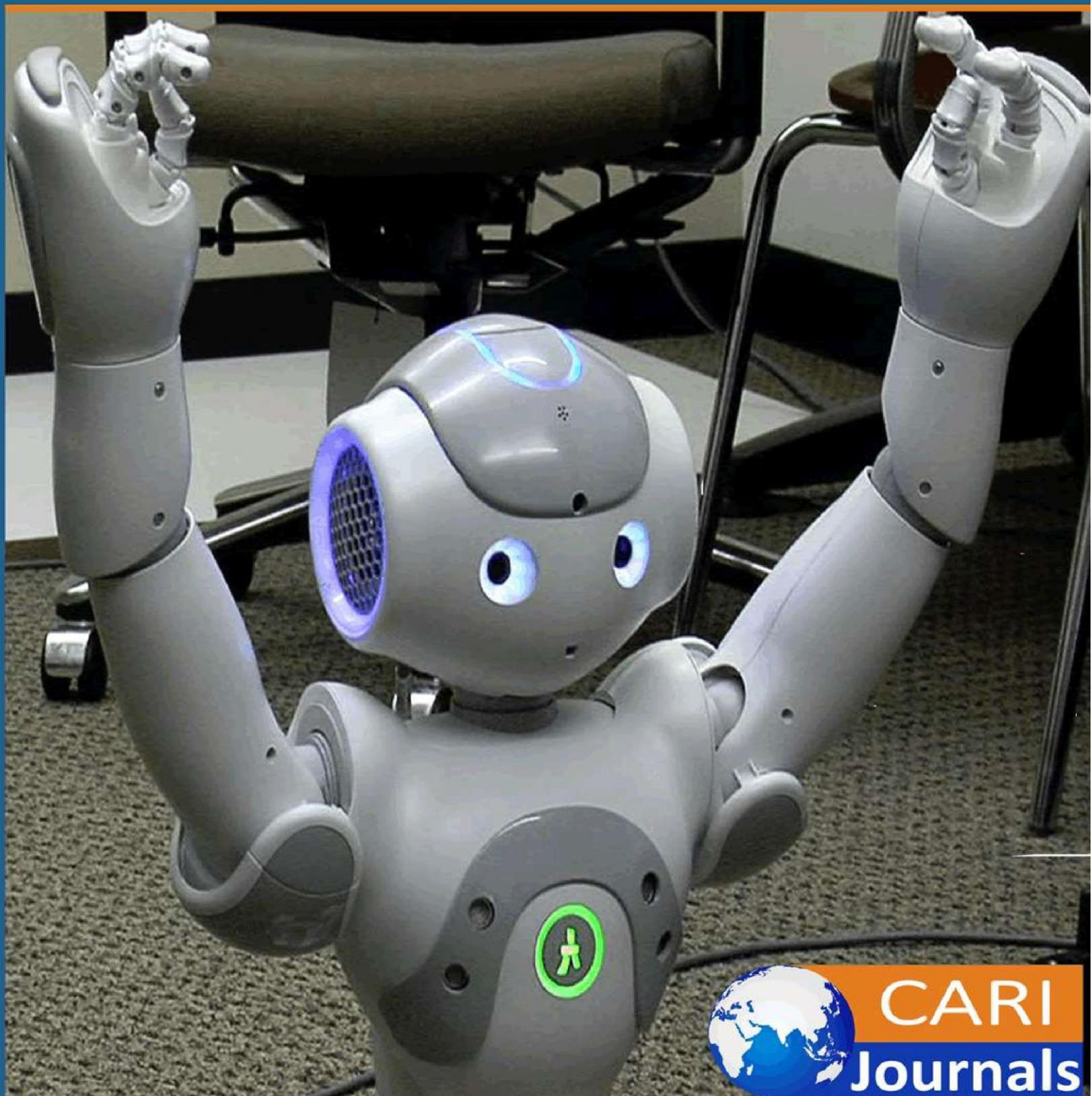
International Journal of
Computing and
Engineering
(IJCE)

Enhancing Software Vulnerability Prediction Models



CARI Journals

International Journal of Computing and Engineering
ISSN 2958-7425 (online)
Vol. 6, Issue No. 5, pp. 10 - 24, 2024
www.carijournals.org

Enhancing Software Vulnerability Prediction Models

Santosh Kumar Kande
https://orcid.org/0009-0000-9086-2015

Accepted: 18th Aug, 2024, Received in Revised Form: 26th Aug, 2024, Published: 18th Sep, 2024

Abstract:

Purpose: The purpose of this study is to evaluate and replicate various Vulnerability Prediction Models (VPMs) to determine their effectiveness in identifying software vulnerabilities. Given the increasing complexity of software, identifying vulnerabilities during development is becoming more challenging. This study aims to enhance the accuracy of vulnerability prediction to improve security inspections and testing.

Methodology: The study involves benchmarking different VPM approaches, including software metrics, text mining, and automated static analysis. These models are evaluated using a dataset that consists of over 100,000 lines of code from multiple open-source projects. The evaluation focuses on assessing the models in terms of precision, recall, and F-Measure.

Findings: The findings indicate that combining multiple VPM techniques results in improved prediction accuracy. The study demonstrates that integrating various approaches enhances the overall effectiveness of vulnerability detection in software development.

Unique Contribution to Theory, Practice, and Policy (Recommendations): The unique contribution of this study lies in its demonstration that a multi-technique approach to VPMs can significantly enhance prediction accuracy. This finding offers valuable insights for both theoretical advancements and practical applications in software security. For practice, it suggests that incorporating a combination of VPM techniques can lead to more effective vulnerability detection. For policy, it underscores the importance of adopting advanced and varied VPM methods to improve software security measures. Future research should focus on expanding datasets to include a broader range of projects and incorporating machine learning techniques to further enhance VPM predictive capabilities.

Keywords: *Vulnerability, Software Engineering, Prediction Models, Metrics, Security, Software, Open Source, VPM, Precision, Recall, Accuracy, Benchmark.*

1 INTRODUCTION

A product weakness is a shortcoming in a product framework that an assailant can exploit to change its way of behaving [1]. As the number of programming frameworks increases daily, so does the number of weaknesses. An assailant can utilize weaknesses to get to the framework, and he could take control to harm it, by sending off new assaults or acquiring some favored data that he can use for their advantage. Considering this, it means quite a bit to know the various sorts of weaknesses, their counteraction, and their location to attempt to keep away from their presence in the last programming rendition of the framework and afterward decrease the chance of assaults and expensive harms. Finding this sort of issue can be valuable for assessing and examining the source code parts more completely. A manual source code review requires human exertion in terms of parts/time because tracking down such weaknesses can be troublesome [1].

Weakness expectation models (VPM) are accepted to guarantee that they give computer programmers direction on where to focus on valuable confirmation assets to look for weaknesses [3]. There are various kinds of VPMs, each with various ways of getting source code data to assemble VPMs that cover various parts of programming. Finding them grants finding the primary highlights to perceive in the event that a part is defenseless or not. In this review, I have recreated the VPMs made by Shin et al. [6] for programming measurements, Scandariato et al. [7] for text mining, and Gegick et al. [2] for the Computerized Static Examination Code (ASA) with a dataset that remembers a few open-source projects for request to test the VPMs with the heterogeneity of the information. Every methodology can be joined between them to develop the procedure's accuracy further.

With this, I propose the following Research questions:

- **RQ1**: Which are the best VPM analyzed?
- **RQ2**: Which classifier obtains a higher error rate and which one a higher accuracy rate?
- **RQ3**: Is it possible to combine different VPMs? In respect of the state-of-art, are they more effective?

The goal of the study is to build upon the contributions of previous research by creating and refining the replication of state-of-the-art Vulnerability Prediction Models (VPMs). The study aims to compare these models to better understand the effectiveness of the replications in discovering vulnerabilities.

The rest of the paper is organized as follows: Section 2 discusses background and related work, Section 3 describes the implementation for each VPM, Section 4 shows the obtained results, and Section 5 describes limitations and future works.

2 Related Work

In this part, the researcher portrays the connected work to my comparison.

A weakness is a mistake in the particular improvement or design of programming that permits the security strategy (verifiable or unequivocal) remembered for an organization's all's product to disregard its execution [6][10]. A shortcoming of the framework permits an outside aggressor to lessen the security of data in the framework. An adventure means to exploit something for one's end, particularly deceptively or ridiculously. An endeavor is a piece of programming that exploits a bug or weakness to make the accidental way of behaving happen on PC programming or equipment. Each weakness has a lifecycle as follows:

1. **Discovery**: when a vulnerability is discovered by a seller, a hacker or others.
2. **Disclosure**: the vulnerability is publicly disclosed; that is, everyone will be informed about the vulnerability.
3. **Exploitation**: an external attacker is capable of using the vulnerability to cause problems.
4. **Patching**: when a vendor resolves the vulnerability.

Normal Weaknesses and Openness (CVE) is the true standard word reference that gives definitions to freely unveiled online protection weaknesses and openings. Presently, the Miter Company keeps up with CVE: this is a philanthropic association that works innovative work habitats subsidized by all significant US bureaucratic governments. CVE expects to normalize the names of all freely known weaknesses and security openings. The objective of CVE is to make it simpler to share information across discrete weak data sets and security tools. Specialists have utilized programming properties to foresee weak code inside programming projects. Theisen et al. [3] performed VPM replication on Mozilla Firefox with 28,750 source code records, including 271 weaknesses utilizing programming measurements, text mining, and crash information. Making a blend of elements from each VPM lastly obtain results from the classifiers reruns.

Zimmermann et al. [1] fostered a weak expectation model for Windows Vista in view of customary computer programmers' measurements, for example, code stir and the number of designers. Gegick et al.[2] fabricated a forecast model utilizing the consequences of the Mechanized Static Investigation instrument "Flexelint" and found that, related at specific programming measurements, it tends to be utilized to foresee powerless components.

Scandariato et al.[7] introduced a VPM in light of AI utilizing the text mining approach, so dissecting straightforwardly the source code rather than programming or designer measurements. They utilized an exploratory approval of 20 Android applications and found that the text mining approach could get an expectation power that is equivalent or better in connection than other techniques. Shin et al. [6] investigated whether issue expectation frameworks could be adjusted for weakness forecast and found that shortcoming expectation performs in basically the same manner to specific weakness indicators. Papadakis et al. [8] directed an examination in view of the replication and correlation of three VPMs with regards to Linux Piece: import and capability calls, programming measurements, and text mining. They found that text mining is the best method while focusing on arbitrary occurrences.

3 Vulnerability Prediction Models

In this segment, the researcher depicts the three VPMs that the researcher will analyze in this study.

3.1 Software Measurements – Stir and Complexity

Shin et al. [6] use stir and intricacy highlights to find parts inside the venture's codebase that are probably going to be helpless. Shin et al. had the option to decrease how much code was reviewed for security exertion by 71% for Mozilla Firefox; they report 12% accuracy and 83% review utilizing comparable measurements. Shin et al. investigated execution intricacy measurements versus static intricacy measurements and found that execution intricacy measurements beat their partner for Mozilla Firefox and Wireshark concerning Record Assessment Decrease [].

3.2 Text Mining

Scandariato et al. [7] beginning from some Java documents that incorporate remarks (in-line remarks and block remarks). Every Java record is tokenized into a vector of terms, likewise called" monograms", and the recurrence of each term in the document is counted. The frequencies are not standardized to the length of the document because of conceivable disintegration of execution. The routine utilized for the tokenization utilizes a bunch of delimiters that incorporate blank area, Java accentuation characters, and both numerical rationale administrators. Scandariato and Walden [7] [9] contrasted text mining and programming measurement ways to deal with weakness expectation. They found that text mining tokens brought about better accuracy and review for weakness expectations for three undertakings: Drupal, PHPmy Administrator, and Moodle. Later investigations on text mining utilized creator-approved weakness set [9]. Moreover, text mining gets some margin to run and has a huge plate space prerequisite. The time taken to tokenize code into highlights is of worry, as it implies that a text mining way to deal with weakness forecast would be contrary with a consistent sending work process.

3.3 Automated Static Analysis

Gegick et al. [2] used Automated Static analysis(ASA) tool results as one of the metrics. A static analysis tool is used to analyze the software code to find defects without executing the code. The output of an ASA tool is an alert, a notification of a potential fault identified in the source code. This technique provides an early, automated, and repeatable analysis to detect faults, but it also provides a high false positive alert. Gegick discovered that ASA results have no resolution in determining if a component is vulnerable, but combined with other techniques, it can provide better results.

4 Dataset

In this section, the researcher discusses the dataset for this experiment.

Sabetta et al. [10] created a dataset of vulnerabilities of open-source software, collecting the vulnerabilities from the National Vulnerability Database (NVD) and from the project's Web resources that the authors monitored on a continuous basis. The dataset consists of a set of 4-tuples:

(vulnerability_id, repositopry_id, commit_hash, class)

- **vulnerability id**: is the identifier of a vulnerability that is fixed in the commit.
- **repository url**: URL of the repository
- **commit hash**: hash value that identifies the commit.
- **class**: this value establishes if a commit fixes a vulnerability (has ”pos” value) or it’s likely not vulnerable (has ”neg”).

The dataset covers 205 distinct Java projects and includes 1282 unique commits corresponding to the fixes to 624 vulnerabilities. In addition, this dataset is different from other ones. Sabetta et al. include vulnerabilities that are not available on the NVD database in the dataset. There are 29 commits with no CVE identifier and 46 vulnerabilities that have been given a CVE identifier by a CVE numbering authority but are not yet published on NVD.

5 Methodology

In this section, the researcher describes the methodology for the replication and the comparison of VPM.

5.1 Mining Software Repository

For the extraction of the components containing vulnerability from the dataset, I used Pydriller [11] [12] to obtain information regarding the repositories contained in the dataset.

The dataset given by Sabetta et al. [10] contains a hash value that identifies the *commit_fix*, and it permits us to reach the modified classes considering the BFIM (before image) and the classes that are introduced in that commit. In MSR theory, the BFIM is defined as the instant in which a file is modified or created before the commit fix. Therefore, I used the above image to obtain the instance containing the vulnerability. My data extraction processing considers only Java projects because this filter is already done in the dataset’s context. After that, I collected and organized in specific folders the several repositories and their commits.

However, there have been some problems with using pydrillers, including incomplete hash commits, a non-existent repository URL, and memory overhead during the MSR phase. The incomplete hash was handled manually, verifying the existence of the repository via web browser - since the latter requires a minimum number of hash commit characters to perform the search, instead PyDriller requires the whole commit and looking for the commit hash in the history commit of the repository to which it corresponded partially and then modify it. The problem with the non-existence of the project repository and therefore its commit hash was handled using the git APIs. In addition, a log file was created that tracks all non-existent repositories. Finally, the overhead problem derives directly from PyDriller. The latter, during the MSR, copied the entire repository contained in the dataset (except for the non- existent ones) to a local workspace, causing a filling of the hard disk memory space without consequent emptying. The problem was handled by opening an issue on the PyDriller repository and notifying the problem.

5.2 Software Metrics Extraction

The process of software metrics extraction for each java file is done through Understand. In line with the experiment done by Theisen et al., [3], I have collected 9 metrics, as shown in Table 1. These metrics are initially collected for each component of the projects, in different granularities. Morrison et al. [5] performed a replication of VPMs on two granularity levels: binary level and file level. The results of this experiment showed that the prediction at the source level is actionable. Therefore, from all the instances of metrics of granularities extracted by the Understand tool, I selected the file-level granularities.

Table 1. Software metrics used for my replication

Name	Description	Understand Name
CountLineCode	Number of lines containing source code. [aka LOC]	CountLineCode
CountDeclClass	Number of declared classes in the source code file.	CountDeclClass
CountDeclFunction	Number of declared functions in the source code file.	CountDeclFunction
CountLineCodeDecl	Number of lines containing declarative source code.	CountLineCodeDecl
SumEssential	Sum of essential complexity of all nested functions or methods.	SumEssential
SumCyclomaticStrict	Sum of strict cyclomatic complexity of all nested functions or methods.	SumCyclomaticStrict
MaxEssential	Max of essential complexity of all nested functions or methods.	MaxEssential
MaxCyclomaticStrict	Maximum strict cyclomatic complexity of nested functions or methods.	MaxCyclomaticStrict
MaxNesting	Maximum nesting level of control constructs.	MaxNesting

5.3 Text Mining

The realization of Text mining gave a first approach that respects the state-of-the-art criteria, as described in Section ref sec: Section2. The aim is to work with Java classes and to tokenize words in ”monogram” sets, where each monogram will have an associated counter that refers to the number of occurrences found. Below an example of a Java class:

```
Public class Example(){

/* I’m a
```

```
block comment
*/
        Public void doRetrieve () {
                //in-line comment
                System.out.println("Hello Guys");
        }
}
```

Before tokenizing, it is necessary to remove in-line comments and block comments, all logical operators, and any constants. After tokenization, a result turns out to be as follows:

{public: 2, class: 1, void: 1, Example: 1, retrieve: 1, System: 1, out: 1, println:1}

Using a dataset with different projects, this type of execution leads to the construction of a dictionary with a huge quantity of words, increasing the execution times and taking up a lot of disk space. So, I got the strengths of this first approach and thought of applying it in a different way, which makes execution times fast and takes up less disk space. Both quantities are, therefore, directly proportional. I used a standard normalization process that aims to carry out further ”pre-processing” phases before tokenization:

1. **Split CamelCase**
2. **Lower case reduction**
3. **Removing special chars and programming keywords**

These further steps have been applied to the file obtained from the execution of the first approach, obtaining significant improvements in terms of execution speed (less than a minute for all repositories) and disk space. The approach described by Scandariato et al. Cite Scandariato created a file of approximately 1.20GB, and with approximately 254,500 different words. By applying my pre-processing phases, I reduced the file by 90%, obtaining a file size of about 125MB with 13.780 different words. This makes us think that the three phases of ”text normalization” are fundamental for finding more occurrences of the same words.

5.4 Automated Static Analysis (ASA) Extraction

SonarQube collects ASA alerts. To export the results of the static analysis, I installed the plugin” CNESREPORT”. Initially, SonarQube’s report contained information about all alert types: code smells, security hotspots, bugs, and vulnerabilities. Due to the number of these types, I created a new quality profile to execute an analysis based only on vulnerability alerts. Therefore, I performed the analysis on the 39 rules available by the tool (excluding the deprecated rules). From these rules, I studied the results to understand which rules are essential to identify the vulnerability of my dataset, looking for the rules that aren’t violated in all Java files of my dataset, and then I excluded them. The analysis results showed that 19 types of vulnerability are present in several projects of

the dataset. I have created a new dataset that contains the number of vulnerabilities of that rule for each file analyzed. Let's define S as the set of files and R as the set of vulnerability. I define M [i, j] with i ⊂ S and j ⊂ R in which N is number of vulnerabilities per class:

$$M[i, j] = \begin{cases} N, if\ there\ are\ vulnerabilities\ of\ rule\ j\ in\ the\ class\ i. \\ 0, otherwise. \end{cases}$$

The resulting dataset presents 1251 files that violate the 19 rules considered.

5.5 Modelling

To execute the classification with these models, I used Weka. In line with the VPMs presented in the previous experiments, I used the following classifiers:

- Logistic Regression
- Support Vector Machine (SVM)
- Naive Bayes
- Random Forest

The selection of these Machine Learning Algorithms for the classification is due to the goal of obtaining a benchmark of the VPMs replication in the state-of-the-art, highlighting the best algorithm in terms of performance and considering the measurements defined in 5.7

For each selected model, I performed a k-fold cross-validation, using part of the dataset to fit the model and the other part to test it. In my replication, I use five-fold cross-validation, which has five components: one for the test set and one for the training. In relation to the dataset, five has shown to be a good value for the number of folds of cross-validation.??

5.6 Combination of the features

I have created different combinations of techniques; each one will be evaluated with the classifiers specified in 5.5. Referring to the combinatorial calculation, I define as P_n the number of permutations without repetitions, on the replicated techniques of n. P_n = n! = 6 different permutation techniques in which the single techniques described in the Section 5 are also considered, which are:

- Software Metrics (SM)
- Text Mining (TM)
- Automated Static Analysis (ASA)
- Software Metrics and Text Mining
- Text Mining and Automated Static Analysis
- Software Metrics and Automated Static Analysis

Given the heterogeneity of the datasets deriving from the individual techniques for the number of tuples and for the type of attributes, it was necessary to make an ad-hoc association by creating a unique *id* for each instance in the dataset, formed by "*commit hash/filename.java*".

Subsequently it was possible to create a dynamic combination in the following way:

Define D1 (with a_1, a_2, a_n the set of attributes in D_1) and D_2 (with b_1, b_2, $b_n$the set of attributes in D_2) two datasets of two techniques studied. The combination of the two techniques will be determined by D where each $d[i] \in D$ is defined:

d[i] = {$d_1[i]$, $d_2[i]$}

where $d_1[i]$ is the tuple of $D_1$with id i and $d_2[i]$ is the tuple of D2 with id i.

5.7 VPM Comparison

After the replication of each single methodology explained in Subsection 5.2, Subsection 5.3, Subsection 5.4 I have considered the possibility of making com- parisons, so as to verify which technique it was less performing with different types of classifiers and if the combination of multiple techniques could provide better results than the single ones. Each technique was evaluated with the following measurements regarding the performance of the classification:

- Precision: Represents the probability that the VPM's declaration of vulnerable code is accurate. It is a function of the True Positive (TP) rate and False Positive (FP) rate of vulnerable source code files. Precision is calculated as:

  Precision = $\frac{TP}{TP+FP}$

- Recall: Represents the probability that the VPM will find a source code file that contains at least one vulnerability. It is a function of the True Positive (TP) and True Negative (TN) rate of vulnerable source code files. Recall is calculated as:

  Recall = $\frac{TP}{TP+TN}$

- F-Score: Represents the geometric mean of precision and recall. Higher indicates better overall accuracy, assuming that precision and recall are weighted equally. F-Score is calculated as:

  $F_1 = (\frac{recall^{-1}+precision^{-1}}{2})^{-1}$

These actions, taken together, address the precision and execution of each VPM in my review. I present these actions independently, so the qualities of each model are still up in the air in lieu of a solitary precision figure. For instance, one model might have a high review but low accuracy, showing a high bogus positive rate. Another model might have high accuracy; however, it is poorly demonstrated that the model seldom gives misleading up-sides yet misses numerous weaknesses.

6 Results

In this section, I present the results of my case study on vulnerability prediction models run against the dataset explained in Section 4.

I have used *W eka* in order to evaluate my VPM with different classifier.

6.1 VPM Comparison

RQ1: Which are the best models analyzed?

Table 2: Median precision, recall and F1 scire for each vulnerability prediction model, using Random Forest classifier.

VPMs	Precision	Recall	F1
Software Metrics	0,606	0,620	0,603
Text Mining	0,776	0,766	0,754
ASA Results	0,562	0,596	0,561
SM + TM	0,763	0,755	0,742
SM + ASA	0,620	0,632	0,615
TM + ASA	0,776	0,766	0,755
All	0,783	0,772	0,761

Table 3: Median precision, recall and F1 score for each vulnerability prediction model using native-Bayes classifier.

VPMs	Precision	Recall	F1
Software Metrics	0,548	0,490	0,479
Text Mining	0,653	0,626	0,630
ASA Results	0,544	0,604	0,525
SM + TM	0,665	0,640	0,643
SM + ASA	0,552	0,544	0,547
TM + ASA	0,656	0,628	0,631
All	0,654	0,631	0,634

Studying the results of the single techniques present, I note that in Table 2, the VPMs use a classifier Random Forest. Analyzing the VPMs with this classifier, I note that the Software Metric (calculated with Understand) and Text

Table 4: Median precision, recall and F1 score for each vulnerability prediction model, using Simple Logistic classifier.

VPMs	Precision	Recall	F1
Software Metrics	0,543	0,598	0,453
Text Mining	0,720	0,709	0,685
ASA Results	0,570	0,620	0,505
SM + TM	0,711	0,701	0,674
SM + ASA	0,566	0,598	0,472
TM + ASA	0,732	0,725	0,709
All	0,727	0,72	0,702

Table 5: Median precision, recall and F1 score for each vulnerability prediction model, using Support Vector Machine classifier.

VPMs	Precision	Recall	F1
Software Metrics	0,530	0,531	0,531
Text Mining	0,746	0,749	0,745
ASA Results	0,439	0,617	0,475
SM + TM	0,757	0,760	0,757
SM + ASA	0,559	0,597	0,467
TM + ASA	0,757	0,759	0,755
All	0,751	0,754	0,751

Mining (with normalization process) show an increase in precision, recall and F-Score compared to those in [3]:

Software Metrics:

- Precision: da 0.45 a 0.60
- Recall: da 0.05 a 0.62
- F-Score: da 0.09 a 0.60
- Precision: da 0.47 a 0.77
- Recall: da 0.05 a 0.76

- F-Score: da 0.09 a 0.60

The obtained results by the ASA are not comparable with the state of theart, as the experiments are based on a different application domain, based onthe Recursive Partitioning classifier and on a set of homogeneous data, creatinga total difference in the context of the experiment. Despite this, my results couldbe acceptable:

Automated Static Analysis:

- Precision: 0,57
- Recall: 0,62
- F-Score: 0,50

The results in the previous tables show a significant increase in terms of precision, recall and f-score for each classifier analyzed for the text mining techniquecompared to software metrics and ASA.

I can, therefore, believe that among single VPMs, the technique of text mining in a context with heterogeneous data leads to better results than those described in the state of the art. I also believe it is important to apply the text mining normalization process also to a non-heterogeneous application domain,in order to assert its strengths with certainty.

6.1 VPM Accuracy rate and Error rate

RQ2: Which classifier obtains a higher error rate, and which one has a higher accuracy rate?

For this research question it is necessary to consider the Accuracy rate and Error rate, using the formulas:

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

Which indicates the accuracy of the model used. The best accuracy assumes a value of 1, while the worst is 0.

$$\text{Error} = \frac{FP+FN}{TP+TN+FP+FN} \text{ o Error} = 1 - \text{Accuracy}$$

which is calculated as the number of all incorrect predictions, divided by the total number of the data set. On the opposite here, the closer you get to 0, the lower the margin of error committed by the classifier, the closer you get to 1 the greater the margin of error.

Referring to Table 6, I calculated the accuracy for each VPM with the respective classifiers, then I calculated an average of accuracy for each classifier, in order to establish the lowest performing one:

1. Random Forest: 0,718
2. Support Vector Machine: 0,690
3. Logistic Regression: 0,674
4. Naive-Bayes: 0,592

Therefore, I identify that the Random Forest obtains a higher accuracy value, while the Naive-Bayes the lowest.

6.3 VPM Combination

RQ3: Is it possible to combine different VPMs? In respect of the state-of-art, are they more effective?

Table 6: Accuracy for each vulnerability prediction model and classifier.

VPMs	RF	NB	LR	SVM
Software Metrics	0,619	0,490	0,598	0,531
Text Mining	0,766	0,626	0,709	0,748
ASA Results	0,595	0,603	0,619	0,617
SM + TM	0,754	0,639	0,700	0,759
SM + ASA	0,632	0,543	0,597	0,596
TM + ASA	0,766	0,627	0,725	0,758
All	0,772	0,630	0,719	0,753

The text mining technique return good results regardless the type of the classifier, in fact the implementations of the VPMs combined with the text mining affects positively on the obtained data, compared to other combined VPMs with- out text mining. Note, in fact, that for each classifier the combined techniques with the text mining get better results than this single technique. Moreover, the results in the previous tables show that it's not always possible to obtain improvements from combining the VPMs, but even in some cases, the addition of features can decrease the model's performance. I consider it important to study the single techniques in order to choose accurately the types of combi- nations. Therefore, it turns out possible to combine the VPMs to improve the performance of a model, and, even if it's not possible to retrieve in the state-of-the-art all combinations, I consider that the combination of the Text Mining and Software Metrics obtains better results than those in [3], having an increase of Precision, Recall and F-Score with all the described classifiers. With these results, I can say that my models are not only more precise (Precision of 0.76), but even more sensitive (Recall of 0.76).

7 Conclusion and Future Works

In this paper, the researcher presents the replications of the vulnerability prediction models in state-of-the-art settings.

These replications are performed on a cross-project dataset, which contains several types of vulnerabilities for each project. Moreover, every single project in this dataset is open-source. The approaches presented from the replications analyze the source code through text mining, the software metrics and the alerts of the automated static analysis.

The results of this study show that text mining is the best technique for predicting vulnerable components e in general every single technique perform better results than those of the state-of-the-art. Subsequently these techniques were combined with each other in order to try to increase the predictive power of the VPM. Nevertheless, while combining the techniquesA and obtaining better results than the state-of-the-art. The researcher have noticed that the deviation of prediction from text mining is minimal. This points out that the techniques combined with text mining contribute very little to improving predictions. As a counterproof, the researcher studied the data obtained from the combination that does not use text mining (Software Metrics and Automated Static Analysis), obtaining lower results in some cases.

In the future, the researcher have planned to work on the development of a general framework for the use of VPM, in which there will be an evaluation and prediction phase, where in the evaluation phase an evaluation of the learning schemes will be obtained and the best. Later in the predicting phase the best learning scheme is used to create a predictor with all historical data and use the latter as a component to predict the vulnerabilities on new data. The first phase of the future general framework relating to evaluation is proposed in this research work. As another future development, a re-engineering process can be provided that allows the use of methodologies as a fundamental tool for predicting vulnerabilities.

In the end, the researcher have planned to perform an information gain technique in order to discover the “amount” of information of the presented VPMs attributes. This technique can be useful for a better understanding of the prediction power of the VPMs.

References

1. Zimmermann, T., Nagappan, N., and Williams, L. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In Software Testing, Verification and Validation (ICST), 2010 Third International Conference on (2010)
2. M. Gegick, P. Rotella, and L. Williams, “Predicting Attack-prone Components,” in ICST’09.
3. C.Theisen, L.Williams, ”Better together: Comparing vulnerability prediction models”, North Carolina USA, 2019
4. T. Hastie, R. Tibshirani, and J. H. Friedman, The Elements of Statistical Learning, New York, Springer, 2001.

5. P. Morrison, K. Herzig, B. Murphy, L. Williams, “Challenges with Applying Vulnerability Prediction Models”.
6. Y.Shin, A. Meneely, L.Williams, J.A Osborne, ”Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, IEEE Trans. Softw. Eng. 37 (6) (2011) 772-787.
7. R.Scandariato, J.Walden, A.Hovsepyan, W.Joose, ”Predicting vulnerable software components via text mining”, IEEE Trans.Softw Eng. 40 (10)(2014) 993-1006.
8. M. Jimenez, M. Papadakis, Y. Le Traon, “Vulnerability Prediction Models: A case study on the Linux Kernel”, 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2016.
9. Y. Shin, L. Williams, An initial study on the use of execution complexity metrics as indicators of software vulnerabilities, in: Proceedings of the 7th International Workshop on Software Engineering for Secure Systems, SESS ’11, ACM, New York, NY, USA, 2011, pp. 1–7, doi:10.1145/1988630.1988632
10. J. Walden, J. Stuckman, R. Scandariato, Predicting vulnerable components: soft- ware metrics vs text mining, in: Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, IEEE, 2014, pp. 23–33.
11. S.E. Ponta and H. Plate and A. Sabetta, M. Bezzi , C. Dangremont,”A Manually Curated Dataset of Fixes to Vulnerabilities of Open-Source Software”,Proceedings of the 16th International Conference on Mining Software Repositories
12. D.Spadini, M.Aniche, A.Bacchelli, ”PyDriller: Python Framework for Mining Soft- ware Repositories”,26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’18)
13. https://github.com/ishepard/pydriller