2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)

Sniffing Android Code Smells: An Association Rules Mining-based Approach

Jehan Rubin[1], Adel Nassim Henniche[2], Naouel Moha[1], Mohamed Bouguessa[1], Nabila Bousbia[2]
University of Quebec in Montreal[1] - Higher National School of Computer Science[2]
Montreal, Canada - Algiers, Algeria
rubin.jehan@courrier.uqam.ca, a_henniche@esi.dz, {moha.naouel, bouguessa.mohamed}@uqam.ca, n_bousbia@esi.dz

***Abstract*—Interest in mobile applications (mobile apps) has grown significantly in recent years and has become an important part of the software development market. Indeed, mobile apps become more and more complex and evolve constantly, while their development time decreases. This complexity and time pressure might lead developers to adopt bad design and implementation choices, which are known as code smells. Code smells in mobile apps could lead to performance issues such as overconsumption of hardware resources (CPU, RAM, battery) or even downtime and crashes. Some tools have been proposed for the detection of code smells in Android apps, such as PAPRIKA or ADOCTOR tools. These tools rely on metrics-based detection rules, which are defined manually according to code smell definitions. However, manually defined rules might be inaccurate and subjective because they are based on user interpretations. In this paper, we present a tool-based approach, called FAKIE, which allows the automatic inference of detection rules by analysing code smells data using an association rules algorithm: FP-GROWTH. We validated FAKIE by applying it on a manually analysed validation dataset of 48 opensource mobile apps. We were able to generate detection rules for a dozen code smells, with an average F-measure of 0.95. After all of that, we performed an empirical study by applying FAKIE on 2,993 apps downloaded from ANDROZOO, a repository of mobile apps.**

***Index Terms*—Android, code smells, detection, association rules, mobile applications.**

I. INTRODUCTION

In recent years, mobile applications (apps) gained a considerable importance in the software market. Indeed, mobile apps become more complex and evolve constantly, while their development time decreases. These factors might lead developers to make bad design choices, which are known as *code smells* [1]. Code smells tend to increase the complexity of a system [2] and affect software maintainability [3], [4], and thus make difficult the evolution of systems. In particular, code smells in mobile apps cause performance issues such as overconsumption of hardware resources (CPU, RAM, battery) or even downtime and crashes. There is a recent and growing interest in Android code smells, which are identified using common object-oriented (OO) detection techniques [5]. However, Android apps are different from common OO systems because the Android platform has several limitations and constraints on resources like memory, CPU or screen sizes. Therefore, Android apps have their own code smells in addition to the common OO ones. Reinmann *et al.* [6] define 30 code smells specific to Android. There are only very few detection tools specialised in Android code smells, such as PAPRIKA [7] and ADOCTOR [8]. Their detection methods use manually defined rules based on the definitions of OO code smells and Android code smells from Reimann *et al.* [6].

Manually defined detection rules can have several limitations because of the manual interpretation of the code smell definitions. In this paper, we present FAKIE, an automatic tool-based approach to generate Android code smells detection rules by performing a static analysis of Android apps. Our approach is a complement to existing code smells detection tools. The purpose of our work is not to create a new detection tool, but an alternative for the definition of the rules. We use the FP-GROWTH association rules algorithm to analyse the data generated by the static analysis.

We validated our approach on 48 opensource apps from F-Droid [9]. The proposed approach generated automatically a dozen detection rules with an average F-measure of 0.95. We also performed a large empirical study on 2,993 Android mobile apps downloaded from AndroZoo [10], the results of this study show the capacity of our approach to extract information from a dataset of code smells and highlight some development trends.

II. BACKGROUND INFORMATION AND RELATED WORK

A. Detection tools

Android apps are developed using OO languages like Java. The properties of a software project written using OO can be described by quality metrics that measure software quality [11]–[15]. These metrics have been used to develop OO code smells detection tools like DECOR [16] or the tool proposed by Kothari *et al.* [17]. However, only a few tools are specialised in code smells detection for Android apps, we can mention: Hecth *et al.* [7] who used rules based on quality metrics to detect 13 Android-specific code smells, and Palomba *et al.* [8] who detected 15 code smells from the catalogue of Reimann *et al.* [6] using also a rule-based technique.

B. Code smells and Machine learning

There are only few works using machine learning to detect code smells, and even less for Android code smells. Maiga *et al.* [18], [19] use the SVM algorithm to detect four code smells by analysing metrics generated with a source code analysis framework. Khomh *et al.* [20] use a Bayesian approach. Other

978-1-7281-3395-9/19/$31.00 ©2019 IEEE
DOI 10.1109/MOBILESoft.2019.00025

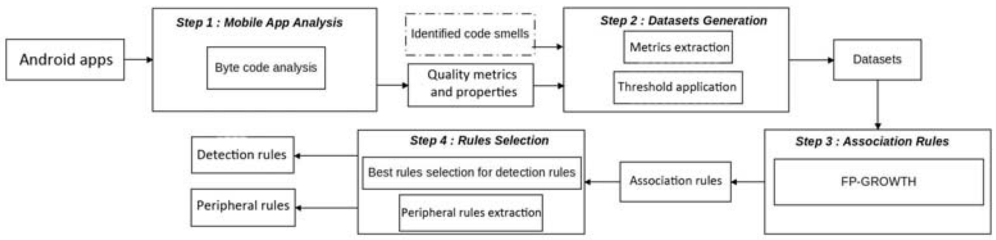

Figure 1. Overview of the approach for rule generation.

Table I
LIST OF QUALITY METRICS AND PROPERTIES.

Metrics and Properties	Entities	Description
NoM	Class	Number of Methods
DoI	Class	Depth of Inheritance
NoII	Class	Number of Implemented Interfaces
NoA	Class	Number of attributes
CLC	Class	Class Complexity
LoC	Class	Lack of Cohesion in methods
isAbstract	Class, Method	Class or Method is Abstract
isStatic	Class, Method	Class or Method is Static
isInnerClass	Class	Class is Inner Class
isInterface	Class	Class is Interface
isActivity	Class	Class is Activity
isBroadcastReceiver	Class	Class is Broadcast Receiver
isAsyncTask	Class	Class is asynchrone
ownOnLowMemory	Class	Class owns On Low Memory Method
NoP	Method	Number of Parameters
NoI	Method	Number of Instructions
NoDC	Method	Number of Direct Calls
NoC	Method	Number of Callers
CC	Method	Cyclomatic Complexity
NatureOfClass	Method	Nature of the Class containing the current method
callExternalMethod	Method	Method calls at least one External Method
callMethod	Method	Method calls at least one Method
useVariable	Method	Method uses at least one Instance variable
callInit	Method	Method calls a constructor

existing approaches use association rules techniques. For example, Palomba *et al.* [21] uses the APRIORI [22] algorithm to generate association rules to analyse co-occurrences between code smells. Palomba *et al.* [23] and Fu *et al.* [24] also use association rules to detect code smells by analysing the history of the evolution of the different versions of a system.

The detection made by machine learning algorithms is always carried out on very different code smells [18]–[20], [25], [26], which facilitates the classification. In a real context, however, code smells can co-occur together and increase the complexity of the classification for code smells detection and sometimes they have similar characteristics.

In this paper, we generate 10 detection rules for OO and Android code smells. The definition of each code smell follows:

Blob Class (BLOB): Blob class or God class is a class containing a large number of attributes and methods. Blob classes are hard to maintain and increase the difficulty to modify the software.
No Low Memory Resolver (NLMR): In Android, the Activity method onLowMemory() is called by the system when running low on memory. This method should free allocated and unused memory spaces, if it is not implemented, the system may kill the process.
Swiss Army Knife (SAK): A Swiss army knife is a very complex interface that regroups a lot of methods. This kind of interface makes child classes hard to understand.
Long Method (LM): Long methods are implemented with much more lines of code than other methods. These methods can be split into smaller methods to fix the problem.
Complex Class (CC): Complex class is a class containing complex methods.
Leaking Inner Class (LIC): In Android, anonymous and non-static inner classes hold a reference of the containing class. This can prevent the garbage collector from freeing the memory space of the outer class even when it is not used anymore, and thus causing memory leaks.
Heavy AsyncTask (HAS): An AsyncTask should not implement the methods onPostExecute, onPreExecute and onProgressUpdate. This can lead to a lack of responsiveness, or stopping the app's execution.
Heavy Broadcast Receiver (HBR): The presence of heavy or blocking operations in a broadcast receiver and more precisely in the OnReceive method can cause crashes of apps.
Member Ignoring Method (MIM): In Android, a method that is not a constructor and does not access non-static attributes must be static to increase performance.
Hashmap Usage (HMU): In Android, the usage of HashMap is not recommended when managing small sets. Instead, the framework provides the ArrayMap and SimpleArrayMap classes.

III. APPROACH

The main idea of our approach is to identify a combination of quality metrics and properties involved in the presence of a code smell. Our four steps are illustrated in Fig.1.

A. *Step 1: Mobile App Analysis*

In this step, we extract information about Android apps. Table I contains the list of quality metrics and the generated properties extracted from the apps. During the analysis, the relationships between entities (class or method) are also collected. To extract the information from the apps, we used the framework SOOT with the DEXPLER [27] module to generate a model with properties and quality metrics. Then, we store this model in an annotated graph using the graph oriented database system NEO4J [28].

B. Step 2: Datasets Generation

This step takes as input a list of code smells identified in previously analysed mobile apps. To identify the code smells in our apps, we used a semi-manual approach described in Section IV. Based on this list, we extract entities containing those code smells to build datasets with quality metrics and properties. The datasets contain only entities infected by code smells and are divided into two levels of granularity: classes and methods. The datasets are composed of numerical and nominal values according to the quality metrics and properties. However, the next step corresponding to the automatic rules generation requires the use of datasets containing only discrete properties. To define thresholds, we used the Tukey box [29]. We end up with a dataset in this form:

```
NoP>25 NoP>52.4 NoDL>12 NoI>22 NoDC>3 NoC>34
CC>18 isGetter ... callInit CodeSmell
true true false false true true true false ... false HAS
true false true false true true false false ... false HAS
```

C. Step 3 : Association Rules

Using the FP-GROWTH [30] association rules algorithm, We analyse datasets previously generated to detect frequent elements. The ultimate goal is to generate association rules under the form : $X, Y \Rightarrow Z$ where X and Y are metrics and property values (LoC=25, $isAbstract$=true...) and Z is a code smell instance (BLOB, NLMR...).The arrow implies: If X and Y are present, then Z is potentially present too. The rules generated by the algorithm must be relevant. To assess the relevance of the association rules, there are two important parameters to consider: support and confidence [31]. We used the FP-GROWTH implementation of the WEKA [32] library to generate our association rules.

D. Step 4: Rules Selection

This step consists in selecting the rules extracted by the algorithm that are most suitable to be used as detection rules. Indeed, the most interesting rules are those containing the largest number of items and must be in the form *many to one*, *one* representing the code smell type and *many* the metrics and properties. The rules selected are those with the highest confidence and support. Our algorithm is wrote on Java.

IV. STUDY DESIGN

To evaluate FAKIE, we conducted two studies: a validation study and an empirical study. For the validation study, we use a code smells dataset created by two experts by performing a semi-manual analysis in 48 opensource Android apps. The second study is an empirical study performed by applying FAKIE on 2,993 mobile apps from AndroZoo [10].

A. The validation study

The first study consists in generating detection rules with FAKIE using 70% of the code smells dataset. Then, the second step is to perform an automatic detection by applying these rules on the other 30% of the dataset.

Dataset. To create the dataset for the validation, we perform a semi-manual analysis of mobile apps. We used both quality metrics and manual code source analysis to identify code smells. The semi-manual detection of code smells is performed in several steps : (1) the analysis of the APKs to generate the NEO4J graph. (2) Using NEO4J queries on the graph, we apply a first filter on the entities of the graph (class and method). For example, to detect *SAK*, we use a query to obtain only classes that are interfaces. (3) Then, we perform a manual analysis of the source code of each entity in the list to identify code smells. If the smelliness of a code smell is correlated to numerical values like the number of instructions, we add the values of each numerical metric to the results of the queries. If a value seems to be atypical, we keep it. Then we perform a manual analysis of the source code to identify the smelliness. This process is applied by both experts separately, once the identification is completed, we cross the results and keep the common results.

Once the datasets are created, we apply steps 3 and 4 described in Section III. After the rules are generated, we perform a detection with them in the form of NEO4J requests. To evaluate the relevance of the generated detection rules, we use three measurements: accuracy, recall and F-measure for each code smell [33].

B. The empirical study

When we generate detection rules with FAKIE, some rules are not selected as detection rules. We call these rules *peripheral rules*. Peripheral rules are less relevant for detection, but they can contain interesting information about code smells. We consider a rule as peripheral if it has at least 0.1 support. This threshold allows the generation of association rules concerning at least 10% of the transactions. We perform a detection with the detection rules generated by FAKIE on 2,993 apps. Then we generate association rules with FAKIE.

V. RESULTS AND DISCUSSIONS

Table II presents the detection rules generated with FAKIE. We use the rules with the highest confidence and support to get the most relevant rules. It enables to get a maximum of true positives. All these rules have a confidence of 1 and a support of 1, except for the rule for *HAS*, which gets 0.96 support. This code smell is present when methods named 'onPrexecute', 'onPostExecute' or 'onProgressUpdate' perform heavy tasks. Thus, the property "name" of the method can take three different values. Currently, FAKIE cannot generate rules when a metric or property can have multiple values for the same code smell. The reason is that FP-GROWTH can extract rules with only one value of an attribute. FP-GROWTH generates rules on transactions, so it is impossible to have different values for one item in the same rule.

In Table III, we provide precision, recall and F-measure of the detection done with FAKIE's rules. The detection has an average of 0.95 F-measure. For *BLOB*, *CC*, *LIC*, *LM*, *SAK*, *HMU* and *HBR*, we obtain a very effective detection by using the automatically generated rules. The recall value for the *HAS*

Table II
AUTOMATICALLY GENERATED RULES

Smells	Generated Rules	Confidence - Support
BLOB	[NoM>14 = true, NoA>8.5=true, LOC>25=true] ⇒ [class=BLOB]	1 - 1
CC	[CLC>28=true] ⇒ [class=CC]	1 - 1
HAS	[NoI>17=true, natureOfClass=is_async_task, CC>3.5=true, callExternalMethod=true, NoDC>2.5=true, **name=onPostExecute**] ⇒ [class=HAS]	1 - 0.96
LIC	[isStatic=false, isInnerClass=true] ⇒ [class=LIC]	1 - 1
LM	[NoI>17=true] ⇒ [class=LM]	1 - 1
MIM	[isInit=false, isStatic=false, **NoC>5=false**, callExternalMethod=false, callMethod useVariable=false] ⇒ [class=MIM]	1 - 1
NLMR	[ownOnLowMemory=false, isActivity=true] ⇒ [class=NLMR]	1 - 1
SAK	[isInterface=true, **isAbstract=true**, NoM>14=true] ⇒ [class=SAK]	1 - 1
HMU	[useHashMap=true, callExternalMethod=true] ⇒ [class=HMU]	1 - 1
HBR	[NoI>17=tue, **NoDC>2.5=true**, natureOfClass=is_broadcast_receiver, name=onReceive, **isOverridden=true**, CC=high, callExternalMethod=true] ⇒ [class=HBR]	1 - 1

Table III
PRECISION, RECALL AND F-MEASURE OF RULES

Smells	Occurences	Precision	Recall	F-measure
BLOB	404	1	1	1
CC	886	1	1	1
HAS	47	1	0.92	0.95
LIC	48	1	1	1
LM	197	1	1	1
MIM	2136	0.67	1	0.80
NLMR	5807	0.66	1	0.79
SAK	1644	1	1	1
HMU	218	1	1	1
HBR	34	1	1	1

Table IV
SOME PERIPHERAL RULES

Rules (Confidence, Support)
NoM>14=true, NoA>8.5=true, LOC>25=true **CLC>28=true** ⇒ class = BLOB (1, 0.81)
[NoM>14=true, NoA>8.5=true, LOC>25=true, **DoI>5=true, isActivity=true**] ⇒ class=BLOB (1, 0.44)
[useHashMap=true, **callInit=true** , callExternalMethod=true, **NoI>26=true**] ⇒ class=HMU (1, 0.58)
[useHashMap=true, **callInit=true**, callExternalMethod=true, **CC>5=true**] ⇒ class=HMU(1, 0.37)

rule is also due to the fact that a property can take several values as explained in the previous paragraph.

The rule for *NLMR* detection has a precision of 0.66. When we used our rule to detect *NLMR*, we detected a higher number of *NLMR* instances. It appears that the generated rules cannot contain any information about the inheritance hierarchies. The reason is because during our dataset generation, we do not analyse the inheritance hierarchies in the graph. It consumes a lot of resources during the generation. However, an activity can inherit the 'onLowMemory' method from another class. So, we have detected some false positives.

Finally, the rule for *MIM* detection has a precision of 0.67, a recall of 1 and a F-measure of 0.80. This can be explained by the use of thresholds for metrics values. The rule detects a *MIM* if the metric NoC is less than 5. The problem is that, for better precision, we must consider the null value. For *MIM*, we must exclude entities with NoC with a null value because it is not interesting to detect entities that are not used in the application.

FAKIE generates relevant detection rules with an average F-measure of 0.95.

Concerning the empirical study, table IV shows some of the selected peripheral rules: Some rules highlight trends on certain code smells. We can see that *BLOB* is mainly present in Activities (44%). Activities represent about 7% of all classes in our dataset and represent main entities in mobile development. *BLOBs* are also very complex, which is normal given that the number of methods, the lack of cohesion and the number of attributes are factors related to the complexity of a class. A large part of *BLOBs* have a great depth of inheritance, which can be explained by the presence of many attributes and methods. In addition, we can see a trend of *HMUs* to occur with the *LM* code smells. Indeed, as shown in Table IV, 58% of *HMUs* have a number of instructions greater than 26.

Developers have a tendancy to implement very heavy Activities containing *BLOBs* (44%).

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed the tool FAKIE to generate automatically Android and OO code smells detection rules using FP-GROWTH, an association rules algorithm. We used a bottom-up approach to infer detection rules from data. First, we inferred and we evaluated the detection rules generated with FAKIE on manually analysed apps. We obtained an F-measure average of 0.95 for ten code smells. This shows the relevance of our rules. Finally, we conducted an empirical study on 2,993 Android apps to extract additional information about code smells. This shows a tendency of developers to develop complex Activities in Android apps. For future work, we will add additional metrics and properties to our datasets for improving the precision of detection rules and generate new detection rules for additional code smells.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[2] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 51–60.

[3] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 306–315.

[4] A. Yamashit a and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 682–691.

[5] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: a threat to the success of android apps," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, 2013, pp. 477–487.

[6] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," in *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*, vol. 2014, 2014.

[7] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of android applications along their evolution (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 236–247.

[8] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 487–491.

[9] "F-DROID," Online; accessed February-2019, https://f-droid.org/.

[10] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 468–471.

[11] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study," *Software process: Improvement and practice*, vol. 14, no. 1, pp. 39–62, 2009.

[12] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[13] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[14] Y. Singh, A. Kaur, and R. Malhotra, "Empirical validation of object-oriented metrics for predicting fault proneness models," *Software quality journal*, vol. 18, no. 1, p. 3, 2010.

[15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[17] S. C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty, "A pattern-based framework for software anomaly detection," *Software Quality Journal*, vol. 12, no. 2, pp. 99–120, 2004.

[18] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 278–281.

[19] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *Reverse engineering (WCRE), 2012 19th working conference on*. IEEE, 2012, pp. 466–475.

[20] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[21] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Workshop on*. IEEE, 2017, pp. 8–13.

[22] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[23] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[24] S. Fu and B. Shen, "Code bad smell detection through evolutionary data mining," in *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*. IEEE, 2015, pp. 1–9.

[25] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[26] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, "Code smell detection: Towards a machine learning-based approach," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 396–399.

[27] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. ACM, 2012, pp. 27–38.

[28] "NEO4J," Online; accessed February-2019, https://neo4j.com.

[29] J. W. Tukey, *Exploratory data analysis*. Reading, Mass., 1977, vol. 2.

[30] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM sigmod record*, vol. 29, no. 2. ACM, 2000, pp. 1–12.

[31] 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. ACM, 1993, pp. 207–216.

[32] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[33] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *European Conference on Information Retrieval*. Springer, 2005, pp. 345–359.