Does Source Code Quality Reflect the Ratings of Apps?

Gemma Catolino
Department of Computer Science, University of Salerno, Italy

ABSTRACT

In the past, bad code quality has been associated with higher bug-proneness. At the same time, the main reason why mobile users negatively rate an app is due to the presence of bugs leading to crashes. In this paper, we conducted a preliminary investigation on the extent to which code quality metrics can be exploited to predict the rating of mobile apps. Key results suggest the existence of a relation between code quality and rating; indeed, we found that inheritance and information hiding metrics represent important indicators and therefore should be carefully monitored by developers.

CCS CONCEPTS

• **Software and its engineering** → *Software creation and management*; • **General and reference** → **Mobile Apps**;

KEYWORDS

Mobile Applications, Metrics, Software Quality

ACM Reference Format:
Gemma Catolino. 2019. Does Source Code Quality Reflect the Ratings of Apps? . In *Proceedings of IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft '18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, *"all we need is at our fingertips thanks to mobile apps"*. No doubt, their advent plays an indispensable role in our society, indeed their growth and popularity is the proof: looking at the app-markets, *iOS app store* counts around 2 million applications while *Google play store* has over 2.2 million apps and these numbers are set to rise in upcoming years. Such a growth convinced several industrial companies in creating specialized teams in order to enter in the market of apps; this led to fierce competition between companies in order to release mobile apps with high quality and successful. In order to guarantee their success, companies usually follow trend of markets, *e.g.,* categories more downloaded, features more appealing. This was also confirmed by the research community that showed how the choice of features influences the success of mobile apps, based on their rating [22]. As for their quality, mobile apps as well as traditional software require proper monitoring of source code quality during both development and maintenance phases [3].

MOBILESoft '18, May 2018, Gothenburg, Sweden

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

An external indicator of quality is the mechanism that allows end users to rate apps using scores and reviews [3, 4], but it seems to be not enough. Indeed, as shown by Palomba *et al.* [19, 20], the quality of source code has a noticeable impact on the bug-proneness of a system. Moving this concept in the mobile field, the presence of bugs in mobile apps might possibly lead to malfunctions, *e.g.,* crashes; as shown in the study of Ickin *et al.* [15] this represents one of the most relevant factors that causes low rating and in some extreme cases the uninstall of a mobile app.

Starting from these findings we want to understand whether code quality metrics can be exploited to predict the commercial success of mobile apps, *i.e.,* based on their rating — the number of stars. More specifically, in this paper, the goal is to empirically evaluate the predictive power of code quality metrics when determining the rate of a mobile app. Only one previous study by Corral *et al.* [9] tried to perform this analysis using only CK Metrics [8] on 100 mobile apps showing marginal results.

For this reason, in our study we built (i) a prediction model that takes into account a large set of metrics *i.e.,* 12 code metrics (*e.g.,* CK Metrics [8], Android Metrics [12]) and 9 code smell types [11] and (ii) a prediction model where we apply a feature selection algorithm, *i.e., Wrapper* [17], in order to extract only the relevant features starting from the initial set of metrics considered. We assessed and compared the performance of the considered models on a dataset of 439 mobile apps. Key results of our study indicate how (i) the model where we applied the *Wrapper* algorithm performed better than the other (the value of *F-Measure* improved of 9%) and (ii) inheritance and information hiding metrics should be carefully monitored during the development and maintenance activities of mobile apps as a way to control their rating.

2 STUDY DESIGN

The *goal* of the study is to evaluate the usefulness of code quality metrics when predicting the rating of a mobile app, with the *purpose* to understand whether code quality can be actually exploited to provide developers with continuous information on the rating of their apps. The *quality focus* is on the accuracy of prediction models based on code quality attributes, while the *perspective* is of mobile app developers and tool vendors: (i) The first ones interested in understanding the value of code quality; (ii) The second ones interested in how to better support developers.

As for the *context* of the study, we relied on a publicly available dataset previously defined in the work by Grano *et al.* [12]. This includes information about (i) rates (from 1 to 5), (ii) reviews (extracted from the *Google Play Store*), (iii) code metrics, and (iv) presence of code smells of 630 open-source Android apps mined from F-Droid and having different size and scope. The dataset contains 395 unique apps, while multiple versions are available for most of them. In our study, we selected all the apps having at least two versions, leading to a selection of 439 total apps. All the pieces of information are reported at app-level, meaning that the dataset

Table 1: Code Metrics used in the study

Metrics	Description
Number of Byte-code Instructions (NBI)	This metric counts the total number of *smali* byte-code instructions, ignoring comment lines and blank lines.
Number of Classes (NOC)	This metric computes the number of classes within the app package.
Number of Methods (NOM)	This metric estimates the amount of methods within the app package.
Instructions per Method (IPM)	This metric is computed by the proportion between the total number of instructions (*i.e.,* NBI) and the total number of methods (*i.e.,* NOM).
Cyclomatic Complexity (CC)	This is a complexity metric introduced by Thomas McCabe. This metric measures the number of linearly independent paths contained in the control flow of the program.
Weighted Methods per Class (WMC)	This is a complexity metric introduced by Chidamber and Kemerer. The WMC metric is the sum of the complexities of all class methods.
Number of Children (NOCH)	This metric indicates the number of immediate sub-classes subordinated to a class in the class hierarchy. Greater is the number of children, greater is the reuse, but if a class has a large number of children, it may require more testing of the methods in that class.
Depth of Inheritance Tree (DIT)	This metric indicates the depth (*i.e.,* the length of the maximal path from the node representing the class to the root of the tree) of the class in the inheritance tree. Deeper trees constitute greater design complexity, since more methods and classes are involved.
Lack of Cohesion in Methods (LCOM)	This metric indicates the level of cohesion between methods and the level of cohesion is retrieved by calculating the number of access toattributes of a class.
Coupling Between Objects (CBO)	This metric indicates the dependency degree of a class by another one.
Percent Public Instance Variables (PPIV)	This metric indicates the ratio of variables introduced by a public modifier.
Access to Public Data (APD)	This metric counts the number of accesses to public or protected attributes of each class.

Table 2: Code Smells used in the study

Code Smell	Description
Bad Smell Method Calls (BSMC)	Research Community individuated 10 Android methods throwing exceptions that can cause app crashes. These methods have to be invoked in a try-catch block.
Swiss Army Knife (SAK)	A Swiss army knife is a class with numerous interface signatures, resulting in a very complex class interface designed to handle a wide diversity of abstractions.
Long Method (LM)	Long methods are implemented with much more lines of code than other methods.
Member Ignoring Method (MIM)	In Android, when a method does not access an object attribute, it is recommended to use a static method because they are about 15%–20% faster than a dynamic invocation.
No Low Memory Resolver (NLMR)	When the Android system is running low on memory, the system calls the method onLowMemory() of running activities, which are supposed to trim their memory usage. If this method is not implemented by the activity, the Android system kills the process in order to free memory, and can cause an abnormal termination of programs.
Blob Class (BLOB)	A Blob class is a class with a large number of attributes and/or operations.
Internal Getter/Setter (IGS)	On Android, fields should be accessed directly within a class to increase performance. The usage of an internal getter or a setter converts into a virtual invoke, which makes the operation three times slower than a direct access.
Leaking Inner Class (LIC)	In Java, non-static inner and anonymous classes are holding a reference to the outer class, whereas static inner classes are not. This could provoke a memory leak in Android systems.
Complex Class (CCA)	A complex class is a class containing complex methods.

contains the average of each metric computed over all the classes composing an app. It is important to note that all the metrics used in this study are available in this dataset.

We formulated the following research questions:

RQ1: *To what extent can source code quality metrics be exploited to predict the rating of mobile apps?*

RQ2: *Can the application of a feature selection technique have an impact on the performance of the rate prediction models?*

As detailed in the following, the first research question **RQ1** is aimed at investigating the contribution given by code quality metrics and code smells, within a prediction model, in evaluating the rating of mobile apps in terms of rating. Finally, in **RQ2** we aimed at measuring the actual gain provided by the application of

a feature selection technique in improving the performance of the rate prediction model.

To answer our research questions, we first need to design the rate prediction model for mobile apps. This step requires the definition of several aspects such as (i) the selection of the independent variables to use in the model, (ii) the formulation of the dependent variable that the model will predict, (iii) the choice of a feature selection technique to avoid multi-collinearity (applied only for **RQ2**), and (iv) the definition of training and validation strategies.

Independent Variables. We relied on 12 code quality metrics (*e.g.,* CK Metrics [8], Android Metrics [12]) as well as the 9 code smell types (*e.g.,* Blob [11, 19, 21]). The reason behind our choice is based on the fact that previous studies showed how code metrics but also code smells are correlated with the *"defectiveness"* of both standard and mobile apps [4, 13, 14, 21]. The larger the number of bugs the larger is the probability to evaluate the mobile app negatively or in the worst case to uninstall it, as shown in a survey of Ickin *et al.* [15]. The metrics used are available in Table 1 and 2.

Dependent Variable. Starting from the dataset [12], we firstly extracted the metric values for the 439 mobile apps (*i.e.,* the independent variables). Besides the value of metrics, each app is associated with a rate, which is the average ratings assigned to them by the end users. We used such information as the dependent variable. In particular, we discriminate high- and low-rated apps using the heuristic by Khalid *et al.* [16]: Apps whose average rating was strictly higher than 3.5 were considered as high-rated, otherwise, they were marked as low-rated.

Training Strategy. Once defined the set of independent and dependent variables, we focused on training a machine learning model. In this step, we also took into account the data unbalance problem since the percentage of app classified as high rated (*i.e.,* 365) was larger than that one classified as low rated (*i.e.,* 74). To handle this issue, we apply *Synthetic Minority Over-sampling Technique* (SMOTE), proposed by Chawla *et al.* [7] to make the training set uniform between high- and low-rated instances. Since this approach can be run once per time to over-sample a certain minority class, we repeated the over-sampling until all the classes considered have a similar number of instances. Afterward, we built a *Random Forest* [10] model: this choice was driven by previous findings showing that it is among the top machine learning approaches to use, since it is robust to noisy data [13, 18].

Validation Strategy. We adopted the *10-Fold Cross Validation* [23]. This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling. A single fold is used as a test set, while the remaining ones are used as a training set. The process was repeated 10 times, using each time a different fold as a test set. Then, the model performances were reported using the mean achieved over the ten runs.

Feature Selection Technique. Although *Random Forest* should automatically select the most appropriate features to use [10], in **RQ2**, we tested whether the application of a second step of feature selection might boost the overall performance of the approach. To this aim, we made sure to avoid model over-fitting by considering the best combination of predictors given as output by the *Wrapper* algorithm [17] that considers the selection of a set of features as a search problem, where different combinations are prepared, evaluated and compared to other combinations. In particular, we followed the same steps described above in order to build the prediction model but with the difference that in this model we discarded the features considered not relevant by the *Wrapper algorithm*. As a result, only 4 metrics were considered meaningful *i.e., Number of Children* (NOCH), *Depth of Inheritance Tree* (DIT), *Percent Public Instance Variables* (PPIV) and *Access to Public Data* (APD). These metrics were used to train the Random Forest classifier.

Evaluation Metrics. Finally, to evaluate and compare the performance of the two models we adopted five widely used metrics, such as *precision* (P), *recall* (R), *F-Measure* (F-M), *Matthews correlation coefficient* (MCC), and *Area Under the ROC Curve* (AUC-ROC) [2]. In particular, *MCC* allows us to gauge how well our classification model/function is performing, while the latter quantifies the overall ability of a prediction model to discriminate between *high* and *low* rate. The closer $AUC - ROC$ to 1, the higher the discrimination power of the classifier, while an $AUC - ROC$ closer to 0.5 means that the model is not very different from a random one.

3 RESULTS

Table 3 reports the overall performance achieved by the experimented models. The first row of the table shows the performance of the model without the second step of feature selection (NFS), while the second row reports the results for the model where we applied the Wrapper algorithm (FS).

Table 3: Overall Performance (in %) of the prediction model.

MODEL	P	R	F-M	MCC	AR
NFS	56	53	54	8	53
FS	65	61	63	26	65
IMPROVEMENT	**+9%**	**+8%**	**+9%**	**+18%**	**+12%**

The first thing that leaps to the eyes is the large difference between the performance achieved by NFS and the FS model.

Indeed, using a feature selection technique (FS model) the value *F-Measure* improves of 9%. This is mainly due to the improvement of the overall *Recall i.e.,* 8%. From a practical point of view, this means that the model is able to discover a greater number of true positive instances, thus being able to better discriminate between low- and high-rated apps. The improvement is observable for all the other evaluation metrics, especially looking at the values of *MCC* and *AUC-ROC* that improve of 18% and 12%, respectively; this means that the FS model is a robust classifier and is able to recognize and distinguish between high and low rated instances. Thus, we claim that a second step of feature selection can improve the prediction capabilities, even in cases where the classifier has a way to reduce the risk of multi-collinearity.

Looking more in-depth into the results, we focused our attention on the metrics that the *Wrapper* technique selected as more relevant. Interestingly, NOCH and DIT belong to the category of inheritance metrics, *i.e.,* indicators of the extent to which a class makes use of inheritance mechanisms. Yu *et al.* [24] showed that such inheritance metrics can strongly affect the bug-proneness of software systems. Bearing in mind the importance of bug prevention and bug detection associated to the rating of mobile apps, it is reasonable to think that inheritance-based metrics play an important role as a predictor of high/low rating of a mobile app.

Similarly, PPIV and APD are metrics that quantify the degree a class exposes its data to other classes: As shown by Aggarwal *et al.* [1], low information hiding leads to higher bug-proneness. This concept is very close to the smell *"Class data should be private"* [11]. From a practical point of view, a public data can be accessed from outside the class. If something goes wrong, a bug can be anywhere, and so in order to track down the problem, it might be necessary have to look at quite a lot of code, this lead to increase the effort and cost of the software [11, 19]. As a conclusion, this means that a "negative" value of the four more relevant metrics might lead to an increase of the probability that a system will contain bugs. As a likely consequence, an app might be rated lower than apps that keep under control the value of such metrics [20].

As a consequence of our findings, *we argue that code quality and, in particular, inheritance and information hiding metrics should be carefully monitored during the development and maintenance activities of mobile apps as a way to control their rating.*

4 DISCUSSION AND FUTURE IMPLICATION

With mobile app growing in size and complexity, and developers having to work under frequent deadlines (especially in mobile context), the introduction of bugs does not really come as a surprise. Unfortunately, Ickin *et al.* [15] showed how the presence of malfunctions in a mobile app lead to assigning low rating by end users and in extreme case to its uninstall. Since user acts as an external indicator of quality only when the app is released trough rating and reviews, there is the need of monitoring the quality of a mobile app since its development in order to guarantee its high rating, so its success. To deal with this, the research community has extensively investigated methods and approaches that try to match users reviews with source code [20], but what developers need is something to monitor before a mobile app is released [3, 4]. Starting from this conjecture, we tried to exploit code metrics [8], but also code smells [11] in order to predict the rating of mobile apps (*i.e.,* high rated and low rated). To this aim, we built (i) a prediction model that takes into account 12 code metrics and 9 code smell types and (ii) a prediction model where we apply a feature selection algorithm, *i.e.,* Wrapper [17], in order to extract only the relevant features starting from the initial set of considered metrics, evaluating if the performance of the model increase. Finally, we assessed and compared the performance of the considered models on a dataset of 439 mobile apps. Our study showed how code quality metrics might be exploited in order to predict the rating of mobile apps; indeed the model where we applied the Wrapper algorithm performed better than the other (the value of F-Measure improved of 9%) and (ii) inheritance and information hiding metrics should be carefully monitored during the development and maintenance activities of mobile apps as a way to control their rating. We are aware that there are other metrics that could be exploited, for this reason, we are actually working on the integration of additional features *i.e.,* developer-related factors; the reason behind our choice is based on the fact that we already showed in the previous studies their benefits, in different topics [3–6]. Indeed, from a practical perspective, a factor like developers' experience could be useful within a rate prediction model and in building a profiling approach of developers in order to assign task during both development and maintenance phases of a mobile app, this might guarantee a good quality of a mobile app during its evolution.

REFERENCES

[1] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. 2009. Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study. *Soft. process: Improvement and practice* (2009).

[2] Ricardo Baeza-Yates, Berthier de Araújo Neto Ribeiro, et al. 2011. *Modern information retrieval.* New York: ACM Press; Harlow, England: Addison-Wesley,.

[3] Gemma Catolino. 2017. Just-in-time bug prediction in mobile applications: the domain matters!. In *Proceedings of the 4th International Conference on Mobile Soft.Engineering and Systems.*

[4] Gemma Catolino, Dario Di Nucci, and Filomena Ferrcci. 2019. Cross-Project Just-in-Time Bug Prediction for Mobile Apps: An Empirical Assessment. In *6th IEEE/ACM International Conference on Mobile Software Engineering and Systems - to appear.*

[5] Gemma Catolino, Fabio Palomba, Andrea De Lucia, Filomena Ferrucci, and Andy Zaidman. 2018. Enhancing Change Prediction Models using Developer-Related Factors. *Journal of Systems and Software* 143, 9 (2018), 14–28.

[6] Gemma Catolino, Fabio Palomba, Damian Andrew Tamburri, Alexander Serebrenik, and Filomena Ferrucci. 2019. Gender Diversity and Women in Software Teams: How Do They Affect Community Smells?. In *41st International Conference on Software Engineering, Software Engineering in Society.*

[7] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* (2002).

[8] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[9] Luis Corral and Ilenia Fronza. 2015. Better code for better apps: a study on source code quality and market success of Android applications. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems.* IEEE, 22–32.

[10] Tom Dietterich. 1995. Overfitting and Undercomputing in Machine Learning. *ACM Comput. Surv.* (1995).

[11] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[12] Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado A Visaggio, Gerardo Canfora, and Sebastiano Panichella. [n. d.]. Android apps and user feedback: a dataset for software evolution and quality improvement. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics.*

[13] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304.

[14] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An empirical study of the performance impacts of android code smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems.* ACM, 59–69.

[15] Selim Ickin, Kai Petersen, and Javier Gonzalez-Huerta. 2017. Why Do Users Install and Delete Apps? A Survey Study. In *International Conference of Software Business.* Springer, 186–191.

[16] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. 2015. What do mobile app users complain about? *IEEE Soft.* (2015).

[17] Ron Kohavi and George H John. 1997. Wrappers for feature subset selection. *Artificial intelligence* (1997).

[18] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ACM/IEEE 30th International Conference on Soft.Engineering.* IEEE.

[19] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2017. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Soft. Engineering* (2017).

[20] Fabio Palomba, Mario Linares-Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2018. Crowdsourcing user reviews to support the evolution of mobile apps. *Journal of Systems and Soft.* (2018).

[21] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. 2017. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering* (2017).

[22] Federica Sarro, Mark Harman, Yue Jia, and Yuanyuan Zhang. 2018. Customer rating reactions can be predicted purely using app features. In *2018 IEEE 26th International Requirements Engineering Conference (RE).* IEEE, 76–87.

[23] Mervyn Stone. 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society. Series B (Methodological)* (1974), 111–147.

[24] Ping Yu, Tarja Systa, and Hausi Muller. 2002. Predicting fault-proneness using OO metrics. An industrial case study. In *6th Eur. Conference on Soft. Maintenance and Reengineering.* IEEE.