.1, 0.63(0.17)]	[0.39, 0.14(0.21)]	[0.5, 0.01(0.02)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]
apache - Fault 3;Y:N=127:31137; BL=0.0| :	[0.02, 0.8(0.04)]	[0.15, 0.38(0.22)]	[0.29, 0.03(0.05)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]
apache - Fault 5;Y:N=559:30705; BL=0.02| :	[0.05, 0.94(0.09)]	[0.14, 0.74(0.24)]	[0.39, 0.23(0.29)]	[0.67, 0.02(0.04)]	[1.0, 0.0(0.0)]	[0.0, 0.0(NA)]
wget - Fault 1;Y:N=64:3955; BL=0.02| :	[0.09, 0.81(0.16)]	[0.31, 0.27(0.29)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]
wget - Fault 3;Y:N=20:3999; BL=0.0| :	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]
wget - Fault 5;Y:N=130:3889; BL=0.03| :	[0.13, 0.82(0.22)]	[0.29, 0.39(0.33)]	[0.33, 0.02(0.04)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]	[0.0, 0.0(NA)]
videolan - Fault 1;Y:N=2527:78906; BL=0.03| :	[0.03, 1.0(0.06)]	[0.06, 0.92(0.11)]	[0.13, 0.63(0.22)]	[0.25, 0.19(0.22)]	[0.26, 0.04(0.07)]	[0.3, 0.0(0.0)]
videolan - Fault 3;Y:N=1126:80307; BL=0.01| :	[0.01, 1.0(0.02)]	[0.03, 0.92(0.06)]	[0.09, 0.63(0.16)]	[0.19, 0.14(0.16)]	[0.19, 0.02(0.04)]	[0.0, 0.0(NA)]
videolan - Fault 5;Y:N=2653:78780; BL=0.03| :	[0.03, 1.0(0.06)]	[0.07, 0.93(0.13)]	[0.17, 0.67(0.27)]	[0.39, 0.27(0.32)]	[0.45, 0.06(0.11)]	[0.5, 0.01(0.02)]

For most datasets the fault classification accuracy precision increase at the cost of recall as we increasing the voting threshold (as expected with a voting based ensemble). However, the ensemble is unable to provide accuracy improvement over the better performing individual classifiers.

In Table 4(b) we provide results for another ensemble over the same ten classifiers, where we now employ XGBoost as a *meta-classifier* over the output of the individual classifiers.

Table 4(b): CLASSIFIER ENSEMBLE: XGBOOST (meta-classifier) BASED

Dataset	Accuracy [Precision, Recall (F-Measure)]	Dataset	Accuracy [Precision, Recall (F-Measure)]
nginx , Fault 1;Y:N=129:6426; BL=0.02	[0.57, 0.73(0.64)]	wget , Fault 1;Y:N=64:3955; BL=0.02	[0.36, 0.11(0.17)]
nginx , Fault 5;Y:N=471:6084; BL=0.07	[0.76, 0.84(0.8)]	wget , Fault 3;Y:N=20:3999; BL=0.0	[0.0, 0.0(NA)]
apache , Fault 1;Y:N=190:31074; BL=0.01	[0.29, 0.45(0.35)]	wget , Fault 5;Y:N=130:3889; BL=0.03	[0.49, 0.24(0.32)]
apache , Fault 3;Y:N=127:31137; BL=0.0	[0.3, 0.39(0.34)]	videolan/vlc , Fault 1;Y:N=2527:78906; BL=0.03	[0.23, 0.41(0.29)]
apache , Fault 5;Y:N=559:30705; BL=0.02	[0.35, 0.48(0.4)]	videolan/vlc , Fault 3;Y:N=1126:80307; BL=0.01	[0.19, 0.41(0.26)]
		videolan/vlc , Fault 5;Y:N=2653:78780; BL=0.03	[0.31, 0.45(0.37)]

Table 4(c): CLASSIFIER ENSEMBLE: XGBOOST (meta-classifier) BASED with only Decision Tree and Random Forest components

nginx , Fault 1;Y:N=129:6426; BL=0.02	[0.53, 0.73(0.61)]	wget , Fault 1;Y:N=64:3955; BL=0.02	[0.49, 0.49(0.49)]
nginx , Fault 5;Y:N=471:6084; BL=0.07	[0.77, 0.8(0.78)]	wget , Fault 3;Y:N=20:3999; BL=0.0	[0.0, 0.0(NA)]
apache , Fault 1;Y:N=190:31074; BL=0.01	[0.3, 0.51(0.38)]	wget , Fault 5;Y:N=130:3889; BL=0.03	[0.4, 0.42(0.41)]
apache , Fault 3;Y:N=127:31137; BL=0.0	[0.35, 0.42(0.38)]	videolan/vlc , Fault 1;Y:N=2527:78906; BL=0.03	[0.23, 0.38(0.29)]
apache , Fault 5;Y:N=559:30705; BL=0.02	[0.37, 0.46(0.41)]	videolan/vlc , Fault 3;Y:N=1126:80307; BL=0.01	[0.19, 0.43(0.26)]
		videolan/vlc , Fault 5;Y:N=2653:78780; BL=0.03	[0.32, 0.47(0.38)]

Even with XGBoost the ensemble accuracy does not outperform the accuracy of the best performing individual classifier (the Decision Tree).

4.3.6 Deep Learning Classifiers

Next we evaluate deep learning classifiers which are of a fundamentally different kind. We used automated machine learning, specifically Google Cloud AutoML. In AutoML Tables we can provide the same set of features as a signature associated with each commit. AutoML then constructs an "effective" deep learning classifier for this data automatically

Table 5. Automated machine learning

Dataset	Best In-house (F-Measure)	Best In-house (Precision)	AutoML
nginx, 1	0.59,0.74 (0.66)	0.59,0.74 (0.66)	0.38, 0.25 (0.31)
apache, 3	0.34,0.46(0.39)	0.35, 0.42(0.38)	0.14, 0.04 (0.06)
curl, 1	0.21, 0.23(0.22)	0.57, 0.04(0.07)	0.52, 0.08 (0.14)
wget, 3	0.00,0.0(NA)	0.00,0.0(NA)	0.00, 0.00 (0.00)
videolan, 5	0.43,0.43(0.43)	0.43,0.43(0.43)	0.53, 0.29 (0.38)
podofo, 1	0.22, 0.62(0.32)	0.33, 0.15(0.21)	0.33, 0.33 (0.33)

Of the six datasets evaluated auto ml is better in only one.

4.3.7 fastText Integration

The final addition to the suite of classifiers was fastText. As described above, the text classifier applied to the various text fields is an SGDC based text classifier. The model developed in such classifiers is based on the distribution of tokens (words) in the text documents. fastText is a *word embeddings* based text classifier that employs a vector based representation of words in the text. The vector based representation helps determine semantic similarity between words and a text classification model developed over this representation is more sophisticated (compared to naïve bayes or gradient descent based text models).

In the fastText addition we consider the entire *code* in a code file associated with a commit as text. We generated fastText classifications over the code as text in three different ways. We trained a *global model* which was trained on code (text) across multiple commits. We also trained two variations of *local models* where for each commit only the code from files associated with that commit is used.

- FastText has various classifier models which are encoded as
  - FLM = File local model, GM = Global model, LM = Local model, LM2 = Local model 2
  - In each table below, we always take the File local model classifier (FLM), and one of the global model and two local models
- Results are provided as pairs of (precision, recall)

Tables 6 (a), (b), (c) provide the fault classification accuracy, as (precision, recall), for the fastText classifiers. We also provide the accuracy achieved with the other feature-driven classifiers we have used, for comparison. Note that FLM stand for fastText Local Model and FGM stands for fastText Global Model.

Table 6(a): FastText Global

	FLM	GM	RF	DT	ABV	GNB	SVM	QDA	KNN	TCM	TCF	TCC
nginx	(0.01, 0.43)	(0.0, 0.0)	(0.64, 0.5)	(0.35, 0.64)	(0.05, 0.21)	(0.05, 0.57)	(0.08, 0.21)	(0.08, 0.21)	(0.0, 0.0)	(0.07, 0.29)	(0.03, 0.71)	(0.02, 0.36)
apache	(0.01, 0.88)	(0.01, 0.12)	(0.44, 0.55)	(0.05, 0.83)	(0.0, 0.02)	(0.01, 0.55)	(0.04, 0.1)	(0.04, 0.1)	(0.0, 0.0)	(0.02, 0.29)	(0.02, 0.9)	(0.03, 0.74)
wget	(0.04, 0.69)	(0.04, 0.04)	(0.4, 0.31)	(0.14, 0.92)	(0.12, 0.08)	(0.05, 0.38)	(0.0, 0.0)	(0.0, 0.0)	(0.25, 0.04)	(0.07, 0.08)	(0.04, 0.5)	(0.0, 0.0)

Table 6(b): FastText Local

	FLM	FGM	RF	DT	ABV	GNB	SVM	QDA	KNN	TCM	TCF	TCC
nginx	(0.06, 0.5)	(0.0, 0.0)	(0.64, 0.5)	(0.35, 0.64)	(0.05, 0.21)	(0.05, 0.57)	(0.08, 0.21)	(0.08, 0.21)	(0.0, 0.0)	(0.07, 0.29)	(0.03, 0.71)	(0.02, 0.36)
apache	(0.03, 0.64)	(0.01, 0.12)	(0.44, 0.55)	(0.05, 0.83)	(0.0, 0.02)	(0.01, 0.55)	(0.04, 0.1)	(0.04, 0.1)	(0.0, 0.0)	(0.02, 0.29)	(0.02, 0.9)	(0.03, 0.74)
wget	(0.05, 0.08)	(0.04, 0.04)	(0.4, 0.31)	(0.14, 0.92)	(0.12, 0.08)	(0.05, 0.38)	(0.0, 0.0)	(0.0, 0.0)	(0.25, 0.04)	(0.07, 0.08)	(0.04, 0.5)	(0.0, 0.0)

Table 6(c): FastText Local 2

FLM	FGM	RF	DT	ABV	GNB	SVM	QDA	KNN	TCM	TCF	TCC

nginx	(0.0, 0.0)	(0.0, 0.0)	(0.64, 0.5)	(0.35, 0.64)	(0.05, 0.21)	(0.05, 0.57)	(0.08, 0.21)	(0.08, 0.21)	(0.0, 0.0)	(0.07, 0.29)	(0.03, 0.71)	(0.02, 0.36)
apache	(0.0, 0.0)	(0.01, 0.12)	(0.44, 0.55)	(0.05, 0.83)	(0.0, 0.02)	(0.01, 0.55)	(0.04, 0.1)	(0.04, 0.1)	(0.0, 0.0)	(0.02, 0.29)	(0.02, 0.9)	(0.03, 0.74)
wget	(0.0, 0.0)	(0.04, 0.04)	(0.4, 0.31)	(0.14, 0.92)	(0.12, 0.08)	(0.05, 0.38)	(0.0, 0.0)	(0.0, 0.0)	(0.25, 0.04)	(0.07, 0.08)	(0.04, 0.5)	(0.0, 0.0)

None of the fastText models are able to demonstrate a viable classification accuracy. The accuracy achieved is comparable with the weakest of the feature driven classifiers.

We also integrated the fastText classifiers into a (voting based) ensemble with the existing ten feature driven classifiers. Tables 7(a), (b), (c) provide the results for the three different fastText models integrated into the ensemble.

Table 7(a): FastText Global

VOTES	**2**	**4**	**6**	**8**	**10**	**11**
nginx	(0.02, 0.71)	(0.07, 0.43)	(0.29, 0.29)	(1.0, 0.21)	(1.0, 0.14)	(0, 0.0)
apache	(0.01, 1.0)	(0.04, 0.86)	(0.11, 0.36)	(0.6, 0.07)	(0, 0.0)	(0, 0.0)
wget	(0.07, 0.96)	(0.17, 0.35)	(0.33, 0.04)	(0, 0.0)	(0, 0.0)	(0, 0.0)

Table 7(b): FastText Local

VOTES	**2**	**4**	**6**	**8**	**10**	**11**
nginx	(0.04, 0.79)	(0.11, 0.5)	(0.4, 0.29)	(1.0, 0.21)	(1.0, 0.14)	(0, 0.0)
apache	(0.02, 0.98)	(0.05, 0.76)	(0.13, 0.31)	(0.67, 0.1)	(0, 0.0)	(0, 0.0)
wget	(0.1, 0.85)	(0.15, 0.15)	(0.0, 0.0)	(0, 0.0)	(0, 0.0)	(0, 0.0)

Table 7(c): FastText Local 2

VOTES	**2**	**4**	**6**	**8**	**10**	**11**
nginx	(0.04, 0.71)	(0.12, 0.43)	(0.38, 0.21)	(1.0, 0.14)	(0, 0.0)	(0, 0.0)
apache	(0.02, 0.98)	(0.06, 0.69)	(0.17, 0.17)	(0.0, 0.0)	(0, 0.0)	(0, 0.0)
wget	(0.11, 0.81)	(0.2, 0.15)	(0, 0.0)	(0, 0.0)	(0, 0.0)	(0, 0.0)

4.3.8 Explain Yourself: Feature Contribution to Classification

The above evaluations demonstrated that certain classifiers can provide a high classification accuracy in predicting faulty commits. Specifically classifiers such as Decision Tress that can leverage discrete features or (gradient descent based) text classifiers over particular text features. Knowing which *features* are most predictive for fault risk is of direct practical interest for improving code quality. For instance if we determined code faults (in a particular project) are highly correlated (even negatively) with certain days of the week that the commits are made then corrective action can be taken.

We conducted a *feature importance analysis* for various classifiers to determine significant features, for three different types of classifiers that demonstrated good fault classification accuracy.

Feature importance of discrete features We selected the Decision Tree classifier and conducted feature importance analysis using built-in feature importance scoring capabilities in the SciKit Learn toolkit. Table 8 provides, for several datasets, the list of features with non-zero feature importance scores. The importance scores are also provided along with.

Table 8: Feature importance

1,1	2,5	6,3	11,1
author_name 0.93816	author_date_hour 0.91284	author_name 0.87569	author_name 0.94124
additions_count_commit 0.04141	author_name 0.03254	additions_count_commit 0.06819	additions_count_commit 0.03879
modifications_count_commit 0.01663	additions_count_commit 0.01465	modifications_count_commit 0.02683	deletions_count_commit 0.00879
last_month_commits 0.00324	committer_name 0.01210	last_month_commits 0.01550	modifications_count_commit 0.00612
deletions_count_commit 0.00041	committer_date_hour 0.00885	deletions_count_commit 0.01080	last_month_commits 0.00456
last_month_faulty_commits 0.00014	last_month_faulty_commits 0.00851	last_month_faulty_commits 0.00295	last_month_faulty_commits 0.00045
	modifications_count_commit 0.00291		
1,5	last_month_commits 0.00264	6,5	12,1
committer_name 0.78727	deletions_count_commit 0.00198	author_name 0.91372	committer_date_hour 0.61274
modifications_count_commit 0.10395	committer_date_weekday 0.00155	last_month_faulty_commits 0.04784	modifications_count_commit 0.16359
author_name 0.08546	committer_email_type 0.00139	additions_count_commit 0.03061	committer_name 0.07454

deletions_count_commit 0.00907		deletions_count_commit 0.00578	author_name 0.05906
additions_count_commit 0.00675	4,1	modifications_count_commit 0.00198	additions_count_commit 0.05896
last_month_commits 0.00548	committer_date_hour 0.90417	last_month_commits 0.00007	committer_email_type 0.01382
last_month_faulty_commits 0.00200	author_name 0.04672		last_month_faulty_commits 0.00893
	modifications_count_commit 0.04024	7,1	deletions_count_commit 0.00447
2,1	committer_name 0.00498	author_name 0.80274	last_month_commits 0.00390
author_name 0.96165	additions_count_commit 0.00193	modifications_count_commit 0.14327	
additions_count_commit 0.03637	deletions_count_commit 0.00071	additions_count_commit 0.04876	12,3
modifications_count_commit 0.00128	committer_email_type 0.00068	deletions_count_commit 0.00261	committer_date_hour 0.71650
last_month_commits 0.00051	last_month_commits 0.00053	last_month_commits 0.00132	deletions_count_commit 0.09869
deletions_count_commit 0.00017	last_month_faulty_commits 0.00000	last_month_faulty_commits 0.00132	modifications_count_commit 0.06664
last_month_faulty_commits 0.00017			author_name 0.05477
	4,5	7,3	committer_name 0.03478
2,3	author_name 0.87956	author_name 0.88351	additions_count_commit 0.01737
committer_date_weekday 0.75268	additions_count_commit 0.05294	modifications_count_commit 0.08301	committer_email_type 0.00860
author_name 0.07270	modifications_count_commit 0.03502	deletions_count_commit 0.01331	last_month_faulty_commits 0.00222
additions_count_commit 0.05066	last_month_commits 0.01374	additions_count_commit 0.01300	last_month_commits 0.00042
committer_date_hour 0.05057	last_month_faulty_commits 0.01048	last_month_commits 0.00586	
committer_name 0.04726	deletions_count_commit 0.00835	last_month_faulty_commits 0.00134	12,5
deletions_count_commit 0.00996			committer_date_hour 0.55581
last_month_faulty_commits 0.00675	6,1	7,5	modifications_count_commit 0.15488
modifications_count_commit 0.00590	author_name 0.98253	author_name 0.84193	committer_name 0.09069
last_month_commits 0.00325	additions_count_commit 0.01477	modifications_count_commit 0.15131	deletions_count_commit 0.07367
committer_email_type 0.00025	modifications_count_commit 0.00120	additions_count_commit 0.00421	author_name 0.04848
	last_month_faulty_commits 0.00098	deletions_count_commit 0.00085	additions_count_commit 0.03587
	last_month_commits 0.00044	last_month_commits 0.00085	last_month_faulty_commits 0.01884
	deletions_count_commit 0.00011	last_month_faulty_commits 0.00085	committer_email_type 0.01417

Conclusions:

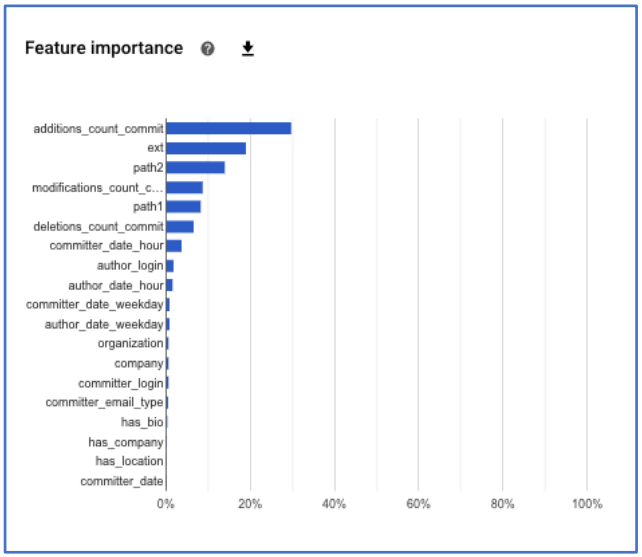
- Developer identifying features, namely author_name and committer_name are the most important features correlated with fault risk.
- Commit activity indicating features which are {additions_count, deletions_count, modificiations_count} are the most important features.
- History based features, such as last_month_commits, last_month_faulty_commits etc. are also correlated
- None of the other features are correlated.

Text Tokens Features For the text classifiers we determined feature importance in terms of the Tfidf scores of the text tokens. The Appendix provides a table with the top 10 most important tokens by Tfidf score, for each dataset

AutoML Feature Importance AutoML also provides built-in feature importance determination functions. Table 9 provides the AutoML feature importance for one dataset. The feature importance for the Decision Tree classifier for the same dataset, is also provided alongside.

Table 9: AutoML feature importance

AutoML



Decision Tree

feature	importance score
author_login	0.92738
additions_count_commit	0.03291
modifications_count_commit	0.02726
deletions_count_commit	0.01244

In Section 2 we described the supervised machine learning classification based formulation of code fault identification. From a practical perspective we are interested in estimating the risk of code faults. Which is the likelihood that any commit will be a fault in the future. A binary prediction of whether or not (Y/N) a commit will be faulty is unrealistic and not even required in application. More useful would be a quantitative assessment of the risk of fault.

In general, for any machine learning classification based approach, the final classification is a probability distribution over the target class instances. In this case the target class “fault” has two instances Y and N. The probability that it belongs to Y (is a fault) is taken as a quantitative measure of fault risk. We then also developed two other schemes for quantifying commit risk, based on statistical measures. We use these measure to conduct a “real world” analysis of fault risk. By real world we mean we take data from real code repositories (as for the classifier based analysis) but we also consider the predicting fault risk for commits in the “future”. We now define the three different risk assessment scoring schemes as well as specifics of this real world evaluation.

Quantitative risk scoring Each commit is assigned a Commit Risk Score (CRS). Commit risk scores can be of the following types.

(i) Ratio based
   a. Risk-Ratio based (RR): For each commit, the commit risk score (CRS) is the product of the (fault) *risk* ratios of individual metadata attributes (attributes with some non-zero correlation with faults)
   b. Odds-Ratio based (OR): For each commit, the commit risk score (CRS) is the product of the (fault) *odds* ratios of individual metadata attributes (attributes with some non-zero correlation with faults)

(ii) Classifier probability based (PR): For each commit, the commit risk score (CRS) is the *probabilistic confidence* provided by the machine learning classifier or particular ensemble meta-classifier employed

Key aspects of the evaluation

- Commits in any dataset are sorted by time, commits prior to a certain point in time are taken in the training set and the remainder (post that time) form the test set
    - The first (earlier, by time) 75% of this time sorted data forms the training set, the remainder 25% "future" data forms the test set
    - It is important to note that we have two different kinds of test / train splits overall:
        - 1) The "cross validation" split where we cross validate evaluation, as in the case of the individual classifier and ensembles evaluations above, splitting in a manner agnostic to commit time.
        - 2) The "temporal" split where data (commit instances) is split by commit time - past and future
- For the classifier based scheme we have applied the ensemble classifier using XGBoost as meta-classifier
- In ratio based schemes we report (only) results with the odds ratio scheme (OR) as we have determined that to be consistently superior to the risk ratio based scheme (RR)
- The test set commits are ranked by the OR and PR schemes

(Commit) History based features:

- One day history
    - Features from commits on the previous day
        - If commit is dated June 30 , 2020 take features from commits on June 29, 2020
- One week history
    - Features from commits on the previous 7 days
        - If commit is dated June 30, 2020 take features from commits with dates June 23 - June 29, 2020
- Previous month history
    - Features from commits in (entire) previous month
        - If commit is dated June 30, or June 5, or June 16 2020 take features from commits in (all of) May 2020

We evaluate the effectiveness of the various risk assessment (scoring) schemes from the perspective of a code quality team wanting to efficiently identify the commits at risk. Efficiency here is simply finding the maximum number of faulty commits, with the lest effort i.e., in examining commits. The assessment is

simple where we first sort the future (test) commits in a dataset by their risk score, OR and also PR. We take the top N commits from this sorted set (N could be 10, 20, 50 or 100 etc.). We count the number of faulty comments in this set of top N. We compare this with random selection of commits with is the baseline density. For a dataset where we have a fault density of 1% we would expect to have 1 faulty commit among 100 commits selected at random. We compare this with a more informed selection of commits of top N, sorted by a commit risk score. Table 10 provides the results for the odds ratio based scheme (OR) as well as the machine learning based classification approach with different sets (five in all) of features.

Table 10: Faults in Top N (10,20,50,100) for OR and PR schemes

Note: In these results a measure like 6/142 means 142 instances were classified as faulty with confidence =1, of which 6 are true faults

Repo	**Fault basis**	**Test set #commits Fault=Y**	**Test set #commits Fault=N**	**Odds ratio based #faults** Top 10/20/50/100	**Classifier based #faults** Top 10/20/50 /100	**Classifier (with one day history) based #faults** Top 10/ 20/ 50/100	**Classifier (with one week history) based #faults** Top 10/ 20/ 50/100	**Classifier (with previous month history) based #faults** Top 10/ 20/ 50 /100	**Classifier (with activity history but no fault knowledge) based #faults** Top 10/20/50 /100	**Test set fault density**
apache	1	28	3098	0 / 1 / 2 / 3	n/a 182 have risk score=1 of which 5 are faults	n/a 152 have risk score=1 of which 13 are faults	n/a 168 have risk score=1 of which 24 are faults	n/a 145 have risk score=1 of which 18 are faults	n/a 144 have risk score=1 of which 15 are faults	1 in 100
	3	62	7754	0 / 1 / 2/ 3	n/a 104 have risk score=1 of which 5 are faults	n/a 220 have risk score=1 of which 45 are faults	n/a 207 have risk score=1 of which 38 are faults	n/a 257 have risk score=1 of which 47 are faults	n/a 246 have risk score=1 of which 43 are faults	1 in 120
	5	45	3081	0 / 0 / 0 / 1	n/a 262 have risk score=1 of which 23 are faults	n/a 251 have risk score=1 of which 33 are faults	n/a 250 have risk score=1 of which 32 are faults	n/a 243 have risk score=1 of which 29 are faults	n/a 241 have risk score=1 of which 29 are faults	1 in 75
image magick	1	10	1504	0 / 0 / 0 / 1	1 / 1 / 1 / 3	0 / 0 / 3 / 5	0 / 0 / 3 / 5	1 / 2 / 3 / 4	1 /3 / 4/ 7	1 in 150
curl	1	18	2398	0 / 1 / 3 / 6	n/a 237 have risk score=1 of which 5 are faults	n/a 145 have risk score=1 of which 7 are faults	n/a 151 have risk score=1 of which 7 are faults	n/a 152 have risk score=1 of which 6 are faults	n/a 155 have risk score=1 of which 6 are faults	1 in 120

	3	5	2411	0 / 0 / 1 / 1	n/a 122 have risk score=1 of which 1 are faults	0 / 0 / 0 / 1	1 / 1 / 1 / 2	0 / 0 / 1 / 2	0 / 0 / 1 / 2	1 in 500
	5	49	2365	0 / 1 / 5 / 5	n/a 325 have risk score=1 of which 26 are faults	n/a 242 have risk score=1 of which 35 are faults	n/a 250 have risk score=1 of which 30 are faults	n/a 243 have risk score=1 of which 36 are faults	n/a 248 have risk score=1 of which 34 are faults	1 in 50
wget	1	5	397	0 / 1 / 3 / 5	0 / 0 / 1 / 1	1 / 1 / 1 / 1	1 / 1 / 1 / 1	1 / 1 / 1 / 1	0 / 0 / 1 / 1	1 in 80
	3	8	2202	0 / 0 / 0 / 0	0 / 0 / 0 / 0	0 / 0 / 1 / 2	0 / 0 / 2 / 2	0 / 0 / 1 / 2	0 / 0 / 0 / 2	1 in 270
	5	28	976	1 / 1 / 1 / 5	1 / 2 / 3 / 6	1 / 3 / 7 / 11	1 / 2 / 6 / 6	1 / 2 / 7 / 11	3 / 3 / 9 / 10	1 in 30
openssl	1	31	2368	1 / 1 / 3 / 3	n/a 175 have risk score=1 of which 11 are faults	n/a 144 have risk score=1 of which 18 are faults	n/a 134 have risk score=1 of which 20 are faults	n/a 109 have risk score=1 of which 14 are faults	n/a 130 have risk score=1 of which 17 are faults	1 in 80
	3	6	2393	0 / 0 / 1/ 3	0 / 0 / 0 / 0	0 / 0 / 0 / 1	0 / 0 / 1 / 2	0 / 0 / 1 / 2	1 /2 / 3 / 4	1 in 400
	5	15	2384	3 / 3 / 4 / 5	n/a 186 have risk score=1 of which 3 are faults	n/a 120 have risk score=1 of which 9 are faults	n/a 121 have risk score=1 of which 4 are faults	n/a 115 have risk score=1 of which 5 are faults	n/a 120 have risk score=1 of which 10 are faults	1 in 160
nginx	1	14	1625	0 / 0 / 5 / 5	3 /4 / 5 / 6	1 / 2 / 6 / 7	1 /3 / 6 / 7	2 / 3 / 8 / 9	2 / 4 / 9 / 9	1 in 100
libraw	1	30	582	1 / 3 / 10 / 12	4 / 7 / 14 / 18	5 / 12 / 23 / 25	4 / 11 / 22 / 24	7 / 13 / 27 / 27	6 / 13 / 17 / 22	1 in 20
	5	4	608	0 / 1 / 2 / 2	0 / 0 / 3 / 3	1 / 2 / 3 / 4	1 / 2 / 3 / 3	2 / 2 / 3 / 4	1 / 3 / 3 / 3	1 in 150

In Table 11 we provide these results in terms of fault density. We also provide the factor improvement over the baseline, which is a direct measure of the efficacy of risk scoring. For instance in the first row (apache, fault basis 1) the test set fault density (baseline) is 1 in 100 which means for every 100 commits we consider we find 1 fault. With the classifier with monthly history it is 1 in 7 i.e., for every 100 commits we consider we find about 15 faults !

Table 11: Fault densities

Repo	**Fault basis**	**Odds ratio scheme fault density:** Top 20 | Top 50 |Top 100			**Classifier scheme fault density:** Based on top 100 or ALL predicted faults with probabilistic risk score=1 in case that number is > 100	**Classifier with monthly history scheme fault density:** Based on top 100 or ALL predicted faults with probabilistic risk score=1 in case that number is > 100	**Classifier with activity history not including faults scheme fault density:** Based on top 100 or ALL predicted faults with probabilistic risk score=1 in case that number is > 100	**Test set fault density**	**Fault density factor improvement:** Odds ratio scheme	**Fault density factor improvement:** Classifier scheme	**Fault density factor improvement:** Classifier with previous month history scheme	**Fault density factor improvement:** Classifier with activity history but no fault knowledge scheme
apache	1	1 in 20	1 in 25	1 in 30	1 in 50	1 in 7	1 in 10	1 in 100	100/30=3	100/50=2	100/7=15	100/10=10
	3	1 in 20	1 in 25	1 in 30	1 in 14	1 in 5	1 in 6	1 in 120	4	8	25	20
	5	0	0	1 in 100	1 in 15	1 in 8	1 in 8	1 in 75	1	5	10	10
image magick	1	0	0	1 in 100	1 in 40	1 in 25	1 in 15	1 in 150	1	2	6	10
curl	1	1 in 20	1 in 15	1 in 15	1 in 25	1 in 25	1 in 25	1 in 120	5	3	5	5

	3	0	1 in 50	1 in 100	0	1 in 50	1 in 50	1 in 500	4	0	10	10
	5	1 in 20	1 in 20	1 in 20	1 in 12	1 in 7	1 in 7	1 in 50	1	3	7	7
wget	1	1 in 20	1 in 15	1 in 20	1 in 100	1 in 100	1 in 100	1 in 80	1	1	1	1
	3	0	0	0	1 in 100	1 in 50	1 in 50	1 in 270	0	3	5	5
	5	1 in 20	1 in 50	1 in 20	1 in 20	1 in 10	1 in 10	1 in 30	1	1	3	3
openssl	1	1 in 20	1 in 15	1 in 30	1 in 16	1 in 8	1 in 7	1 in 80	3	5	10	11
	3	0	1 in 50	1 in 30	0	1 in 50	1 in 25	1 in 400	13	0	8	16
	5	1 in 7	1 in 12	1 in 20	1 in 60	1 in 11	1 in 12	1 in 160	8	3	15	13
nginx	1	0	1 in 10	1 in 20	1 in 20	1 in 10	1 in 10	1 in 100	5	5	10	10
libraw	1	1 in 2	1 in 7	1 in 5	1 in 5	1 in 4	1 in 5	1 in 20	4	4	5	5
	5	1 in 20	1 in 25	1 in 50	1 in 30	1 in 25	1 in 30	1 in 150	3	5	6	6

Conclusions summary for above results

Odds Ratio Based

Mean improvement factor = 3.8 ; Minimum = 0 ; Maximum = 13

68-95-99.1: [0.6, 7] [0,10.2] [0,13.4]

Classifier (Ensemble)

Mean improvement factor = 3.2 ; Minimum = 0, Maximum = 8

68-95-99.1: [1.25, 5.25] [0, 7.25] [0, 9.25]

Classifier with monthly history including faults(Ensemble)

Mean improvement factor = 8.9 ; Minimum = 1, Maximum = 25

68-95-99.1: [3.2, 14.4] [0, 20.0] [0, 25.4]

Classifier with activity history NOT including faults(Ensemble)

Mean improvement factor = 8.9 ; Minimum = 1, Maximum = 20

68-95-99.1: [4.2, 13.6] [0, 18.3] [0, 23.0]

4.3.9 Learning Curve Analysis

The final analysis is to determine the limits of machine learning classification in the fault risk prediction problem. We do this with *learning curve analysis*. In learning curve analysis we split any given dataset into disjoint train and test data, the basic idea is to vary (gradually increment) the size of the train and the remainder forms the test date. We keep increasing the train data size till a convergence or plateau is reached. We could choose a metric that we want to maximize, for instance the classifier accuracy in terms of F-Measure, precision etc. We then gradually increase the train size till this measure (F-Measure or Precision) *plateaus*. In our case we chosen precision as the metric. The plateau informs us of the highest precision that is theoretically possible with classification. We can also use classifier error, specifically the train error along with the validation error. We increment the train size till the train error and validation error *converge*. This error indicates the *minimum error* that will exist with any machine learning classifier for this data and classification task. Figures 1 and 2 show the learning curve analysis for some of the datasets.

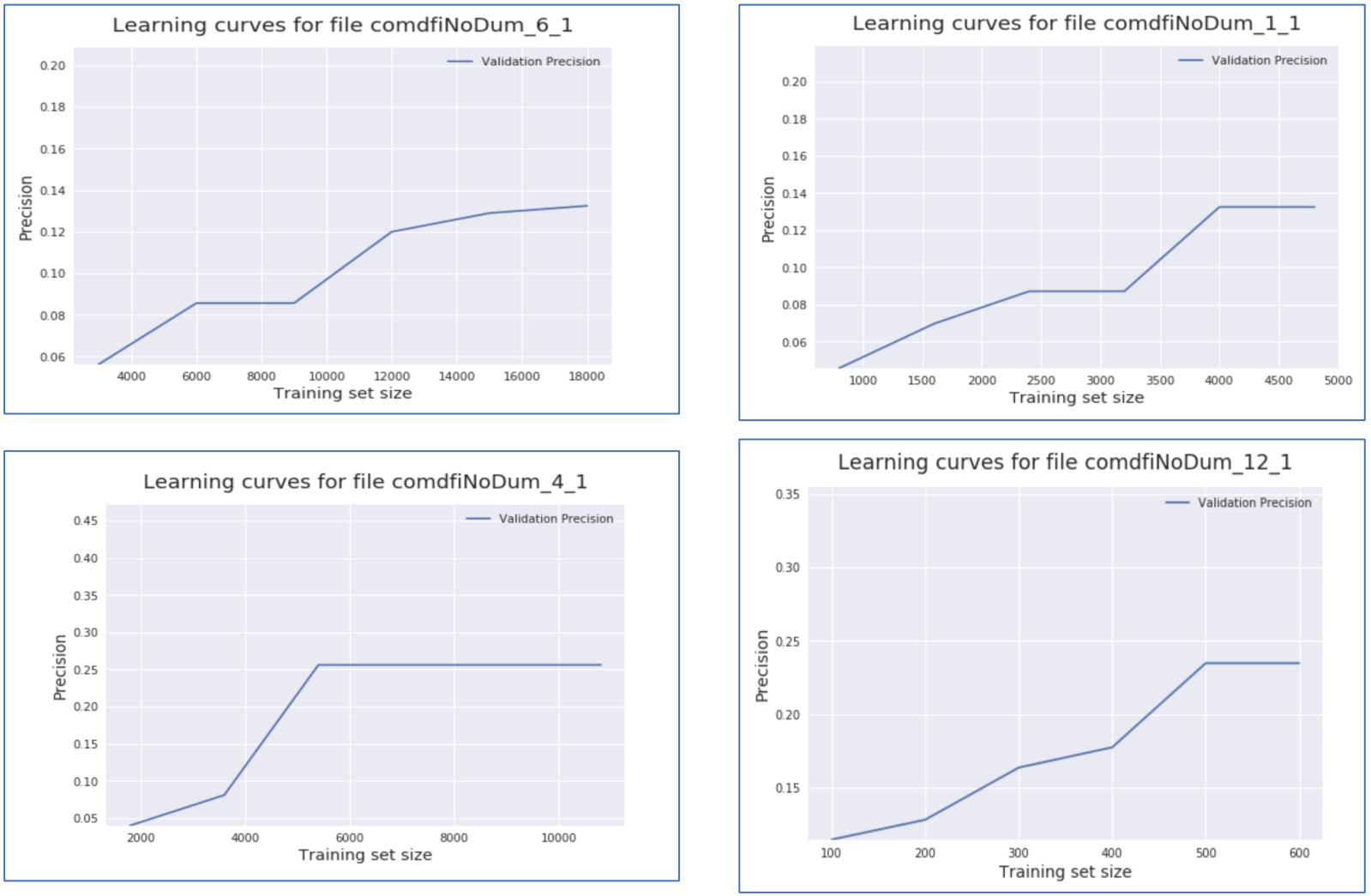


Figure 1. Learning curves using precision

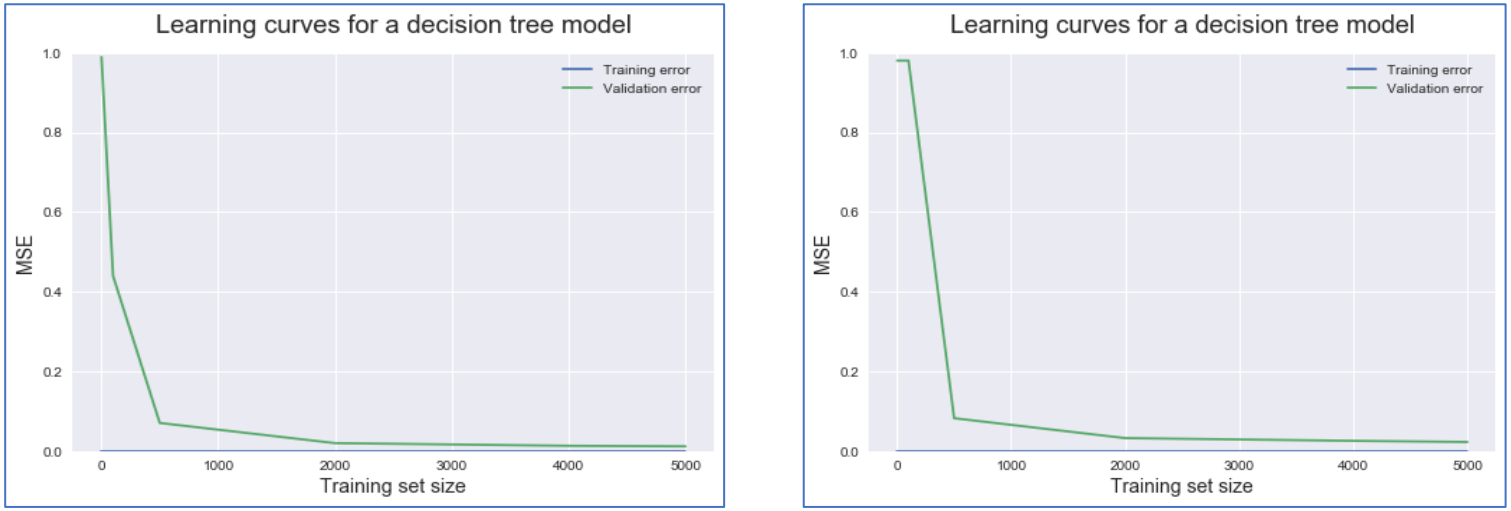


Figure 2. Learning curves using classifier error

We observe that for most data sets classifier stability is obtained when the train data size is in the 1000-4000 commits range.

4.4 Conclusions

The conclusions from this investigation, development of the fault risk assessment pipeline, and experimental evaluation all center on the key takeaway that **a machine learning classification based approach and system for predicting faulty commits is effective**. At a more specific level we learned several aspects, namely:

- From amongst a spectrum of machine learning classification approaches and frameworks evaluated, a "simple" hand-crafted feature based Decision Tree classifier has the best performance. Compared to other feature driven classifiers (such as Random Forests, SVMs etc.), enembles of such classifiers, and even deep learning frameworks (AutoML).
- Only a very small set of features seem influential for faulty commit prediction, based on our feature importance analysis of a very wide spectrum of features evaluated.
- Factoring information of very recent development activity on the repository (from the last few days, weeks and months) significantly improves fault prediction capabilities. By factoring features based on recent history we were able to achieve an average of 10-fold improvement in fault prediction (over the baseline), and it was as high as 20-fold for certain repositories evaluated.

Overall, this effort has resulted in an approach and system that is effective in fault risk prediction and has also provided us with a confident understanding of the limits (prediction accuracies possible) in such a task. This forms a foundation for a tool that we are building for software developers and software project managers that can employ data for improving the quality of their code as well as the efficiency of the software development process

5 Phase II Option Period

The goal of our Phase II option was to move beyond software engineering behavior per se, and to focus more deeply on social behavior of the open source contributors and whether this was a harbinger of risk. For example, do new participants in a project represent a risk? Do those who associate with other questionable online entities represent a greater than usual risk? Are people who file bugs interested in getting them fixed or interested in indirectly engineering a weakness in a product? In doing this work, InferLink collaborated with Margin Research for 50% of the effort.

We began our analysis by focusing on formulating new techniques for profiling potentially significant contributors and conversations on Firefox's bug tracker, Bugzilla. Operating off the tracked User Statistics in Bugzilla, we focused on exploring insights derived from analyzing these values at scale. InferLink built tools to automatically gather the records in the bug database. Performing analysis of these user statistics allowed us to infer facts regarding the nature of the users in bug tracker threads. We were specifically highlighting outliers that might indicate abnormal or harmful behavior and insights as to how certain types of community members engage with the platform.

The core of this analysis is intended to support the detection of malicious actors that may be disrupting the development processes of the open-source Firefox project. While attempting to profile bug fix threads that devolved into "flame wars," we discovered a false positive where we were flagging threads with many contributions from brand new accounts. After investigating, we determined that these threads were indeed flame wars but typically oriented around UI/UX changes. In one instance, a bug was submitted after Firefox had changed the behavior of the URL bar for installations on Linux. This created a massive thread of brand new accounts, in some cases explicitly stating they created the account to complain in the thread, berating the developers and derailing any workflow that may have otherwise occurred in that bug report. We determined that this highlighted a threat model we had not previously considered; malicious actors may coordinate on a different platform, in this case, Reddit, and direct a community to commit disruptive actions. Our algorithm was able to find these types of accounts very effectively. We also determined that a leading

indicator of these disruptions is the sudden presence of a social media post that links back to that thread. If we search for these events in any open source project, we can anticipate disruptions from people external to the development community.

A specific name kept coming up while searching for cross-references between social media and the Firefox Bugzilla community. Upon further analysis, we found that this user was not only one of the primary users submitting bugs but also a moderator of the firefox subreddit. This person's job, official or not, was to take bugs reported via the subreddit and share them in the Bugzilla tracker so that they could be fixed. Finding people like this, who have critical roles in the community but are not explicitly defined as such, and are therefore in a precarious position, was a significant objective of this project. There are a couple of reasons why we wanted to discover these types of users:

1. If they were to stop doing their job suddenly one day, the project would immediately lose a critical vulnerability reporting pipeline.

2. We wanted to examine the degree to which the criticality of their role was known to the community. From our exploration, we determined that while these roles are likely known to people active within the community, they are not explicitly defined, making them prone to abuse. Very little work is done that supports the transfer of skills between people in these unofficial albeit critical roles and people who might support or replace them in time.

3. The risks of account hijacking are magnified.

If someone like Linus Torvalds, a well-known open source contributor, had their email taken over, it would likely be apparent quickly. They are a significant player in the Kernel, and they likely have several other mechanisms by which they communicate and engage with other authorities on the project. In other words, because he holds both a primary and explicitly defined role in the community, there are mechanisms to provide some degree of support if things go wrong. If someone like the aforementioned Redditor were to experience an account takeover, they might not have those same pipelines in place. Thus, the risks and consequences are magnified.

This revelation led us to explore the possibility of leveraging grey-market collections of compromised accounts to gauge risks experienced by certain members of the Firefox community. Using tools like **HaveIBeenPwned**, we can immediately determine the number of account breaches within which a given email has been exposed. Depending on the nature of that person's role in the community, there may be a higher or lower risk. An exciting result of this effort was the realization that accounts being compromised is, in fact, a reliable indicator of the authenticity of a given email. More specifically, if an email is purpose-built to be used on a single platform, its exposure is reduced to such a degree that it likely does not exist in breach databases. This allows us to extrapolate facts regarding the nature of specific accounts. For example, if a bad actor decided to harass people on the platform or created an account to antagonize developers, we likely would not see that account in many breach databases because it was purpose-built. Unfortunately, this type of analysis was limited by the nature of the Bugzilla platform. It deliberately does not expose emails to users on the platform.

Margin Research has access to a database of collections from known hacker forums that we decided to use instead. While some users do include a screen name or handle in their account name, it was often not complex enough to reliably flag any other observed account names in hacker forums. The instances in which we did flag names were often simple names based on 2-4 letter words that were ultimately benign or false positives. This type of analysis may bear fruit with a larger dataset. However, it will inevitably deal with collisions on simple names, which could be exploited by adversaries looking to disrupt communities under the moniker "John Smith."

Following this effort, we decided to focus our attention on extending analysis capabilities when starting with known bad actors. Focusing on what at the time was a current event rapidly unfolding, we examined

the process by which the log4j exploit was disclosed. We felt it provided an excellent template for analyzing disruptions to open source development cycles on GitHub. We attempted to uncover insights into how adversaries disrupt the patch cycle of a critical vulnerability by analyzing this event. In this particular instance, a critical vuln was disclosed to Apache, which had patched it and was adding the fix to their next release. At which point, several Chinese researchers jumped into the PR complaining about it not receiving enough attention (despite being fixed). Shortly after that, PoC code was shared on Twitter, enabling widespread exploitation of the vulnerability.

We performed a deep dive into the individuals involved in this particular pull request. We examined the timeline of events that occurred based on direct accounts from the team patching the bug on Apache's side and the Alibaba security researchers that had disclosed the vulnerability. Because the weaponization of the vulnerability occurred in a region of the Chinese internet that we have little access to, we explored the possibility of generating communities based on seed profiles like the individuals targeting the developers at Apache. The generated lists based on "followers of followers" effectively outlined the projects and communities that the suspect users were a part of. In the interest of extending that analysis, we comprised a list of researchers at Alibaba who allegedly discovered and disclosed the log4j vulnerability, offensive researchers from Kunlun Labs and Quihoo360, and Western research groups like Project Zero. After analyzing each group, we examined each group's social sphere and any intersections that exist—doing so effectively allowed us an eagle's eye view of the global malware research and development community. Further work might involve applying conventional trending algorithms to extract insights into ongoing research focuses of this community and possibly preempt threats.

To support the types of analysis done above by Margin Research, InferLink built tools for social media monitoring and social network extraction. For example, we built a social media "news gathering" solution that automatically identified unique interests of any subgroup being monitored; the idea is that if one wants to analyze the unique interests of a group (e.g., foreign hackers, vulnerability hunters, etc.), the system we developed can aid with automating that analysis. Twitter was the example domain to which we applied this tool.

The system works as follows. In general, to identify "unique interests" of a group, it compares the aggregation of tweets by that group with a random sample of conversation at the same source. In our main working example, we were monitoring 200 well known security researchers, comparing the aggregation of their tweets with a sample of random tweets. By doing this, we can automatically eliminate common topics (e.g., items in the news that are of general interest to anyone) and focus on those unique to the security group. For example, "cve" is unlikely to be a common term for the population at large, but it would be expected to be common for the security researcher tweet corpus. Our solution ran over any time period (e.g., once per week or once per day). It compares data from the last time period (e.g., the last day or past week) with the time period of same length that precedes it (e.g., the day before last, the week before last). Its analysis is focused on identifying language trends including terms, phrases, hashtags. For each period, the system generates:

- Most popular terms/phrases/hashtags
- New terms/phrases/hashtags
- Trending terms/phrases/hashtags

Sample results from a recent run are shown below. Specifically, we can see the interest in log4j and related topics (e.g., Minecraft and iCloud, which were affected). Our software surfaced these interests automatically. We are exploring deployment of this system to other groups (e.g., industrial control system security experts, etc.) and other sources (e.g., GitHub).

```
//This shows the most common phrases from 2021-12-16 that were not present in the results from 2021-12-09
46      log4j vulnerability
38      apache log4j
26      log4j cve202144228
18      cve202144228 vulnerability
17      log4j cve202144228 vulnerability
15      extremely bad
15      log4j logging
14      millions applications log4j
14      applications log4j logging
14      vulnerability extremely bad
14      extremely bad millions
14      logging attacker app
14      cve202144228 vulnerability extremely
14      log4j logging attacker
14      bad millions applications
14      attacker app log
14      special string icloud
14      app log special
14      string icloud steam
14      icloud steam minecraft
14      steam minecraft confirmed
14      log special string
13      vulnerable log4j
13      imessage exploit
13      zeroclick imessage
12      version log4j
12      zeroclick imessage exploit
```

A second software effort has been to develop a strategy for identifying "social peers" using a straightforward network analysis approach. In particular, we are interested in the case where some person P1 has a set of followers F1. We then look at the set of people besides P1 that each member of F1 follows. We then compile the most popular figures and add them to our "peers" list and continue the process (as applicable). For example, if Mark Hamill (star of Star Wars) was the subject of focus, then we would expect Harrison Ford and Carrie Fisher (co-stars) to also be identified as popular peers, because it is very likely that Mark Hamill's followers would have such an interest. We would like to extend this network analysis to social media identities of interest (e.g., hacker groups, etc).

More generally, we developed a "followings of followers" solution that allowed us to identify a community based on a seed of subjects. For example, given identities `X, Y`, and `Z` on a social network, identify the sorted, ranked set of X_{ff} `U` Y_{ff} `U` Z_{ff} where X_{ff} is the set of followings for the followers of X. Thus, we wish to identify popular peers of X, Y, and Z, as indirectly communicated by their followers. The hope is that if we have hacker X, Y, and Z, we use the follower-of-followings approach to identify J, K, L, and M peers – also hackers, but previously not known to us -- yet deemed to be part of the same community because of their popularity among followers of X, Y, and Z. There are numerous similar community-finding algorithms such as these that exist, and many have been documented in the literature.

In starting with something relatively simple, we analyzed the data found from a set of log4j related personalities on GitHub. Our results suggest that this type of examination can act as an analog for centrality analysis that a graph database would typically perform without requiring prior knowledge of the cluster of users in question. After examining a group that was intended to represent "cyber security focused" individuals, we confirmed the findings based on our knowledge of the industry. As an opportunity to extend this analysis, we offered a number of suspect users that had demonstrated knowledge of the log4j vulnerability before most of the western world learned via Twitter. This is because the exploit was weaponized in Chinese social media, a famously hard target for OSINT analysis. By performing this "followings of followers" analysis, we hope to uncover the social sphere of individuals associated with the weaponization of this vulnerability without requiring access to the social media accounts in question. The log4j use case helps validate the utility of our community-finding system.

Through this option period, we were able to make significant progress on multiple objectives and were able to conclude:

- There is often good reason to use social media behavior as an indicator of risk/threat for a given open source software repository
- It is possible to build tools to semi-automatically monitor activities such that one can potentially be warned of possible risk to such repos
- There seems to be promise in mining social network graphs to identify potential risky actors

These and other indicators suggest future work in this area would be useful. A key challenge is, of course, finding enough real examples to build tools that automatically identify these risks. In the case of our Phase II work before the option, we settled on an approach that allowed us to do that – i.e., we mined commit log messages. We could have mined JIRA database as well. In the Option period, there is a tougher challenge of what to monitor, but perhaps the solution is to simply build tools that enable quicker and richer human analysis without passing judgment.

6 Appendix

6.1 Full results of all repositories tested

	RF	DT	ABV	GNB	SVM	QDA	KNN	TCM	TCF	TCC
Repo: nginx										
Fault 1;Y:N=129:6426; BL=0.02|	0.11,0.79(0.19)	0.59,0.74(0.66)	0.09,0.71(0.16)	0.20,0.36(0.26)	0.00,0.0(NA)	0.19,0.37(0.25)	0.10,0.2(0.13)	0.12,0.4(0.18)	0.05,0.84(0.09)	0.08,0.14(0.1)
Fault 5;Y:N=471:6084; BL=0.07|	0.41,0.94(0.57)	0.75,0.84(0.79)	0.37,0.86(0.52)	0.60,0.32(0.42)	0.07,0.03(0.04)	0.53,0.4(0.46)	0.42,0.55(0.48)	0.63,0.82(0.71)	0.40,0.78(0.53)	0.25,0.72(0.37)
Repo: apache										
Fault 1;Y:N=190:31074; BL=0.01|	0.05,0.93(0.09)	0.30,0.48(0.37)	0.02,0.57(0.04)	0.06,0.15(0.09)	0.00,0.01(0.0)	0.05,0.12(0.07)	0.01,0.06(0.02)	0.03,0.29(0.05)	0.02,0.82(0.04)	0.02,0.53(0.04)
Fault 3;Y:N=127:31137; BL=0.0|	0.03,0.74(0.06)	0.34,0.46(0.39)	0.01,0.56(0.02)	0.03,0.07(0.04)	0.00,0.0(NA)	0.03,0.09(0.04)	0.00,0.0(NA)	0.07,0.19(0.1)	0.02,0.74(0.04)	0.02,0.05(0.03)
Fault 5;Y:N=559:30705; BL=0.02|	0.12,0.91(0.21)	0.33,0.47(0.39)	0.04,0.74(0.08)	0.08,0.07(0.07)	0.01,0.03(0.01)	0.07,0.08(0.07)	0.02,0.09(0.03)	0.06,0.47(0.11)	0.04,0.92(0.08)	0.07,0.56(0.12)
Repo: image magick										
Fault 1;Y:N=158:14972; BL=0.01|	0.07,0.77(0.13)	0.22,0.42(0.29)	0.03,0.56(0.06)	0.00,0.0(NA)	0.00,0.03(0.0)	0.09,0.01(0.02)	0.01,0.07(0.02)	0.01,0.11(0.02)	0.03,0.53(0.06)	0.02,0.05(0.03)
Fault 5;Y:N=83:15047; BL=0.01|	0.09,0.65(0.16)	0.16,0.26(0.2)	0.01,0.24(0.02)	0.00,0.0(NA)	0.00,0.02(0.0)	0.00,0.0(NA)	0.00,0.02(0.0)	0.01,0.04(0.02)	0.02,0.5(0.04)	0.00,0.0(NA)
Fault 7;Y:N=88:15042; BL=0.01|	0.13,0.76(0.22)	0.18,0.26(0.21)	0.01,0.57(0.02)	0.00,0.0(NA)	0.01,0.11(0.02)	0.00,0.0(NA)	0.01,0.04(0.02)	0.00,0.37(0.0)	0.03,0.52(0.06)	0.02,0.04(0.03)
Repo: curl										
Fault 1;Y:N=542:23623; BL=0.02|	0.10,0.77(0.18)	0.34,0.41(0.37)	0.06,0.62(0.11)	0.15,0.13(0.14)	0.02,0.04(0.03)	0.17,0.12(0.14)	0.03,0.14(0.05)	0.06,0.45(0.11)	0.06,0.82(0.11)	0.07,0.38(0.12)
Fault 3;Y:N=76:24089; BL=0.0|	0.02,0.6(0.04)	0.23,0.23(0.23)	0.01,0.42(0.02)	0.04,0.19(0.07)	0.00,0.0(NA)	0.04,0.15(0.06)	0.00,0.0(NA)	0.00,0.0(NA)	0.02,0.69(0.04)	0.01,0.02(0.01)
Fault 5;Y:N=934:23231; BL=0.04|	0.17,0.86(0.28)	0.39,0.46(0.42)	0.10,0.62(0.17)	0.25,0.14(0.18)	0.04,0.09(0.06)	0.26,0.12(0.16)	0.07,0.24(0.11)	0.09,0.57(0.16)	0.10,0.87(0.18)	0.12,0.4(0.18)
Repo: wget										
Fault 1;Y:N=64:3955; BL=0.02|	0.22,0.78(0.34)	0.50,0.68(0.58)	0.04,0.54(0.07)	0.00,0.0(NA)	0.01,0.03(0.01)	0.00,0.0(NA)	0.03,0.08(0.04)	0.12,0.11(0.11)	0.03,0.43(0.06)	0.29,0.05(0.09)

Fault 3;Y:N=20:3999; BL=0.0| 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.01,0.11(0.02) 0.00,0.0(NA)

Fault 5;Y:N=130:3889; BL=0.03| 0.13,0.7(0.22) 0.43,0.43(0.43) 0.09,0.61(0.16) 0.00,0.0(NA) 0.05,0.02(0.03) 0.00,0.0(NA) 0.08,0.12(0.1) 0.14,0.2(0.16) 0.10,0.82(0.18) 0.06,0.06(0.06)

Repo: videolan/vlc

Fault 1;Y:N=2527:78906; BL=0.03| 0.09,0.75(0.16) 0.23,0.4(0.29) 0.08,0.69(0.14) 0.03,0.99(0.06) 0.03,0.09(0.04) 0.03,0.99(0.06) 0.04,0.2(0.07) 0.07,0.59(0.13) 0.05,0.82(0.09) 0.06,0.49(0.11)

Fault 3;Y:N=1126:80307; BL=0.01| 0.07,0.85(0.13) 0.18,0.39(0.25) 0.03,0.75(0.06) 0.01,0.98(0.02) 0.01,0.04(0.02) 0.01,0.99(0.02) 0.01,0.11(0.02) 0.04,0.52(0.07) 0.03,0.7(0.06) 0.03,0.52(0.06)

Fault 5;Y:N=2653:78780; BL=0.03| 0.10,0.8(0.18) 0.31,0.46(0.37) 0.08,0.77(0.14) 0.03,0.98(0.06) 0.03,0.09(0.04) 0.03,0.98(0.06) 0.05,0.23(0.08) 0.10,0.66(0.17) 0.09,0.71(0.16) 0.07,0.56(0.12)

Repo: nghttp2

Fault 1;Y:N=16:6267; BL=0.0| 0.04,1.0(0.08) 0.80,0.5(0.62) 0.00,0.0(NA) 0.07,0.38(0.12) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.01,0.75(0.02) 0.00,0.0(NA)

Fault 5;Y:N=92:6191; BL=0.01| 0.15,0.96(0.26) 0.31,0.38(0.34) 0.04,0.46(0.07) 0.11,0.11(0.11) 0.00,0.0(NA) 0.11,0.12(0.11) 0.04,0.09(0.06) 0.06,0.09(0.07) 0.06,0.95(0.11) 0.07,0.23(0.11)

Repo: mekentosi/podofo

Fault 1;Y:N=92:1171; BL=0.07| 0.38,0.79(0.51) 0.58,0.52(0.55) 0.20,0.49(0.28) 0.08,0.95(0.15) 0.17,0.02(0.04) 0.08,0.93(0.15) 0.23,0.25(0.24) 0.41,0.15(0.22) 0.23,0.69(0.35) 0.18,0.2(0.19)

Fault 3;Y:N=92:1171; BL=0.07| 0.35,0.8(0.49) 0.50,0.47(0.48) 0.17,0.53(0.26) 0.07,0.91(0.13) 0.00,0.0(NA) 0.07,0.89(0.13) 0.16,0.27(0.2) 0.14,0.09(0.11) 0.16,0.67(0.26) 0.26,0.35(0.3)

Fault 5;Y:N=97:1166; BL=0.08| 0.37,0.88(0.52) 0.50,0.64(0.56) 0.15,0.33(0.21) 0.08,0.92(0.15) 0.06,0.02(0.03) 0.08,0.94(0.15) 0.14,0.17(0.15) 0.26,0.09(0.13) 0.23,0.56(0.33) 0.36,0.08(0.13)

Repo: openssl

Fault 1;Y:N=336:23655; BL=0.01| 0.06,0.81(0.11) 0.26,0.45(0.33) 0.04,0.7(0.08) 0.11,0.12(0.11) 0.01,0.03(0.01) 0.10,0.1(0.1) 0.02,0.1(0.03) 0.04,0.32(0.07) 0.04,0.78(0.08) 0.04,0.44(0.07)

Fault 3;Y:N=75:23916; BL=0.0| 0.01,0.56(0.02) 0.14,0.41(0.21) 0.01,0.32(0.02) 0.04,0.22(0.07) 0.00,0.0(NA) 0.04,0.2(0.07) 0.01,0.1(0.02) 0.00,0.02(0.0) 0.01,0.8(0.02) 0.01,0.32(0.02)

Fault 5;Y:N=428:23563; BL=0.02| 0.08,0.83(0.15) 0.28,0.49(0.36) 0.05,0.7(0.09) 0.12,0.12(0.12) 0.01,0.05(0.02) 0.12,0.15(0.13) 0.05,0.16(0.08) 0.05,0.31(0.09) 0.05,0.76(0.09) 0.05,0.44(0.09)

Repo: libuv

Fault 1;Y:N=101:4276; BL=0.02| 0.13,0.43(0.2) 0.12,0.22(0.16) 0.04,0.39(0.07) 0.02,0.91(0.04) 0.02,0.07(0.03) 0.02,0.91(0.04) 0.02,0.17(0.04) 0.10,0.24(0.14) 0.05,0.61(0.09) 0.09,0.04(0.06)

Fault 3;Y:N=18:4359; BL=0.0| 0.03,0.67(0.06) 0.14,0.44(0.21) 0.02,0.11(0.03) 0.00,1.0(0.0) 0.00,0.0(NA) 0.00,0.89(0.0) 0.00,0.0(NA) 0.00,0.0(NA) 0.04,0.67(0.08) 0.00,0.0(NA)

Fault 5;Y:N=157:4220; BL=0.04| 0.19,0.73(0.3) 0.32,0.39(0.35) 0.10,0.49(0.17) 0.03,0.98(0.06) 0.02,0.03(0.02) 0.03,0.97(0.06) 0.09,0.18(0.12) 0.09,0.2(0.12) 0.08,0.73(0.14) 0.08,0.04(0.05)

Repo: EQEmu

Fault 1;Y:N=32:7299; BL=0.0| 0.02,0.53(0.04) 0.15,0.35(0.21) 0.02,0.18(0.04) 0.04,0.06(0.05) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.01,0.41(0.02) 0.04,0.12(0.06)

Fault 3;Y:N=119:7212; BL=0.02| 0.07,0.67(0.13) 0.22,0.3(0.25) 0.04,0.59(0.07) 0.08,0.01(0.02) 0.00,0.0(NA) 0.08,0.01(0.02) 0.05,0.09(0.06) 0.04,0.04(0.04) 0.04,0.6(0.07) 0.02,0.07(0.03)

Fault 5;Y:N=202:7129; BL=0.03| 0.11,0.81(0.19) 0.29,0.42(0.34) 0.06,0.62(0.11) 0.14,0.02(0.04) 0.01,0.02(0.01) 0.10,0.03(0.05) 0.04,0.09(0.06) 0.09,0.17(0.12) 0.07,0.56(0.12) 0.00,0.0(NA)

Repo: mangosthree

Fault 1;Y:N=87:8151; BL=0.01| 0.06,0.81(0.11) 0.22,0.37(0.28) 0.03,0.46(0.06) 0.01,0.9(0.02) 0.00,0.0(NA) 0.01,0.9(0.02) 0.03,0.08(0.04) 0.13,0.17(0.15) 0.03,0.77(0.06) 0.08,0.02(0.03)

Fault 3;Y:N=382:7856; BL=0.05| 0.20,0.77(0.32) 0.27,0.28(0.27) 0.12,0.7(0.2) 0.05,0.95(0.1) 0.05,0.06(0.05) 0.05,0.95(0.1) 0.09,0.2(0.12) 0.12,0.44(0.19) 0.12,0.81(0.21) 0.08,0.02(0.03)

Fault 5;Y:N=334:7904; BL=0.04| 0.15,0.79(0.25) 0.26,0.34(0.29) 0.10,0.67(0.17) 0.11,0.01(0.02) 0.01,0.01(0.01) 0.04,0.96(0.08) 0.05,0.17(0.08) 0.09,0.29(0.14) 0.10,0.78(0.18) 0.11,0.04(0.06)

Repo: libexif

Fault 1;Y:N=34:1109; BL=0.03| 0.14,0.79(0.24) 0.53,0.47(0.5) 0.02,0.11(0.03) 0.02,0.79(0.04) 0.04,0.05(0.04) 0.02,0.79(0.04) 0.05,0.16(0.08) 0.13,0.16(0.14) 0.13,1.0(0.23) 0.05,0.11(0.07)

Fault 3;Y:N=11:1132; BL=0.01| 0.02,0.6(0.04) 0.75,0.6(0.67) 0.00,0.0(NA) 0.01,1.0(0.02) 0.00,0.0(NA) 0.01,1.0(0.02) 0.00,0.0(NA) 0.05,0.2(0.08) 0.03,0.8(0.06) 0.00,0.0(NA)

Fault 5;Y:N=44:1099; BL=0.04| 0.24,0.73(0.36) 0.48,0.5(0.49) 0.09,0.27(0.14) 0.04,0.92(0.08) 0.00,0.0(NA) 0.04,0.92(0.08) 0.04,0.08(0.05) 0.00,0.0(NA) 0.07,0.77(0.13) 0.19,0.27(0.22)

Repo: libpcap

Fault 1;Y:N=18:4416; BL=0.0| 0.05,0.5(0.09) 0.13,0.25(0.17) 0.00,0.0(NA) 0.00,0.75(0.0) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.01,0.38(0.02) 0.00,0.0(NA)

Fault 5;Y:N=58:4376; BL=0.01| 0.04,0.83(0.08) 0.49,0.69(0.57) 0.04,0.4(0.07) 0.15,0.23(0.18) 0.00,0.0(NA) 0.17,0.11(0.13) 0.06,0.11(0.08) 0.18,0.09(0.12) 0.04,0.74(0.08) 0.08,0.03(0.04)

Repo: trafficserver

Fault 1;Y:N=14:12097; BL=0.0| 0.00,0.0(NA) 0.50,0.18(0.26) 0.00,0.0(NA) 0.04,0.18(0.07) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.07,0.45(0.12) 0.00,0.0(NA)

Fault 5;Y:N=15:12096; BL=0.0| 0.25,0.18(0.21) 0.14,0.09(0.11) 0.00,0.0(NA) 0.05,0.18(0.08) 0.00,0.0(NA) 0.05,0.18(0.08) 0.00,0.0(NA) 0.00,0.0(NA) 0.05,0.55(0.09) 0.00,0.91(0.0)

Repo: zaphoyd

Fault 5;Y:N=13:1685; BL=0.01| 0.15,0.25(0.19) 0.50,0.75(0.6) 0.25,0.12(0.16) 0.11,0.25(0.15) 0.00,0.0(NA) 0.00,0.0(NA) 0.13,0.25(0.17) 0.00,0.0(NA) 0.01,0.12(0.02) 0.00,0.0(NA)

Repo: libarchive

Fault 1;Y:N=91:5501; BL=0.02| 0.15,0.85(0.26) 0.29,0.49(0.36) 0.05,0.53(0.09) 0.16,0.15(0.15) 0.00,0.0(NA) 0.16,0.15(0.15) 0.04,0.21(0.07) 0.06,0.13(0.08) 0.03,0.79(0.06) 0.06,0.02(0.03)

Fault 3;Y:N=13:5579; BL=0.0| 0.01,0.5(0.02) 0.29,0.5(0.37) 0.00,0.0(NA) 0.04,0.25(0.07) 0.00,0.0(NA) 0.04,0.25(0.07) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA) 0.00,0.0(NA)

Fault 5;Y:N=129:5463; BL=0.02| 0.11,0.67(0.19) 0.30,0.36(0.33) 0.06,0.47(0.11) 0.14,0.07(0.09) 0.00,0.0(NA) 0.15,0.07(0.1) 0.07,0.13(0.09) 0.18,0.15(0.16) 0.04,0.59(0.07) 0.00,0.0(NA)

Repo: libav

Fault 1;Y:N=375:43573; BL=0.01| 0.05,0.82(0.09) 0.23,0.41(0.29) 0.03,0.7(0.06) 0.12,0.18(0.14) 0.01,0.04(0.02) 0.09,0.19(0.12) 0.02,0.14(0.04) 0.03,0.29(0.05) 0.02,0.64(0.04) 0.02,0.03(0.02)

Fault 3;Y:N=223:43973; BL=0.01| 0.03,0.57(0.06) 0.25,0.34(0.29) 0.02,0.6(0.04) 0.09,0.22(0.13) 0.00,0.01(0.0) 0.13,0.2(0.16) 0.02,0.1(0.03) 0.02,0.19(0.04) 0.02,0.56(0.04) 0.03,0.01(0.01)

Fault 5;Y:N=759:43148; BL=0.02| 0.08,0.7(0.14) 0.27,0.38(0.32) 0.05,0.79(0.09) 0.29,0.18(0.22) 0.02,0.04(0.03) 0.29,0.15(0.2) 0.04,0.16(0.06) 0.05,0.55(0.09) 0.05,0.66(0.09) 0.04,0.02(0.03)

Repo: sqlite

Fault 1;Y:N=624:20249; BL=0.03| 0.13,0.78(0.22) 0.31,0.32(0.31) 0.07,0.7(0.13) 0.10,0.04(0.06) 0.02,0.04(0.03) 0.07,0.02(0.03) 0.05,0.17(0.08) 0.08,0.65(0.14) 0.05,0.76(0.09) 0.06,0.06(0.06)

Fault 3;Y:N=237:20636; BL=0.01| 0.03,0.64(0.06) 0.31,0.49(0.38) 0.02,0.7(0.04) 0.03,0.04(0.03) 0.01,0.01(0.01) 0.07,0.03(0.04) 0.02,0.06(0.03) 0.05,0.29(0.09) 0.03,0.78(0.06) 0.03,0.03(0.03)

Fault 5;Y:N=1061:19812; BL=0.05| 0.16,0.78(0.27) 0.34,0.46(0.39) 0.11,0.68(0.19) 0.19,0.06(0.09) 0.03,0.04(0.03) 0.05,0.96(0.1) 0.09,0.22(0.13) 0.11,0.61(0.19) 0.08,0.75(0.14) 0.08,0.12(0.1)

Repo: libpnq

Fault 1;Y:N=38:4031; BL=0.01| 0.07,0.45(0.12) 0.24,0.41(0.3) 0.03,0.14(0.05) 0.02,0.05(0.03) 0.00,0.0(NA) 0.00,0.0(NA) 0.03,0.05(0.04) 0.00,0.0(NA) 0.06,0.55(0.11) 0.02,0.14(0.04)

Fault 5;Y:N=134:3935; BL=0.03| 0.22,0.74(0.34) 0.29,0.48(0.36) 0.09,0.33(0.14) 0.14,0.14(0.14) 0.02,0.01(0.01) 0.20,0.08(0.11) 0.07,0.14(0.09) 0.09,0.04(0.06) 0.11,0.55(0.18) 0.13,0.19(0.15)

Repo: libming

Fault 5;Y:N=93:24; BL=0.79| 1.00,0.86(0.92) 1.00,0.95(0.97) 0.82,0.55(0.66) 0.82,0.86(0.84) 0.86,0.1(0.18) 0.81,0.83(0.82) 0.86,0.72(0.78) 0.91,0.86(0.88) 1.00,0.88(0.94) 0.81,0.98(0.89)

Repo: libraw

Fault 1;Y:N=44:2399; BL=0.02| 0.24,0.83(0.37) 0.47,0.67(0.55) 0.04,0.29(0.07) 0.01,0.88(0.02) 0.00,0.0(NA) 0.01,0.83(0.02) 0.08,0.17(0.11) 0.17,0.21(0.19) 0.02,0.46(0.04) 0.08,0.04(0.05)

Fault 5;Y:N=39:2404; BL=0.02| 0.14,0.59(0.23) 0.38,0.22(0.28) 0.20,0.3(0.24) 0.05,0.04(0.04) 0.25,0.07(0.11) 0.06,0.04(0.05) 0.09,0.07(0.08) 0.23,0.11(0.15) 0.04,0.81(0.08) 0.03,0.07(0.04)

Repo: libtiff

Fault 1;Y:N=27:2979; BL=0.01| 0.07,0.79(0.13) 0.33,0.21(0.26) 0.00,0.0(NA) 0.01,0.86(0.02) 0.00,0.0(NA) 0.01,0.64(0.02) 0.00,0.0(NA) 0.06,0.07(0.06) 0.01,0.14(0.02) 0.00,0.0(NA)

Fault 3;Y:N=14:2992; BL=0.0| 0.02,0.12(0.03) 0.29,0.25(0.27) 0.08,0.12(0.1) 0.00,0.88(0.0) 0.00,0.0(NA) 0.00,0.88(0.0) 0.07,0.12(0.09) 0.00,0.0(NA) 0.01,0.38(0.02) 0.00,0.0(NA)

Fault 5;Y:N=245:2761; BL=0.08| 0.31,0.74(0.44) 0.45,0.59(0.51) 0.14,0.61(0.23) 0.07,0.93(0.13) 0.06,0.04(0.05) 0.07,0.91(0.13) 0.11,0.32(0.16) 0.18,0.36(0.24) 0.16,0.89(0.27) 0.15,0.06(0.09)

Repo: libzmq

Fault 1;Y:N=226:7285; BL=0.03| 0.18,0.83(0.3) 0.49,0.49(0.49) 0.05,0.5(0.09) 0.07,0.02(0.03) 0.03,0.06(0.04) 0.10,0.1(0.1) 0.04,0.14(0.06) 0.19,0.44(0.27) 0.10,0.87(0.18) 0.10,0.18(0.13)

Fault 5;Y:N=108:7403; BL=0.01| 0.06,0.75(0.11) 0.23,0.32(0.27) 0.02,0.25(0.04) 0.01,0.85(0.02) 0.02,0.02(0.02) 0.01,0.85(0.02) 0.03,0.07(0.04) 0.09,0.35(0.14) 0.04,0.77(0.08) 0.01,0.07(0.02)

Repo: tesserat

Fault 5;Y:N=140:1500; BL=0.09| 0.43,0.87(0.58) 0.60,0.76(0.67) 0.24,0.52(0.33) 0.65,0.12(0.2) 0.05,0.01(0.02) 0.46,0.07(0.12) 0.22,0.28(0.25) 0.27,0.46(0.34) 0.29,0.71(0.41) 0.00,0.0(NA)

Repo: protobuf

Fault 5;Y:N=127:1500; BL=0.08| 0.59,0.89(0.71) 0.73,0.81(0.77) 0.18,0.57(0.27) 0.58,0.17(0.26) 0.00,0.0(NA) 0.62,0.16(0.25) 0.19,0.31(0.24) 0.36,0.35(0.35) 0.30,0.93(0.45) 0.70,0.09(0.16)

Repo: october

Fault 5;Y:N=42:1500; BL=0.03| 0.25,0.7(0.37) 0.58,0.61(0.59) 0.16,0.43(0.23) 0.02,0.83(0.04) 0.00,0.0(NA) 0.02,0.78(0.04) 0.04,0.04(0.04) 0.18,0.13(0.15) 0.09,0.57(0.16) 0.02,1.0(0.04)

Repo: WordPress

Fault 5;Y:N=113:1500; BL=0.07| 0.49,0.88(0.63) 0.76,0.78(0.77) 0.18,0.43(0.25) 0.52,0.23(0.32) 0.04,0.01(0.02) 0.52,0.23(0.32) 0.11,0.36(0.17) 0.64,0.31(0.42) 0.35,0.91(0.51) 0.08,1.0(0.15)

Repo: phpredis

Fault 5;Y:N=80:1500; BL=0.05| 0.22,0.76(0.34) 0.33,0.29(0.31) 0.13,0.37(0.19) 0.16,0.2(0.18) 0.00,0.0(NA) 0.16,0.1(0.12) 0.11,0.18(0.14) 0.26,0.12(0.16) 0.11,0.84(0.19) 0.05,0.92(0.09)

Repo: symfony

Fault 5;Y:N=31:1500; BL=0.02| 0.14,0.55(0.22) 0.67,0.5(0.57) 0.05,0.2(0.08) 0.02,0.95(0.04) 0.00,0.0(NA) 0.02,0.95(0.04) 0.01,0.05(0.02) 0.50,0.1(0.17) 0.30,0.7(0.42) 0.02,1.0(0.04)

Repo: woocommerce

Fault 5;Y:N=40:1500; BL=0.03| 0.29,0.71(0.41) 0.73,0.67(0.7) 0.04,0.12(0.06) 0.03,0.96(0.06) 0.09,0.08(0.08) 0.02,0.92(0.04) 0.04,0.12(0.06) 0.20,0.04(0.07) 0.23,0.79(0.36) 0.00,0.0(NA)

Repo: yii2

Fault 5;Y:N=48:1500; BL=0.03| 0.42,0.73(0.53) 0.69,0.67(0.68) 0.07,0.3(0.11) 0.03,0.83(0.06) 0.00,0.0(NA) 0.03,0.83(0.06) 0.03,0.17(0.05) 0.44,0.23(0.3) 0.59,0.87(0.7) 0.03,1.0(0.06)

Repo: django

Fault 5;Y:N=1047:1500; BL=0.41| 0.71,0.92(0.8) 0.80,0.79(0.79) 0.56,0.69(0.62) 0.82,0.05(0.09) 0.43,0.22(0.29) 0.78,0.05(0.09) 0.46,0.57(0.51) 0.65,0.75(0.7) 0.58,0.85(0.69) 0.88,0.03(0.06)

Repo: ansible

Fault 5;Y:N=2051:1500; BL=0.58| 0.77,0.87(0.82) 0.82,0.79(0.8) 0.74,0.67(0.7) 0.57,0.96(0.72) 0.72,0.77(0.74) 0.83,0.01(0.02) 0.65,0.62(0.63) 0.71,0.73(0.72) 0.71,0.91(0.8) 0.58,1.0(0.73)

Repo: scrapy

Fault 5;Y:N=430:1500; BL=0.22| 0.48,0.92(0.63) 0.72,0.79(0.75) 0.34,0.77(0.47) 0.22,0.95(0.36) 0.21,0.14(0.17) 0.22,0.95(0.36) 0.26,0.52(0.35) 0.47,0.58(0.52) 0.45,0.88(0.6) 0.67,0.02(0.04)

6.2 Voting ensembles, using additional user meta-data attributes that are history based

Votes: 2 4 6 8 9 10

nginx - Fault 1;Y:N=129:6426; BL=0.02| : [0.09, 0.81(0.16)] [0.2, 0.71(0.31)] [0.31, 0.37(0.34)] [0.75, 0.09(0.16)] [1.0, 0.01(0.02)] [0.0, 0.0(NA)]
nginx - Fault 5;Y:N=471:6084; BL=0.07| : [0.32, 0.97(0.48)] [0.61, 0.88(0.72)] [0.82, 0.69(0.75)] [0.98, 0.33(0.49)] [0.98, 0.19(0.32)] [0.0, 0.0(NA)]
apache - Fault 1;Y:N=190:31074; BL=0.01| : [0.03, 0.93(0.06)] [0.1, 0.63(0.17)] [0.39, 0.14(0.21)] [0.5, 0.01(0.02)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
apache - Fault 3;Y:N=127:31137; BL=0.0| : [0.02, 0.8(0.04)] [0.15, 0.38(0.22)] [0.29, 0.03(0.05)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
apache - Fault 5;Y:N=559:30705; BL=0.02| : [0.05, 0.94(0.09)] [0.14, 0.74(0.24)] [0.39, 0.23(0.29)] [0.67, 0.02(0.04)] [1.0, 0.0(0.0)] [0.0, 0.0(NA)]
image magick - Fault 1;Y:N=158:14972; BL=0.01| : [0.03, 0.75(0.06)] [0.14, 0.26(0.18)] [0.29, 0.02(0.04)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
image magick - Fault 5;Y:N=83:15047; BL=0.01| : [0.03, 0.56(0.06)] [0.09, 0.09(0.09)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
image magick - Fault 7;Y:N=88:15042; BL=0.01| : [0.02, 0.74(0.04)] [0.08, 0.31(0.13)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
curl - Fault 1;Y:N=542:23623; BL=0.02| : [0.06, 0.9(0.11)] [0.14, 0.58(0.23)] [0.31, 0.17(0.22)] [0.68, 0.04(0.08)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
curl - Fault 3;Y:N=76:24089; BL=0.0| : [0.02, 0.65(0.04)] [0.11, 0.25(0.15)] [0.33, 0.02(0.04)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
curl - Fault 5;Y:N=934:23231; BL=0.04| : [0.09, 0.93(0.16)] [0.22, 0.66(0.33)] [0.49, 0.29(0.36)] [0.82, 0.05(0.09)] [1.0, 0.01(0.02)] [0.0, 0.0(NA)]
wget - Fault 1;Y:N=64:3955; BL=0.02| : [0.09, 0.81(0.16)] [0.31, 0.27(0.29)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
wget - Fault 3;Y:N=20:3999; BL=0.0| : [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
wget - Fault 5;Y:N=130:3889; BL=0.03| : [0.13, 0.82(0.22)] [0.29, 0.39(0.33)] [0.33, 0.02(0.04)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
videolan/vlc - Fault 1;Y:N=2527:78906; BL=0.03| : [0.03, 1.0(0.06)] [0.06, 0.92(0.11)] [0.13, 0.63(0.22)] [0.25, 0.19(0.22)] [0.26, 0.04(0.07)] [0.3, 0.0(0.0)]
videolan/vlc - Fault 3;Y:N=1126:80307; BL=0.01| : [0.01, 1.0(0.02)] [0.03, 0.92(0.06)] [0.09, 0.63(0.16)] [0.19, 0.14(0.16)] [0.19, 0.02(0.04)] [0.0, 0.0(NA)]
videolan/vlc - Fault 5;Y:N=2653:78780; BL=0.03| : [0.03, 1.0(0.06)] [0.07, 0.93(0.13)] [0.17, 0.67(0.27)] [0.39, 0.27(0.32)] [0.45, 0.06(0.11)] [0.5, 0.01(0.02)]
nghttp2 - Fault 1;Y:N=16:6267; BL=0.0| : [0.04, 0.88(0.08)] [0.25, 0.25(0.25)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
nghttp2 - Fault 5;Y:N=92:6191; BL=0.01| : [0.09, 0.98(0.16)] [0.24, 0.41(0.3)] [0.45, 0.09(0.15)] [1.0, 0.04(0.08)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
mekentosi/podofo - Fault 1;Y:N=92:1171; BL=0.07| : [0.08, 1.0(0.15)] [0.29, 0.72(0.41)] [0.52, 0.44(0.48)] [0.8, 0.07(0.13)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
mekentosi/podofo - Fault 3;Y:N=92:1171; BL=0.07| : [0.07, 0.98(0.13)] [0.22, 0.8(0.35)] [0.45, 0.42(0.43)] [0.43, 0.05(0.09)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
mekentosi/podofo - Fault 5;Y:N=97:1166; BL=0.08| : [0.09, 1.0(0.17)] [0.29, 0.77(0.42)] [0.56, 0.3(0.39)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
openssl - Fault 1;Y:N=336:23655; BL=0.01| : [0.04, 0.88(0.08)] [0.12, 0.62(0.2)] [0.33, 0.14(0.2)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
openssl - Fault 3;Y:N=75:23916; BL=0.0| : [0.01, 0.88(0.02)] [0.06, 0.34(0.1)] [0.3, 0.07(0.11)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
openssl - Fault 5;Y:N=428:23563; BL=0.02| : [0.05, 0.9(0.09)] [0.13, 0.64(0.22)] [0.33, 0.19(0.24)] [0.8, 0.02(0.04)] [1.0, 0.0(0.0)] [0.0, 0.0(NA)]
libuv - Fault 1;Y:N=101:4276; BL=0.02| : [0.02, 0.96(0.04)] [0.06, 0.61(0.11)] [0.1, 0.13(0.11)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libuv - Fault 3;Y:N=18:4359; BL=0.0| : [0.0, 1.0(0.0)] [0.08, 0.67(0.14)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libuv - Fault 5;Y:N=157:4220; BL=0.04| : [0.03, 1.0(0.06)] [0.12, 0.76(0.21)] [0.32, 0.33(0.32)] [0.5, 0.03(0.06)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
EQEmu - Fault 1;Y:N=32:7299; BL=0.0| : [0.03, 0.41(0.06)] [0.19, 0.18(0.18)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
EQEmu - Fault 3;Y:N=119:7212; BL=0.02| : [0.05, 0.7(0.09)] [0.13, 0.24(0.17)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
EQEmu - Fault 5;Y:N=202:7129; BL=0.03| : [0.09, 0.8(0.16)] [0.2, 0.29(0.24)] [0.38, 0.03(0.06)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
mangosthree - Fault 1;Y:N=87:8151; BL=0.01| : [0.01, 0.96(0.02)] [0.05, 0.71(0.09)] [0.2, 0.29(0.24)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
mangosthree - Fault 3;Y:N=382:7856; BL=0.05| : [0.05, 0.99(0.1)] [0.11, 0.8(0.19)] [0.27, 0.47(0.34)] [0.43, 0.05(0.09)] [1.0, 0.0(0.0)] [0.0, 0.0(NA)]

mangosthree - Fault 5;Y:N=334:7904; BL=0.04| : [0.07, 0.92(0.13)] [0.14, 0.67(0.23)] [0.29, 0.19(0.23)] [0.33, 0.0(0.0)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libexif - Fault 1;Y:N=34:1109; BL=0.03| : [0.03, 1.0(0.06)] [0.11, 0.74(0.19)] [0.14, 0.16(0.15)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libexif - Fault 3;Y:N=11:1132; BL=0.01| : [0.01, 1.0(0.02)] [0.04, 0.6(0.07)] [0.5, 0.2(0.29)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libexif - Fault 5;Y:N=44:1099; BL=0.04| : [0.04, 1.0(0.08)] [0.15, 0.77(0.25)] [0.24, 0.23(0.23)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libpcap - Fault 1;Y:N=18:4416; BL=0.0| : [0.01, 0.5(0.02)] [0.04, 0.12(0.06)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libpcap - Fault 5;Y:N=58:4376; BL=0.01| : [0.07, 0.94(0.13)] [0.29, 0.46(0.36)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
trafficserver - Fault 1;Y:N=14:12097; BL=0.0| : [0.17, 0.27(0.21)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
trafficserver - Fault 5;Y:N=15:12096; BL=0.0| : [0.04, 0.64(0.08)] [0.04, 0.09(0.06)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
zaphoyd - Fault 5;Y:N=13:1685; BL=0.01| : [0.2, 0.5(0.29)] [1.0, 0.12(0.21)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libarchive - Fault 1;Y:N=91:5501; BL=0.02| : [0.08, 0.85(0.15)] [0.22, 0.4(0.28)] [0.3, 0.13(0.18)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libarchive - Fault 3;Y:N=13:5579; BL=0.0| : [0.02, 0.5(0.04)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libarchive - Fault 5;Y:N=129:5463; BL=0.02| : [0.09, 0.72(0.16)] [0.2, 0.23(0.21)] [0.17, 0.01(0.02)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libav - Fault 1;Y:N=375:43573; BL=0.01| : [0.03, 0.83(0.06)] [0.13, 0.48(0.2)] [0.34, 0.15(0.21)] [1.0, 0.01(0.02)] [1.0, 0.0(0.0)] [0.0, 0.0(NA)]
libav - Fault 3;Y:N=223:43973; BL=0.01| : [0.02, 0.68(0.04)] [0.12, 0.36(0.18)] [0.49, 0.13(0.21)] [1.0, 0.03(0.06)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libav - Fault 5;Y:N=759:43148; BL=0.02| : [0.06, 0.87(0.11)] [0.2, 0.53(0.29)] [0.49, 0.15(0.23)] [0.88, 0.01(0.02)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
sqlite - Fault 1;Y:N=624:20249; BL=0.03| : [0.07, 0.89(0.13)] [0.19, 0.57(0.28)] [0.4, 0.07(0.12)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
sqlite - Fault 3;Y:N=237:20636; BL=0.01| : [0.03, 0.85(0.06)] [0.16, 0.43(0.23)] [0.31, 0.03(0.05)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
sqlite - Fault 5;Y:N=1061:19812; BL=0.05| : [0.07, 0.97(0.13)] [0.15, 0.77(0.25)] [0.29, 0.33(0.31)] [0.39, 0.02(0.04)] [0.5, 0.0(0.0)] [0.0, 0.0(NA)]
libpnq - Fault 1;Y:N=38:4031; BL=0.01| : [0.08, 0.5(0.14)] [0.11, 0.09(0.1)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libpnq - Fault 5;Y:N=134:3935; BL=0.03| : [0.16, 0.76(0.26)] [0.27, 0.31(0.29)] [0.53, 0.11(0.18)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libming - Fault 5;Y:N=93:24; BL=0.79| : [0.82, 1.0(0.9)] [0.85, 0.91(0.88)] [0.94, 0.84(0.89)] [1.0, 0.67(0.8)] [1.0, 0.45(0.62)] [1.0, 0.09(0.17)]
libraw - Fault 1;Y:N=44:2399; BL=0.02| : [0.02, 0.96(0.04)] [0.12, 0.79(0.21)] [0.24, 0.21(0.22)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libraw - Fault 5;Y:N=39:2404; BL=0.02| : [0.11, 0.67(0.19)] [0.4, 0.15(0.22)] [1.0, 0.07(0.13)] [1.0, 0.04(0.08)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libtiff - Fault 1;Y:N=27:2979; BL=0.01| : [0.01, 0.86(0.02)] [0.06, 0.36(0.1)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libtiff - Fault 3;Y:N=14:2992; BL=0.0| : [0.0, 0.88(0.0)] [0.05, 0.25(0.08)] [1.0, 0.12(0.21)] [0.0, 0.0(NA)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libtiff - Fault 5;Y:N=245:2761; BL=0.08| : [0.08, 0.98(0.15)] [0.17, 0.87(0.28)] [0.32, 0.52(0.4)] [0.45, 0.1(0.16)] [0.4, 0.01(0.02)] [0.0, 0.0(NA)]
libzmq - Fault 1;Y:N=226:7285; BL=0.03| : [0.1, 0.94(0.18)] [0.37, 0.56(0.45)] [0.62, 0.1(0.17)] [1.0, 0.01(0.02)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]
libzmq - Fault 5;Y:N=108:7403; BL=0.01| : [0.01, 0.97(0.02)] [0.05, 0.67(0.09)] [0.12, 0.2(0.15)] [0.38, 0.05(0.09)] [0.0, 0.0(NA)] [0.0, 0.0(NA)]

6.3 XGBOOST Classifier ensembles

Dataset	Accuracy [Precision, Recall (F-Measure)]
nginx , Fault 1;Y:N=129:6426; BL=0.02	[0.57, 0.73(0.64)]
nginx , Fault 5;Y:N=471:6084; BL=0.07	[0.76, 0.84(0.8)]
apache , Fault 1;Y:N=190:31074; BL=0.01	[0.29, 0.45(0.35)]
apache , Fault 3;Y:N=127:31137; BL=0.0	[0.3, 0.39(0.34)]
apache , Fault 5;Y:N=559:30705; BL=0.02	[0.35, 0.48(0.4)]
image magick , Fault 1;Y:N=158:14972; BL=0.01	[0.24, 0.45(0.31)]
image magick , Fault 5;Y:N=83:15047; BL=0.01	[0.21, 0.3(0.25)]
image magick , Fault 7;Y:N=88:15042; BL=0.01	[0.16, 0.28(0.2)]
curl , Fault 1;Y:N=542:23623; BL=0.02	[0.29, 0.37(0.33)]
curl , Fault 3;Y:N=76:24089; BL=0.0	[0.0, 0.0(NA)]
curl , Fault 5;Y:N=934:23231; BL=0.04	[0.38, 0.45(0.41)]
wget , Fault 1;Y:N=64:3955; BL=0.02	[0.36, 0.11(0.17)]
wget , Fault 3;Y:N=20:3999; BL=0.0	[0.0, 0.0(NA)]
wget , Fault 5;Y:N=130:3889; BL=0.03	[0.49, 0.24(0.32)]
videolan/vlc , Fault 1;Y:N=2527:78906; BL=0.03	[0.23, 0.41(0.29)]
videolan/vlc , Fault 3;Y:N=1126:80307; BL=0.01	[0.19, 0.41(0.26)]
videolan/vlc , Fault 5;Y:N=2653:78780; BL=0.03	[0.31, 0.45(0.37)]
nghttp2 , Fault 1;Y:N=16:6267; BL=0.0	[0.0, 0.0(NA)]
nghttp2 , Fault 5;Y:N=92:6191; BL=0.01	[0.36, 0.09(0.14)]

Dataset	Accuracy [Precision, Recall (F-Measure)]
mangosthree , Fault 1;Y:N=87:8151; BL=0.01	[0.56, 0.17(0.26)]
mangosthree , Fault 3;Y:N=382:7856; BL=0.05	[0.28, 0.32(0.3)]
mangosthree , Fault 5;Y:N=334:7904; BL=0.04	[0.26, 0.32(0.29)]
libexif , Fault 1;Y:N=34:1109; BL=0.03	[0.62, 0.26(0.37)]
libexif , Fault 3;Y:N=11:1132; BL=0.01	[0.5, 0.2(0.29)]
libexif , Fault 5;Y:N=44:1099; BL=0.04	[0.38, 0.12(0.18)]
libpcap , Fault 1;Y:N=18:4416; BL=0.0	[0.0, 0.0(NA)]
libpcap , Fault 5;Y:N=58:4376; BL=0.01	[0.5, 0.09(0.15)]
trafficserver , Fault 1;Y:N=14:12097; BL=0.0	[0.0, 0.0(NA)]
trafficserver , Fault 5;Y:N=15:12096; BL=0.0	[0.0, 0.0(NA)]
zaphoyd , Fault 5;Y:N=13:1685; BL=0.01	[0.5, 0.12(0.19)]
libarchive , Fault 1;Y:N=91:5501; BL=0.02	[0.31, 0.47(0.37)]
libarchive , Fault 3;Y:N=13:5579; BL=0.0	[0.0, 0.0(NA)]
libarchive , Fault 5;Y:N=129:5463; BL=0.02	[0.48, 0.14(0.22)]
libav , Fault 1;Y:N=375:43573; BL=0.01	[0.23, 0.4(0.29)]
libav , Fault 3;Y:N=223:43973; BL=0.01	[0.24, 0.28(0.26)]
libav , Fault 5;Y:N=759:43148; BL=0.02	[0.27, 0.43(0.33)]
sqlite , Fault 1;Y:N=624:20249; BL=0.03	[0.36, 0.4(0.38)]
sqlite , Fault 3;Y:N=237:20636; BL=0.01	[0.36, 0.54(0.43)]

mekentosi/podofo , Fault 1;Y:N=92:1171; BL=0.07	[0.63, 0.2(0.3)]	sqlite , Fault 5;Y:N=1061:19812; BL=0.05	[0.34, 0.46(0.39)]
mekentosi/podofo , Fault 3;Y:N=92:1171; BL=0.07	[0.5, 0.24(0.32)]	libpnq , Fault 1;Y:N=38:4031; BL=0.01	[0.0, 0.0(NA)]
mekentosi/podofo , Fault 5;Y:N=97:1166; BL=0.08	[0.53, 0.15(0.23)]	libpnq , Fault 5;Y:N=134:3935; BL=0.03	[0.36, 0.35(0.35)]
openssl , Fault 1;Y:N=336:23655; BL=0.01	[0.25, 0.47(0.33)]	libming , Fault 5;Y:N=93:24; BL=0.79	[1.0, 0.91(0.95)]
openssl , Fault 3;Y:N=75:23916; BL=0.0	[0.08, 0.27(0.12)]	libraw , Fault 1;Y:N=44:2399; BL=0.02	[0.41, 0.38(0.39)]
openssl , Fault 5;Y:N=428:23563; BL=0.02	[0.25, 0.46(0.32)]	libraw , Fault 5;Y:N=39:2404; BL=0.02	[0.5, 0.07(0.12)]
libuv , Fault 1;Y:N=101:4276; BL=0.02	[0.18, 0.31(0.23)]	libtiff , Fault 1;Y:N=27:2979; BL=0.01	[0.0, 0.0(NA)]
libuv , Fault 3;Y:N=18:4359; BL=0.0	[0.5, 0.11(0.18)]	libtiff , Fault 3;Y:N=14:2992; BL=0.0	[0.5, 0.12(0.19)]
libuv , Fault 5;Y:N=157:4220; BL=0.04	[0.33, 0.45(0.38)]	libtiff , Fault 5;Y:N=245:2761; BL=0.08	[0.45, 0.58(0.51)]
EQEmu , Fault 1;Y:N=32:7299; BL=0.0	[0.0, 0.0(NA)]	libzmq , Fault 1;Y:N=226:7285; BL=0.03	[0.41, 0.5(0.45)]
EQEmu , Fault 3;Y:N=119:7212; BL=0.02	[0.42, 0.07(0.12)]	libzmq , Fault 5;Y:N=108:7403; BL=0.01	[0.24, 0.27(0.25)]
EQEmu , Fault 5;Y:N=202:7129; BL=0.03	[0.29, 0.44(0.35)]		

6.4 XGBOOST (meta-classifier) BASED with ONLY Decision Tree and Random Forest

nginx , Fault 1;Y:N=129:6426; BL=0.02 [0.53, 0.73(0.61)]
nginx , Fault 5;Y:N=471:6084; BL=0.07 [0.77, 0.8(0.78)]
apache , Fault 1;Y:N=190:31074; BL=0.01 [0.3, 0.51(0.38)]
apache , Fault 3;Y:N=127:31137; BL=0.0 [0.35, 0.42(0.38)]
apache , Fault 5;Y:N=559:30705; BL=0.02 [0.37, 0.46(0.41)]
image magick , Fault 1;Y:N=158:14972; BL=0.01 [0.28, 0.37(0.32)]
image magick , Fault 5;Y:N=83:15047; BL=0.01 [0.13, 0.15(0.14)]
image magick , Fault 7;Y:N=88:15042; BL=0.01 [0.15, 0.24(0.18)]
curl , Fault 1;Y:N=542:23623; BL=0.02 [0.33, 0.39(0.36)]
curl , Fault 3;Y:N=76:24089; BL=0.0 [0.27, 0.27(0.27)]
curl , Fault 5;Y:N=934:23231; BL=0.04 [0.38, 0.46(0.42)]
wget , Fault 1;Y:N=64:3955; BL=0.02 [0.49, 0.49(0.49)]
wget , Fault 3;Y:N=20:3999; BL=0.0 [0.0, 0.0(NA)]
wget , Fault 5;Y:N=130:3889; BL=0.03 [0.4, 0.42(0.41)]
videolan/vlc , Fault 1;Y:N=2527:78906; BL=0.03 [0.23, 0.38(0.29)]
videolan/vlc , Fault 3;Y:N=1126:80307; BL=0.01 [0.19, 0.43(0.26)]
videolan/vlc , Fault 5;Y:N=2653:78780; BL=0.03 [0.32, 0.47(0.38)]
nghttp2 , Fault 1;Y:N=16:6267; BL=0.0 [1.0, 0.38(0.55)]
nghttp2 , Fault 5;Y:N=92:6191; BL=0.01 [0.27, 0.36(0.31)]
mekentosi/podofo , Fault 1;Y:N=92:1171; BL=0.07 [0.54, 0.51(0.52)]
mekentosi/podofo , Fault 3;Y:N=92:1171; BL=0.07 [0.37, 0.51(0.43)]
mekentosi/podofo , Fault 5;Y:N=97:1166; BL=0.08 [0.44, 0.55(0.49)]
openssl , Fault 1;Y:N=336:23655; BL=0.01 [0.24, 0.44(0.31)]
openssl , Fault 3;Y:N=75:23916; BL=0.0 [0.1, 0.32(0.15)]
mangosthree , Fault 1;Y:N=87:8151; BL=0.01 [0.18, 0.35(0.24)]
mangosthree , Fault 3;Y:N=382:7856; BL=0.05 [0.31, 0.39(0.35)]
mangosthree , Fault 5;Y:N=334:7904; BL=0.04 [0.32, 0.42(0.36)]
libexif , Fault 1;Y:N=34:1109; BL=0.03 [0.53, 0.47(0.5)]
libexif , Fault 3;Y:N=11:1132; BL=0.01 [0.8, 0.8(0.8)]
libexif , Fault 5;Y:N=44:1099; BL=0.04 [0.48, 0.58(0.53)]
libpcap , Fault 1;Y:N=18:4416; BL=0.0 [0.15, 0.25(0.19)]
libpcap , Fault 5;Y:N=58:4376; BL=0.01 [0.42, 0.66(0.51)]
trafficserver , Fault 1;Y:N=14:12097; BL=0.0 [0.5, 0.18(0.26)]
trafficserver , Fault 5;Y:N=15:12096; BL=0.0 [0.14, 0.09(0.11)]

openssl , Fault 5;Y:N=428:23563; BL=0.02 [0.25, 0.47(0.33)] libuv , Fault 1;Y:N=101:4276; BL=0.02 [0.17, 0.3(0.22)] libuv , Fault 3;Y:N=18:4359; BL=0.0 [0.07, 0.11(0.09)] libuv , Fault 5;Y:N=157:4220; BL=0.04 [0.36, 0.44(0.4)] EQEmu , Fault 1;Y:N=32:7299; BL=0.0 [0.19, 0.47(0.27)] EQEmu , Fault 3;Y:N=119:7212; BL=0.02 [0.28, 0.37(0.32)] EQEmu , Fault 5;Y:N=202:7129; BL=0.03 [0.28, 0.38(0.32)]	

6.5 Important Grams in Code

Repo 1 , Fault 1

For Fault commits:

Top 10 correlated unigrams: index , line , emerg , handler , type , connection , kqueue , void , cache , errno

Top 10 correlated bigrams: socklen sizeof , error ngx , errno err , sockaddr struct , sizeof ngx , emerg cf , log alert , log emerg , pool sizeof , char ngx

For NO fault commits:

Top 10 correlated unigrams: head , warn , table , domain , addr , wev , goto , atoi , nan , dst

Top 10 correlated bigrams: data return , return endif , af inet , inet ngx , char buf , mutex unlock , addr addr , continue ngx , data value , ngx atoi

Repo 1 , Fault 5

For Fault commits:

Top 10 correlated unigrams: ctx , hunk , wev , fd , return , event , ev , ngx , endif , nan

Top 10 correlated bigrams: palloc pool , err ngx , log ngx , ngx fd , ngx pid , event ngx , log info , ngx http , ngx log , include ngx

For NO fault commits:

Top 10 correlated unigrams: ranges , fastcgi , stderr , crlf , body , status , current , code , handle , valid

Top 10 correlated bigrams: ch ch , ngx invalid , data char , start ngx , shm zone , value data , http header , request body , debug http , ctx len

Repo 2 , Fault 1

For Fault commits:

Top 10 correlated unigrams: errfile , response , nan , dobj , reap , queue , compare , actual , rp , io

Top 10 correlated bigrams: sc server , conn connection , headers headers , stream pool , tmp bb , brigade split , static ap , filter return , proxy function , pool conn

For NO fault commits:

Top 10 correlated unigrams: level , doesn , check , aborted , real , note , work , rerror , ifndef , path

Top 10 correlated bigrams: byteranges instead , ifdef sigwinch , bucket transient , transient create , egeneral return , ap filter , ap hook , return http , log rerror , rerror aplog

Repo 2 , Fault 3

For Fault commits:

Top 10 correlated unigrams: struct , accessed , transferred , read , start , ifndef , eos , config , endif , et

Top 10 correlated bigrams: ap select , egeneral return , value kip , break ap , int type , rec request , return ap , brigade bb , string len , bucket brigade

For NO fault commits:

Top 10 correlated unigrams: assert , allocation , auto , item , desc , counter , shm , slotmem , task , beam

Top 10 correlated bigrams: include util , io io , char line , return status , beam beam , ap debug , task task , debug assert , input beam , task input

Repo 2 , Fault 5

For Fault commits:

Top 10 correlated unigrams: apr , ap , sequence , hp , handles , buff , obj , refcount , task , nan

Top 10 correlated bigrams: ap debug , struct hostent , destroy pool , use atomics , len read , ssl callback , task output , create pool , debug assert , obj refcount

For NO fault commits:

Top 10 correlated unigrams: tmp , filetype , rc , message , based , present , str , work , pthread , called

Top 10 correlated bigrams: char err , text html , char end , ap strchr , length apr , apr dir , type apr , ok return , server error , core dir

Repo 4 , Fault 1

For Fault commits:

Top 10 correlated unigrams: character , 17 , 01 , errno , ong , artifacts , 00 , cli , sun , morphology

Top 10 correlated bigrams: info id , image header , quantum info , size ead , count ssize , ssize length , length break , wand wand , info page , cli wand

For NO fault commits:

Top 10 correlated unigrams: eset , signature , ocale , black , png , critical , resize , windows , continue , export

Top 10 correlated bigrams: const unsigned , authentic pixels , red image , static void , image image , size type , agick size , false status , ernel info , locale file

Repo 4 , Fault 5

For Fault commits:

Top 10 correlated unigrams: composite , local , function , specification , integer , trans , goto , bitmap , eturn , device

Top 10 correlated bigrams: psd info , composite op , png color , source colorspace , memory size , alpha traits , op case , number colors , info id , type sync

For NO fault commits:

Top 10 correlated unigrams: etermine , epsilon , ync , index , ax , class , method , long , storage , register

Top 10 correlated bigrams: channel blue , proceed defined , continue channel , ueue cache , endif proceed , attributes assert , storage class , void et , image estroy , mage null

Repo 4 , Fault 7

For Fault commits:

Top 10 correlated unigrams: global , field , tiffg , mapped , build , cli , fourier , block , ourier , single

Top 10 correlated bigrams: field tiff , signature signature , sizeof pixel , mapped agick , double kernel , color red , cli wand , info mapped , null draw , memory info

For NO fault commits:

Top 10 correlated unigrams: data , event , channel , hue , component , authentic , debug , extent , og , mage

Top 10 correlated bigrams: image attributes , attributes assert , pixel channel , void og , og magick , event et , magick event , elinquish magick , pixel channels , image channel

Repo 6 , Fault 1

For Fault commits:

Top 10 correlated unigrams: alloc , result , external , tok , far , outfile , nan , free , fds , char

Top 10 correlated bigrams: add buffer , result false , return null , data change , null char , change proxy , base 64 , conn firstsocket , define curl , connect conn

For NO fault commits:

Top 10 correlated unigrams: blocking , ignored , terminated , open , dd , ase , sslversion , wait , body , ingle

Top 10 correlated bigrams: ransfer ncoding , int nread , cleanup struct , ssl version , new url , ase 64 , len strlen , conn send , conn secondarysocket , function returns

Repo 6 , Fault 3

For Fault commits:

Top 10 correlated unigrams: recipient , opy , completion , holding , nitialize , basically , necessary , uery , secpkg , passdwp

Top 10 correlated bigrams: recipient arameters , message ready , data session , userp passwdp , char passwdp , outlen curl , passdwp user , userp user , char userp , userp const

For NO fault commits:

Top 10 correlated unigrams: ase , existing , index , os , 32 , win , mac , target , obsolete , usage

Top 10 correlated bigrams: conn data , url add , sys select , data easy , case curle , win 32 , int res , proxy host , data null , 32 cleanup

Repo 6 , Fault 5

For Fault commits:

Top 10 correlated unigrams: char , send , problem , conn , ftp , data , connected , url , false , nan

Top 10 correlated bigrams: struct ession , ession handle , handle data , handle multi , ssl connection , curle ok , bits tunnel , data conn , easy handle , char ptr

For NO fault commits:

Top 10 correlated unigrams: subject , note , timestamp , headerfile , decode , big , ld , method , nodelay , encoded

Top 10 correlated bigrams: conn allocptr , data progress , url base , curle ssl , static curl , win 32 , ifdef curl , passwd result , host result , tcp nodelay

Repo 7 , Fault 1

For Fault commits:

Top 10 correlated unigrams: used , user , null , init , response , ptr , path , store , st , type

Top 10 correlated bigrams: static int , logprintf log , log verbose , local file , resp header , auth finished , user passwd , ip address , static bool , static char

For NO fault commits:

Top 10 correlated unigrams: ind , owever , ote , know , reverse , contents , eep , strcasecmp , onnection , log

Top 10 correlated bigrams: header req , char end , logputs log , null return , set header , request set , convert links , end return , html files , rel value

Repo 7 , Fault 3

For Fault commits:

Top 10 correlated unigrams: errors , command , pointer , ote , required , add , authorization , right , multiple , reverse

Top 10 correlated bigrams: return null , len strlen , enable ipv , int file , address addr , null return , return return , resp header , int fd , char end

For NO fault commits:

Top 10 correlated unigrams: static , secs , used , cookie , downloaded , char , seen , conversion , path , maybe

Top 10 correlated bigrams: char const , int int , hash table , params const , ifdef windows , set header , request set , rel value , header req , static bool

Repo 7 , Fault 5

For Fault commits:

Top 10 correlated unigrams: filename , char , struct , data , using , close , port , res , log , nan

Top 10 correlated bigrams: url file , logputs log , char const , log notquiet , output document , opt output , opt verbose , local file , static void , static char

For NO fault commits:

Top 10 correlated unigrams: defined , init , old , store , sure , run , hen , default , abort , current

Top 10 correlated bigrams: address addr , set header , header req , request set , rel value , char end , return return , return true , static int , static struct

Repo 10 , Fault 1

For Fault commits:

Top 10 correlated unigrams: fclose , free , editions , types , dlclose , model , input , protected , psz , nan

Top 10 correlated bigrams: sd psz , free psz , psz meta , bg mux , sys types , return total , stream stream , ead line , psz buffer , pes size

For NO fault commits:

Top 10 correlated unigrams: actory , x05 , identity , chromas , truncated , code , orce , strcmp , coord , double

Top 10 correlated bigrams: psz vlc , output aout , duration et , strdup psz , visible height , sizeof vout , sys video , ufd fd , module vlc , output stream

Repo 10 , Fault 3

For Fault commits:

Top 10 correlated unigrams: png , vasprintf , folder , equal , tab , allow , 00 , kludge , flags , olor

Top 10 correlated bigrams: init sys , psz vlc , qp oint , bg obj , input var , button new , fd vlc , psz header , ql ist , tmp return

For NO fault commits:

Top 10 correlated unigrams: comment , mute , spudec , submodules , menu , gpu , strncpy , onfiguration , nan , ebuild

Top 10 correlated bigrams: lockval address , stack central , unlock lockval , include intf , push var , free input , item category , aout input , menu null , define frame

Repo 10 , Fault 5

For Fault commits:

Top 10 correlated unigrams: tree , dec , dvd , tests , intf , vdec , css , vpar , netlist , nan

Top 10 correlated bigrams: rr msg , es decoder , bytes line , fprintf stderr , ac dec , intf bg , intf rr , widget size , bg msg , vout bytes

For NO fault commits:

Top 10 correlated unigrams: dvdnav , panels , got , hoose , nsupported , high , pointers , esc , ugly , body

Top 10 correlated bigrams: display mode , arn demux , media type , libvlc event , display width , main panel , intf volume , mouse moved , demux null , null continue

Repo 11 , Fault 1

For Fault commits:

Top 10 correlated unigrams: stream , check , pri , ssize , nvlen , head , include , num , valuelen , readlen

Top 10 correlated bigrams: data frame , hd stream , flag end , nghttp session , stream dep , frame hd , nghttp shut , end stream , stream nghttp , stream session

For NO fault commits:

Top 10 correlated unigrams: raw , tablelen , ctrl , don , provider , ignore , cat , pack , temporal , function

Top 10 correlated bigrams: frame type , callback session , uint flags , error nghttp , session add , init nghttp , frame return , init frame , frame session , data spdylay

Repo 11 , Fault 5

For Fault commits:

Top 10 correlated unigrams: priority , http , spec , res , free , pri , payload , nghttp , stream , readlen

Top 10 correlated bigrams: nghttp nv , effective weight , window size , case nghttp , break case , nghttp data , weight nghttp , stream free , pri spec , return stream

For NO fault commits:

Top 10 correlated unigrams: promise , capacity , mask , raw , ping , nomem , opening , del , static , default

Top 10 correlated bigrams: inflate init , preface len , session add , uint flags , data data , init nghttp , session spdylay , data spdylay , stream init , nghttp hd

Repo 12 , Fault 1

For Fault commits:

Top 10 correlated unigrams: params , variant , clear , free , int , error , buffer , df , dictionary , nan

Top 10 correlated bigrams: psz filename , et dictionary , object et , output stream , df dictionary , dictionary dd , df object , dd key , df variant , df error

For NO fault commits:

Top 10 correlated unigrams: rite , len , table , tring , info , 14 , string , op , operator , range

Top 10 correlated bigrams: buffer len , df reference , operator rhs , std vector , object obj , static cast , df output , const iterator , buffer size , df document

Repo 12 , Fault 3

For Fault commits:

Top 10 correlated unigrams: test , font , ifdef , ase , reate , case , width , writer , var , 14

Top 10 correlated bigrams: variant var , error ut , ut memory , const int , len buffer , const df , filter filter , object object , df font , font et

For NO fault commits:

Top 10 correlated unigrams: null , delete , format , cache , ref , tmp , load , ap , code , nan

Top 10 correlated bigrams: output stream , df object , color space , object obj , object et , df page , object eference , char buffer , df error , df reference

Repo 12 , Fault 5

For Fault commits:

Top 10 correlated unigrams: number , return , path , parser , xr , objects , buffer , psz , writer , nan

Top 10 correlated bigrams: df font , object df , object obj , vec objects , char psz , buffer size , reate font , df object , xr ef , char buffer

For NO fault commits:

Top 10 correlated unigrams: obj , descriptor , reference , malloc , set , key , list , ase , trailer , handle

Top 10 correlated bigrams: et buffer , variant var , output device , char malloc , cast char , df output , podofo raise , et dictionary , stream et , font et

Repo 13 , Fault 1

For Fault commits:

Top 10 correlated unigrams: hash , err , abort , good , async , total , gnore , nan , job , necessary

Top 10 correlated bigrams: update md , void buf , return 509 , buffer null , async job , igest update , openssl crypto , int ok , hash evp , free bio

For NO fault commits:

Top 10 correlated unigrams: req , rt , bits , just , know , initial , hint , application , fips , retry

Top 10 correlated bigrams: client hello , openssl psk , key sizeof , al fatal , set cert , openssl sys , state ssl , ifdef openssl , int type , flags ssl

Repo 13 , Fault 3

For Fault commits:

Top 10 correlated unigrams: mentioning , eric , core , 65 , packet , 256 , digest , 12 , session , pkey

Top 10 correlated bigrams: ll advertising , distribution ll , advertising materials , features use , software display , mentioning features , materials mentioning , default return , int main , reserved edistribution

For NO fault commits:

Top 10 correlated unigrams: rights , nid , 255 , sent , selftest , xff , obj , rec , aa , ve

Top 10 correlated bigrams: obj nid , sizeof ctx , return evp , int char , break endif , ssl debug , md endif , ifdef ssl , debug fprintf , obj obj

Repo 13 , Fault 5

For Fault commits:

Top 10 correlated unigrams: fixture , crl , packet , mismatch , cookie , tag , nist , mdebug , pkt , nan

Top 10 correlated bigrams: al ssl , debug fprintf , tlsext err , hash evp , uint 32 , rand bytes , 509 crl , decode error , crypto mdebug , bn nist

For NO fault commits:

Top 10 correlated unigrams: allow , memcpy , min , ctrl , certificate , prime , dup , input , long , neg

Top 10 correlated bigrams: ctx bn , version ssl , bn zero , return tls , tls 12 , len null , ec group , public key , key null , ossl statem

Repo 14 , Fault 1

For Fault commits:

Top 10 correlated unigrams: error , status , addr , size , sizeof , eintr , io , writable , ngx , sockaddr

Top 10 correlated bigrams: sys error , req handle , uv req , uv stream , uv tcp , watcher uv , int uv , errno eintr , ngx queue , struct sockaddr

For NO fault commits:

Top 10 correlated unigrams: arg , mode , file , goto , int , req , strdup , length , current , result

Top 10 correlated bigrams: size return , error error , buffer size , handle int , defined apple , apple defined , fs req , loop init , uv init , int result

Repo 14 , Fault 3

For Fault commits:

Top 10 correlated unigrams: char , win , ip , type , path , user , count , line , ticks , address

Top 10 correlated bigrams: int uv , ngx queue , io watcher , void uv , static void , uv io , int fd , unsigned int , uv signal , char path

For NO fault commits:

Top 10 correlated unigrams: valgrind , self , guard , wsag , ipv , open , access , switch , insert , include

Top 10 correlated bigrams: loop int , return assert , ev return , return void , return loop , cb int , ev unref , stop uv , data uv , init ev

Repo 14 , Fault 5

For Fault commits:

Top 10 correlated unigrams: watcher , run , eintr , struct , ngx , 64 , want , sa , family , type

Top 10 correlated bigrams: type uv , sockaddr addr , queue init , init uv , ngx queue , errno eintr , req req , req handle , static int , const char

For NO fault commits:

Top 10 correlated unigrams: options , written , reading , dir , malloc , flags , memset , send , handle , line

Top 10 correlated bigrams: error loop , malloc sizeof , pipe handle , handle read , handle pipe , handle return , uv init , uv handle , handle flags , flags uv

Repo 15 , Fault 1

For Fault commits:

Top 10 correlated unigrams: timers , xffffffff , results , work , mysqlerror , ag , short , return , log , int

Top 10 correlated bigrams: auto row , id id , inst inv , item ame , err mysqlerror , begin mu , result og , result row , et inv , array query

For NO fault commits:

Top 10 correlated unigrams: outapp , select , packet , total , app , ffect , entries , spdat , ump , changed

Top 10 correlated bigrams: row row , result safe , char id , mysql res , res result , query select , result mysql , id emp , packet outapp , ump packet

Repo 15 , Fault 3

For Fault commits:

Top 10 correlated unigrams: target , orpse , ttack , erc , aug , iterator , extern , ana , effect , merc

Top 10 correlated bigrams: essage tring , tem class , emove merc , ast client , et group , sc lient , group et , spell spell , et merc , et id

For NO fault commits:

Top 10 correlated unigrams: message , string , true , array , uery , zone , const , value , item , pell

Top 10 correlated bigrams: result og , row query , delete array , id tem , safe delete , id atoi , errmsg size , mysql errmsg , row row , res result

Repo 15 , Fault 5

For Fault commits:

Top 10 correlated unigrams: damage , temp , changed , dont , 20 , xport , target , ry , added , lua

Top 10 correlated bigrams: et class , 100 command , truct item , sep arg , ataset result , eqa pplication , pplication packet , win 32 , database oad , et target

For NO fault commits:

Top 10 correlated unigrams: 11 , list , leader , hate , corpse , rom , online , file , needed , ake

Top 10 correlated bigrams: zone id , zone et , return false , hate list , id zone , list et , group id , id uint , eqe mu , query mysql

Repo 16 , Fault 1

For Fault commits:

Top 10 correlated unigrams: offline , applied , requirements , data , 20 , paladin , 29 , creature , type , et

Top 10 correlated bigrams: type iterator , id true , type uint , et battle , id result , et auras , data data , spellfamily druid , pet guid , row delete

For NO fault commits:

Top 10 correlated unigrams: odify , og , log , loaded , guess , vent , day , load , team , uild

Top 10 correlated bigrams: data smsg , log debug , bject accessor , target spell , guid et , class class , db reature , message type , accessor et , type const

Repo 16 , Fault 3

For Fault commits:

Top 10 correlated unigrams: size , dd , time , mail , flags , 16 , pos , triggered , etch , map

Top 10 correlated bigrams: fields result , entry const , spell effect , uint 64 , uery select , bject mgr , uery result , result result , result etch , typeid player

For NO fault commits:

Top 10 correlated unigrams: event , 32 , script , eset , list , create , combat , load , item , far

Top 10 correlated bigrams: caster et , delete itr , game object , dummy aura , pack guid , uint 32 , db reature , layer caster , et item , item item

Repo 16 , Fault 5

For Fault commits:

Top 10 correlated unigrams: bg , ch , realm , sec , config , arena , world , save , og , roup

Top 10 correlated bigrams: dummy aura , id player , bg et , world config , et instance , achievement mgr , uint 64 , player player , pet guid , const char

For NO fault commits:

Top 10 correlated unigrams: se , dd , nstance , plr , lower , items , used , int , debug , false

Top 10 correlated bigrams: et player , guid return , object mgr , ext row , iterator itr , attle ground , emove auras , spell spell , result result , uery result

Repo 17 , Fault 1

For Fault commits:

Top 10 correlated unigrams: strlen , undefined , long , nikon , return , break , order , default , case , exif

Top 10 correlated bigrams: exif format , data data , case exif , data size , exif data , data priv , ifd exif , exif entry , format undefined , case mnote

For NO fault commits:

Top 10 correlated unigrams: entry , printf , xif , memset , olympus , debug , null , mnote , int , count

Top 10 correlated bigrams: case strncpy , maxlen break , entry components , buf size , entry format , sizeof xif , val maxlen , unsigned int , snprintf val , mnote data

Repo 17 , Fault 3

For Fault commits:

Top 10 correlated unigrams: entry , buf , break , strncpy , vs , val , strlen , case , sizeof , xif

Top 10 correlated bigrams: val maxlen , exif short , exif tag , maxlen break , break case , strncpy val , entry data , sizeof xif , case exif , xif entry

For NO fault commits:

Top 10 correlated unigrams: offset , parent , format , count , order , note , min , priv , nikon , void

Top 10 correlated bigrams: data priv , entry components , cf entry , data entry , cc entry , exif format , entry size , entry order , components maxlen , snprintf maxlen

Repo 17 , Fault 5

For Fault commits:

Top 10 correlated unigrams: nan , format , snprintf , olympus , cc , maxlen , unsigned , int , entry , strncpy

Top 10 correlated bigrams: exif format , exif short , format short , cc entry , strncpy val , entry order , unsigned char , val maxlen , maxlen cc , snprintf val

For NO fault commits:

Top 10 correlated unigrams: include , memset , offset , strlen , min , maker , long , debug , endif , return

Top 10 correlated bigrams: sizeof xif , data data , case exif , mnote data , exif data , exif ifd , data size , maker note , exif entry , tag exif

Repo 18 , Fault 1

For Fault commits:

Top 10 correlated unigrams: ust , sec , strncpy , siocgifflags , appears , beginning , did , hat , reason , local

Top 10 correlated bigrams: errno free , don support , null endif , header sizeof , size pcap , int ret , ifr sizeof , ifr ioctl , fd siocgifflags , return error

For NO fault commits:

Top 10 correlated unigrams: open , including , bother , snapshot , message , live , nl , string , err , bufsize

Top 10 correlated bigrams: pcap strerror , return pcap , return null , null return , pcap close , errno return , null snprintf , static int , fp null , live capture

Repo 18 , Fault 5

For Fault commits:

Top 10 correlated unigrams: strdup , min , appears , strerror , handle , dag , ll , md , offset , aix

Top 10 correlated bigrams: snprintf ebuf , null snprintf , include sys , 802 11 , case dlt , ieee 802 , static int , op pcap , handle md , dlt ieee

For NO fault commits:

Top 10 correlated unigrams: assume , strncpy , tv , means , siocgifflags , sec , hat , dd , int , pcap

Top 10 correlated bigrams: bpf filter , null pcap , errbuf return , void snprintf , snprintf handle , ifr sizeof , int ret , fd siocgifflags , ifr ioctl , tv sec

Repo 19 , Fault 1

For Fault commits:

Top 10 correlated unigrams: tr , client , addr , def , raffic , cls , erver , request , set , oc

Top 10 correlated bigrams: proxy config , ts lua , rocesses efault , http ctx , tr rocesses , raffic erver , config http

For NO fault commits:

Top 10 correlated unigrams: import , http , ts , config , nan , int , port , len , size , char

Top 10 correlated bigrams: config http , raffic erver , tr rocesses , http ctx , rocesses efault , ts lua , proxy config

Repo 19 , Fault 5

For Fault commits:

Top 10 correlated unigrams: proxy , http , config , nan , ts , error , null , value , field , loc

Top 10 correlated bigrams: raffic erver , rocesses efault , tr rocesses , config http , http ctx , ts lua , proxy config

For NO fault commits:

Top 10 correlated unigrams: request , hdr , raffic , efault , erver , rror , make , tsh , release , check

Top 10 correlated bigrams: proxy config , ts lua , http ctx , config http , tr rocesses , rocesses efault , raffic erver

Repo 20 , Fault 5

For Fault commits:

Top 10 correlated unigrams: utility , write , bool , eb , elog , websocket , alog , uint , status , websocketpp

Top 10 correlated bigrams: return true , utility hex , hex std , websocketpp utility , cout websocketpp , typedef websocketpp , websocketpp log , eb socket , lib error , ptr new

For NO fault commits:

Top 10 correlated unigrams: bind , placeholders , handle , read , echo , async , start , boost , asio , return

Top 10 correlated bigrams: std stringstream , connection std , connection ptr , io service , asio io , msg payload , boost bind , asio placeholders , placeholders error , boost asio

Repo 21 , Fault 1

For Fault commits:

Top 10 correlated unigrams: onvert , multi , ead , denied , cpio , use , void , data , entry , ae

Top 10 correlated bigrams: const char , error access , access denied , error lasterr , lasterr et , dosmaperr lasterr , denied errno , dword lasterr , archive entry , static int

For NO fault commits:

Top 10 correlated unigrams: sc , bidder , standard , unsigned , file , information , lseek , regular , rar , info

Top 10 correlated bigrams: static const , archive fatal , st st , enomem allocate , iso 9660 , len return , ret archive , unsigned int , st mode , unsigned char

Repo 21 , Fault 3

For Fault commits:

Top 10 correlated unigrams: char , free , end , zip , start , open , buffer , skip , option , uncompressed

Top 10 correlated bigrams: archive ok , const char , archive entry , archive read , return archive , struct archive , static int , stat st , struct stat , data struct

For NO fault commits:

Top 10 correlated unigrams: xe , se , uad , integer , rite , ctime , denied , est , 0 , does

Top 10 correlated bigrams: char utf , entry ctime , ctime ae , atime ae , ws int , wchar ws , fd errno , type handle , null free , support format

Repo 21 , Fault 5

For Fault commits:

Top 10 correlated unigrams: 64 , private , fprintf , code , unused , type , new , null , base , char

Top 10 correlated bigrams: path return , rite file , malloc sizeof , fd const , archive ok , static void , int fd , fprintf stderr , archive archive , return null

For NO fault commits:

Top 10 correlated unigrams: goto , sc , int , argv , read , target , stderr , filter , erify , ssize

Top 10 correlated bigrams: write filter , read ahead , break default , read consume , break case , unsigned int , archive read , include archive , int set , ret ret

Repo 22 , Fault 1

For Fault commits:

Top 10 correlated unigrams: enomem , start , avcodec , needed , odec , default , sample , video , frames , 256

Top 10 correlated bigrams: 16 le , type int , context avp , format int , odec context , codec codec , id codec , int 64 , avc odec , return ret

For NO fault commits:

Top 10 correlated unigrams: headers , lum , like , order , chr , ioc , yte , specific , request , function

Top 10 correlated bigrams: width int , ff thread , int width , int height , current picture , picture ptr , ioc ontext , yte ioc , ontext pb , frame number

Repo 22 , Fault 3

For Fault commits:

Top 10 correlated unigrams: release , ebml , dc , alac , try , exit , left , program , blocksize , right

Top 10 correlated bigrams: size av , typedef struct , avctx priv , ecode context , end avc , avctx release , int max , data size , exit program , context const

For NO fault commits:

Top 10 correlated unigrams: run , unsigned , log , width , pb , 64 , uint , break , skip , min

Top 10 correlated bigrams: format int , const avp , return static , rgb 32 , unsigned char , break default , 16 uint , av log , log error , static void

Repo 22 , Fault 5

For Fault commits:

Top 10 correlated unigrams: bitrate , framerate , height , pcm , ogg , true , avi , print , remaining , dv

Top 10 correlated bigrams: pict type , frame bits , pixels tab , type type , len len , static uint , type 16 , ecode context , uint src , 16 dst

For NO fault commits:

Top 10 correlated unigrams: audio , rame , cache , output , count , pix , flags , null , samples , add

Top 10 correlated bigrams: stream ost , 16 le , return static , 32 16 , 16 uint , type case , return endif , codec id , avf rame , uint buf

Repo 23 , Fault 1

For Fault commits:

Top 10 correlated unigrams: table , debug , delete , exists , able , trans , single , interp , prev , version

Top 10 correlated bigrams: 3f ind , parse expr , static int , rc rc , sort order , col tab , sqlite os , sqlite error , char buf , sqlite malloc

For NO fault commits:

Top 10 correlated unigrams: reserved , page , cnt , collate , hash , pghdr , ndebug , pager , ii , depth

Top 10 correlated bigrams: tab list , sqlite hash , sqlite ok , ok goto , db parse , assert sqlite , static char , ifndef ndebug , ager pager , int ii

Repo 23 , Fault 3

For Fault commits:

Top 10 correlated unigrams: em , cookie , trigger , byte , enc , ota , ndex , sorter , tbl , iter

Top 10 correlated bigrams: sort order , dbe coverage , ts 5i , int byte , table sqlite , iter iter , token token , unsigned int , ager pager , key info

For NO fault commits:

Top 10 correlated unigrams: num , associated , cursor , debug , connection , test , char , sure , itmask , converted

Top 10 correlated bigrams: result set , sqlite enable , src tab , malloc failed , return sqlite , end sqlite , char buf , db parse , col tab , sqlite debug

Repo 23 , Fault 5

For Fault commits:

Top 10 correlated unigrams: term , wrc , xplain , prev , way , format , nt , seek , best , ref

Top 10 correlated bigrams: gh dr , sqlite vdbe , op ull , assert sqlite , db sql , src tab , op sqlite , int ii , expr return , return wrc

For NO fault commits:

Top 10 correlated unigrams: compute , testing , oto , atabase , registers , given , ai , ws , sort , rite

Top 10 correlated bigrams: 3 ind , seq parse , coll seq , expr op , stmt stmt , sqlite 3 , aff assert , ws flags , parse reg , list expr

Repo 24 , Fault 1

For Fault commits:

Top 10 correlated unigrams: doing , add , way , sub , destroy , structrp , ex , routine , correct , won

Top 10 correlated bigrams: chunks png , static png , end png , endif ifdef , const bytep , data png , rows png , png infop , structrp png , static int

For NO fault commits:

Top 10 correlated unigrams: define , libpng , palette , null , void , gamma , version , dp , crc , return

Top 10 correlated bigrams: png read , bit depth , defined png , png malloc , null png , malloc png , ptr bit , row info , palette png , row null

Repo 24 , Fault 5

For Fault commits:

Top 10 correlated unigrams: malloc , define , chunk , crc , row , rp , bpp , idat , pixel , ihdr

Top 10 correlated bigrams: png size , ptr nvalid , ptr chunk , pixel depth , read 16 , png idat , chunk png , unsigned int , endif endif , row info

For NO fault commits:

Top 10 correlated unigrams: zlib , method , multiple , rgba , supported , read , filter , value , null , buf

Top 10 correlated bigrams: png debug , png structp , supported ifdef , end png , endif png , 32 png , info ptr , png read , png free , debug png

Repo 25 , Fault 1

For Fault commits:

Top 10 correlated unigrams: line , struct , offset , mp , version , extern , scale , bounds , use , swfv

Top 10 correlated bigrams: static int , read ui , utput write , swfr ect , swfs ound , int num , write ui , int flags , ovie add , return null

For NO fault commits:

Top 10 correlated unigrams: code , action , swf , println , case , actions , indent , decompile , puts , swfaction

Top 10 correlated bigrams: swfm ovie , method data , swff ont , swfi nput , ont font , swf error , swfo utput , break case , swf action , case swfaction

Repo 25 , Fault 5

For Fault commits:

Top 10 correlated unigrams: ction , begin , case , println , puts , indent , sact , decompile , actions , swfaction

Top 10 correlated bigrams: static int , swff ont , swfm ovie , write ui , swfb lock , utput block , method data , swf action , begin swf , case swfaction

For NO fault commits:

Top 10 correlated unigrams: utput , file , printf , write , offset , swfm , swfo , error , number , struct

Top 10 correlated bigrams: swff ont , static int , nt 16 , swfi nput , read ui , swfm atrix , ui nt , ovie add , swfo utput , utput write

Repo 26 , Fault 1

For Fault commits:

Top 10 correlated unigrams: flags , forc , height , 3 , parse , 16 , 10 , width , id , 32

Top 10 correlated bigrams: idata make , present uchar , make anon , offset seek , 9050 present , 940 ushort , 940 present , dcraw dcraw , save seek , ifp fseek

For NO fault commits:

Top 10 correlated unigrams: buf , start , ens , version , table , case , iso , lens , 0 , short

Top 10 correlated bigrams: unsigned len , buf 9050 , imgdata image , buf 940 , imgdata lens , imgdata sizes , ib start , lens makernotes , table buf , makernotes ens

Repo 26 , Fault 5

For Fault commits:

Top 10 correlated unigrams: adobe , ushort , length , mul , crop , len , printf , sizes , sorder , wb

Top 10 correlated bigrams: ib raw , imgdata color , ifdef libraw , library build , libraw library , load raw , raw class , ftell ifp , tag type , imgdata sizes

For NO fault commits:

Top 10 correlated unigrams: dx , 18 , iheight , cur , x0 , linear , writer , switch , 4 , imgdata

Top 10 correlated bigrams: 940 present , save ftell , margin col , image row , int col , row row , int row , height row , ushort imgdata , col width

Repo 27 , Fault 1

For Fault commits:

Top 10 correlated unigrams: heck , image , 24 , sbyte , open , byte , rgb , ile , ata , allocate

Top 10 correlated bigrams: tif tifftag , tag tiff , tiff sbyte , clientdata module , tif uint , 32 tiff , dir td , tiffd ata , type tiff , tiff float

For NO fault commits:

Top 10 correlated unigrams: compression , cp , short , bytes , cc , row , input , memcpy , mode , tsize

Top 10 correlated bigrams: et field , tiffe rror , tifff ield , uint 32 , tiff free , td td , tif tif , uint 16 , tiff memcpy , tiff short

Repo 27 , Fault 3

For Fault commits:

Top 10 correlated unigrams: tiff , tif , int , uint , dir , unsigned , dp , 16 , tiffd , entry

Top 10 correlated bigrams: tif dir , tiffe rror , tifff ield , tiff free , tiffs et , et field , td td , uint 32 , tif tif , uint 16

For NO fault commits:

Top 10 correlated unigrams: configuration , byte , tiffw , values , ata , switch , realloc , tiffc , open , allocate

Top 10 correlated bigrams: ind field , tag tiff , return tif , tifff ind , tiff long , tif clientdata , ext tif , rror ext , tif uint , clientdata module

Repo 27 , Fault 5

For Fault commits:

Top 10 correlated unigrams: tiffc , tiffg , write , colorimetry , sbyte , header , space , bytes , array , ok

Top 10 correlated bigrams: tif clientdata , ext tif , err tiffr , entry err , free data , tiff long , static void , tiff sbyte , colorimetry support , case tifftag

For NO fault commits:

Top 10 correlated unigrams: tdir , jpeg , nan , count , values , compression , cp , define , tsize , sizeof

Top 10 correlated bigrams: tiff free , et field , type tiff , yc bc , break case , tiff short , static int , return return , tiff memset , char module

Repo 28 , Fault 1

For Fault commits:

Top 10 correlated unigrams: data , valid , ok , cast , written , blocking , std , new , efault , ctx

Top 10 correlated bigrams: zmq ctx , errno einval , void zmq , msg msg , int rc , rc errno , errno assert , static cast , const int , errno efault

For NO fault commits:

Top 10 correlated unigrams: log , term , number , pipes , args , process , pgm , nbytes , transport , openpgm

Top 10 correlated bigrams: fd handle , ifdef zmq , current active , assert false , return false , assert nbytes , msg flags , zmq assert , cmd args , zmq openpgm

Repo 28 , Fault 5

For Fault commits:

Top 10 correlated unigrams: include , terminate , push , object , return , don , af , decoder , string , state

Top 10 correlated bigrams: char buffer , slot sync , errno einval , zmq recv , einval return , sizeof int , int assert , void zmq , return false , close errno

For NO fault commits:

Top 10 correlated unigrams: uses , getsockopt , bi , directional , pipepair , decode , structure , openbsd , false , data

Top 10 correlated bigrams: void socket , init errno , based select , return return , zmq openbsd , endpoint max , string char , wsa error , errno wsag , engine write

6.6 Feature importance for selected repositories

30: nginx_mock,

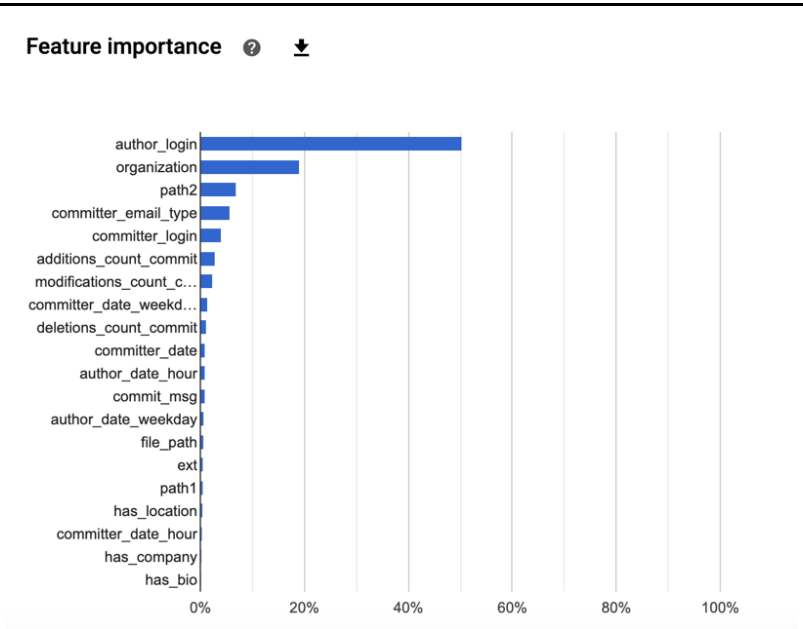
```
Fault 1;Y:N=129:6426; BL=0.02,
```

SciKit Feature Importance

```
[                              gain
 feature
 modifications_count_commit  0.43797
 author_login                0.34511
 committer_login             0.09768
 committer_date_hour         0.03975
 additions_count_commit      0.02318
 committer_date_weekday      0.01889
 deletions_count_commit      0.01683
 committer_email_type        0.01262
 author_date_hour            0.00801],
```

AutoML Feature Importance

```
author_login_chobits 0.06265
author_login_NMorozxov 0.05593
committer_login_Khaless 0.05227
author_login_LinuxJedi 0.05189
committer_login_tSed 0.04818
deletions_count_commit 0.04764
author_login_xeioex 0.03862
additions_count_commit 0.03804
author_login_Khaless 0.02357
author_login_bartw72 0.02198
committer_login_LinuxJedi 0.02131
committer_login_VBart 0.02037
committer_login_pushrax 0.01955
committer_login_FdaSilvaYY 0.01939
author_login_leblanc-simon 0.0193
committer_login_NMorozxov 0.01751
committer_login_maximkonovalov 0.01743
committer_login_chobits 0.01738
author_login_vl-homutov 0.01627
author_login_lepatryk 0.01448
author_login_maage 0.01339
committer_login_arut 0.01292
author_login_tSed 0.01289
author_login_ottok 0.01112
```



Weka Classification and Feature Importance

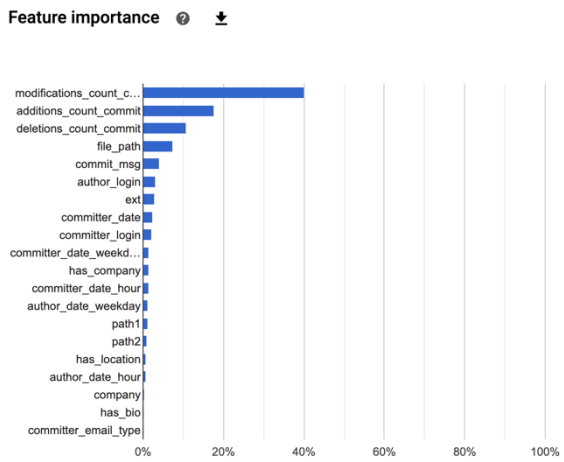
Weka: Precision = 0, Recall = 0

32: ssl_mock,

Fault 1;Y:N=336:23655; BL=0.01,

SciKit Feature Importance	AutoML Feature Importance

```
[                                gain
 feature
 deletions_count_commit       0.53116
 modifications_count_commit   0.45573
 author_login                 0.01243
 additions_count_commit       0.00073],
```



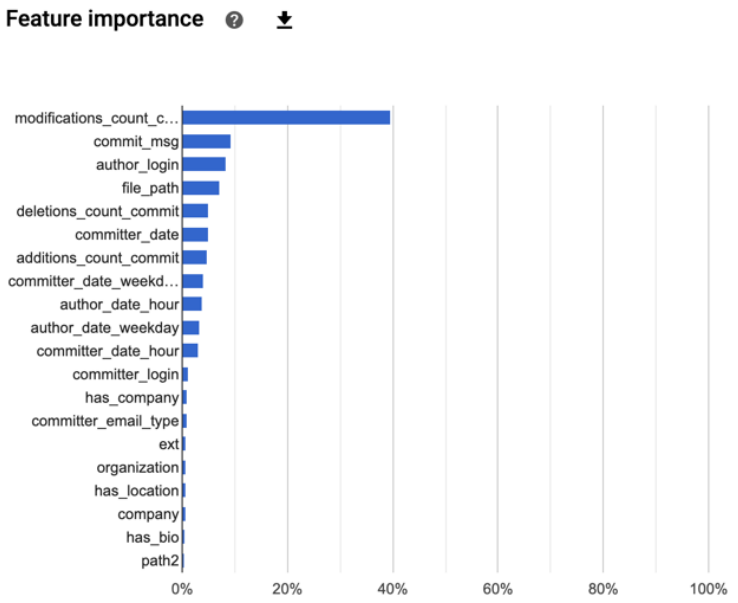
33: libuv_mock,

Fault 1;Y:N=101:4276; BL=0.02,

SciKit Feature Importance	AutoML Feature Importance

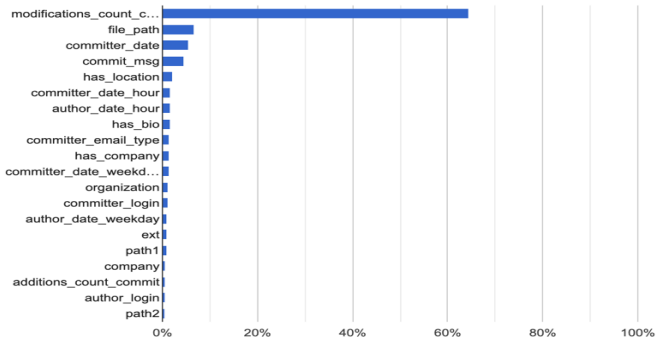
```
[                                gain
 feature
author_login                 0.91938
modifications_count_commit   0.03728
```

```
additions_count_commit      0.02721
deletions_count_commit      0.01610
],
author_login_1st1 0.07958
author_login_Trott 0.04894
author_login_ashaffer 0.04256
author_login_Qix- 0.04116
author_login_btrask 0.03992
modifications_count_commit 0.03728
author_login_bkize 0.02924
additions_count_commit 0.02721
author_login_bontibon 0.02539
author_login_Gottox 0.0253
author_login_Keno 0.02422
author_login_AvianFlu 0.02384
author_login_chopdown 0.02026
author_login_EdSchouten 0.01942
author_login_alex 0.0192
author_login_CurlyMoo 0.01829
author_login_HungMingWu 0.01716
author_login_ararslan 0.01673
author_login_XadillaX 0.0161
deletions_count_commit 0.0161
author_login_bajtos 0.01513
author_login_PeterJohnson 0.01438
author_login_bonifaido 0.01403
author_login_ai-rayven 0.01302
author_login_KryDos 0.01262
```



34: vlc_mock,

Fault 3;Y:N=1126:80363; BL=0.01,

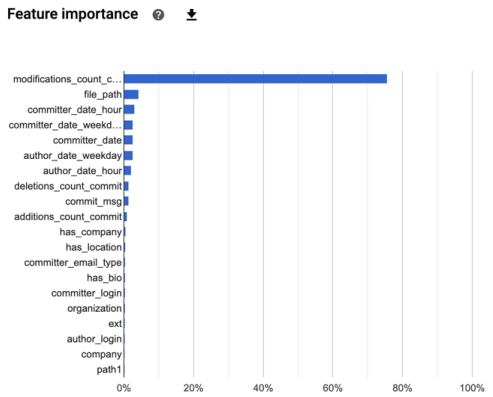
SciKit Feature Importance	AutoML Feature Importance
[gain feature modifications_count_commit 0.99975 additions_count_commit 0.00025],	

Note: For vlc_mock there seems to be an inflection point at modifications_count of 800,000. Above that almost all instances are Fault=Y. Below that NO instance is Fault = Y

35: server_mock,

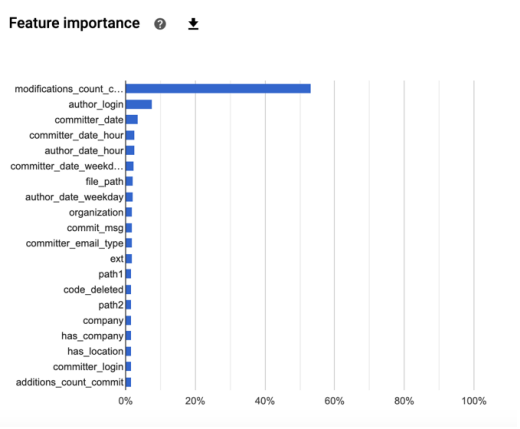
Fault 3;Y:N=382:7856; BL=0.05,

SciKit Feature Importance	AutoML Feature Importance
[gain feature modifications_count_commit 0.55289	

author_login 0.43844 deletions_count_commit 0.00616 additions_count_commit 0.00251],	

36: sqlite_mock,

Fault 3;Y:N=237:20636; BL=0.01,

SciKit Feature Importance	AutoML Feature Importance
[gain feature modifications_count_commit 0.99976 additions_count_commit 0.00024],	

37: curl_mock,

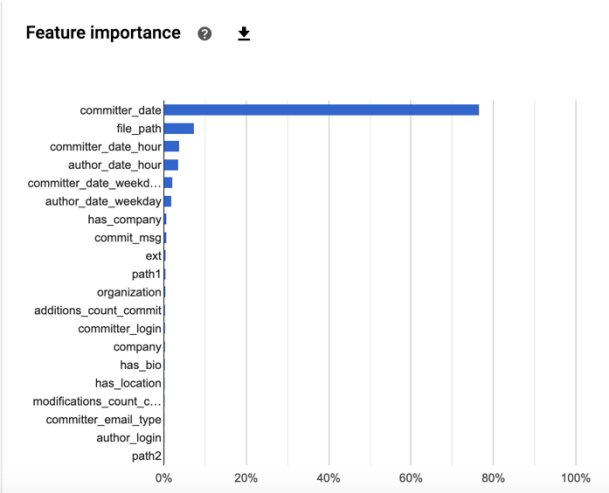
Fault 5;Y:N=934:23231; BL=0.04,

SciKit Feature Importance

```
[                              gain
 feature
 author_login                0.99941
 additions_count_commit      0.00041
 deletions_count_commit      0.00011
 modifications_count_commit  0.00010],

author_login_AndreHeinecke         0.97953
author_login_ArchangelSDY          0.01757
additions_count_commit    0.00041
author_login_Aulddays              0.00021
author_login_DimStar77    0.00021
author_login_Andersbakken          0.00021
author_login_Jactry                0.00011
author_login_CarloWood    0.00011
author_login_LEW21                 0.00011
author_login_ComputerDruid         0.00011
author_login_3dyd                  0.00011
author_login_Hawk777               0.00011
author_login_JeremR                0.00011
```

AutoML Feature Importance



author_login_Karlson2k 0.00011 author_login_FabianFrank 0.00011 author_login_CaViCcHi 0.00011 author_login_Alexxz 0.00011 deletions_count_commit 0.00011 author_login_CrazyHackGUT 0.00011 author_login_BurningEnlightenment 0.0001 author_login_Jan-E 0.0001 author_login_IhorKarpenko 0.0001 modifications_count_commit 0.0001 author_login_AdrianPeniak 4e-05 author_login_ErikMinekus 1e-05	

Weka Classification and Feature Importance

Weka: Decision Tree (J48). Precision = 0.996 Recall = 1.00

Information Gain Ranking Filter

Ranked attributes:

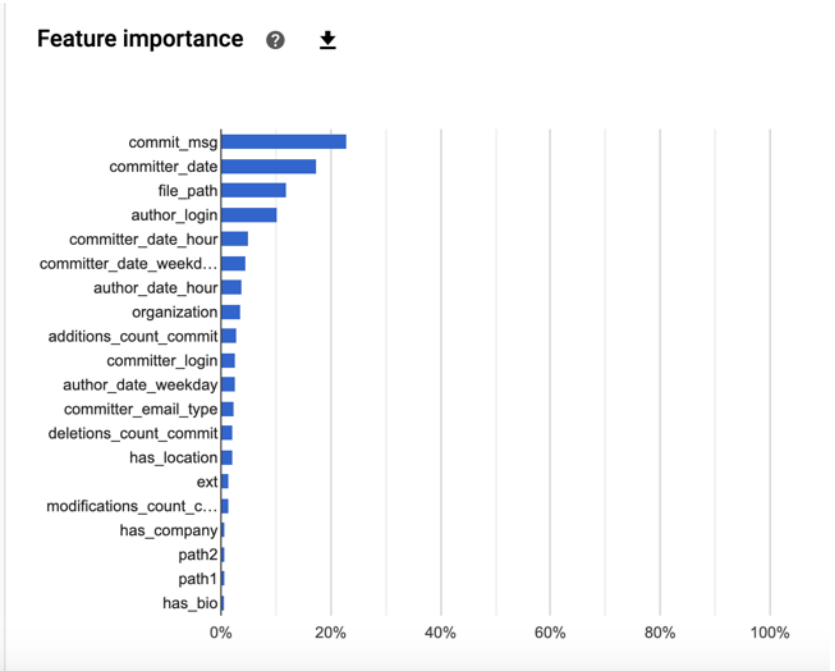
```
 0.236072   7 committer_date
 0.200384   8 committer_date_hour
 0.19899   10 author_date_hour
 0.09661   11 author_date_weekday
 0.096132   9 committer_date_weekday
 0.051154  18 path2
 0.030761   5 additions_count_commit
 0.029956   4 modifications_count_commit
 0.029824  19 ext
```

```
0.023498   6 deletions_count_commit
0.022194  17 path1
0.010134   1 author_login
0.003325  16 organization
0.002295   2 committer_login
0.000643  12 company
0.000632  14 has_bio
0.000506  15 has_company
0.000397   3 committer_email_type
0.000243  13 has_location
```

38: eqemu_mock,

```
Fault 5;Y:N=202:7129; BL=0.03,
```

SciKit Feature Importance	AutoML Feature Importance
[gain feature author_login 0.92086 additions_count_commit 0.05020 deletions_count_commit 0.01721 committer_login 0.00904 modifications_count_commit 0.00269], author_login_Derision 0.45327 author_login_AMDmi3 0.28062 author_login_Siroro 0.07093 additions_count_commit 0.0502 author_login_AthrogatePEQ 0.04712	

deletions_count_commit 0.01721 author_login_spdkils 0.0132 author_login_dencelle 0.01063 author_login_mackal 0.01024 committer_login_Golgie 0.00592 author_login_cavedude00 0.00427 modifications_count_commit 0.00269 author_login_Uleat 0.00234 author_login_briankinney 0.00228 author_login_Drajor 0.00214 author_login_Natedog2012 0.00116 author_login_SCMcLaughlin 0.00115 author_login_zerosum0x0 0.00103 author_login_Corysia 0.001 committer_login_Derision 0.00097 author_login_fryguy503 0.00096 author_login_Shendare 0.00096 author_login_SecretsOTheP 0.00088 author_login_httm 0.00071 author_login_Cilraaz 0.0007	

Weka Classification and Feature Importance

```
Weka Decision Tree (J48): Precision = 0.7 Recall = 0.69

Information Gain Ranking Filter
Ranked attributes:
 0.177819586    7 committer_date
 0.053244525    8 committer_date_hour
 0.052605641   10 author_date_hour
```

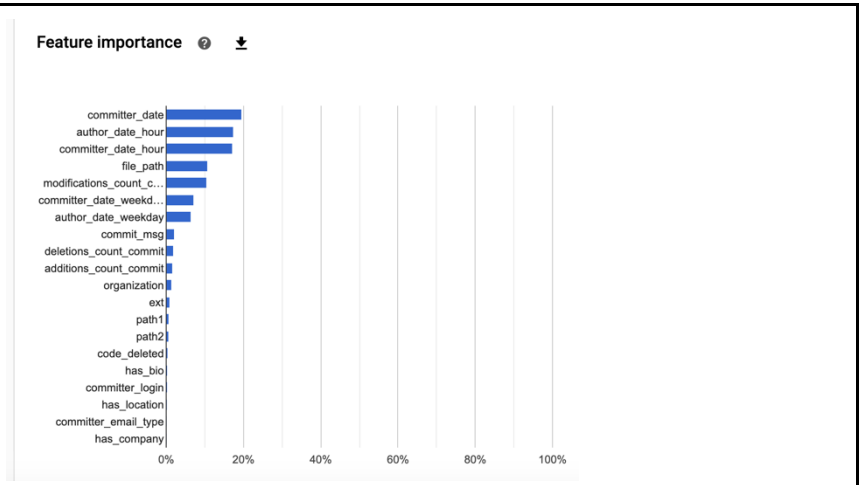
```
0.040213529  18 path2
0.03703775    9 committer_date_weekday
0.035848207  11 author_date_weekday
0.025067947   4 modifications_count_commit
0.024729462   5 additions_count_commit
0.022081905   6 deletions_count_commit
0.013254739   1 author_login
0.008327429   2 committer_login
0.005775394  19 ext
0.005615917  17 path1
0.005400034  16 organization
0.001542587   3 committer_email_type
0.001208503  14 has_bio
0.000033919  13 has_location
0.000029483  15 has_company
0.000000155  12 company
```

39: libtiff_mock

SciKit Feature Importance	AutoML Feature Importance

```
                              gain
feature
additions_count_commit      0.93852
deletions_count_commit      0.05787
committer_email_type        0.00098
modifications_count_commit  0.00090
committer_date_hour         0.00087
```

author_login	0.00086
committer_login	0.00001



Weka Classification and Feature Importance

```
Weka Decision Tree (J48):  Precision=0.99, Recall=1.0


        Information Gain Ranking Filter
Ranked attributes:
 0.253563    7 committer_date
 0.183307    8 committer_date_hour
 0.183307   10 author_date_hour
 0.121511   11 author_date_weekday
 0.121511    9 committer_date_weekday
 0.086407    4 modifications_count_commit
 0.042354   18 path2
 0.027185   17 path1
 0.026971   19 ext
 0.021536    5 additions_count_commit
```

```
0.018311   6 deletions_count_commit
0.015065  16 organization
0.000428   3 committer_email_type
0           2 committer_login
0          12 company
0          13 has_location
0          15 has_company
0          14 has_bio
0           1 author_login
```

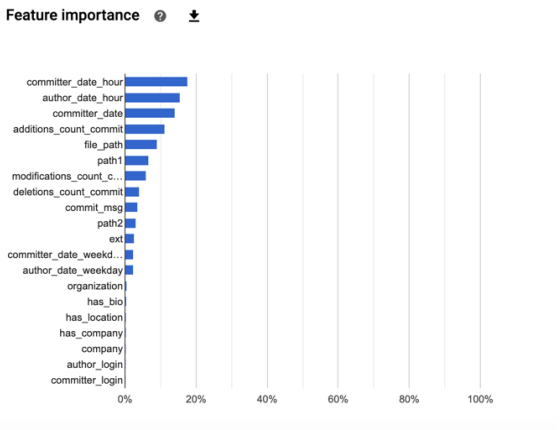
40: imagemagick_mock,

Fault 1;Y:N=158:15032; BL=0.01,

SciKit Feature Importance

```
[                              gain
 feature
 additions_count_commit      0.33887
 author_login                0.29072
 committer_email_type        0.06618
 path1                       0.05787
 path2                       0.04848
 committer_date_hour         0.04056
 author_date_hour            0.03568
 modifications_count_commit  0.03383
 deletions_count_commit      0.03179
 company                     0.01914
 organization                0.01656
 has_bio                     0.01001
 committer_login             0.00634
```

AutoML Feature Importance



committer_date_weekday 0.00204 author_date_weekday 0.00173 has_location 0.00032	

41: libming_mock

SciKit Feature Importance	AutoML Feature Importance
gain feature additions_count_commit 0.99591 modifications_count_commit 0.00409