What Are the Characteristics of High-Rated Apps? A Case Study on Free Android Applications

Yuan Tian*, Meiyappan Nagappan†, David Lo*, and Ahmed E. Hassan‡
*Singapore Management University, Singapore
{yuan.tian.2012,davidlo}@smu.edu.sg
†Rochester Institute of Technology, Rochester, USA
mei@se.rit.edu
‡Queen's University, Kingston, Canada
ahmed@cs.queensu.ca

***Abstract*—The tremendous rate of growth in the mobile app market over the past few years has attracted many developers to build mobile apps. However, while there is no shortage of stories of how lone developers have made great fortunes from their apps, the majority of developers are struggling to break even. For those struggling developers, knowing the "DNA" (i.e., characteristics) of high-rated apps is the first step towards successful development and evolution of their apps.**

In this paper, we investigate 28 factors along eight dimensions to understand how high-rated apps are different from low-rated apps. We also investigate what are the most influential factors by applying a random-forest classifier to identify high-rated apps. Through a case study on 1,492 high-rated and low-rated free apps mined from the Google Play store, we find that high-rated apps are statistically significantly different in 17 out of the 28 factors that we considered. Our experiment also shows that the size of an app, the number of promotional images that the app displays on its web store page, and the target SDK version of an app are the most influential factors.

I. Introduction

In recent years, the "mobile app economy", which refers to the economy that has been created around the development and delivery of software applications for smartphones and tablets [9], has been growing rapidly. According to a 2013 report[1], the global app economy was worth $53 billion in 2012, and is expected to rise to $143 billion in 2016. The fast growing mobile app market continues to attract more and more developers with more than 2.3 million developers already having their own apps published on app stores.[2] However, while few developers have made great fortunes from their app, the majority of the app developers are still struggling to break even [40]. Similar to a DNA that determines an organism's structure, there might be a specific "DNA" associated with a successful application. Therefore, for those struggling developers, knowing the "DNA" (i.e., characteristics) of existing successful apps can be one of the initial steps towards building successful apps.

Measuring the success of a software system is difficult as there is neither a universal metric nor a ranking scheme [12]. Similarly, for a mobile app, various values could be regarded as indicators of app success. In this study, same as previous studies [2], [14], [17], [23], [43], we choose app rating as proxy for app success. There are other possibilities; for example, one could consider using number of downloads to identify successful apps. However, many users may download an app without using it. Furthermore, in the Google Play store, for each app, the number of downloads is shown as a range (e.g., from 100,000 to 500,000), instead of an actual number, making it hard to differentiate apps that fall in the same range. Another possibility is to analyze comments that users may post while rating an app. However, many users rate without giving comments, and accurate automated identification of sentiments from comments is still beyond the reach of state-of-the-art natural language processing tools, c.f., [44].

Recent papers on analyzing factors relevant to app rating find that there are relationships between app rating and factors such as change and fault proneness of adopted Android APIs, complexity of user interface, and application churn [2], [14], [17], [23], [43]. Although these prior studies give some initial hints about the characteristics of high-rated apps, a slew of additional factors can be considered. For instance, code complexity is claimed to be significantly associated with defects in software [37]. Thus code complexity of a mobile app could potentially impact the rating of an app because more defects might lead to poor rating. Besides code complexity, many other characteristics could be investigated by mining app store. In addition, given the fact that app rating is affected by multiple factors, how these factors integrate together to impact app rating is still unknown. To fill the gaps in current research, in this study, we aim to examine much more factors that are potentially associated to app rating and investigate what are the most influential factors for identifying high-rated apps.

We focus on factors that developers can control and strive to improve or change. For example, high-rated apps might have larger volume of downloads, however number of downloads is not considered as a factor in this work because it can not be controlled by the developers. Our 28 factors (c.f., Section II) are mined from information recovered from an app's binary, i.e., its Android Application package (APK), and its Google Play store page. Our factors are grouped

[1]http://www.developereconomics.com/reports/app-economy-forecasts-2013-2016/

[2]http://www.forbes.com/sites/tristanlouis/2013/08/10/how-much-do-average-apps-make/

along eight dimensions: 1) size of app, 2) complexity of code, 3) dependence on libraries, 4) quality of library code, 5) complexity of user interface, 6) requirements on users, 7) marketing effort, and 8) category of app. We crawled 1,492 apps from the Play store and used our factors to explore the following two research questions:

- **RQ1: Is there a relationship between each factor and app rating?**
  We collect 746 high-rated apps and 746 low-rated free apps from the Google Play store and calculate the values of the 28 factors for each app. For each factor, we apply the Mann-Whitney U test to examine whether high-rated apps are statistically significantly different from low-rated ones. We also compute the effect size of the differences. We find that in 17 out of the 28 factors, high-rated apps are statistically significantly different from low-rated ones. Generally, high-rated apps have larger sizes, more complex code, more requirements on users, more marketing efforts, more dependence on libraries, and adopt higher quality Android APIs.
- **RQ2: What are the important factors that could indicate, with high probability, that an app will be high-rated?**
  To compare the importance of the factors, we learn a random-forest classifier using the factors as input features to identify whether an app will be high-rated or not. Correlation and redundancy analyses are applied to better model the integrated impact of the factors on app rating. We find that the install size of an app (from the size dimension), the number of promotional images (from the marketing effort dimension), and the target SDK version (from the requirements on user dimension) are the top 3 most influential factors.

The structure of this paper is as follows. In Section II, we present the factors that might impact the likelihood of an app being a high-rated app. We describe our factor extraction methodology and data processing steps in Section III. We present and analyze the results for RQ1 and RQ2 in Section IV. We discuss additional points in Section V. Related work is described in Section VI. We finally conclude and briefly mention future directions in Section VII.

II. Factors Potentially Affecting App Ratings

In this study, we consider 28 factors along eight dimensions, that might be correlated with app rating. We describe the meaning and rationale of each factor in Table I.

Size of App includes factors that capture the size of an app in various ways. Larger apps might contain more features or better functionality. Thus they might have higher ratings. However, larger volume of code imply higher probability to contain a bug [48], and hence might lead to lower ratings. Note that, in Android, an APK file contains the code of an app, as well as the resources (e.g., images or other media files) that an app needs, thus install size captures additional information rather than simply the size of the code.

Complexity of Code includes factors that represent the complexity of an app code in various ways. Typically, high complexity code has a higher chance to contain more bugs [37]. More bugs might cause more crashes while users are using the app which can result in lower ratings. In this dimension, we use six well-known Chidamber and Kemerer's object-oriented complexity metrics (CK metrics) [6], along with two other metrics, average afferent coupling of classes (CA) [25], and number of public methods (NPM) [1] to capture code complexity.

Dependence on Libraries includes factors that represents mobile app's dependence on library code. Compared with traditional software, Android apps are reported to be heavily dependent on libraries including the Android base libraries, and other third-party libraries [41]. However, too much dependence on the APIs might lock an app into the platform or third party libraries, which might impact the quality of the code [42]. For example, the rapid evolution of Android APIs makes it hard for app developers to keep their app working on newer API versions, leading to defects and inconsistencies that impact the end user, which might lead to poor rating.

Quality of Library Code includes factors that are related to change and fault proneness of library code used by an app. They are reported to be related with app rating, i.e., high-rated apps depend on more stable and reliable libraries [2], [23].

Complexity of UI includes factors that are related to the user interface of an app. Inappropriate use of input and output elements in a mobile screen of limited size might make an app less user-friendly, which might lead to low ratings. For instance, a user might feel frustrated, if too many input fields need to be filled to complete a task or overwhelmed if too many output elements are shown at the same time.

Requirements on Users includes factors related to requirements that a user must meet before they can use an app. They are related to the specification and settings of the user's device. They might impact app rating in various ways. Larger minimum sdk version and more required device features might suggest that the app has provided more complex features. Larger target sdk version might suggest that developers have updated their app towards latest sdk version. Number of permission might indicate the risk of causing privacy problems, thus it might impact app rating.

Marketing Effort includes factors related to app-specific information that is shown on the application store to entice users to download the app. They are shown to users before they install the app, and thus give the first impression to users. For this dimension, we consider app description and promotional images.

Category of App is a categorical variable to group a set of related apps together. The category of an app might impact how other factors affect ratings and might influence user's expectation of an app (e.g., users might be more forgiving for a bug in a game app versus a financial app).

TABLE I: Factors potentially affecting the rating of an app.

Dimension	Factor (Name)	Explanation	Rationale
Size of App	Install size (installSize).	Binary size of the APK file (measured in KB)	Larger install size, number of classes, number of project classes might indicate more features or better functionality. Activities and services are specific components of Android apps: an activity provides a screen for users to interact, whereas a service is used to support long-running operations and interaction in the background. All of them suggest more elaborate functionality.
	Total classes (num_class).	Total number of classes (including library code).	
	App classes (num_projectClass).	Total number of app specific classes.	
	Number of activities (num_activity).	Total number of activities defined in the AndroidManifest.xml file.	
	Number of services (num_service).	Total number of services defined in the AndroidManifest.xml file.	
Complexity of Code	Weighted methods per class (wmc_mean).	Mean of the number of methods in each class.	Chidamber and Kemerer's object oriented complexity metrics (CK metrics), along with CA and NPM, are metrics to measure code complexity [36]. They are also reported to be relevant to code quality, more complex code tends to have more bugs [37].
	Depth of inheritance tree (dit_mean).	Mean of the length of the inheritance chain of each class.	
	Number of children (noc_mean).	Mean of the number of immediate sub-classes of each class.	
	Coupling between object classes (cbo_mean).	Mean of the number of classes with which each class is coupled.	
	Response for a class (rfc_mean).	Mean of the number of different methods that can be executed when an object of each class receives a message.	
	Lack of cohesion in methods (lcom_mean).	Mean of the number of pairs of methods in each class that do not share at least one field.	
	Afferent coupling (ca_mean).	Mean of the number of other classes that depend upon each class.	
	Number of public methods (npm_mean).	Mean of the number of public methods in each class.	
Dependence on Library	Total dependency on Android libraries (dep_android).	Total number of calls to libraries that start with "android.".	Library code might bring more elaborate functionality. However, too much dependence on the APIs might impact the quality of the code [42].
	Total dependency on third-party libraries (dep_third).	Total number of calls to third party libraries.	
	Percentage dependency on Android libraries (dep_per_android).	Percentage of calls to libraries that start with "android.".	Similar with dep_android and dep_third, but these two factors consider proportion.
	Percentage dependency on third-party libraries (dep_per_third).	Percentage of calls to third party libraries.	
Quality of Library Code	Change of used Android API (api_change).	Mean number of methods changed in the used Android API – see [2], [23].	Change and defect-proneness of adopted Android APIs might represent quality of the used Android APIs [2], [23].
	Faultiness of used Android API (api_bug).	Mean number of bugs in the used Android API – see [2], [23].	
Complexity of UI	Input elements per layout (ui_input).	Mean number of input elements per layout.	Complexity of user interface (in terms of input elements, and output elements) might impact the success of an Android application [43].
	Output elements per layout (ui_ouput).	Mean number of output elements per layout.	
Requirements on Users	Minimum SDK version (minSDK).	The minimum SDK version required for the app to run.	High minimum SDK version suggests that apps are frequently updated to exploit the latest features provided by the newest versions of the SDK.
	Target SDK version (targetSDK).	The SDK version that the app targets. If not set, the default value equals to minSDK.	A high target SDK version suggests that developers have tested their app on recent SDKs.
	Required device features (device_feature).	Number of required features from user's device (e.g., camera).	Required device features might indicate the functional complexity of the app.
	Required user permission (user_permission).	Number of permissions needed from user.	Some user might be sensitive to the number of permissions, due to privacy concerns [5].
Marketing Effort	Length of description (lenDescription).	Number of words appearing in the description of the app on its Play store page.	Users might prefer clear function presentation and more features.
	Promotional images (num_img).	Number of images shown on the app's store page.	
Category of App	App category (appCategory)	Category of the app.	Users might have differing expectations of apps based on their category.

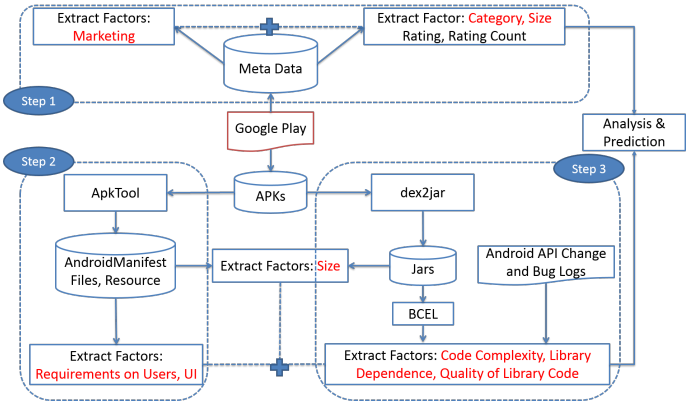

Fig. 1: Overall framework to calculate factor values. Information on Google Play (app's APK and store page), Android APIs' change history and bug-fix logs are the inputs to the whole process.

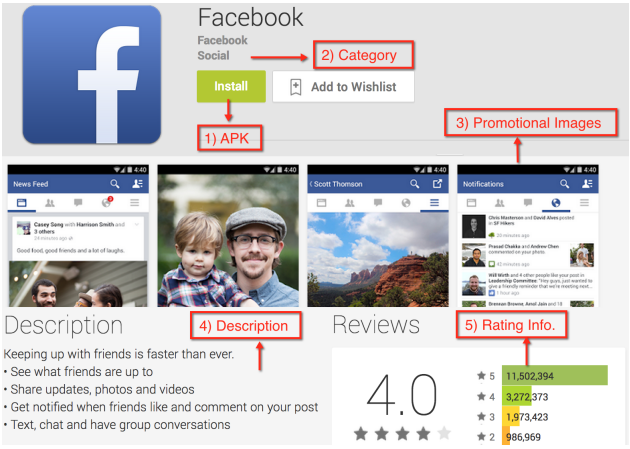

Fig. 2: A sample Google Play store's page of an app.

III. Factor Extraction and Data Collection

In this section, we first describe our methodology to extract the factors that we defined in Section II. We then describe how the dataset that we use to answer the two research questions is collected.

A. Factor Extraction

As shown in Figure 1, the overall process of our approach contains three major steps:

Step 1: Process app's store page. We crawl an app's page in Google Play and extract information in it. Figure 2 shows a sample store page of an app. Category, install size, and rating information can be directly read from the crawled page. Factors in the marketing effort dimension are also easy to extract by processing the app's description and images in the page. The APK of an app could be downloaded from the store.

Step 2: Process app's AndroidManifest.xml file and resources. We extract an app's AndroidManifest.xml files and the resources that the app uses by processing the APK file using "apktool"[3]. AndroidManifest.xml file is a special file that every Android app must have in its root directory. Factors in the **requirements on users** dimension and some of the **size** factors (i.e., num_activity and num_service) can be easily collected from the AndroidManifest.xml file. For instance, required user permissions are defined in elements named "uses-permission" in the xml file. The root directory also contains a "resource" folder which contains resources, such as images and strings, that the app uses. We use the contents of the "resource" folder to extract UI related factors. The "resource" folder has a subfolder named "layout" which contains XML files that define the layout of the app's UI. We extract input and output fields from these XML files following the process described by Taba et al. [43]. We then count the average number of input and output components per layout.

Step 3: Process app's class files and libraries. Factors in the code complexity, library dependence and quality of library code dimensions, and some factors in the size dimension (i.e, num_class and num_projectClass) require code analysis.

To compute code complexity factors, we first extract bytecode files (i.e., .class files) from the app's APK file using the dex2jar2 tool[4]. Next, we use a tool named "ckjm" [35] to collect values of the eight code complexity metrics. In the computation of the complexity metrics at the class level, both project classes and library classes are considered. To heuristically identify whether a class belongs to the project, we check whether the package name is the same as the one defined in the AndroidManifest.xml file. The same process is also used by Linares-Vásquez et al. [24].

To compute library factors, we need to extract dependencies between project files and libraries. To capture these dependencies, we analyze Java bytecode files using the Apache Commons BCEL Java library[5]. The BCEL library transforms the bytecode files into readable JVM instructions, which can be analyzed to identify method invocations indicating dependencies. To differentiate dependencies to Android libraries and other third-party libraries, we use the following heuristic: we regard a library dependency as an Android library dependency if the package of the called method starts with "android.", we regard other dependencies as third-party library dependencies. To compute change and fault-proneness of Android libraries, we make use of the publicly available data provided by Linares-Vásquez et al.[6].

B. Data Collection and Filtering

We randomly selected 10,000 apps from the list of apps that were crawled by Dienst and Berger in 2011 [10]. These apps cover all the categories in Google Play store and they have a wide range of ratings (i.e., 1.3 to 5 stars). For each of them, we perform the factor extraction steps described in Section III-A. ApkTool and dex2jar fail for 14 out of these 10,000 application, and thus we omit them in our study. For

[3]https://code.google.com/p/android-apktool/

[4]https://code.google.com/p/dex2jar/

[5]http://commons.apache.org/proper/commons-bcel/

[6]http://www.cs.wm.edu/semeru/data/fse-android-api/

the remaining apps, we further filter them if they satisfy one of the criteria shown below:

1) **They have less than 10 ratings.** We use this criteria to make sure that the average user-rating that each app has is reliable [30].
2) **Their project code does not exist in the package specified in the AndroidManifest.xml file or they have obfuscated code in their project package.** We use this criteria because hidden and obfuscated code have impact on the reliability of the computation of the code complexity factors [24]. We use the same heuristic as Linares-Vásquez et al. [24] to detect obfuscated code: search for a file named "a.class" under the package that contains project code.

Based on the above filtering criteria, we end up with 7,365 applications across 30 store categories. We list a summary of the number of apps and range of ratings for each of the 30 categories in Table II.

Next, we sorted apps based on their ratings, and then consider the top 10% apps with at least 100 ratings as high-rated apps and bottom 10% apps as low-rated apps. In the end, we get 746 high-rated apps and 746 low-rated apps. These apps form the data set for our case study. We present the distribution of ratings and number of downloads of the two groups of high-rated and low-rated apps in Figure 3. Note that as the number of downloads is stored as a range in the Play store, we use the minimum boundary to roughly represent the number of downloads. The median rating of the high-rated apps is 4.571, whereas that of the low-rated apps is 2.867. The median number of downloads for high-rated apps is 100,000, whereas that of low-rated apps is 10,000.

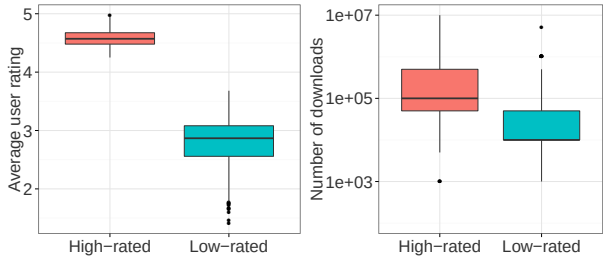

Fig. 3: Boxplot of average user ratings (left side) and number of downloads (right side) for the selected high-rated and low-rated apps.

IV. Case Study Results

In this section, we present and discuss the answer to the two research questions we set out to examine in Section I.

A. (RQ1) Is there a relationship between each factor and app rating?

Motivation: Prior work has examined the relationship between factors such as change and fault-proneness of used Android APIs [2], [23], user interface complexity [43] and rating. However, there are many other factors that also might impact the rating of an app, such as size, design, or marketing of the app. In this research question, we are interested in investigating how each factor presented in Section II is related with the rating. App developers, app store owners and researchers, could use the results of this question to understand how high-rated apps differ from low-rated apps.

TABLE II: Characteristics of the remaining 7,365 apps after filtering.

Category	#Apps (%)	Rating (min-max)
Tools	752 (10.21%)	1.831-4.902
Entertainment	610 (8.28%)	1.787-4.826
Personalization	487 (6.61%)	3.000-4.851
Casual	477 (6.48%)	1.662-4.592
Lifestyle	411 (5.58%)	1.939-4.974
Brain	391 (5.31%)	1.643-4.860
Arcade	374 (5.08%)	1.727-4.682
Education	325 (4.41%)	2.013-4.939
Productivity	264 (3.58%)	1.967-4.737
Books and reference	259 (3.52%)	1.743-4.941
News and magazines	248 (3.37%)	1.593-4.889
Communication	233 (3.16%)	2.537-4.870
Health and fitness	231 (3.14%)	1.468-4.773
Travel and Local	231 (3.14%)	1.780-4.882
Photography	225 (3.05%)	1.759-4.742
Music and audio	221 (3.00%)	2.314-4.876
Finance	209 (2.84%)	1.414-4.769
Social	196 (2.66%)	1.843-4.942
Sports	192 (2.61%)	2.517-4.958
Media and video	161 (2.19%)	1.759-4.950
Cards	145 (1.97%)	2.000-4.626
Shopping	128 (1,74%)	1.676-4.725
Business	119 (1.62%)	2.259-4.682
Transportation	108 (1.47%)	2.255-4.700
Comics	77 (1.04%)	1.944-4.790
Sports games	72 (0.98%)	2.572-4.581
Medial	63 (0.86%)	2.863-4.852
Libraries and demo	52 (0.71%)	1.771-4.976
Racing	52 (0.71%)	2.531-4.545
Weather	52 (0.71%)	2.895-4.610

Approach: We compare the values of each factor between selected high-rated apps and low-rated apps. We first analyze the statistical significance of the difference between the two groups of apps, i.e., high-rated apps and low-rated apps, by applying the Mann-Whitney U test at $p-value = 0.01$ [7]. To show the effect size of the difference between the two groups, we compute Cliff's Delta (or d), which is a non-parametric effect size measure [13]. We use the *effsize* package in R[7] to compute Cliff's d. Following the guidelines in [13], [23], we interpret the effect size values as small for $0.147 < d < 0.33$, medium for $0.33 < d < 0.474$, and large for $d > 0.474$.

Results: Table III shows the factors that have $p - value < 0.01$, and $d > 0.147$ (i.e., statistically significant difference with at least a small effect size). We find that the selected high-rated and low-rated apps have statistically significant differences with at least a small effect size in 17 out of the 28 factors. The effect sizes are small for most of the 17 factors, except for install size, number of images, and target SDK version, which have medium effect sizes.

Now we investigate each dimension one by one. For size

[7]http://softeng.polito.it/software/effsize/

TABLE III: Relationship between each factor and rating. **Dim.** = dimension. **Rel.** = relationship. "+" = high-rated apps have significantly higher value on this factor. "-" = low-rated apps have significantly higher value on this factor. Factors with medium d-value are highlighted in bold.

Dim.	Factor Name	Rel.	d-value
Size	**installSize**	+	**0.363 (Medium)**
	num_class	+	0.260 (Small)
	num_projectClass	+	0.243 (Small)
	num_activity	+	0.239 (Small)
Code Complexity	wmc_mean	+	0.173 (Small)
	cbo_mean	+	0.191 (Small)
	rfc_mean	+	0.198 (Small)
	lcom_mean	+	0.162 (Small)
	ca_mean	+	0.174 (Small)
Library Dependence	dep_android	+	0.244 (Small)
	dep_third	+	0.191 (Small)
Library Quality	api_change	-	0.181 (Small)
	api_bug	-	0.187 (Small)
User Requirement	minSDK	+	0.158 (Small)
	targetSDK	+	**0.354 (Medium)**
Marketing	lenDescription	+	0.271 (Small)
	num_img	+	**0.421 (Medium)**

dimension, four out of the five factors can differentiate high-rated apps from low-rated apps, except for the number of services. Typically, when an app gets larger in size, it becomes more difficult to maintain the app and there is a greater chance for field failures. However, we observe that high-rated apps are usually significantly larger then small apps. Similarly, high complexity is usually related with fault proneness [37]. However, from the results of the code complexity dimension, we observe that high-rated apps tend to have more complex project code than low-rated apps. One possible reason could be that high-rated apps are feature rich in comparison to low-rated apps, and the implementation of these features causes larger size and additional complexity.

For the quality of library code dimension, low-rated apps tend to use APIs from the Android libraries that have higher change and fault proneness. This result is consistent with the findings by Linares-Vásquez et al. [2], [23]. For the requirement on users dimension, high-rated apps have a higher minimum and target SDK requirement. This suggests that compared with low-rated apps, high-rated apps usually are more frequently updated to exploit the latest features provided by the newest versions of the SDK. For the marketing effort dimension, we observe that developers of high-rated apps, tend to use significantly more text in the app description and more images to advertise their apps.

High-rated apps are statistically significantly different from low-rated apps in 17 out of the 28 factors. Generally, high-rated apps are larger with more complex code, more preconditions, more marketing efforts, more dependence on libraries, and they make use of higher quality Android libraries.

B. (RQ2) What are the important factors that could indicate, with high probability, that an app will be high-rated?

Motivation: In reality, app rating is impacted by multiple factors rather than one. Among these, some factors might be more influential on app rating than others, for which developers would need to pay more attention. Therefore, in this research question we are interested in finding these important factors. In order to compare the importance of multiple factors on app rating, we build a random-forest classifier to predict whether an app will be high-rated, given the values of the various factors. The factors that contribute more to the classifier are more important.

Evaluation: The effectiveness of the classifier in correctly predicting high-rated apps represents the prediction power of the factors. In this study, we use F-measure and AUC to measure the effectiveness of the classifier. These metrics are commonly used in classification tasks [8], [46]. F-measure is the harmonic means of precision and recall. Precision measures the correctness of the classifier in predicting whether an app is a high-rated app. Recall measures the completeness of the classifier in predicting high-rated apps. AUC is another commonly-used measure for binary classification problems. AUC refers to the area under the Receiver Operating Characteristic (ROC) curve [16].

Approach: To identify whether an app is a high-rated app or not, we process the factors using the random-forest classifier [4]. We choose the random-forest classifier because it is known to have several advantages, such as being robust to noise and outliers, which means that some non-typical high-rated or low-rated apps in the training data are likely to have little impact on the learned model in our case. Also, it can handle a mixture of categorical and continuous predictor variables. In addition, the classification power of the random-forest classifier has been demonstrated by its application to automate many software engineering tasks [8], [15], [22], [32]. We use the implementation of the random-forest classifier in the **bigrf** R package[8] and set the number of trees as 100.

Before we build a model, we need to perform variable selection because correlated variables might lead to poor models (classifiers) which are hard to interpret [28]. The overall approach contains four main stages:

Stage 1: Correlation analysis. We use a variable clustering analysis, implemented in a R package named **Hmisc**[9], to construct a hierarchical overview of the correlations among the factors. For sub-hierarchies of factors with correlations larger than 0.7, we select only one variable from the sub-hierarchy for inclusion in our models. In particular, out of 28 factors, there are four pairs of variables that have correlations larger than 0.7: 1) api_change and api_bug; 2) num_projectClass and dep_android; 3) wmc_mean and npm_mean; 4) wmc_mean and rfc_mean. We remove api_change, num_projectClass,

[8]http://cran.r-project.org/web/packages/bigrf/bigrf.pdf
[9]http://cran.r-project.org/web/packages/Hmisc/index.html

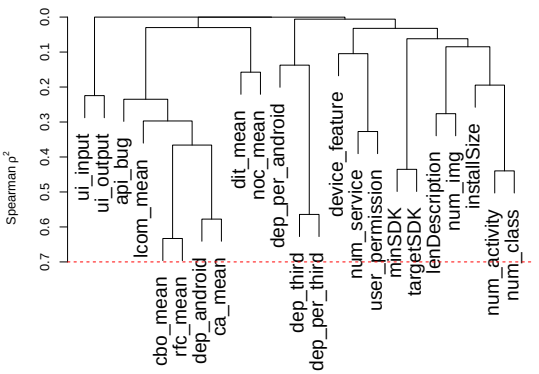

Fig. 4: Clustered factors after correlation analysis.

npm_mean, and wmc_mean. The final hierarchical overview is presented in Figure 4.

Stage 2: Redundancy analysis. Correlation analysis reduces collinearity among the factors, but it may not detect all of the redundant factors, i.e., factors that do not have a unique signal relative to the other factors. Redundant factors in an explanatory model will interfere with one another, distorting the modeled relationship between the factors and predictors. We remove redundant factors by using the implementation provided by the **redun** function in the **rms** R package. In particular, from the leftover 24 variables from the previous step, we remove rfc_mean and dep_per_third because they can be represented using other factors.

Stage 3: Building a random-forest classifier and testing it. After the two stages of removing redundant variables, we have 22 remaining factors: api_change, num_projectClass, npm_mean, wmc_mean, rfc_mean and dep_per_third are removed. We use 10-fold cross validation [11] to evaluate the effectiveness of our model. We randomly separate our data into ten folds and perform ten rounds of experiments. For each round, we select nine folds to train the random-forest classifier, and then apply the learned classifier on the remaining fold. We repeat this process ten times and aggregate the results to measure the overall performance for the whole data.

Stage 4: Calculating factor importance through multiple runs. To identify the most influential factors in our random forest model, we use the **varimp** function in **bigrf** package. This function computes the influence of a factor in the training process, based on out of the bag (OOB) estimates, an internal error estimate of a random forest classifier [47]. The underlying idea is to permute each factor randomly one by one and see whether the OOB estimates will be reduced significantly or not.

For each run of the 10-fold cross validation we get a variable importance value for each factor. In order to determine the factors that are most influential for the whole dataset, we take the values from all 10 runs and apply the Scott-Knott test.[10] This test takes as input a set of distributions (one for each variable) and identifies groups of variables that are statistically significantly different from one another.

[10]http://cran.r-project.org/web/packages/ScottKnott/ScottKnott.pdf

TABLE IV: Scott-Knott test results when comparing the mean rank of feature importance, divided into distinct groups that have a statistically significant difference in the mean ($p - value < 0.05$).

Group	Factor Name	Mean Rank	Highest Rank	Lowest Rank
G1	installSize	1.0	1	1
G2	num_img	2.4	2	3
	targetSDK	2.6	2	3
G3	appCategory	4.3	4	6
G4	api_bug	5.8	4	7
	num_class	6.2	5	8
	dep_android	6.3	4	8
G5	lenDescription	7.8	5	10
G6	cbo_mean	9.5	8	11
G7	ca_mean	11.2	8	13
	dep_third	12.3	9	16
	lcom_mean	12.5	9	18
G8	user_permission	12.9	11	16
	num_activity	13.6	9	17
	minSDK	13.9	10	17
G9	ui_output	16.2	12	19
	dit_mean	16.6	12	19
	dep_per_android	17.4	15	19
	ui_input	17.6	14	20
G10	num_service	20.2	19	21
	noc_mean	20.7	20	21
G11	device_feature	22.0	22	22

Results: The 10-fold cross validation shows that the model can successfully infer high-rated apps leveraging multiple factors with an F-measure of 0.74, and an AUC of 0.81. This result confirms the impact of our proposed factors on app rating because an AUC beyond 0.7 is generally considered reasonably good [18]. Additionally, in the Software Engineering domain, many accepted predictors have AUC values between 0.7-0.8 [8], [22], [45].

Table IV shows the factors ranked by their importance, as determined by the Scott-Knott test. Different groups of variables whose variable importance values are statistically significantly different from other groups of variables are shown in Table IV. The results show that the install size, the number of images on an app's store page, and the target SDK version are the top three most important factors that influence our random forest model. Recall that in the results of RQ1 (see Section IV-A), install size, number of images, and target SDK version are the top three factors that are significantly different between high-rated and low-rated apps. Thus, the results of RQ1 and RQ2 are consistent with each other.

Larger install size means bigger app which is likely to translate to richer features. More number of promotional images in the app's page in the Google Play store implies that users' first impression is important. Moreover, developers usually use images to show various functionalities of an app. More images might translate to more functionalities. Larger target SDK version implies more recent version of the Android SDK. These results suggests that apps in our dataset that are

larger, with more images and targeted to a newer SDK are more likely to have higher ratings. These factors helped most in discriminating high-rated apps from low-rated ones.

The size of an app, the number of promotional images on its store page, and the target SDK are the three most influential factors in determining the likelihood of an app being a high-rated app.

V. Discussion

Comparisons with Past Findings: Our study revisits some of the previous findings on the correlation between some factors (i.e., api_bug, ui_input, ui_output, user_permission) with app rating. Linares-Vásquez et al. find that low-rated apps have method calls to APIs that are more change or fault prone [2], [23]. Our findings agree with theirs; fault proneness of the used APIs is relevant to app rating, as seen in Table III. However, it is not the most important factor (i.e., it is ranked fifth as shown in Table IV). Moreover, the fault proneness of the used APIs is highly correlated with the number of changes of the used APIs (c.f., Stage 1 of RQ2), which suggests that these two factors capture similar information for identifying high-rated apps. Taba et al. find that generally low-rated apps have more complex UI [43], however in our dataset, UI complexity (ui_input and ui_output) is not significantly related with app rating. Chia et al. find that the number of required permissions (user_permission) is only weakly positively correlated with app's popularity [5]. Our results support Chia et al.'s finding; user_permission ranks in the middle (i.e., thirteenth) among the 28 factors.

Compared with previous work, the top three important factors that we find, i.e., install size of an app, number of promotional images, and target SDK version are never mentioned in previous studies. Besides, the experiment result shows that the category of an app, which is used to control the impact of other factors on app rating, is an important factor (i.e., fourth). The associations between these factors and app rating are strong and we recommend to include them as variables in models that can be used to determine high-rated apps (similar to how LOC is used in defect prediction models [22]).

Beside the above mentioned factors, we examine factors from additional dimensions, e.g., code complexity dimension and library dependence dimension, that are reported to be positively related to number of defects inside software [37], [42]. Our initial thought is these factors are negatively associated to app rating, however, we observe the opposite: high-rated apps usually have more complex code and depend more on libraries.

The Power of Considering Multiple Factors: Table V shows the prediction results using only factors from one dimension. We observe that factors from the size dimension performs the best however its performance is still poorer than using factors from all dimensions. A model built using all factors from the eight dimensions can outperform models built using factors from only one dimension by 9.7% (size dimension) - 82.2% (category dimension), and 8.9% (size dimension) - 166.2% (category dimension), in terms of F-measure and AUC, respectively. This result reconfirms our assumption that app rating is impacted by multiple factors.

TABLE V: Prediction results.

Dimension	Recall	Precision	F-measure	AUC
All	0.744	0.728	0.736	0.812
Size	0.663	0.678	0.671	0.746
Code Comp.	0.638	0.630	0.634	0.704
Dependence	0.627	0.628	0.628	0.684
Library Quality	0.574	0.575	0.574	0.612
UI Comp.	0.524	0.572	0.547	0.614
Req. on Users	0.572	0.666	0.616	0.700
Marketing	0.594	0.640	0.616	0.695
Category	0.438	0.374	0.404	0.305

Manual Analysis of Wrongly Identified Cases: We manually examined the reviews of apps that had a good rating, but our approach could not identify them as high-rated apps to understand the reason for the incorrect predictions. We found that often such apps are extremely small, like in the case of com.acdcwallpaper.bogmix. Even though these apps are small, they work smoothly without any crashes or hangs. For example a user of the app com.acdcwallpaper.bogmix commented that 'Nothing to it really... The photos are great and the app works so 5 stars for it'. The fact that there were wrongly predicted apps, implies that a one-size-fits-all global model might not be sufficient. It will be interesting to investigate a composition of local and global models – similar to the approaches presented by Bettenburg et al. [3] and Menzies et al. [26].

Association vs. Causation: The purpose of our study is to examine whether there are factors that are *associated* to app rating, and how they jointly impact app rating. Note that association does not imply causation. Furthermore, since we only consider factors that can be collected by mining the binary of apps and their profile pages, we do not have a complete picture. This limitation is shared by previous studies that analyze relationships between app characteristics and success (or popularity) by mining code and software repositories [2], [14], [23], [29], [43]. Still, these studies along with ours are the first steps towards a better understanding of the "DNA" of successful apps. They suggest factors that developers need to take note of while developing and evolving their apps. Future studies may take the reported findings and go beyond mining code and software repositories, e.g., by interviewing a large number of developers and users, to more rigorously validate the findings. Additionally, more advanced statistical analysis, e.g., causality analysis [33], can also be employed.

Threats to Validity: Threats to internal validity relate to the conditions under which experiments are performed. To build the random forest classifier, we set the number of trees to 100, which is similar to the setting used by a prior work where random forest was also used [8]. Another threat to internal validity relates to the way the high-rated and low-rated apps considered in this work are selected. Firstly, some

apps may have high ratings due to fake ratings (e.g., ratings made by paid raters, ratings by the app developers themselves, etc.). To address this issue, assuming that the number of fake ratings are not too many, we filter high-rated apps with less than 100 ratings. Secondly, to not bias our findings to a particular category, we pick high and low-rated apps across many categories.

Threats to external validity relate to the generalizability of our findings. In this work, we only consider free apps, since we can get access to their APKs for free. It is unclear whether the same observations still hold for paid apps. However in 2013, it was estimated that 80% of all apps were free apps and 91% of all downloads in the Google Play store were downloads of free apps.[11] Our study is performed on a data set composed of 1,492 Android apps. Our results might not generalize to all apps on the Android platform (or other mobile platforms). Regarding the factors that we have considered, there might be additional factors that could be more relevant to app rating. Still, the results of our prediction experiment shows the power of the 28 factors considered in this work. In the future, to further reduce these threats, we plan to investigate additional apps and consider more factors.

VI. Related Work

A few studies have examined the relationship between the rating of an app and a particular feature of an app [2], [5], [17], [23], [29], [43]. Harman et al. examine the correlations between application's description, download count, and average user rating among thousands of BlackBerry apps [17]. Ruiz et al. investigate the relationship between the number of integrated ad libraries and rating, and find that there is no relationship between them [29]. Guerrouj et al. analyze 154 free apps to study the impact of app churn on the app rating and find that high app churn leads to lower user ratings [14]. For other related work [2], [5], [23], [43], we revisit their observations on our dataset and provide a detailed comparison in Section V. Our study is similar to the above studies in that we too are examining the relationships between different factors of an app and its rating. However, we consider a slew of other factors that have not been investigated before, and ignore some factors (e.g., number of ad libraries [29]) that have been shown to have no relationship with ratings. Also, we are the first to compare these factors and rank them according to the strength of their impact on the rating of an app.

There have been several studies that have analyzed reviews of Android apps [19], [20], [21], [31]. Different from these studies, we only analyze the ratings of apps, and consider a different goal, i.e., to identify factors that characterize high-rated apps.

There are a number of other studies on Android apps. Shabtai et al. apply machine learning techniques to features extracted from Android apps in order to classify Android apps as either tools and games [34]. Syer et al. compare the source code of open source mobile apps for the Android and BlackBerry platforms [41]. Minelli et al. analyze the source code of 20 Android apps and find that Android apps are smaller than traditional software systems [27]. Linares-Vásquez et al. investigate two threats that affect the validity of studies analyzing APK files: code obfuscation and library usages [24]. They find that removing neither obfuscated code nor library code cause a big difference in Android code reuse study results.

VII. Conclusion and Future Work

A number of past studies have investigated a few factors that are related to app ratings. However, there exist a slew of other factors that have not been investigated yet. Moreover, how these factors integrate together to impact the likelihood of an app being a high-rated app is still unknown. To fill the gaps in current research, we consider 28 factors along eight dimensions that could potentially be associated to app ratings. We analyze more than a thousand Android apps and investigate the differences between high-rated and low-rated apps. We find that high-rated apps are statistically significantly different from low-rated apps in 17 out of these 28 factors. These differences are significant enough such that a random forest classifier built on these factors can differentiate high-rated from low-rated apps with fairly good result (i.e., an F-measure of 0.74 and an AUC of 0.81). Thus there is initial evidence to suggest that a DNA of high-rated apps exists. Among the 28 factors, three of them, which have not been investigated in prior studies, are related the most to app ratings: install size of an app, number of promotional images, and target SDK version.

In the future, we would like to explore additional factors that can potentially impact app rating, e.g., app developer characteristics (c.f., [39], [38]), etc. We also plan to extend the study to contrast characteristics of successful and unsuccessful apps within each app category. Moreover, while we have only investigated high and low rated apps in this work, in the future, we plan to investigate borderline ones and build suitable classification or regression models to capture relationships between various factors and ratings of apps from a whole range of ratings. We have also only analyzed app ratings and ignored reviews given to these apps; in the future, we plan to analyze the reviews too. We also would like to collect a large number of versions of these apps to apply causality analysis. It would also be interesting to investigate how the app stakeholders' opinions match with the mined data, and thus we plan to survey Android app developers and users for their thoughts on factors that are associated with app success.

Acknowledgement

We would like to thank Linares-Vásquez, Bavota, Bernal-Cárdenas, Di Penta, Oliveto, and Poshyvanyk, for making the Android API's change and bug dataset used in their work [23] publicly available. We would also like to thank Khalid from the Software Analysis and Intelligence Lab at Queen's University for crawling the apps used in our case study.

[11] http://www.gartner.com/newsroom/id/2592315

REFERENCES

[1] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28(1):4–17, 2002.

[2] G. Bavota, M. L. Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. The impact of API change- and fault-proneness on the user ratings of android apps. *IEEE Trans. Software Eng.*, 41(4):384–407, 2015.

[3] N. Bettenburg, M. Nagappan, and A. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 60–69, June 2012.

[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[5] P. H. Chia, Y. Yamamoto, and N. Asokan. Is this app safe?: a large scale study on application permissions and risk signals. In *Proceedings of the 21st international conference on World Wide Web*, pages 311–320. ACM, 2012.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[7] W. J. Conover and W. Conover. Practical nonparametric statistics, 3rd edition. 1998.

[8] D. A. da Costa, S. L. Abebe, S. McIntosh, U. Kulesza, and A. E. Hassan. An empirical study of delays in the integration of addressed issues. In *ICSME*, 2014.

[9] J. den Haan. http://www.theenterprisearchitect.eu/blog/2012/06/27/7-ways-a-platform-can-fuel-the-app-economy/.

[10] S. Dienst and T. Berger. Static analysis of app dependencies in android bytecode. http://www.informatik.uni-leipzig.de/berger/tr/2012-dienst.pdf, 2012.

[11] B. Efron. Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983.

[12] D. M. German. Using software distributions to understand the relationship among free and open source software projects. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 24–24. IEEE, 2007.

[13] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[14] L. Guerrouj, S. Azad, and P. C. Rigby. The influence of app churn on app success and stackoverflow discussions. In *SANER*. IEEE, 2015.

[15] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 417–428. IEEE, 2004.

[16] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2011.

[17] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 108–111. IEEE, 2012.

[18] D. W. Hosmer Jr and S. Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.

[19] C. Iacob and R. Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 41–44. IEEE, 2013.

[20] C. Iacob, V. Veerappa, and R. Harrison. What are you complaining about?: a study of online reviews of mobile applications. In *Proceedings of the 27th International BCS Human Computer Interaction Conference*, page 29. British Computer Society, 2013.

[21] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the devices to test your app on: A case study of android game apps. In *FSE*, 2014.

[22] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.

[23] 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*, pages 477–487. ACM, 2013.

[24] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 242–251. ACM, 2014.

[25] R. Martin. OO design quality metrics - an analysis of dependencies. In *Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics @ OOPSLA*, 1994.

[26] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *Software Engineering, IEEE Transactions on*, 39(6):822–834, June 2013.

[27] R. Minelli and M. Lanza. Software analytics for mobile applications–insights and lessons learned. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 144–153. IEEE, 2013.

[28] A. Mockus, M. Nagappan, and A. E. Hassan. Best practices and pitfalls for statistical analysis of se data. In *ICSE*, 2014.

[29] I. Mojica Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan. On the relationship between the number of ad libraries in an android app and its rating. *Software, IEEE*, PP(99):1–1, 2014.

[30] I. J. Mojica Ruiz. Large-scale empirical studies of mobile apps. 2013.

[31] D. Pagano and W. Maalej. User feedback in the appstore: An empirical study. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 125–134. IEEE, 2013.

[32] F. Peters, T. Menzies, and A. Marcus. Better cross company defect prediction. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 409–418. IEEE, 2013.

[33] R. D. Retherford and M. K. Choe. *Statistical models for causal analysis*. John Wiley & Sons, 2011.

[34] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333, Dec 2010.

[35] D. Spinellis. Tool writing: a forgotten art?(software tools). *Software, IEEE*, 22(4):9–11, 2005.

[36] D. Spinellis. *Code quality: the open source perspective*. Adobe Press, 2006.

[37] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Software Engineering, IEEE Transactions on*, 29(4):297–310, 2003.

[38] D. Surian, N. Liu, D. Lo, H. Tong, E. Lim, and C. Faloutsos. Recommending people in developers' collaboration network. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pages 379–388, 2011.

[39] D. Surian, D. Lo, and E. Lim. Mining collaboration patterns from a large developer network. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 269–273, 2010.

[40] A. P. D. Survey. http://app-promo.com/wp-content/uploads/2013/06/SlowSteady-AppPromo-WhitePaper2013.pdf.

[41] M. D. Syer, B. Adams, Y. Zou, and A. E. Hassan. Exploring the development of micro-apps: A case study on the blackberry and android platforms. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 55–64. IEEE, 2011.

[42] M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan. Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal*, pages 1–24, 2014.

[43] S. E. S. Taba, I. Keivanloo, Y. Zou, J. Ng, and T. Ng. An exploratory study on the relation between user interface complexity and the perceived quality of android applications. In *ICWE*, 2014.

[44] M. Thelwall, K. Buckley, and G. Paltoglou. Sentiment strength detection for the social web. *Journal of the American Society for Information Science and Technology*, 63(1):163–173, 2012.

[45] F. Thung, D. Lo, and L. Jiang. Automatic defect categorization. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 205–214. IEEE, 2012.

[46] Y. Tian, D. Lo, X. Xia, and C. Sun. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, pages 1–30, 2014.

[47] D. H. Wolpert and W. G. Macready. An efficient method to estimate bagging's generalization error. *Machine Learning*, 35(1):41–55, 1999.

[48] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.