Maschine Learning based Code Analysis for Software Quality Assurance

Eldar Sultanow
Capgemini
Nuremberg, Germany
eldar.sultanow@capgemini.com

André Ullrich
Dept. of Business Informatics, University of Potsdam
Potsdam, Germany
aullrich@lswi.de

Stefan Konopik
IT Dept. of the Federal Employment Agency
Nuremberg, Germany
stefan.konopik@arbeitsagentur.de

Gergana Vladova
Dept. of Business Informatics, University of Potsdam
Potsdam, Germany
gvladova@lswi.de

Abstract—**Machine Learning is often associated with predictive analytics, for example with the prediction of buying and termination behavior, with maintenance times or the lifespan of parts, tools or products. However, Machine Learning can also serve other purposes such as identifying potential errors in a mission-critical large-scale IT process of the public sector. A delay of troubleshooting can be expensive depending on the error's severity – a hotfix may become essential. This paper examines an approach, which is particularly suitable for Static Code Analysis in such a critical environment. For this, we utilize a specially developed Machine Learning based approach including a prototype that finds hidden potential for failure that classical Static Code Analysis does not detect.**

Keywords—association rule mining, Machine Learning, Static Code Analysis, German Federal Employment Agency

I. Introduction

A characteristic feature of large software development projects are immense requirements for ensuring code quality. The case of the German Federal Employment Agency (FEA) shows how complex it is to meet such requirements, and demonstrates some failure consequences for the development team and the organization. ALLEGRO is a mission-critical IT process of FEA with a monthly disbursement volume of approximately two billion euros (~25 billion euros annually) to 10 million households (20 million eligible persons). Approximately 50 thousand power users operate the system in parallel. About 80 developers implement ALLEGRO continuously, which comprises more than 800 thousand lines of code. The software development is subject to a strong quality-assured procedure and uses the established Static Code Analysis (SCA) tool SonarQube. In spite of this, not all errors can be discovered before going live. Depending on the severity of an error in production, this often has the consequence that a hotfix needs to be delivered, which causes additional time and resources.

Newly added code may have "lookalikes" in the already existing code. Even if the developers have a broad and deep knowledge of the system, for them it is very hard to remember and locate existing code that differs minimally from newly created code. Human cognitive abilities to recognize such repetitions and patterns reach their natural limits at this scale. In case of a recent hotfix, the problem occurred in context of a not-null check that was missing subsequently to a specific variable declaration.

Classical SCA reaches its limits in the given context and complexity. Not every errors in code is recognized, testing rules needs to be predefined, classical SCA tools do not learn from errors that already occurred, and finally their support to programmers is focused on syntactical and grammatical corrections, and it is limited to corrections that base on predefined rules. Machine Learning (ML) opens up new ways to lift these restrictions. A ML based system identifies patterns in very large source code, which an individual is not able to comprehend anymore.

Until now, Machine Learning has been used only in few contexts of SCA, for example to detect duplicate code or suggest better variable names and documentation. However, ML methods allow pattern recognition, detection of programming rules from existing verified source code and violations in newly added code against these previously identified rules. Supporting developers in writing code, finding bugs, proposing better code conventions and detecting code clones are all promising applications of Machine Learning. ML is thus able to contribute to Static Code Analysis extensively.

Against this background, the present paper demonstrates a new approach of SCA that incorporates ML techniques for recognizing patterns in large source code, which emerge in software development, and which cannot be comprehended by humans.

The subsequent part of this paper is organized as follows: Section 2 discusses theoretical concepts that are relevant for the prototype. Section 3 introduces a Machine Learning based approach for pattern recognition. The prototype is presented in Section 4. Section 5 provides conclusions, limitations, and an outlook.

II. Theoretical Background

Source code is a collection of instructions and functions written by a programmer and processable by a machine, which can be statically or dynamically analyzed in order to find, for example, security flaws during automated tests.

Static Code Analysis is a selection of algorithms and techniques used to analyze source code. It applies to code that is not running and detects vulnerabilities, errors or poorly written code at compile time [1]. Hence SCA can reduce the cost of fixing security issues [2]. SCA tools are usually applied during early development to ensure code quality and security [3].

Dynamic Code Analysis (DCA) follows the opposite approach: instead of analyzing the software at compile time, under the approach of DCA, software is analyzed while it is operating. In more concrete terms, Dynamic Code Analysis

“will monitor system memory, functional behavior, response time, and overall performance of the system” [2]. An advantage of DCA is the ability of identifying memory accesses and buffer overflows [3]. Dynamic Code Analysis is used during and after deployment to consider live performance or detect potential issues [4], while Static Code Analysis is used to analyze software statically, without attempting to execute the code [5].

Machine Learning independently finds solutions for unsolved problems based on existing data and algorithms by recognizing patterns, regularities and deviations from these regularities. It has been recognized as a valid method for analyzing code [cf. 6, 7, 8] and is considered to be promising for bug detection and prediction [cf. 9, 10]. Additionally, there are various use cases in cyber security [11] or code clone detection [12]. Lechtaler et al. [13] introduce a solution for automated analysis of source code patches using ML algorithms. Allamanis et al. [14, 15] discovered ML to be useful for learning to adapt source code or for learning natural coding conventions. Additionally, there exist first scientific thrust towards code analysis that incorporates ML approaches. For example, a group of git repositories named *MAST (Machine Learning for the Analysis of Source Code Text)* is available open source [16]. Furthermore, Singh, Srikant and Aggarwal [17] introduce an approach for “question independent” software grading using ML. Johnson, Song, Murphy-Hill and Bowdidge [18] investigated the developers’ usage of SCA and came inter alia to the conclusion that finding bugs or software defects using static analysis tools is faster and cheaper than manual inspections.

There are a handful of ways to distinguish between ML algorithms; for instance regarding the learning style that an algorithm adopts and uses. They are usually classified into Supervised-, Semi-supervised-, and Unsupervised Learning. Relevant in the current context is the latter.

Unsupervised Learning needs neither predefined target values nor feedback from the environment. The learning machine tries to detect patterns in the input data itself. Since the outcomes are unknown, there is no evaluation of the accuracy. In other words, the learning algorithm needs to find commonalities among input data, when the outcome in training data is not predefined [19]. Frequent Pattern Mining and Sequential Pattern Mining are both Unsupervised Learning methods.

Frequent Pattern Mining aims to find relationships among the items in a database. Pattern mining refers to algorithms that discover interesting, unexpected or useful patterns in data. Frequent pattern mining was first proposed by Agrawal, Imieliński and Swami [20] in form of association rule mining for market baskets analysis. Therein, a transaction is defined as a set of distinct items. Given a set of transactions, association rule mining finds the rules that enable to predict the occurrence of a specific item based on the other items’ occurrences within transactions. Typical applications for frequent pattern mining are web link analysis, genome analysis, or click stream analysis. The most popular algorithm is *Apriori*, which has been first introduced in 1993. A wide variety of Apriori based algorithms was developed later, such as FP-Growth and Eclat [21].

Sequential Pattern Mining deals with data represented as a set of sequences. A sequence represents a set of transactions. Sequential pattern mining is applied in many cases, such as mining DNA (deoxyribonucleic acid) sequences and genomes or discovering customer-buying patterns; for example the customer buys a laptop, a digital camera, and a card reader within several months [22].

III. Pattern recognition via machine learning

Our idea of applying pattern recognition methods to code analysis is to transfer the principle of analyzing a shopping basket to SCA. The point of departure for shopping basket analysis is a set O of items, and a set F of all transactions, where each transaction $T=(TID, I)$ comprises a subset of the item set $TID \in F$, $I \subseteq O$ as shown in Figure 1.

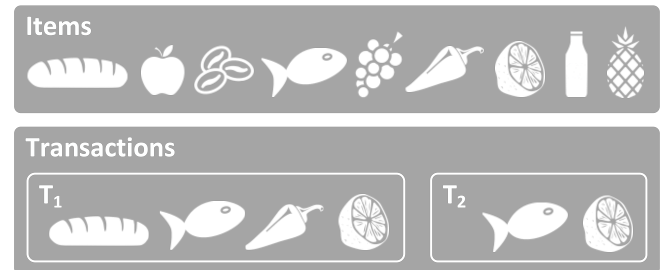

Fig. 1. Transactions and objects in shopping basket analysis

The basic idea is to solve the problem: find item sets such as “fish” and “lemon”, which are part of such transactions that make up a certain minimum percentage of all transactions. These item sets are called “frequent patterns”. These frequent patterns are the starting point for finding association rules such as “fish follows lemon” or more objectively formulated as “who buys fish, also buys lemon”.

We denote R: $X \rightarrow Y$ as an association rule with $X,Y \subseteq O$, $X \cap Y = \emptyset$.

Further, a transaction $T=(TID, I)$ satisfies the rule R, when it contains the disjoint item sets X and Y, $(X \cup Y) \subseteq I$.

Earlier we mentioned a minimum percentage of transactions that contain particular item sets. Related to this, three parameters needs to be considered for completing the defined framework:

1) The level of support of an item set

The level of support named as $support_F(X)$ of an item set X is the proportion of transactions that contain X, measured in the total number of transactions:

$$support_F(X) = \frac{|\{T \in F | T = (TID,\ I), X \subseteq I\}|}{|F|}$$

2) The level of support of an association rule

The level of support named as $support_F(X \rightarrow Y)$ of an association rule R: $X \rightarrow Y$ represents a statistical significance of the rule R, determined by the proportion of transactions that contain $X \cup Y$, measured in the total number of transactions:

$$support_F(X \rightarrow Y) = \frac{|\{T \in F | T = (TID,\ I), X \cup Y \subseteq I\}|}{|F|} = support_F(X \cup Y)$$

3) The confidence of an association rule

The confidence named as $confidence_F(X \rightarrow Y)$ of an association rule describes a degree of confidence for this rule.

It is calculated from the proportion of transactions that contain $X \cup Y$, measured by the number of transactions containing the item set X:

$$confidence_F(X \rightarrow Y) = \frac{support_F(X \rightarrow Y)}{support_F(X)}$$

But what if we do not refer to purchased objects, but to the concept of program code as presented here? The core idea is to consider code instructions, such as method calls, variable declarations or not-null checks as items. In our model, a transaction encloses item sets by a Java method. We derive association rules based on the frequency of occurrences of code instructions within Java methods. In addition, a minimum level of support can be configured, for example *50%, so that items are only considered if they occur in every second transaction*.

However, before any pattern recognition may perform on code, the measurement concept (when do we considered code to be "similar"?) must be determined. It makes less sense to consider two variable declarations as similar, if both variables has the same name. It is better to make a comparison based on the variables' type. In order to perform these comparisons efficiently, the code must be prepared accordingly. Every variable declaration and every method call must therefore be fully qualified. This preparation is part of the first step in the presented approach. The entire verified code base serves as input. The output is an attribute-relation file format (ARFF) file, which contains the transformed code with fully qualified information. In technical jargon, this step is usually referred to as "Code Mining" or "API Mining".

As a result, the algorithm delivers frequent item sets including their support level, and association rules along with their support level and confidence. The identification of rules bases on verified ("clean") code. New code, which naturally is unverified, will be checked against violations of these rules. After new code has passed all quality gates such as unit tests, integration tests and peer reviews, it will be merged into the verified code base. As the code base grows, so does the model. Ultimately, the volume of the transaction database and rule set increase the confidence.

The process is simplified in Figure 2. One of the simplifications is that the development of releases in FEA is not strictly sequential, but partly parallel – in simple terms the development of Release $X + 1$ has already begun at the end of the development of Release X. Moreover, in reality the model is updated more often than once per release and in future the update process will be automated using a Jenkins job (Jenkins is an open source automation server).

Essentially, Figure 2 illustrates the principle that new or newly modified code is checked against the existing rule set during the development of releases. If a rule violation occurs, the developer checks the affected code, corrects it if necessary, and merges the code into the Git repository. Notwithstanding this, each push is peer-reviewed by other experienced developers.

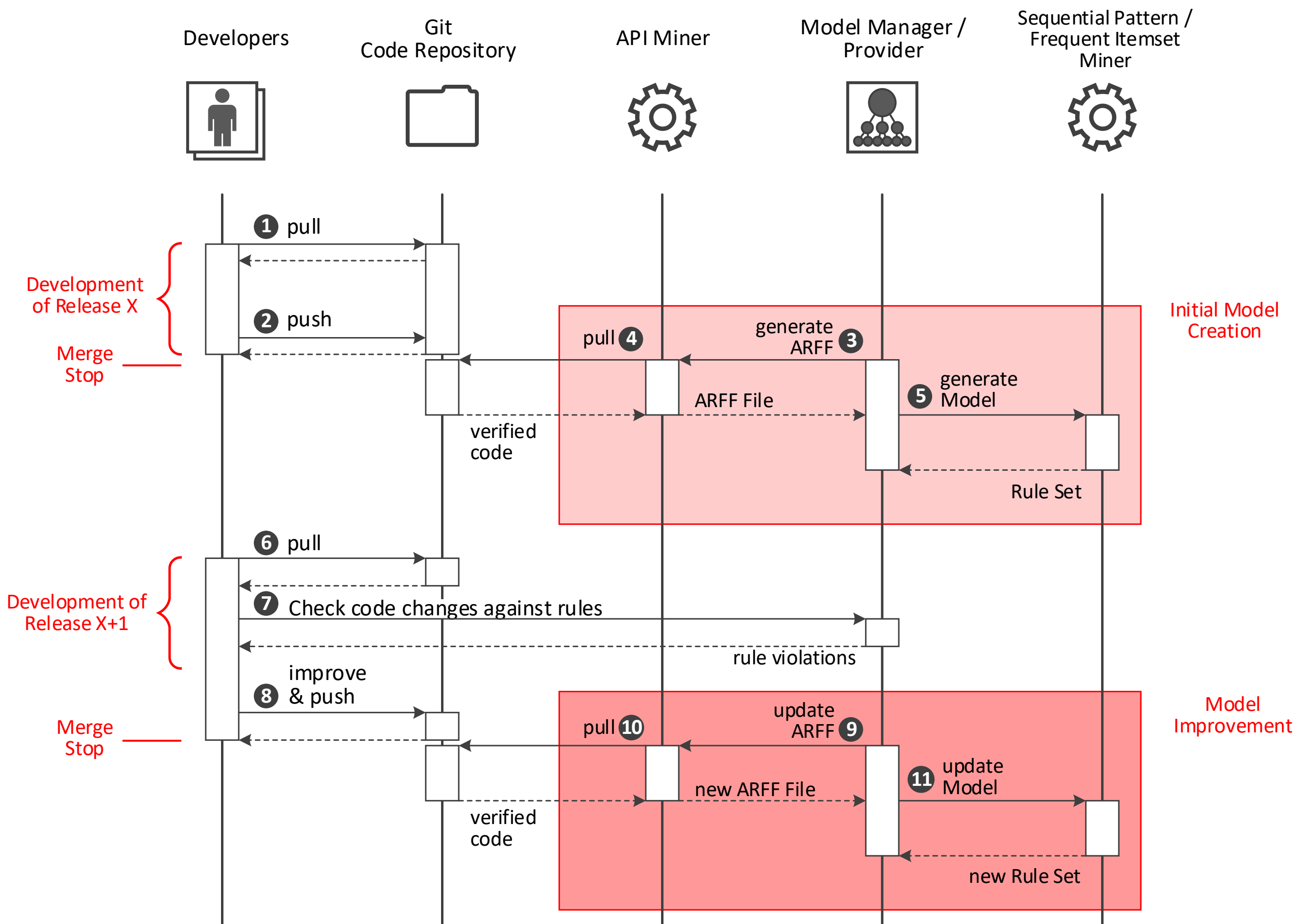

Fig. 2. Theoretical procedure of model development and application

IV. Prototype

A. Technical description

The system uses three Machine Learning procedures: the API Mining for generating an ARFF file (1), the Sequential Pattern Mining (2), and the Frequent Item Set Mining for recognizing patters in the prepared code that is given by the ARFF file (3). Thereby, different open source frameworks are used in combination.

However, it is clearly more demanding to implement appropriate algorithms, as it seems at first glance. It proves to be practicable using synthetic code as a verified code base and place a not-null check at four similar code fragments (following the earlier mentioned hotfix case). In a fifth fragment that is part of new (unverified) code, the not-null check was deliberately omitted. That was our litmus test: it was considered passed if the system detects and indicates the missing not-null check. During the implementation of the ML based system, we gained experiences on how to best use the algorithmic adjusting screws. One of the key findings was that a very low value for minimum support and confidence was needed to find the rule, which was relevant for the mentioned hotfix. The system has the following adjusting screws:

Minimum Support Training specifies the lower limit on the relative frequency of patterns, which are to be considered for the Pattern Mining process. It causes, patterns of lower frequency will not be detected.

Minimum Confidence Training indicates the minimum required confidence of a rule in percent. Let us recall, the confidence $confidence(X \rightarrow Y)$ of a rule is given by the quotient $sup(X \cup Y) / sup(X)$.

Max Antecedents Training specifies the limit of item sets by pattern *X*. Patterns that exceed the limit are excluded. A Subset of an excluded pattern below the limit will remain. The higher the value, the longer the computing time.

Max Consequents Training specifies the limit of item sets by pattern *Y*. Patterns that exceed the limit are excluded. A subset of an excluded pattern below the limit will remain. Again, higher values cause longer computing times.

Max intraprocedural recursion specifies the maximum number of recursive steps in resolving method calls. A value zero deactivates this feature.

Training set directory is a directory that contains the .java files that are used for mining the patterns and rules.

Input Pattern Training is a regular expression that filters Java classes out of the training set directory. The expression is applied to the absolute paths of the .java files. With local training (one method at a time), folders that contain tests are generally filtered out. A possible expression is:

.*BProtErgSachlAbsetzungenErmittler.*|.*SubModelAbsetzung.*|.*DAO.*|.*Constraint.*|.*Helper.*.

Solely files should be considered as input, whose name contain "DAO", "Helper", etc.; in this concrete case, specific business objects, data access objects and helper classes.

Caller Method Space is a regular expression that filters out methods from the analysis. Here, the expression is applied to the full qualified method name.

Call Method Space is a regular expression that filters out method calls from the analysis. In this case, the expression is matched with the full qualified name of a method call.

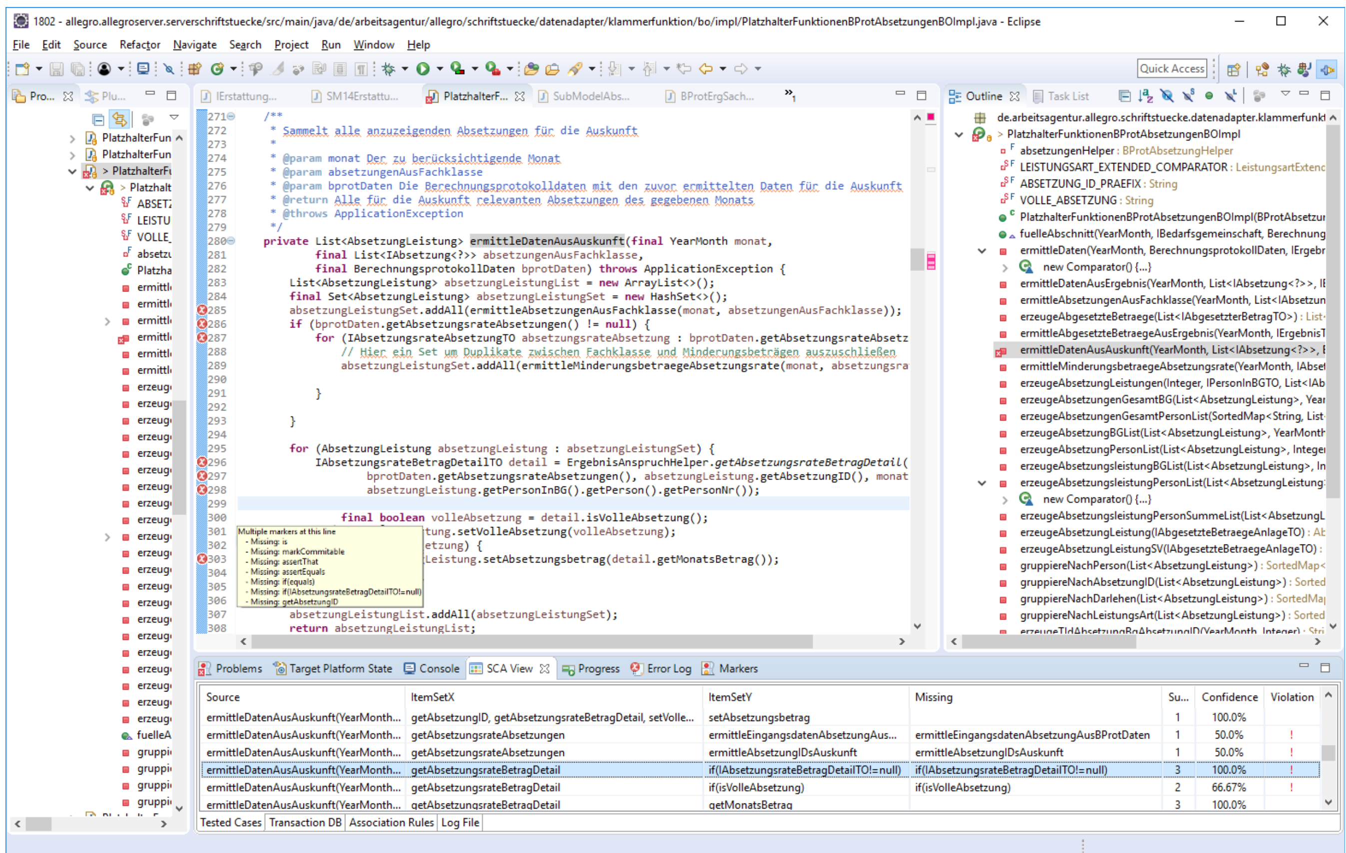

Fig. 3. Screenshot of the Machine Learning based Eclipse extension

Figure 3 shows the Eclipse extension used by the Federal Employment Agency to identify potential errors in ALLEGRO's source code. The screenshot shows the code fragment of the hotfix that we used as a reference example in this paper. In the method *ermittleDatenAusAuskunft* the not-null check was removed (the absence led to the hotfix) and the Machine Learning based system was started to analyze this method. In fact, the system identified the rule, which states that at the relevant position in the code a not-null check must occur; and the system correctly suggested to complete this code fragment. Specifically, the system indicates that the following statement (ItemSetX):

getAbsetzungsrateBetragDetail

needs to be collocated with the statement (ItemSetY):

if*(IAbsetzungsrateBetragDetailTO! =null).*

Accordingly, as shown in figure 3, the missing statement is indicated at the highlighted line of code 299.

B. Application and implications

The system analyzed the source code of ALLEGRO and determined the rule relevant to the hotfix. Related to this, the main advantage is that Machine Learning is able to identify the relevant rules by itself. This is a substantial innovation in contrast to the present situation, where developers need to provide a list of all necessary rules in advance. Beyond that, our ML based system identified further rules.

One of these rules states that a transaction opened by "TransactionalSection.enter" must also be marked as "committable" by "TransactionalSection.markCommitable". The error appeared mainly in test code and is partially caused by a lack of experience, since in test cases transactions must be managed manually, whereas a framework manages transactions in productive code.

Another example is the rule that developers must call the round function in certain cases. Additionally, the system has also provided guidance on the use of reusable, technically appropriate utility/helper methods.

Let us mention a third example, namely a rule stipulating that in test cases a list must be sorted, which contains sums of certain monetary amounts (overpayments of health insurance contributions). This list should be sorted after its entries were summarized in a special processing step. The sorted order is important for test cases, since otherwise during a stepwise comparison the actual value would differ from the expected one.

V. Conclusions

Within this paper, pattern recognition methods were used in order to transfer the principle of shopping basket analysis to Static Code Analysis. This approach was successfully tested within ALLEGRO – a mission-critical IT process and complex software system of the Federal Employment Agency. We presented and discussed our idea, the mathematical foundation, the implementation of the ML based system including the configuration of its parameters, and finally the experiences of the prototypical use of this system.

The concept, provided in this paper forms a key contribution that Machine Learning can make to SCA. However, there exist other possible contributions, which for example include the code clone detection that recognizes duplicate code, even if it is modified or written differently. Vulnerability scans to detect security weaknesses in source code are also addressable by Machine Learning. Another area of application is naming and documentation suggestion for variable names or method names in the code. For this, text-mining procedures for context determination and analysis that originate in the field of ML are suitable.

The practice of complex software development processes has shown that rules with very high confidence are relevant and their violation lead to errors. Even more interesting, however, are rules with very low confidence, since patterns that rarely occur in a large code mass are more difficult to recognize by humans. Consequently, a very low value for the minimum support and confidence may be required for a deep source code analysis. This is an indicator for improving our approach, for example by possibly blacklist certain patterns to minimize the noise floor. Integrating our system as a plug-in for Eclipse opens up new options as well. It is conceivable, for instance, to enable "intelligent" auto completion by means of Eclipse Code Recommenders. In addition, rules could also be included in a revocation list so that they will not continue to be displayed.

In summary, the key benefit of the approach presented here is the ability of finding hidden potential for failure that classic SCA does not detect. Classical SCA requires one to predefine all rules in advance – after a bug becomes apparent in production, the rule set will be adjusted in turn. On the contrary, Machine Learning has the advantage that rules can be found "intelligently" via pattern recognition – before error prone code gets merged into the code base and a bug becomes on the loose in a productive environment.

References

[1] M. Nadeem, "Why SonarQube: An introduction to Static Code Analysis" 2015. Online available from: https://dzone.com/articles/why-sonarqube-1 (accessed 24 February 2018)

[2] N. DuPaul, "Static Testing vs. Dynamic Testing" 2017. Online available from: https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing (accessed 24 February 2018)

[3] B. McCorkendale, T. Xue Feng, G. Sheng, Z. Xiaole, M. Jun, M. Qingchun, H. Ge Hua, and E.H.Wei Guo, "Systems and methods for combining static and dynamic code analysis." U.S. Patent 8,726,392, issued May 13, 2014.

[4] D. Cornell, "Static Analysis, Dynamic Analysis and how to use them together" in: Denim Group. Online available from: https://denimgroup.com/media/pdfs/DenimGroup_ StaticAnalysisTechniquesForTestingApplicationSecurity_OWASPSanAntonio_20080131.pdf (accessed 24 February 2018)

[5] A. Marchenko, and P. Abrahamsson, "Predicting Software Defect Density: A case Study on Automated Static Code Analysis." in Proceedings of 8th International Conference, XP 2007. Agile Processes in Software Engineering and Extreme Programming, Lecture Notes in Computer Science, 4536, DOI: 10.1007/978-3-540-73101-6

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, and J. Vanderplas, "Scikit-learn: Machine learning in Python". *Journal of machine learning research*, *12*(Oct) 2011, pp.2825-2830.

[7] C. Robert, "Machine Learning, a Probabilistic Perspective", CHANGE,27:2 2014, pp. 62-63, DOI: 10.1080/09332480.2014.914768

[8] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning". *Journal of Computer Security*, *19*(4) 2011, pp.639-668.

[9] S. Axelsson, D. Baca, R. Feldt, D. Sidlauskas, and D Kacan, "Detecting defects with an interactive code review tool based on visualisation and Machine Learning". Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering 2009.

[10] M. Fejzer, M. Wojtyna, M.J. Burzańska, P. Wiśniewski, and K.J. Stencel, “Supporting code review by automatic detection of potentially buggy changes” in: S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa D. (eds) “Beyond Databases, Architectures and Structures” BDAS 2015. Communications in Computer and Information Science, vol. 521. Springer, Cham.

[11] NIST SAMATE, “Source Code Security Analyzers” 2016. Online available from: https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html (accessed 29 March 2018)

[12] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, “CCLearner – A deep learning-based clone detection approach“ 2017.

[13] A. C. Lechtaler, J. C. Liporace, M. Cipriano, E. García, A. Maiorano, E. Malvacio, and N. Tapia, “Automated Analysis of Source Code Patches using Machine Learning Algorithms”. IV Workshop de Seguridad Informática (WSI) 2015, XXI Congreso Argentino de Ciencias de la Computación.

[14] M. Allamanis, “Learning natural coding conventions”. Dissertation 2016, Institute for Adaptive and Neural Computation, School of Informatics, University of Edinburgh.

[15] M. Allamanis, H. Peng, and C. Sutton, “A convolutional attention network for extreme summarization of source code”. Proceedings of Machine Learning Research, Volume 48: International Conference on Machine Learning, 20-22 June 2016, New York, New York, USA, pp. 2091-2100

[16] MAST, “MAST – Machine Learning for the analysis of source code text. 2018. Online available from: https://github.com/mast-group/ (accessed 29 March 2018)

[17] G. Singh, S. Srikant, and V. Aggarwal, “Question independent grading using Machine Learning: The case of Computer Program Grading.” KDD ’16, August 13 - 17, 2016, ACM Press: San Francisco, CA. DOI: http://dx.doi.org/10.1145/2939672.2939696 (accessed 29 March 2018)

[18] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” Proceedings of the 2013 International Conference on Software Engineering (ICSE '13), San Francisco, CA, May 18 - 26, 2013, pp. 672-681, Piscataway, NJ: IEEE Press.

[19] S.-S. Shai, and B.-D. Shai, “Understanding Machine Learning: From theory to algorithms“ 2014. Cambridge University Press. Online available from: http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning (accessed 15 February 2018)

[20] R. Agrawal, T. Imieliński, and A. Swami, “Mining association rules between sets of items in large databases“ in *Acm sigmod record* ,Vol. 22, No. 2 1993, pp. 207-216).

[21] C. C. Aggarwal, and J. Han (Eds.), “*Frequent pattern mining“ 2014* Springer.

[22] T.-R. Li, Y. Xu, D. Ruan, and W. Pan, “Sequential pattern mining. Intelligent data mining“. Springer, Berlin, Heidelberg, 2005. 103-122.