Using Machine Learning Towards Early Flagging of Potentially Buggy Software Commits

Aradhana Soni
Department of Industrial and Systems Engineering
University of Tennessee
Knoxville, United States
asoni5@vols.utk.edu

Kalyan S. Perumalla
Department of Industrial and Systems Engineering
University of Tennessee
Knoxville, United States
kperuma3@utk.edu

***Abstract*—Detection and flagging software bugs is in general a difficult problem, especially in collaboratively developed software managed via large and dynamic software development systems. There are various levels at which this problem can be tackled, such as analysis at the level of source code or binaries. Here, we explore a different approach, namely, investigating the potential to tackle it at the level of *metadata* in software repositories and provide a problem formulation suitable for solution using machine learning. We build an implementation of the solution using multiple classifiers, and verify the feasibility of our approach on metadata from synthetic datasets modeled by a characterization of a few large software repositories and developer profiles. The results show that, with sufficient amount of past metadata for training, it is conceivable to flag (as potentially buggy or not) new software commits to repositories. Good accuracy is observed via inferencing on suitable classifiers trained on the metadata, although, on increasing data sizes, some saturation of the accuracy is observed. To further increase the efficiency, additional metadata (such as inter-entity linkages) and more complex machine learning techniques (such as graph neural networks) may be warranted to tap more latent information in the software evolution processes. Nevertheless, the metadata-based learning approach appears promising as an automatable service towards early flagging of potentially buggy commits in software repositories. Such flags can conceivably augment the features provided as services by software hosting companies or institutions with large software bases.**

***Index Terms*—vulnerabilities, characterization of commits, developers' experience, learning model**

I. Introduction

Analyses of software changes while the software evolves can help flag undesirable effects such as bugs and vulnerabilities. In this paper, we focus on metadata-based analyses that can provide early and online flagging of software changes as they are committed to software repositories. Specifically, we consider the potential offered by the inclusion of metadata about the repositories, commits, and developers in learning latent information to inform about potentially buggy commits. For example, how do the developers' historical traits affect the probability of committing bug? How do some of the features of software repositories correlate with the chance of some commits being buggy more than the others? Our study explores sets of features that can be used from software repositories to correlate and learn their contribution to buggy commits. We evaluate the usage of these features and the extent to which learning models can be effective. To the best of our knowledge, such a learning formulation has not been published in the literature.

A. Background and Related Work

Some methods using static and template-based techniques rely on the analysis of source code to develop vulnerability detection methods [1]–[3]. However, they suffer from several false positive alerts. There has also been an increase in the number of vulnerabilities disclosed after the release of software products, which suggests that current techniques require improvement in terms of efficiency and effectiveness [4]. Previous efforts have tried to document, track, and study the reported vulnerabilities [5], [6], while others aimed to classify the severity of the vulnerability using only the vulnerability description [7]. Some of these studies use the data from National Vulnerability Database [8]. An assumption is sometimes made that software security threats may not undergo rapid change, that they do not significantly evolve between analysis and use of analysis in detection [9]. In the case of bugs that are vulnerabilities, it is important to be able to identify the associated software commits as early as possible to be aware of them or address them. Although many analysis efforts deal with the software itself in question, not many studies have built analyses based on the conditions *outside the source code itself* that have correlations (causal or incidental) with certain commits being more prone to bugs than others.

Over the recent decades, there has also been a massive adoption of open source software based on online systems such as GitHub. This has made source code and development history of millions of software projects available to public [10]. The open source code comes with its own challenges. Automated analyses of these open software repositories for early bug detection is an open research challenge. Bugs representing exploitable vulnerabilities in software can pose potential threats to the secure operation of computer systems as proved by Heartbleed [11] and Shellshock [12]. As a simple illustration out of many, a vulnerability in Apache Struts in 2017 resulted in 143 million consumers' financial data to be compromised [13]. Such incidents highlight the importance of investing efforts to improve detection of potentially buggy software changes as early as possible.

A closely related body of research concerns the methods for detection of vulnerabilities in software. Suneja et al. [2] explore the applicability of graph neural network in learning the nuances of source code — whether signatures of vulnerabilities in source code can be learned from their graph representations and studying the relationships between nodes and edges. The idea behind this study is to present a large dataset already tagged as vulnerable or non-vulnerable to a learning-based model so that the model can figure out the properties which differentiate vulnerable code from healthy code. This is analogous to our approach we present here, except that theirs is focused on the *internals* of the code whereas ours uses metadata *external* to the code.

In a study by Rusell et al. [1], the authors use machine learning on C and C++ open-source code available to develop a large-scale, function-level vulnerability detection system. A team of security researchers mapped the categories determined by static analyzers to the corresponding common weakness enumerations (CWEs) and identified which CWEs would likely result in potential security vulnerabilities. This process generated binary labels of "vulnerable" and "not vulnerable," depending on the CWE. Jie et al. [14] and Yamaguchi et al. [15] used machine learning on the internal features of the software to detect vulnerabilities. They classify the machine learning-based vulnerability analysis methods into three categories: program analysis, extracting features and vulnerability knowledge. Yamaguchi et al. identify Application Programming Interface (API) symbols of each function through lexical analysis and embed API symbols in vector space, and apply principal component analysis (PCA) to find the usage of the API modes. Wijayasekara et al. [16] propose a method using text mining technology to mine potential vulnerabilities in public bug databases. They extract description information from open bug databases to identify through the feature vectors whether the bug is a normal bug or a vulnerability.

Gonzalez et al. [17] aim to characterize the vulnerabilities automatically to analyze Common Vulnerability and Exposure (CVE) reports and automatically infer their Vulnerability Description Ontology characteristics. The authors curated 365 vulnerability descriptions from the National Vulnerability Database (NVD), each mapped to 1 of 19 characteristics from the NIST Vulnerability Description Ontology. Han et al. [7] propose a deep learning approach to predict multi-class severity level of software vulnerability using only vulnerability description. The study uses words embeddings and one-layer shallow Convolutional Neural Network (CNN) to automatically capture discriminative word and sentence features of vulnerability descriptions for predicting vulnerability severity into one of the severity levels — critical, high, medium, low — of the Common Vulnerability Scoring System framework.

Another closely related body of research concerns methods for detecting plagiarism. The study by Viuginov et al. [18] uses a dataset of solutions for 62 different competitive programming tasks from service code forces. These tasks were collected from 10 real contests hosted on the platform. The total dataset consists of 80,000 programs on C++. The authors proposed a new feature set, which allowed to consider ACM solutions as feature vectors. Ullah et al. [19] propose a methodology for software plagiarism detection in multiprogramming languages including C, C++, Java, C#, and Python. The software plagiarism detection in multiprogramming languages is a major challenge because each language has different syntax and semantic structures. The authors remove noisy words using tokenization to convert the source codes into meaningful tokens. The authors use Principal Component Analysis for extracting features to convert high dimensional datasets into lower dimensional spaces without losing the actual information. Bandara et al. [20] believe that any algorithm that may be suitable for pattern recognition can be used for source code author identification. The authors mention three such algorithms, including Naïve Bayes Classifier, k-Nearest Neighbor Algorithm and AdaBoost Meta-learning Algorithm. Since not all source code metrics contribute equally for source code author identification, the authors identified nine metrics including line-length (number of characters in one source code line), line-words (number of words in one source code line), and underscores-calculator (number of underscore characters in identifiers).

A tangentially related area of work deals with "learning" to identify and distinguish among software repositories. Rokon et al. [21] identify similar repositories and their clusters in a large online archive, GitHub. They propose a comprehensive embedding approach, Repo2Vec, to represent a repository as a distributed vector by combining features from three types of information sources: (a) metadata, (b) the structure of the repository, and (c) the source code. The authors combine these information types into a single embedding using a series of embedding approaches. The methodology is evaluated with two real datasets from GitHub for a combined 1013 repositories. The authors create an embedding for each type of data: (i) meta2vec for metadata, (ii) source2vec for the source code, and (iii) struct2vec for the directory structure. An embedding vector is created for each of the three data types. Then, all of this is combined into a repository embedding to determine similarity.

B. Our Contributions

Our key contribution is to introduce a machine learning-based formulation for the problem of early flagging of potentially buggy software commits by multiple developers/coders in large software respositories. The promising use of this formulation is demonstrated via an implementation and testing on representative datasets. It fills a gap in the literature, namely, exploiting the information available in the form of metadata of software repositories and evaluating the extent to which they could be useful in classifying the commits as potentially buggy or not.

To aid in the experimental evaluation, we created a synthetic dataset with features including the characteristics of repositories to which commits are made, the characteristics of the commits themselves and the traits of the developers making

the commits. The essential aim of the learning models is to use the input features to classify commits as buggy or not.

II. Learning Problem Formulation

We formulate the learning problem as follows for a machine learning-based early flagging of potentially buggy commits. The system of interest consists of a set of software repositories $R = \{r_1, \ldots, r_{N_R}\}$ and a set of developers $D = \{d_1, \ldots, d_{N_D}\}$. Each repository r has a sequence of commits $C_r = [c_0, \ldots, c_{N_r}]$. Each commit c is defined by its timestamp t_c, developer d_c, and a set of files F_c. Each developer d has a set of traits such as age and years of experience. All commits are by default considered (labeled) as not-buggy. Over time, some subset of commits from the past are determined (and labeled) as buggy. At any time T, the system has the opportunity to learn from the data in a snapshot of R, C, and D of that time. The learning manifests as a classification problem – given the feature vectors of R, C, and D with training labels assigned to C, the problem is to accurately predict the labels on commits, coming after time T, as buggy or not-buggy.

Note that we use the term "buggy" in a generic way, which can be customized for different purposes based on the application. The essential nature in our initial, simplified formulation is that of a binary tagging scheme. For example, "buggy" versus "non-buggy" could be interpreted as "vulnerable" and "non-vulnerable," or as "correct" and "incorrect," and so on, without affecting our learning scheme. Although our experimentation has been on binary tagging, it could in principle be extended to multiple labels for other applications.

The basic assumption behind this approach is that an unknown set of processes drive the buggy versus non-buggy nature of the commits, and some information about the processes gets manifested and encoded in the overall aggregate phenomenon of the system made of R and D. If the unknown set of processes are essentially random, this learning problem essentially fails in learning anything meaningful and hence cannot do a good job of predicting the labels for the subsequent commits. On the other hand, if the underlying processes are preserved in some shape or form in the feature space formed by the union of R, C, and D, then the learning can be effective. Furthermore, the sophistication warranted in the learning model would depend on the linear and non-linear nature of the contribution by the features ultimately to the buggy versus non-buggy labels of the commits.

Here, we implement this formulation of the learning problem using what ultimately boils down to a binary classification problem after the data is appropriately collected, organized, and processed. In the absence of a large amount of accurately labeled data for our evaluation purposes, we experiment with the three scenarios just described: (1) random labeling to model the absence of effective correlation between the underlying processes and the final labels, (2) piece-wise linear contributions of the features to the probabilities of the final labels, and (3) complex, non-linear contributions of the features to the probabilities of the final labels.

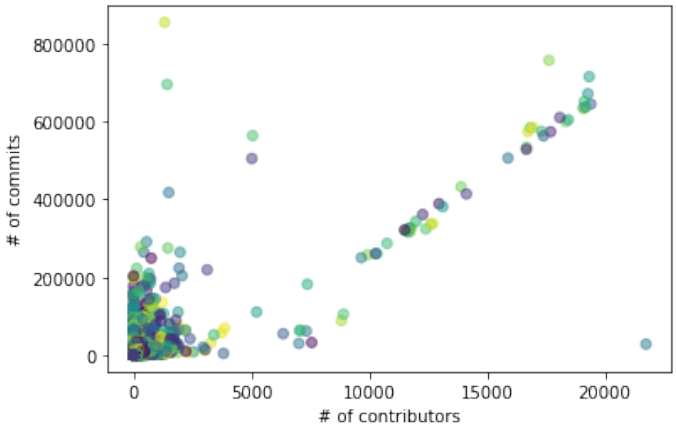

Fig. 1. Number of contributors vs number of commits

III. Data preparation

The datasets for R, C, and D are prepared based on empirically determined distributions from different, publicly available data sources. Note that although not all three elements of R, C, and D are necessarily from a single system, their composition provides a reasonable aggregate system for experimentation. A production version of our approach could clearly be built by interested parties with access to a consistent, large corpus of ground truth.

We used the metadata from a snapshot of GitHub containing 452 million real commits from 16 million GitHub repositories [22], and fit the data to probability density functions (PDF) that model the features. Fig. 1 shows the relation between number of contributors and number of commits in this dataset. For the developers' traits, we used the results of a similar PDF analysis from a Stack Overflow Developer Survey 2021 [23]. Using these sources, we created a synthetic dataset to explore the learning formulation. We initially started with a sample of 5000 commits, and steadily increased the sample size to compute the results at each step until we reached a sample size of 50,000 commits. To verify the appropriateness of our sample size, we used the Cochran's formula [24] and Slovin's formula [25] to calculate the appropriate sample size for our study. We assumed a total of 100 million repositories each with 500 commits. Our assumptions are inspired by the total number of repositories hosted by GitHub [26] and average number of commits in repositories with more than 100 commits in the 452 million metadata.

Since the goal is to predict whether the commit is buggy or not, we use a buggy indicator column with values 1 and 0 indicating the commit is buggy or otherwise, respectively. This dataset captures the characteristics of the commit including the repository to which the commit belongs and the traits of the developer committing.

A. Input Features

We used seven input feature vectors to feed into our learning algorithm. These features are listed next, along with the

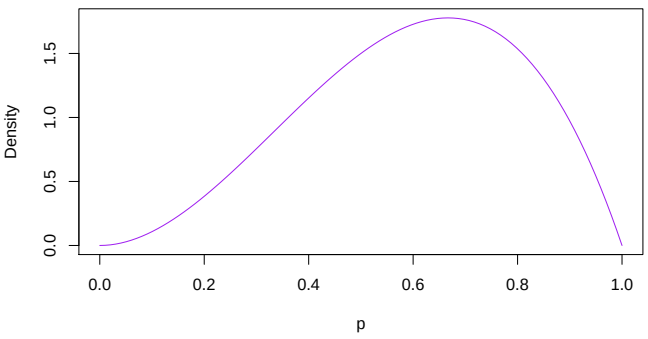

Fig. 2. Beta distribution used for sampling the ages of repositories

method we used in generating the column values. Each row of the data corresponds to a single commit.

1) Total number of commits: The value in this column corresponds to the total number of commits in the repository to which the commit in the record belongs. To create the values in this column, we used our analysis from the GitHub metadata mentioned previously. We characterized the number of commits and found that number of commits follow an exponential distribution with parameter $\lambda = 1.07$. Using this information, we generated this column by sampling the values from the exponential distribution with parameter value $\lambda = 1.07$. To determine the range of this column, we took a sample of 50,000 data points from the 452 million records and noted the maximum number of commits, ignoring the right outliers.
2) Total number of contributors: This column represents the total number of contributors to the repository to which the commit in the record belongs. The analysis of 452 million commits metadata showed that the distribution of the number of contributors is right skewed. Thus, for this column, we sampled the values using a right skewed Gamma distribution ($\alpha = 1$ and $\beta = 10.75$). We again used the sample of 50,000 data points from the 452 million commits to find the maximum number of contributors to be around 250, ignoring the right outliers.
3) Age of the repository (in weeks): This column represents the age of the repository to which the commit has been made. Since the number of repositories is generally increasing, we expect that the age of the repositories would be slightly skewed to the left. For this column, we sampled the values from a Beta distribution [27] (with parameters $\alpha = 3$ and $\beta = 2$). Fig. 2 show the PDF of this beta distribution. The age of repositories ranges from about 6 months to about 25 years.
4) Number of files in the commit: We explored a dataset of over 8000 Android applications [28]. Inspired by the results of this data, we set the number of files in a commit to be positively skewed by an Exponential distribution (we used $\lambda = 1.13$). We sampled the values of this column using exponential distribution and modeled it from one file to few dozen files per commit.
5) Number of types of file in the commit: Lacking prior data analysis on the number of types of files, we set the number of types of files to be distributed between 1 and 20.
6) Years of experience of developers: As mentioned in Section III, we used the raw data from the Stack Overflow Developer Survey 2021 and analyzed the distribution of the years of experience of the developers. The results indicated that years of experience follow an exponential distribution with $\lambda = 1.13$. We generated the values of this column by sampling from an exponential distribution with $\lambda = 1.13$, with the highest developer experience of about 44 years (note that this is consistent with the results of the stack overflow survey).
7) Time of the day the commit was made: We set this for a uniformly selected time across the day and night, accommodating the scenarios of late work nights and world-wide developer base.

B. Buggy Indicator Labels

As previously mentioned, to indicate whether a commit in our dataset is buggy, we introduce a buggy indicator which takes a value of 1 for vulnerable commits and 0 otherwise. We begin by introducing buggy indicators where no relation may be inferred between the indicator and the chance of individual variables contributing to the overall probability of making a buggy commit. We introduce two such sets of indicator labels.

- Random Buggy Indicator: To create this indicator, we uniformly distributed the labels of 0 or 1 for non-buggy and buggy commits respectively, with one-third of the commits set to be buggy.
- Logistic Buggy Indicator: For this scenario, we set a linear relationship between each feature variable and the logit of the baseline indicator variable. That is, we fit a logistic model using the random buggy indicator as the response variable and all the features as the dependent variables. Using the threshold of 40%, we predicted the binary values for the buggy indicator using this logistic model. We used these predicted values as the logistic buggy indicator for the binary classifiers we discuss in Section IV-A.

The essential consideration underlying the data is about how each input feature contributes to the overall chance of committing a bug. Therefore, an obvious next step is to be able to infer some natural relation between an overall probability of committing a bug and the chance of bug by each individual input features. That is, each input feature that we created in the dataset would contribute to the probability of whether a commit is buggy or not. For example, let us say that each contributor adds to the probability of a buggy commit according to a trend. We experimented with the trend that the more the number of contributors in a repository, the higher the probability that the commit may be buggy. Note that

our methodology remains unaffected even if this model must be changed to suit another dataset. Specific probabilities are assigned to each feature and these probabilities are averaged to determine the final probability that the commit is buggy. Below, the assignment of probabilities is discussed.

- Total number of contributors: As the number of contributors increases, the chance of committing a bug is higher. To calculate the probability of committing a bug due to the number of contributors, the number of contributors is normalized to a scale of 0 to 1. These normalized values are then used as the probability of committing a bug in association with this feature. This essentially determines a correlation between the number of contributors and probability of committing a bug.
- Total number of commits, Number of files in the commit, and Number of types of file in the commit: For these three features, we made the same assumption as for the number of contributors. That is, there exists a positive relation between the values of each of these features and probability of committing a bug. Again, each of these features is normalized to a scale of 0 and 1 for use as the probability of committing a bug contributed by each of this feature.
- Age of the repository (in weeks): The probability of committing a bug in relation to the age is modeled to increase at first with the age, then start to decrease and then again increase for older repositories. As we see in Fig. 3, the probability of committing a bug increases for repositories with age upto 10 years, the probability decreases for repositories with age between 10 to 20 years and then again increases for older repositories.
- Years of experience of developer: As the developer gains more experience, the probability of committing a bug is expected to reduce. This model is used to assign probability values based on the years of professional experience (see Fig. 4).
- Time of the day the commit was made: The probability of bugs are modeled to increase later in the day; during normal range of office hours (7 AM to 6 PM), the probability of committing a bug is set lower than the probability during late night hours (11 PM to 5 AM), with probabilities for the rest of the day between these two extremes (see Table I).

TABLE I
PROBABILITY OF COMMITTING A BUG VARIED BY TIME OF THE DAY

Time	Probability of bug
Office hours	Randomly between 1% to 10%
Late night hours	Randomly between 20% to 50%
Rest of the day	Randomly between 10% to 30%

To calculate the final probability of committing a bug, that would incorporate the probabilities we assigned to each individual feature, the average of all the seven individual probabilities associated with each feature is calculated. This is then used to estimate the probability-based buggy indicator by comparing with a random value uniformly distributed between 0 and 1.

Again, we note that other feature models and their contributions to the buggy labels can be varied for another dataset without affecting our basic formulation and classification approach. Here, we evaluate the approach with the aforementioned empirically-based features as representative of the possible target systems where this learning-based flagging can be applied.

Fig. 3. Probability of committing based on age of the repository

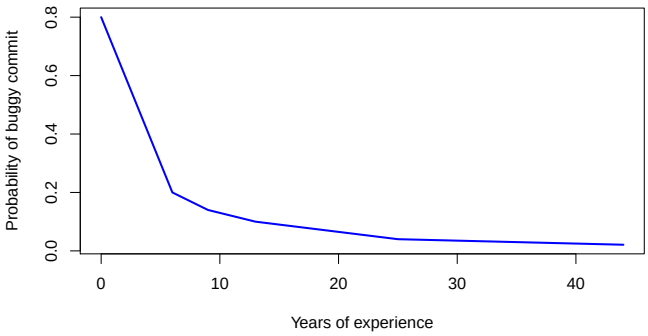

Fig. 4. Probability of committing bug based on years of experience of developer

IV. EVALUATION METHODOLOGY

A. Binary Classifiers

We use four different binary classifiers to identify the buggy commits: (1) Random Forest (RF) [29], (2) TabNet [30], (3) XGBoost [31], and (4) Light Gradient Boosting Machine (LightGBM) [32]. Here we briefly discuss the importance of these classifiers:

- RF is generally highly robust against overfitting. RF also helps in ranking feature importance per the total decrease in the Gini measure of node impurities [33]. For the

RF model, based on preliminary results using out-of-bag (OOB) error, 100 trees were included in the model.

- TabNet was introduced by Google in 2019 [30]. Their paper claims that this neural network outperforms the leading tree-based models across a variety of benchmarks. TabNet uses sequential attention to choose which features to reason from at each decision step. Thus it is considerably more explainable and interpretable than boosted tree models as it has built-in explainability. It is also claimed that TabNet is more efficient as the learning capacity is used for the most salient features. We fit our model using the default parameters with maximum epochs of 200, patience of 15, batch size of 1024 and virtual batch size of 128.
- XGBoost is an efficient and scalable implementation of gradient boosting framework by Friedman [34]. Gradient boosting is a popular and effective approach to classification. It produces a prediction model in an ensemble of weak models, typically decision trees [35].
- GBM is a gradient boosting framework based on decision trees that increases the efficiency of the model. It uses two novel techniques including Gradient-based One Side Sampling and Exclusive Feature Bundling which overcomes the limitations of histogram-based algorithm that is primarily used in all Gradient Boosting Decision Tree frameworks.

Generally speaking, it has been observed in our experiments that RF and LightGBM produced similar results and seem to do better than the other two classifiers for most scenarios.

B. Evaluation Metrics

To evaluate the performance of all the binary classifiers across all the models, we calculate the three standard evaluation metrics including accuracy, F1 Score and area under the curve (AUC). We used the confusion matrix (Table II) to calculate these metrics. For completeness, the definitions we use for these terms are provided next.

- Accuracy: Accuracy is defined as the ratio of correct predictions over total predictions.

$$Accuracy = \frac{a+d}{N}$$

- F1 Score: It is the harmonic mean of precision and recall.

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

  - Precision: Precision is the proportion of positive predictions that are correct.

$$Precision = \frac{a}{a+b}$$

  - Recall: Recall is the proportion of positive predictions that are correctly classified. That is, it measures the effectiveness of a classifier to identify positive labels.

$$Recall = \frac{a}{a+c}$$

- Area under the curve (AUC): AUC is the value that reflects the overall ranking performance of a classifier [36].

TABLE II
CONFUSION MATRIX

		Actual Labels: Positive	Actual Labels: Negative	Total
Predicted Labels	Positive	a	b	$a+b$
	Negative	c	d	$c+d$
	Total	$a+c$	$b+d$	N

V. EXPERIMENTS VARYING THE RELATION OF FEATURES TO BUGS

A. Random Relation of Features to Bugs

Here, we explore the case when there exists a random relation between the input features and the buggy labels. That is, we fit all four binary classification models to predict the random buggy indicator (section III-B) using the seven input features as described in section III-A on 50,000 commits. We calculated the evaluation metrics discussed in section IV-B and present them in the Table III. As we see here, the accuracy is about 63% for all the four classifiers. The binary classifier for the random buggy indicator was assigned randomly so we did not expect the models to do any better than 50%. But it seems like the classifiers did identify some pattern and were able to predict the labels correctly with more than an average chance.

We used feature ranking capability of the RF classifier to rank the features based on their importances. See Fig. 5 for ranking the feature importances using the Gini measure. The most important feature using this measure appears to be the time of the day at which the commit was made, followed by number of commits, age of the repository and others.

TABLE III
RESULTS FROM RANDOM RELATION OF FEATURES TO BUGS

	RF	TabNet	XGBoost	LightGBM
Accuracy	63.00%	62.73%	62.28%	63.00%
F1 Score	29.47%	41.63%	28.07%	28.42%
AUC	54.16%	57.09%	53.13%	53.69%

B. Logistic Relation of Features to Bugs

Next, we fit the classifiers using the logistic buggy indicator label as the dependent variable, using the seven features as inputs. The results are presented in Table IV. Not surprisingly, all the classifiers seem to work almost perfectly in this scenario. This is expected because we created the buggy indicator to follow a linear relationship between each feature variable and the logit of the buggy indicator. All our classifiers are able to identify this relation and make nearly perfect predictions. This model serves the role of a sanity check on the working of the classifiers and confirms the correct working of our model.

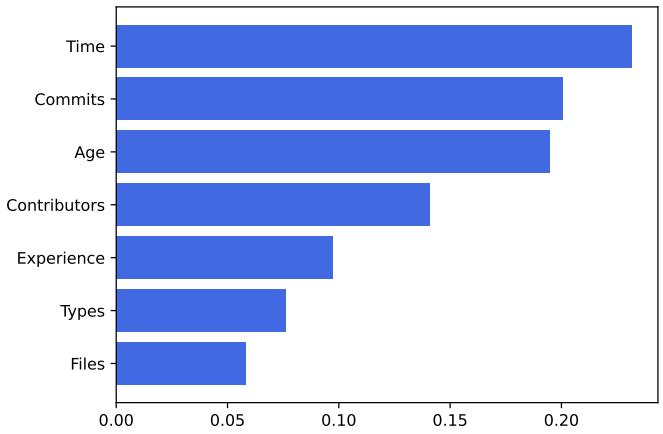

Fig. 5. Ranking of features by importance in the base model

TABLE IV
RESULTS FROM LOGISTIC RELATION OF FEATURES TO BUGS

	RF	TabNet	XGBoost	LightGBM
Accuracy	99.45%	99.73%	99.55%	99.59%
F1 Score	99.19%	99.61%	99.33%	99.39%
AUC	99.35%	99.64%	99.39%	99.46%

C. Probability-based Relation of Features to Bugs

In the more complex model encoding a more complex dependency of the buggy indicator on the features, we fit the binary classifiers on the probability-based buggy indicator that was created using the individual probabilities of committing bug of the input features (as described in section III-B). We again used all the seven features as inputs from section III-A and used the same sample of 50,000 commits. The values of the three evaluations can be found in Table V. The three classifiers – RF, TabNet and LightGBM – predicted the labels with an accuracy of over 75%. Although the accuracy for TabNet is the lowest among the four models, this model outperforms the other three in terms of F1 Score and AUC.

RF and LightGBM appear to produce similar results for this model. Another important observation here is that the F1 Score seems very low across all four classifiers. We investigated this further and found that although precision outcomes for three of the models are pretty high (over 98% for RF, XBGoost and LightGBM), the values for recall are as low as 0.4% for LightGBM. Because F1 Score is a harmonic mean of these two metrics, the recall value is pulling down the F1 Score for all classifiers. Further investigation of this phenomenon remains as part of future work.

We again used RF to find the importances of input features and ranked them. Refer to Fig. 6 for the ranking and importances using the Gini measure. The ranking of features is similar to that of the base case apart from the first two features which are reversed in order here.

TABLE V
RESULTS FROM PROBABILITY-BASED RELATION OF FEATURES TO BUGS

	RF	TabNet	XGBoost	LightGBM
Accuracy	76.06%	59.15%	75.87%	76.34%
F1 Score	4.47%	37.34%	5.53%	0.8%
AUC	50.34%	56.57%	50.65%	50.07%

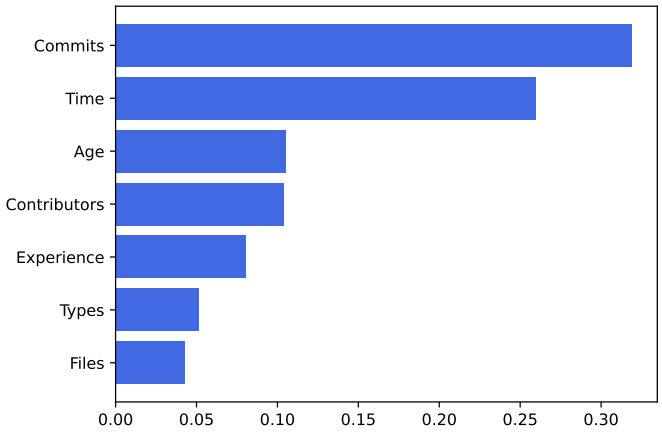

Fig. 6. Ranking of features by importance in the probablity-based model

D. Including the Time of Commit in Features

Next, we include the probability of bug due to the time of the day as an input feature to the probability-based model. Recall that the final probability of committing a bug is computed as the average of probabilities contributed by each of the individual features (see Section III-B). By design, the probability contributed by each feature, except the commit time, is (step-wise) linearly related to that input feature. For the commit time, we introduce a variant in the model by which a non-linear relation is defined between the probability contributed by the feature and the final bug probability. The thinking behind this experiment is to explore if the classifiers are able to identify the non-linear relation thus introduced. The results from this variation can be found in Table VI.

As we observe here that all the classifiers appear to deliver a higher quality of classification than in the case where the probability contributed by the commit time is not considered as an input feature (Table V).

TABLE VI
RESULTS FROM INCLUDING THE COMMIT TIME

	RF	TabNet	XGBoost	LightGBM
Accuracy	88.21%	80.71%	88.11%	88.33%
F1 Score	67.37%	61.95%	67.91%	67.94%
AUC	75.75%	75.86%	76.12%	75.93%

E. Excluding the Time of Commit from Features

As we observed earlier, including the probability contributed by the commit time as an input feature improved the results

significantly. This raises an obvious question about the significance of this feature. To explore this, we experiment with another scenario by dropping this feature altogether from the input features and fit the binary classifiers on the remaining six features. The results of this experiment can be seen in Table VII. The results do not appreciably change from the results of the model whose results are shown in Table V which includes this feature.

TABLE VII
RESULTS FROM DROPPING THE COMMIT TIME

	RF	TabNet	XGBoost	LightGBM
Accuracy	73.64%	59.6%	74.03%	74.51%
F1 Score	9.66%	39.76%	7.85%	3.04%
AUC	50.72%	57.27%	51.04%	50.44%

VI. EXPERIMENTS VARYING DATA SIZES AND DATASETS

A. Varying Data Sizes

The consolidated results of the final model are presented here and the variation of results is analyzed with changes in the sample size. We fit the binary classifiers using a series of increasing sample sizes of 5000 commits, 10,000 commits, 25,000 commits and finally 50,000 commits. The results are presented in Table VIII and Fig. 7. We observe here that the results for the three classifiers RF, XGBoost and LightGBM does not seem to change much across all sample sizes. But we do see the metrics varying significantly for TabNet. We would not worry much about this since, as we pointed before, accuracy for TabNet seems really low throughout when compared with the other classifiers although this classifier seems to outperform the others in terms of F1 Score and AUC.

TABLE VIII
RESULTS FROM VARYING DATA SIZES

	Datasize	RF	TabNet	XGBoost	LightGBM
Accuracy	5,000	77.00%	59.60%	74.67%	76.00%
	10,000	76.35%	72.33%	73.13%	75.07%
	25,000	76.67%	60.91%	75.65%	76.13%
	50,000	76.06%	59.15%	75.87%	76.34%
F1 Score	5,000	6.50%	28.03%	10.38%	4.26%
	10,000	2.47%	20.35%	7.78%	2.60%
	25,000	5.04%	35.70%	8.43%	3.66%
	50,000	4.47%	37.34%	5.53%	0.8%
AUC	5,000	50.94%	50.72%	50.69%	50.12%
	10,000	50.13%	52.59%	49.66%	49.78%
	25,000	50.59%	55.83%	51.08%	50.41%
	50,000	50.34%	56.57%	50.65%	50.07%

B. Using Features from Samples of Real Metadata

Taking a random sample of 50,000 records from the metadata of 452 million commits where the number of commits are between 100 and 50,000. Our exercise was to sample the records of repositories each of which has at least 100 commits

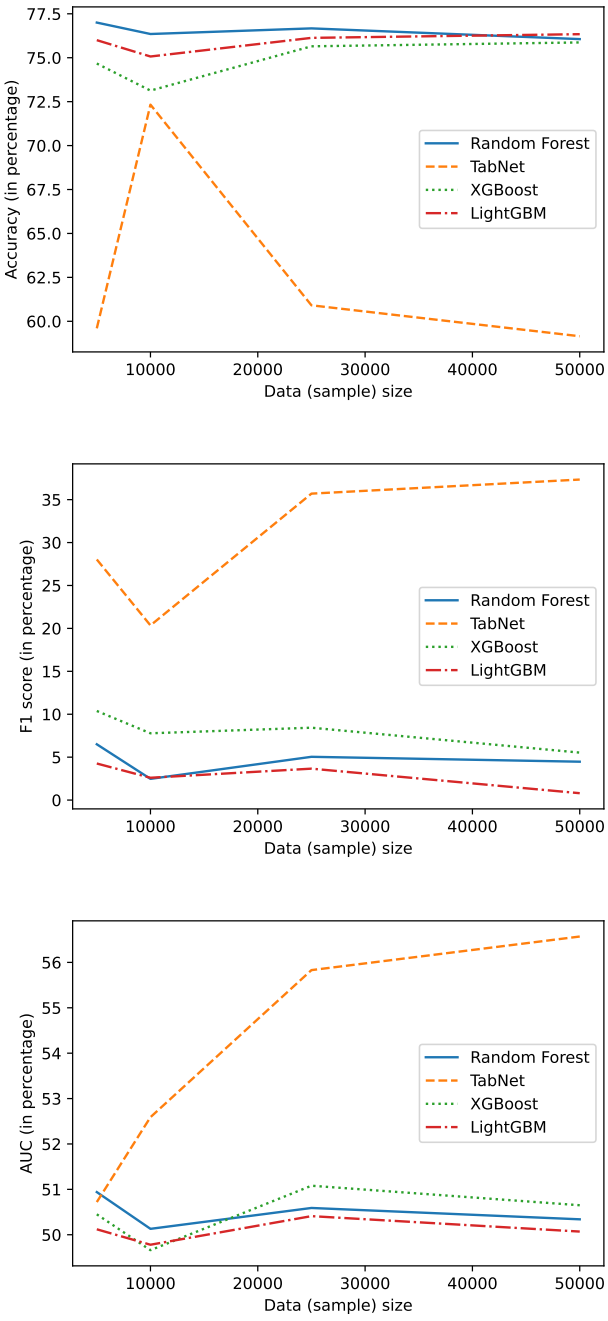

Fig. 7. Accuracy (above), F1 Score (center) and AUC (below) results are shown for varying sample sizes. Note that the results for RF, XGBoost and LightGBM do not seem to change significantly across different sample sizes.

(since the repositories with less than 100 commits represent very low activity). Also, we limited the maximum number of commits as 50,000 similar to our synthetic data.

We calculated the probability of committing a bug using the methodology we adopted for probability-based buggy indicator defined in section III-B. However, this time we restricted the feature vectors to the number of commits and number of contributors. The other features are not included as input for computing the buggy indicator for this experiment. We fit the four models as described in section IV-A using the number of commits and number of contributors as input features and the buggy indicator as the output label.

Table IX show the results for the computed metrics for all four models. We see here that all models are able to predict the buggy labels with high accuracy. Although the accuracy for TabNet is the lowest among the four models, it is still fairly good and ouperforms the other three in terms of F1 Score and AUC. We investigated the F1 Score levels further and the results seem similar as before, that is, the numbers for precision for most the models are quite high (over 93% for RF, XGBoost and LightGBM), but the recall values are as low as 43% for LightGBM.

TABLE IX
CLASSIFICATION RESULTS WITH 50,000 SAMPLES OF COMMIT METADATA

	RF	TabNet	XGBoost	LightGBM
Accuracy	83.64%	77.97%	85.15%	85.29%
F1 Score	57.26%	60.10%	57.15%	56.76%
AUC	71.58%	76.75%	70.65%	70.33%

VII. CONCLUSION

In software evolution, detecting potentially buggy commits to repositories can help mitigate many types of problems. Broadly speaking, the methods for such detection can be based on analyses of the software *internals* such as source code. In contrast to such methods that rely on internals, we proposed an approach that exploits features that are *external* to the actual software. Specifically, we focused on exploiting the metadata associated with the repositories, commits, and developers that are part of the software evolution system. A learning formulation of the detection problem is presented along with a simplified machine learning solution based on binary classification. We evaluated the potential for learning using this approach and found that it is conceivable to get good performance. Our approach has been implemented in software and the results from experiments using this implementation are encouraging. Using samples from metadata of a large software repository warehouse, as well as from synthetic data generated to mimic the distributions of real data from repositories, commits, and developer information, we verified that it is feasible to train and test the system. Overall, the results point to the possibility of using machine learning for early flagging of potentially buggy commits to software repositories. This can be useful in various ways, such as a service offered by the hosting institutions to software development communities. Other uses include expansion of the binary model to more labels and customization beyond early detection of bugs or vulnerabilities at their very origins.

ACKNOWLEDGMENT

Elided for double-blind review.

REFERENCES

[1] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[2] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," *arXiv preprint arXiv:2006.08614*, 2020.

[3] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.

[4] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, "Hackers vs. testers: A comparison of software vulnerability discovery processes," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 374–391.

[5] N. Bhatt, A. Anand, D. Aggrawal, and O. H. Alhazmi, *Categorization of Vulnerabilities in a Software*. CRC Press, Boca Raton, FL, 2018.

[6] R. A. Gandhi, H. Siy, and Y. Wu, "Studying software vulnerabilities," *Crosstalk: The Journal of Defense Software Engineering*, pp. 16–20, 2010.

[7] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 125–136.

[8] H. Booth, D. Rike, and G. Witte, "The national vulnerability database (nvd): Overview," 2013-12-18 2013. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=915172

[9] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *CoRR*, vol. abs/1801.01681, 2018. [Online]. Available: http://arxiv.org/abs/1801.01681

[10] E. S. Raymond and T. O'Reilly, *The Cathedral and the Bazaar*, 1st ed. USA: O'Reilly & Associates, Inc., 1999.

[11] "The heartbleed bug," Jun 2020. [Online]. Available: https://heartbleed.com/

[12] "Shellshock: All you need to know about the bash bug vulnerability," September 2014. [Online]. Available: https://www.broadcom.com/products/cyber-security

[13] B. Dynkin and B. Dynkin, "Derivative liability in the wake of a cyber attack," *Alb. LJ Sci. & Tech.*, vol. 28, p. 23, 2017.

[14] G. Jie, K. Xiao-Hui, and L. Qiang, "Survey on software vulnerability analysis method based on machine learning," in *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*. IEEE, 2016, pp. 642–647.

[15] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX conference on Offensive technologies*, 2011, pp. 13–13.

[16] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *2012 5th International conference on human system interactions*. IEEE, 2012, pp. 89–96.

[17] D. Gonzalez, H. Hastings, and M. Mirakhorli, "Automated characterization of software vulnerabilities," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 135–139.

[18] N. Viuginov, P. Grachev, and A. Filchenkov, "A machine learning based plagiarism detection in source code," in *2020 3rd International Conference on Algorithms, Computing and Artificial Intelligence*, 2020, pp. 1–6.

[19] F. Ullah, J. Wang, M. Farhan, M. Habib, and S. Khalid, "Software plagiarism detection in multiprogramming languages using machine learning approach," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 4, p. e5000, 2021.

[20] U. Bandara and G. Wijayarathna, "A machine learning based tool for source code plagiarism detection," *International Journal of Machine Learning and Computing*, vol. 1, no. 4, p. 337, 2011.

[21] M. O. F. Rokon, P. Yan, R. Islam, and M. Faloutsos, "Repo2vec: A comprehensive embedding approach for determining repository similarity," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 355–365.

[22] M. Vadim, "452m commits on github - dataset by vmarkovtsev," May 2017. [Online]. Available: https://data.world/vmarkovtsev/452-m-commits-on-github

[23] S. Overflow. Stack Overflow developer survey 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021/

[24] W. G. Cochran, *Sampling Techniques*, 3rd ed. New York: John Wiley & Sons, 1977.

[25] T. Yamane, *Statistics: An Introductory Analysis*, 2nd ed. New York: Harper and Row, 1967.

[26] S. Horawalavithana, A. Bhattacharjee, R. Liu, N. Choudhury, L. O. Hall, and A. Iamnitchi, "Mentions of security vulnerabilities on reddit, twitter and github," in *IEEE/WIC/ACM International Conference on Web Intelligence*, 2019, pp. 200–207.

[27] A. K. Gupta, *Beta Distribution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 144–145. [Online]. Available: https://doi.org/10.1007/978-3-642-04898-2_144

[28] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli, "A graph-based dataset of commit history of real-world android apps," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 30–33.

[29] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1. IEEE, 1995, pp. 278–282.

[30] S. O. Arık and T. Pfister, "Tabnet: Attentive interpretable tabular learning," in *AAAI*, vol. 35, 2021, pp. 6679–6687.

[31] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[32] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, pp. 3146–3154, 2017.

[33] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to data mining*. Pearson Education India, 2016.

[34] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)," *The annals of statistics*, vol. 28, no. 2, pp. 337–407, 2000.

[35] L. Zhang and C. Zhan, "Machine learning in rock facies classification: an application of xgboost," in *International Geophysical Conference, Qingdao, China, 17-20 April 2017*. Society of Exploration Geophysicists and Chinese Petroleum Society, 2017, pp. 1371–1374.

[36] M. Hossin and M. Sulaiman, "A review on evaluation metrics for data classification evaluations," *International Journal of Data Mining & Knowledge Management Process*, vol. 5, no. 2, p. 1, 2015.