PRADO: Predicting App Adoption by Learning the Correlation between Developer-Controllable Properties and User Behaviors

XUAN LU, ZHENPENG CHEN, XUANZHE LIU, and HUORAN LI, Peking University, China
TAO XIE, University of Illinois Urbana-Champion, USA
QIAOZHU MEI, University of Michigan, USA

To survive and stand out from the fierce market competition nowadays, it is critical for app developers to know (desirably ahead of time) whether, how well, and why their apps would be adopted by users. Ideally, the adoption of an app could be predicted by factors that can be controlled by app developers in the development process, and factors that app developers are able to take actions on and improve according to the predictions. To this end, this paper proposes PRADO, an approach to measuring various aspects of user adoption, including app download and installation, uninstallation, and user ratings. PRADO employs advanced machine learning algorithms to predict user adoption based on how these metrics correlate to a comprehensive taxonomy of 108 developer-controllable features of the app. To evaluate PRADO, we use 9,824 free apps along with their behavioral data from 12.57 million Android users, demonstrating that user adoption of a new app can be accurately predicted. We also derive insights on which factors are statistically significant to user adoption, and suggest what kinds of actions can be possibly performed by developers in practice.

CCS Concepts: • **Human-centered computing** → **User centered design**; **Mobile computing**; • **Software and its engineering** → *Software usability*; • **Applied computing** → Marketing;

Additional Key Words and Phrases: Mobile apps, usage data, user adoption

ACM Reference Format:
Xuan Lu, Zhenpeng Chen, Xuanzhe Liu, Huoran Li, Tao Xie, and Qiaozhu Mei. 2017. PRADO: Predicting App Adoption by Learning the Correlation between Developer-Controllable Properties and User Behaviors. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 3, Article 79 (September 2017), 30 pages.
https://doi.org/10.1145/3130944

1 INTRODUCTION

In the past few years, the fast growing "app economy" has been a major phenomenon in the ubiquitous and mobile computing community. It was reported that the yearly revenue from the Apple AppStore has reached

This work was supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A202, the Natural Science Foundation of China (Grant No. 61370020, 61421091, 61528201, 61529201), and the Microsoft-PKU Joint Research Program. Tao Xie's work was supported in part by NSF under grants No. CCF-1409423, CNS-1434582, CNS-1513939, CNS-1564274. Qiaozhu Mei's work was supported in part by the National Science Foundation under grant no. IIS-1054199 and an MCubed grant at the University of Michigan.
Author's address: X. Lu, Z. Chen, X. Liu, and H. Li, the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China, 100871, email:{luxuan, chenzhenpeng, liuxuanzhe, lihuoran}@pku.edu.cn; T. Xie, University of Illinois Urbana-Champion, email: taoxie@illinois.edu; Q. Mei, University of Michigan, email: qmei@umich.edu.
Corresponding author: liuxuanzhe@pku.edu.cn.

2474-9567/2017/9-ART79 $15.00
https://doi.org/10.1145/3130944

over 20 billion USD as of 2015[1], and the aggregated revenue from all app stores is expected to reach 70 billion by 2017 [53]. These app stores continuously attract millions of developers to release and distribute their apps. Many industry leaders have either started or revived from an app.

Some "*Killer Apps*," such as Facebook, Snapchat, and WhatsApp, are inherently more popular than others partly because they were born and published on the app stores. However, not every single app is fortunate enough to be a successful one in such an app-centric software-distribution model. Due to the extremely fierce competitions, very few apps are fortunate enough to obtain a large volume of users, receive many positive reviews, or be ranked highly out of similar apps. Indeed, the success of an app may be assessed differently from the perspectives of business owners, product managers, domain experts, and end users [17, 18]. But, for most app developers, the preceding metrics are the very ones that indicate the extent to which an app is appreciated by its users, or how the app is "adopted" by the users. Intuitively, a well-adopted app is expected to own more downloads/users, win higher ratings/rankings, or attract more use rather than being uninstalled. Developers are eager to know how their apps perform in terms of these metrics, or in other words, whether, how well, and even why their apps are being adopted by users. More importantly, they desire to know such information ahead of time – as early as possible in the app's lifecycle, or even before the app is released – so that they can address potential risks or flaws and thus win better user adoption.

Such a desire points to the need of predicting the potential adoption of apps as early as possible, even before the app is actually released and distributed; the purpose of the work in this paper aims to satisfy such need. On one hand, there are indeed too many factors to affect the adoption of innovations along with factors that mix technical, economic, social, psychological, and political dimensions [45], making the prediction task very challenging. On the other hand, not all factors are practical to app developers: there are only certain dimensions that can be controllable by developers. Our goal is to find these predictive and actionable factors and thus to help developers with informative suggestions.

In this paper, we propose a novel approach, namely **PRADO**, to **pr**edicting Android **a**pps' a**do**ption. The core of the PRADO approach is to establish a prediction model by learning the correlations between various "*developer-controllable*" properties extracted from app information (including code-level metrics, textual descriptions, and so on, as elaborated in Section 5) and the adoption behavior collected from large-scale real Android users, including the activities of download and installation, uninstallation, and user rating of apps. For a new app that is to be released or is just released in app stores, this model can predict how well it is going to be adopted by users. In addition, PRADO can identify app properties that are statistically significant to influence user adoption, and thus provide insights on what to improve in the app-development process.

PRADO is practically useful for at least two purposes:

- **Efficient Release Planning**. Based on the predicted user adoption of their apps, developers can better organize and schedule release-planning activities in advance. For example, knowing the potential number of downloads, developers can efficiently estimate the possible workloads and thus better allocate their server-side resources. Such knowledge could be particularly useful for small teams or individual developers who have very limited budget to maintain cloud resources. Meanwhile, developers can optimize manpower and/or marketing resources such as ads investments to increase revenue. Combining the predicted and the observed feedback from the users, developers can more efficiently prioritize their schedule of release planning.
- **Improved App Quality and Ranking**. The prediction of user ratings or abandonment can help app developers to identify potential problems/bugs of their apps, and to suggest some insights on how to make their apps more successful. As a result, developers can pay more attention to those properties that are highly correlated to user adoption and take some possible actions in advance. For example, they can

[1]http://www.computerworld.com/article/3019716/apple-ios/apples-cut-of-2015-app-store-revenue-tops-6b.html.

prioritize or better allocate resources to perform A/B tests to validate whether these features really matter. By doing so, developers can make their apps win higher ranks in app stores and obtain more users as well as revenue. Additionally, developers can also learn more knowledge about advantages and shortcomings of their competitors.

To evaluate PRADO, we use a large-scale data set that includes the collected download and installation, uninstallation, and rating behaviors of 12.57 million Android users over 9,824 free Android apps. To the best of our knowledge, our work is the first to study the real user behavioral data at such a large scale and to be able to understand how app adoption is affected by developer-controllable properties. Empirical results confirm the effectiveness of PRADO for predicting app adoption.

It is important to state that PRADO does not claim necessary *causal* relationship between the app properties and the adoption of apps. Instead, it identifies those developer-controllable features that are only statistically *correlated* with the success of an app. Indeed, correlation analysis is the first step to derive the reasonable casual association. Some of PRADO-identified features may be exactly causal to an app's success, and the developers could perform A/B tests to validate which of them are, before the developers can actually increase user adoption by optimizing these features.

This paper makes the following major contributions:

- The app-adoption measurement model that consists of three indicators derived from actual user behaviors, i.e., the number of downloads, user ratings, and uninstallation ratio. In particular, our new uninstallation-ratio indicator is complementary to the other two and enables a more comprehensive measurement of the app adoption.
- The prediction model that is built upon advanced machine learning algorithms to identify the correlation between extensive app properties and user behavioral data at scale. Such a model can accurately predict the adoption of a new app/version before it is released in app stores and identify some statistically significant properties that can affect app adoption.
- The empirical study that is conducted over thousands of apps along with millions of actual users of these apps. To the best of our knowledge, such a scale is the largest to date and thus can well preserve the effectiveness and confidence of our proposed approach.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 illustrates the overall workflow of the PRADO approach. Section 4 introduces the data set and formulates the user adoption indicators. Section 5 presents the taxonomy of selected features from various dimensions. Section 6 presents how PRADO predicts user adoption of a new app by learning the correlations between selected features and user behaviors. Section 7 illustrates the statistically significant features that can possibly affect user adoption. Section 8 demonstrates the performance of PRADO on one particular app category in comparison to that on all apps. Section 9 presents the limitations and threats to validity. Section 10 concludes the paper.

2 RELATED WORK

In the community of software engineering, determining the factors that influence the successful development and deployment outcomes of software systems has been a major research focus for years. There have been many factors reported to have an effect on the success of software development [17, 18, 22, 27, 36, 37]. Such factors can be traced and leveraged to help predict app success [8]. In this paper, we measure the success of an Android app by how it is liked or disliked by users. More specifically, we focus on how the users adopt an app, i.e., whether the users decide to download or abandon an app, or give high rating of this app.

Undoubtedly, from the perspective of developers, software quality is always the major factor to determine the success of software systems and applications. Researchers have already developed various metrics to measure software quality. As an Android app essentially complies to Object-Oriented programming paradigms, the traditional OO metrics [9, 14] can be applied. Considering the variety of app-centric ecosystem, numerous efforts have been proposed from different perspectives for evaluating app success [10, 13, 16, 23, 24]. Bavota *et al.* [10] investigated how the fault- and change-proneness of APIs used by Android apps are related to their success, which is estimated as the average rating of those apps provided by their users. Coulton *et al.* [16] conducted a case study on game apps created for the WidSets platform and found that popular features such as chatting can increase the popularity of an app. Guerrouj *et al.* [23] studied the impact of app churn on the user ratings of 154 free apps and found that a higher app churn can lead to lower user ratings. Chen *et al.* [12] proposed the AR-Miner to discover information from raw user reviews that are the most "informative" for developers to improve their apps. Tian *et al.* [51] examined the relationships between various features of an app and its ratings. They used only 28 factors to characterize the differences between high-rated apps and low-rated apps.

Learning the lessons from these preceding results, in this paper we also investigate app information from the app developers' perspective, the information that characterizes how they implement and release their apps. In practice, PRADO can substantially improve the state-of-the-art research in two folds.

First, there are various indicators of more successful app adoption, such as a higher number of downloads and higher user ratings. However, these two metrics inherently have some limitations. Users may download an app without using or uninstalling it. In addition, "fake" downloads can be performed by some automatic programs [30, 34]. On the other hand, user ratings may suffer from significant biases, e.g., ratings for some apps are largely sparse [30, 32]. To complement these indicators for more comprehensive measurement of app adoption, PRADO introduces a **new** signal from a uniquely different perspective: the actual user management behaviors of apps, i.e., uninstallation ratio of apps. Such a signal exactly indicates the users' positive/negative attitudes towards an app, and can be quite complementary to the number of downloads and user ratings. In fact, the uninstallation information is typically quite private and cannot be publicly accessed. In collaboration with a leading Android app store in China, called Wandoujia,[2] our research has gained the access to such actual user behavioral data [30] to advance the state of the art in this valuable direction.

Second, PRADO enables us to derive the correlations between every single indicator and an extensive set of features extracted from app information, including the size of apps, complexity of apps, dependency of system/third-party libraries, complexity of user interfaces, requirements of user/device, etc. Based on advanced machine learning algorithms, PRADO can accurately predict the user adoption of a new app, and thus can make it possible for app developers to estimate their release-planning efforts. In addition, PRADO identifies some statistically significant features that can possibly affect user adoption. Although these correlation-analysis results do not necessarily reflect the underlying causality, they are still meaningful to provide valuable insights and guidelines for developers, who can gain deeper understanding on improving their apps.

Indeed, predicting the user adoption of apps is quite crucial for the success on the app-store centric ecosystem. Similar to the search-engine optimization (SEO), various industrial solutions focus on the app-store optimization (ASO), which aims to improve the visibility of an app on app stores. For example, AppAnnie [8] provides analytics services to estimate the downloads, usage, user distribution, and user retention of apps, so as to inform the developers in terms of product, marketing, and investment strategies. The core idea of AppAnnie is to trace the user-behavior data by deploying a stand-alone app. Similar to AppAnnie, PRADO derives the estimation of user adoption based on learning the correlations between behaviors from large-scale actual user behavioral data and developer-controllable properties. However, other than estimation and prediction, PRADO also aims to infer some insights that can help developers understand the factors impacting the user adoption and further optimize their apps.

[2]Wandoujia now owns over 300 million users and 1 million apps.

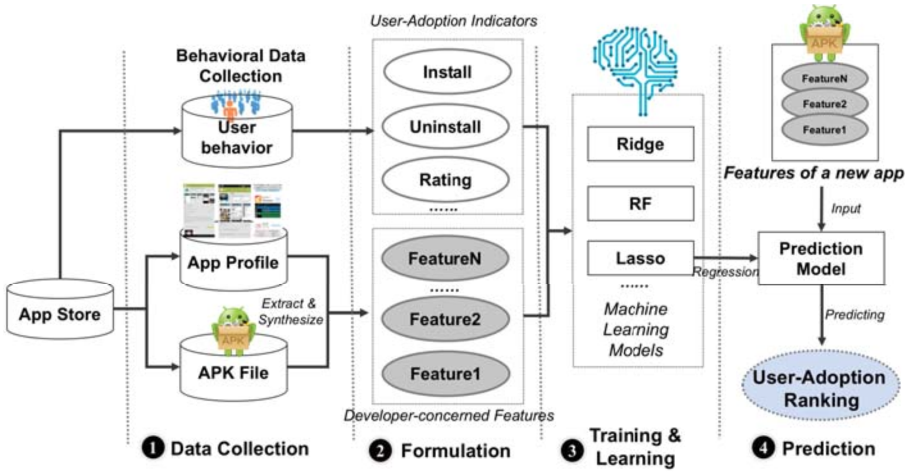


Fig. 1. Approach overview of PRADO.

3 PRADO: IN A NUTSHELL

To address the urgent needs of predicting user adoption of apps, PRADO is proposed to **pr**edict **a**pps' a**do**ption based on the factors that are related to the implementation and releasing activities of the apps. The essential goal of PRADO is to *learn and infer the correlation between various indicators of user adoption and features that can be controlled by developers*.

The general workflow of PRADO is illustrated in Figure 1. In general, PRADO consists of four major phases:

- **User-Behavior Data Collection**. In PRADO, the user adoption of an app is measured by three indicators of user behaviors, including the user rating, the number of downloads, and the uninstallation ratio. Such behavioral data can reflect the users' attitude towards an app. The first two indicators can be publicly collected from app stores, while the uninstallation ratio is quite private and has not been well studied. As introduced later, we are fortunate to collect uninstallation information from the leading Wandoujia app store in China, and thus able to measure the user adoption in a more comprehensive way.
- **Feature Formulation from App Properties**. PRADO aims to correlate the user adoption with the "developer-controllable" properties of an app. To this end, we establish a taxonomy that comprehensively conveys extensive features (described in Section 5) about the implementation and releasing of an app.
- **Model Derivation by Correlation Analysis.** PRADO then derives a model that correlates the developer-controllable features with the indicators that are computed from the user behavioral data. Such a model is trained using various state-of-the-art machine learning algorithms (i.e., *Ridge*, *Lasso*, and *Random Forest*), which can also discover features that have significant influences on specific indicators.
- **User Adoption Prediction**. Finally, for a new app or a new app version that will be released on the app store, PRADO extracts the features of this app and runs the derived model to predict the preceding user-adoption indicators of the app/version.

In a sense, PRADO relies on two inputs at scale: (1) the user behavioral data indicating an app's adoption; (2) the features derived from app properties that can be controlled by the app's developers. PRADO performs the correlation analysis with machine learning algorithms offline, and generates two outputs: (1) the predicted model for user adoption; (2) the statistically significant features that can possibly affect the user adoption. We then introduce these two outputs in Sections 4 and 5, respectively.

4 INDICATORS OF USER-ADOPTION

Usually, whether a user adopts an app can be simply determined by whether or not he/she has downloaded the app, or he/she shows positive/negative attitudes towards this app. Therefore, how well an app is adopted in general can be simply measured by the *Number of Downloads* or the *Rating* of this app. In practice, these two indicators can be crawled from the public app stores. However, it is argued that these two indicators can have some inherent limitations. Downloading an app does not necessarily indicate that a user really needs, likes, or dislikes the app, as the user is likely to leave the app on device but never launches it any longer. The rating of an app can indeed indicate the users' positive or negative attitudes towards the app. However, it was reported that the rating of some apps can be quite sparse and even unavailable. In addition, similar to other online systems, app stores can also possibly experience some fake ratings that are not reliable for being used.

Hence, in addition to these two indicators, we introduce a new indicator, namely the Uninstallation-Installation ratio (abbreviated as the *U-I Ratio*), indicating the extent that the user "exactly" abandons the app. Although the *U-I Ratio* is not a perfect indicator, it indicates the information about users abandoning an app; such information cannot be inferred from the *Number of Downloads* or *Rating*. Additionally, such an indicator is not as sparse as the *Rating*. It captures the users' downloading and uninstalling behaviors and reflects their possible attitudes towards the app. Hence, we can synthesize these indicators to more comprehensively understand the user adoption of the app.

Indeed, the uninstallation behavior is usually unavailable on most app stores. However, we are fortunate to access such data to enable the study. We next describe the data set.

4.1 Data Set

In our previous work [31, 35], we have presented the user behavioral data from Wandoujia, a leading Android app store that had over 300 million users and hosted over 1 million free apps as of 2014[3]. This behavioral data contain the management activities of downloading, updating, and uninstalling an app.

In this paper, we use a 5-month data set (May 1st to September 30th, 2014) of app-management activities recorded by Wandoujia.[4] To make sure that our analysis result is robust enough, we select only the apps that have at least 50 unique active users (identified by the "anonymized" device IDs). The selected 32,141 apps with user management logs constitute the *original* data set. We then crawl these apps' corresponding `.apk` files and webpage profiles from the Wandoujia website, decompile the APK files, and extract the features and indicators of the apps. Finally, we successfully obtain the complete information of 9,824 apps, which we denote as the *final* data set. Indeed, one concern is whether this pre-processing can introduce some potential bias. To this end, we validate the distribution of app categories before and after the pre-processing steps. As shown in Figure 2, we can find that there is no obvious bias between the two distributions; thus, the pre-processing process may not compromise the results.

4.2 Ethical Consideration

Indeed, any data collected from real users need to be carefully processed. We take a series of steps to preserve the privacy of involved users in our data set. First, all raw data collected for this study are kept on the Wandoujia data-warehouse servers (which live behind a company firewall). Second, our data-collection logic and analysis pipelines were governed by Wandoujia employees, who supervised the process of data collection and de-identification, to ensure compliance with the commitments of Wandoujia's privacy policy stated in the *Term-of-Use* statements. Third and the most significantly, Wandoujia employees anonymized the user identifiers before any data analysis.

[3]See http://techcrunch.com/2014/01/12/wandoujia-120m/ for more details.
[4]Part of the data set has been publicly released along with our published ICSE 2016 work [35].

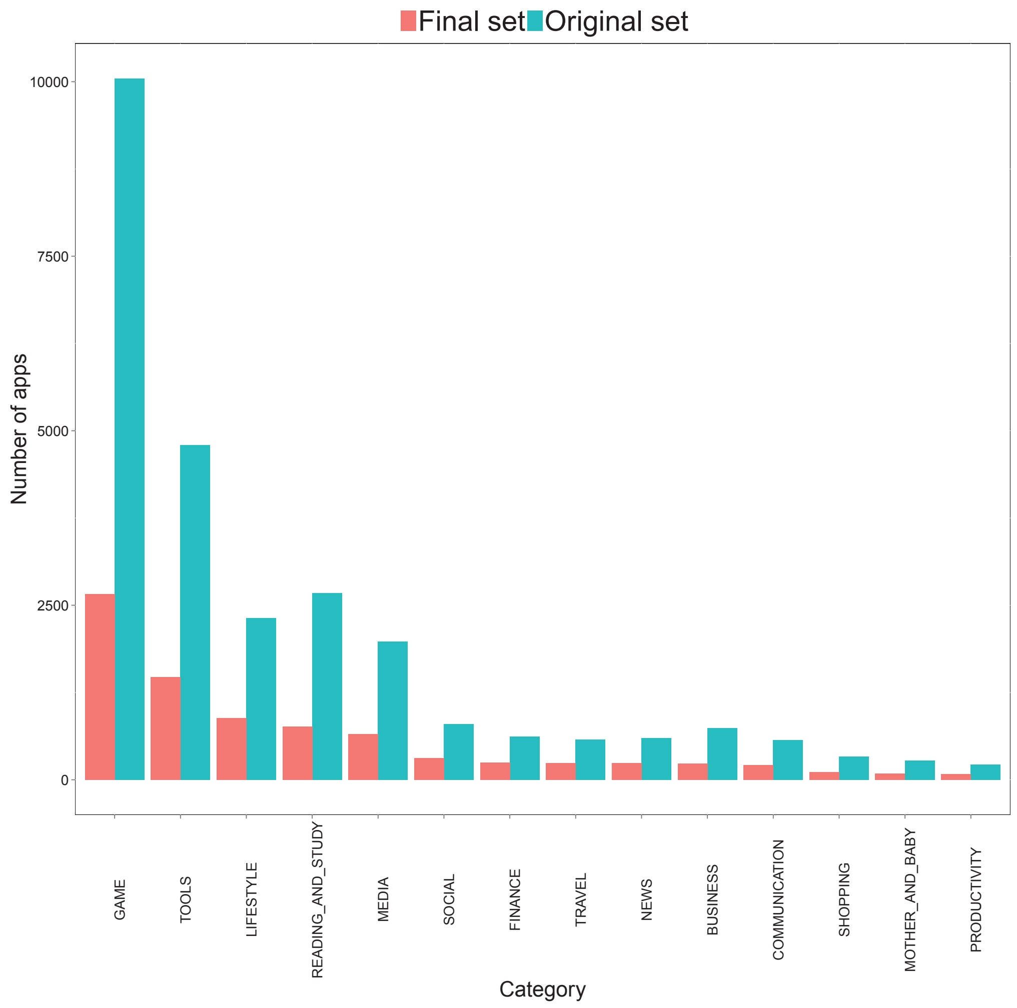


Fig. 2. Distribution of app categories before/after pre-processing.

Only aggregated statistics are produced for the users covered in our study period. Finally, we obtain the IRB approval from the authors' institute to conduct this research.

4.3 Indicator Formulation

We next describe the indicators of how Android apps are adopted by actual users, which can be inferred from our user behavioral data set.

We can immediately obtain the *Number of Downloads* during the time span of our data set. For user ratings, since Wandoujia employs a simple binary rating model ("like" or "dislike") rather than the 5-star model used by Google Play, we measure app ratings by the "*Like Ratio*" indicator that is simply computed as follows:

$$\text{Like Ratio} = \frac{\text{number of } \textit{likes}}{\text{number of } \textit{likes} + \text{number of } \textit{dislikes}} \tag{1}$$

Intuitively, a higher *Like Ratio* indicates that the app is probably more appreciated by the users.

However, as mentioned earlier, these two indicators have possible limitations. To derive more comprehensive knowledge about user adoption, this paper introduces a new indicator, namely the **U-I Ratio**, which is computed as follows:

$$\text{U-I Ratio} = \frac{\text{number of } \textit{uninstallations}}{\text{number of } \textit{installations}} \tag{2}$$

Compared to the preceding two indicators, the *U-I Ratio* essentially reflects the "actual" (negative) attitude towards an app, i.e., a higher *U-I Ratio* indicates that this app is more probably abandoned by its users. Such a metric can be complementary to the number of downloads and the user ratings, and thus makes us have more comprehensive knowledge of user adoption of an app.

5 FEATURE FORMULATION

PRADO aims to correlate the user adoption with the properties of an app. To comprehensively describe various perspectives of an app, we establish a taxonomy of features related to the implementation and releasing of an app, mainly from a software-engineering perspective. All selected features can be controlled by the app developers. We then describe each dimension as well as the selected features in that dimension. The definition and description of all individual features can also be found in Table 5.

5.1 App Size (D1)

The first dimension is the size of the released app's installable file, i.e., the `.apk` file, which includes the code and the resources. Intuitively, a larger app can contain more features and functionalities, and possibly introduces more bugs [57]. Such dual characteristics may affect the user adoption in two opposite directions.

The feature *size_apk* (measured in kilo byte) is the overall measurement of the app size. At a finer granularity, we use two features, the **number of all classes** (*num_class*) and the **number of all methods** (*num_method*) in the app. Since the influence of third-party libraries in app quality is quite complex, careful consideration is needed in determining whether to remove third-party libraries in app analytics or not [33]. Therefore, we decide to consider both the total code and project code (i.e., code without third-party libraries), with two features, the **number of project classes** (*num_proClass*) and the **number of project methods** (*num_proMethod*). Additionally, although researchers often leverage *Lines of Code* (LOC) to represent the size of code, we choose to quantify the byte code instructions by decompiling the `.apk` file, because the source code is not always available. The corresponding features are defined as *num_byteCodeIns* for **byte code instructions in the app**, and *num_proByteCodeIns* for **byte code instructions in the project code**.

To further characterize the functionality richness of an app, we introduce *num_activity* indicating the **number of activities** (in the Android programming model, an *Activity* represents a single screen with a user interface [6]), and *num_service* indicating the **number of services** (in the Android programming model, a *Service* is a component that runs at the background to perform long-running operations without user interactions, and it can keep working even if the app is destroyed [7]).

Intuitively, more resources used in an app may introduce richer user experiences. Despite the dynamically loaded resources, we consider only the **number and volume of pictures** (*num_resPic* and *size_resPic*) in the resources file.

5.2 Code Complexity (D2)

We then measure the code complexity metrics, which are demonstrated to be important in determining software defects [49]. For example, the increase of class size is associated with a high number of defects, which can be related to user experience and further influence user adoption.

Since the Android-app development usually employs C/C++ and Java as programming languages, most features in this dimension are essentially object-oriented (OO) metrics. This paper leverages the six well-known OO metrics proposed by Chidamber and Kemerer (i.e., CK metrics) [14], as illustrated below.

• **Weighted methods per class** (*WMC*) is originally defined as the sum of complexity of all methods in the class under measurement. Consider a class C_1 with methods M_1, ..., M_n defined in the class, and c_1, ..., c_n describe the complexity of the methods, respectively. Then,

$$WMC = \sum_{i=1}^{n} c_i \tag{3}$$

We assign the complexity value as 1 to each method, and therefore the value of the WMC is equal to the number of methods in the class.

• **Depth of inheritance tree** (*DIT*) is the depth of inheritance of the class under measurement. When multiple inheritances exist, the DIT is the maximum length from the node (for the class) to the root of the inheritance tree.

• **Number of children** (*NOC*) is the number of immediate subclasses subordinated to the class under measurement in the class hierarchy.

• **Coupling between object classes** (*CBO*) is the number of other classes to which the class under measurement is coupled. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

• **Response for a class** (*RFC*) measures the size of RS, a set of methods that can potentially be executed in response to a message received by an object of the class under measurement. Theoretically, RS can be expressed as

$$RS = \{M\} \bigcup_{all\ i} \{R_i\} \tag{4}$$

where $\{R_i\}$ is the set of methods called by method i and $\{M\}$ is the set of methods in the class.

• **Lack of cohesion in methods** (*LCOM*) is the number of method pairs whose similarity is zero minus the number of method pairs whose similarity is not zero. The similarity of two methods is measured by their shared instance variables.

Consider a class C_1 with n methods M_1, ..., M_n. Let $\{I_i\}$ be the set of instance variables used by method M_i. In addition, let $P = \{(I_i, I_j) | I_i \bigcap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) | I_i \bigcap I_j \neq \emptyset\}$. If all $\{I_i\}$ are $\emptyset$ then let $P = \emptyset$.

$$LCOM = |P| - |Q| \ (if\ |P| > |Q|)\ or\ 0\ (otherwise) \tag{5}$$

In addition to the CK metrics, we employ the metric of **afferent couplings** (*CA*) [40], which is defined as the number of classes (outside a category of classes) that depend upon classes within this category, and the **number of public methods** (*NPM*) [9], which refers to the number of public methods in the class under measurement.

Data access can possibly impact the performance of an app. We take into account the **percent of public and protected fields** (*pubVarDefPer*) to all the fields defined in the class under measurement, and the **percent of access to public and protected fields** (*pubVarAccPer*), regarding that a lower such metric is desirable [9]. We also consider the total **number of parameters of all the methods** (*para*) in the class under measurement, and the average **number of parameters of a method** (*paraPerMethod*) in the class under measurement.

The preceding features are all defined at the *class* level. To evaluate the complexity of an app instead of the *class*, we leverage the maximum, mean, and standard deviation of the features extracted from all classes. For example, the features corresponding to WMC should contain *mean_wmc*, *max_wmx*, and *std_wmc*.

In addition, we consider the **number of byte code instructions per method (IPM)** (*num_ipm*) and the **cyclomatic complexity** [41] (*num_exePath*). The cyclomatic complexity measures the program complexity based on possible paths of execution. It can be calculated by counting the number of regions in the program graph.

5.3 Code Quality (D3)

In the dimension of code quality, we focus on two kinds of metrics that could result in app crash or unresponsiveness. Users would possibly uninstall such apps or submit a low rating. Similarly, we still consider all the code from the `.apk` file, including the project code and the library code.

• **Exception handling**. Indeed, exception handling is one of the most important features affecting the code quality. As proposed by Kechagia and Spinellis [29], the most common reason for app crash is programming errors (i.e., `NullPointerException`), and the top-10 methods that frequently throw exceptions are `dismiss`, `show`, `setContentView`, `createScaledBitmap`, `onKeyDown`, `isPlaying`, `unregisterReceiver`, `onBackPressed`, `showDialog`, and `create`. Using these methods without a `try-catch` block would possibly result in exceptions and thus lead to app crash. Therefore, we count the **number of calls that include the mentioned 10 methods but without explicit `try-catch blocks`**, denoted as *num_ucDismiss*, *num_show*, and so on.

Handling exceptions with a `try-catch block` can imply the developers' carefulness. However, a `try-catch block` without correct action can be also problematic. We then count the **number of `try-catch blocks`** (*num_catchBlock*), the **number of empty blocks** (*num_noAction*), and the **number of blocks with only logging instructions** (*num_logOnly*).

• **ANR issues**. From the official Android developers' guide [2], an "**Application Not Responding**" (ANR) dialog will be displayed if there is no response to the input event (such as key press or screen touch events) within 5 seconds, or a `BroadcastReceiver` is not finished within 10 seconds. As Android apps run in a single-thread model by default, a UI thread and the main thread are inherently the same one. Hence, anything happening in the UI thread and requiring a long time to complete can trigger the `ANR` dialog because the app is not giving itself a chance to handle the input event or intent broadcasts. As proposed by Yang et al. [55], we take into account four types of API calls in the UI thread as the common causes of `ANR`, as explained as follows. We count the number of occurrences of each category, respectively.

Network access (*num_networkUIThread*). Network operations performed in the UI thread can cause the perception of sluggishness. An exception called `NetworkOnMainThreadException` has been added to the Android APIs since level 11[5], but the network access performed in the UI thread is still allowed in apps targeting earlier SDK versions although it is heavily discouraged.

Flash storage access (*num_fileIOUIThtread*). Even simple disk operations could exhibit significant and unexpected latencies, and should be avoided in the UI thread [20].

Database access (*num_SQLiteUIThread*). Android apps often access the on-device SQLite database, which can generate a substantial number of expensive write operations to the flash storage, and thus can make the response sluggish.

Bitmap processing (*num_bitmapUIThread*). Processing large bitmaps could be too computationally expensive to be adopted in the UI thread [2].

[5]https://developer.android.com/reference/android/os/NetworkOnMainThreadException.html.

5.4 Libraries (D4)

Android apps heavily depend on libraries including the Android's base libraries and third-party libraries. Such dependency can impact the quality of an app, e.g., API usage adaptation code is more defect-prone than the one without API usage adaptation [42]. Hence, we extract the dependence of libraries as follows.

• **Android API**. We first compute the **dependency on Android's base libraries** (*dep_AndroidLib*) as the number of libraries starting with "`android.`" that are called [51]. In practice, app developers cannot always adapt to the fast-evolving APIs [42] on time, and such a mismatch can result in the app's inconsistencies, bugs, or even errors and/or failures. Therefore, we calculate the **average of update frequencies of involved Android APIs** (*mean_AndroidUpdateFre*) by mining the source code on the Git repositories of Android[6]. In particular, we calculate the **average number of bug-fixing actions** (*mean_AndroidBug*) by mining all the updating logs.

• **Third-party libraries**. In addition to Android APIs, many third-party open-source libraries are available and can be simply reused to enrich the functionalities and user experiences. We focus on popular libraries in the 10 categories (i.e., *advertisement*, *social network*, *mobile analytics*, *development aid*, *app market*, *payment*, *UI component*, *game engine*, *utility*, and *map*) that are defined in LibRadar [38]. We count the **dependency on all third-party libraries** (*dep_thirdPartyLib*) by counting the total number of called libraries and the **dependency on libraries in every single category** (*dep_adLib*, *dep_socialNetworkLib*, etc.) by counting the number of called libraries in every single category. We assume that the popularity may relate to the functionality and quality of the library, and thus include the **average popularity of the libraries** (*mean_thirdPartyLibPopu*) by calculating the average of popularities of each called library. The *popularity* values are defined by LibRadar and they are trained from 1 million apps from Google Play.

5.5 Requirements on Devices/Users (D5)

We next move forward to the features that require users to meet some constraints. For example, when a user installs an app, he/she will be informed of hardware, OS versions, and permissions required by the app. We categorize them into two aspects.

• **Device-specific and SDK requirements.** We count the **device-specific features** (*num_deviceFeature*). Usually, requiring too many device-specific features can narrow the functionalities or user experiences on some specific devices, e.g., some game apps require powerful graphics processing and cannot run smoothly on low-end devices.

We also obtain the **minimum SDK version required for the app to run** (*minSDK*) and the **targeted SDK of the app** (*targetSDK*). These two fields declared in the manifest file express an app's compatibility with one or more versions of the Android platform (OS version), by means of an API level integer [1]. An app can only be installed on a device when the API Level of the given Android system on the device is no lower than *minSDK*. Compatibility features will be disabled if the API Level of the device is higher than *targetSDK*. Such two fields can reflect the potential compatibility risks for an app, as the fragmentation is significant for both devices and OS versions [35, 54].

• **Permissions**. The permission requirements are very significant of an app. Users may be sensitive to permissions accessing private data such as contact, SMS, and camera. Android controls the access to system resources with install-time permissions [19]. Google has officially defined the `protectionLevel`[7] to characterize the potential risk implied by a specific permission. All Android permissions are categorized to three threat levels [19] (i.e., *normal*, *dangerous*, and *signature/system*). We count the **total number of permissions** (*num_userPermission*) and the **number of permissions at the three levels** (*num_normalPermission*, *num_dangerousPermission*, and *num_ssPermission*), respectively.

[6]https://android.googlesource.com/.
[7]https://developer.android.com/reference/android/R.attr.html#protectionLevel.

5.6 Energy Drain (D6)

The energy drain is a significant concern for Android users, and thus apps requiring much battery drain may not be appreciated [47]. In this dimension, we take into account factors that tend to drain battery fast, such as those proposed by Sharkey [46]. First, a `WakeLock` is used to keep the screen from turning off after a given period of time, which can be manually set by developers. We count the **number of `WakeLocks` without timeout** (*num_noTimeoutWakeLock*), which will increase battery consumption. Second, we count the **number of `LocationListener` uses** (*num_LocationListener*) and **number of GPS uses** (*num_GPS*), which are inevitably energy consuming. Third, it was reported that event-based parsers are more efficient than tree-based parsers (usually the DOM parser) [46], and hence we count the **number of uses of *DOMParser*** (*num_DOMPaser*) and **the aggregate number of uses of two common event-based parsers**, i.e., the *XMLPullParser* (*num_XMLPullParser*) and the *SAXParser* (*num_SAXParser*) [47]. Finally, we consider the **number of connection timeouts** (*num_connectionTimeout*) and the **number of socket timeouts** (*num_socketTimeout*). These two kinds of timeouts are set to limit the time length that the app can establish a TCP connection and to keep the connection open without receiving any data, respectively.

5.7 Complexity of Graphical User Interface (D7)

The graphical user interface (GUI), where users interact with the app, is significant to the users' adoption of the app. The layout, styling, color, and smoothness can significantly impact the success of apps. The complexity of GUI reflects the usability and user friendliness, and thus potentially influences users' attitudes. For example, too many input or output elements displayed at the same time can be confusing and frustrating to users [26, 28].

In app design, the *layout* defines the visual structure for a user interface, such as the GUI of an activity or app widget [3]. We count **the number of input/output elements in a layout** (*num_inputUI* and *num_outputUI*). A *view* object draws elements on the screen that the user can interact with [4], and a large number (usually over 80) of views in a layout is considered to have more side effects for app performance [5]. We then calculate the **maximum number of views in a layout declared in XML files** (*max_viewsInAnXML*). To evaluate the GUI complexity on the whole, we also include the **number of layouts** (*num_layout*) and the **volume size of the files** (*size_layout*).

Another concern about the *view* is where it is defined. Android development is based on the classical Model-View-Controller (MVC) design pattern, and thus the views should not be defined in controllers; these controllers are realized by *activities* in Android. We count the **number of views defined in controllers** (*num_viewsInController*), and the **percent of views defined in controllers** (*per_viewsInController*). These two metrics can help indicate the compliance of the MVC pattern.

5.8 Marketing Effort (D8)

In the app-centric ecosystem, developers should upload and release their apps for users to download. Hence, they need some marketing efforts. Such efforts are usually reflected in the app's profile page where users can view and download the app. Although developers can promote their apps through various ways to gain revenue, in this dimension we limit the consideration of marketing efforts to only the Wandoujia app store.

The profile page of an app includes the functionalities and the most impressive information, which shall be useful in persuading users to download and use the app. Among all the elements on the profile page, we choose the **length of textual description** (*len_description*), **number of promotion images** (*num_img*), and **number of categorization tags**[8] (*num_tag*) of the app.

[8]Wandoujia annotates apps to indicate various attributes such as category, usage, and other user-defined tags.

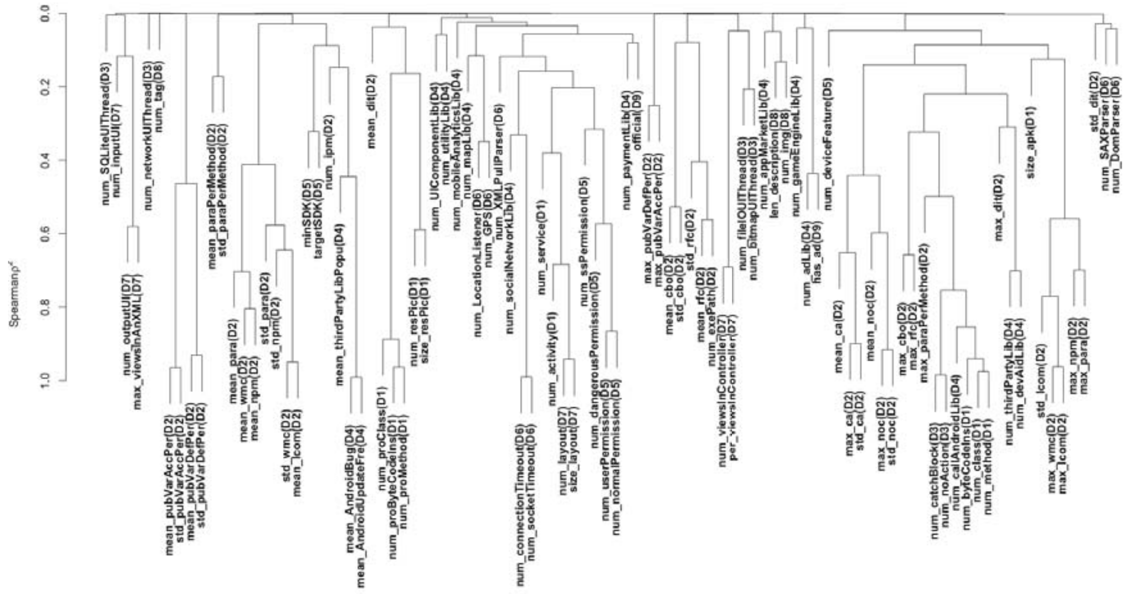


Fig. 3. **Correlation between the developer-controllable features**. 12 of the 108 features with constant values do not apply in the training set. The other 17 features, i.e., 5 features in D1 (*num_proByteCodeIns*, *num_proMethod*, *num_byteCodeIns*, *num_class*, *num_method*), 7 in D2 (*max_ca*, *max_noc*, *std_pubVarAccPer*, *std_pubVarDefPer*, *max_wmc*, *mean_npm*, *mean_lcom*), 1 in D3 (*num_catchBlock*), 1 in D4 (*mean_AndroidBug*), 2 in D5 (*nun_normalPermission*, *num_socketTimeout*), and 1 in D7 (*size_layout*), are filtered out.

5.9 App Store Evaluation (D9)

Serving as the bridge between developers and users, some app stores also make evaluations before the app is released on the market, e.g., anti-virus check, and official-version confirmation. Such a step provides the end users more information and confidence about the app. Indeed, the evaluations are performed by app store operators, but reflect some aspects that are controlled by the developers, e.g., whether the app contains developer-customized features such as ads. These features can possibly affect the user adoption.

According to the Wandoujia data set, we take into account two features. The first one is *has_ad*, indicating whether the app contains in-app ads. The second one is $official$, indicating whether the app is an officially authorized version released by the original developers, as it is possible in Android that third-party developers can customize and repackage an app before they release the app on an app store. Such apps are more likely to be infected as malicious code or libraries [56]. These two metrics are both assigned as binary values (0 or 1).

6 PREDICTING USER ADOPTION

After formulating the indicators and the features, we now manage to build machine learning models to predict user adoption of an app using these features. If such a prediction is effective, the developers may use PRADO to predict user adoption of the app even it is newly developed or added in an app store. The developers may also learn from the prediction and understand which features could play an important role on the success of their app. The goal of PRADO is not only to predict user adoption of apps as accurately as possible, but also to provide interpretable explanations of which features have a significant correlation with the adoption.

To perform the prediction task, we randomly divide the data set into three subsets, i.e., a **training** set with 9,000 apps, a **test** set with 500 apps, and a **validation** set with the remaining 324 apps.

6.1 Feature Correlation Analysis

We begin with a correlation analysis of the features. A straightforward practice is to include all the features described in Section 5 into a prediction model. However, in practice some features are likely to be highly correlated with one another, leading to problems such as collinearity and can seriously affect the robustness of the model and the interpretation of individual features. Therefore, we conduct a feature correlation analysis and carefully select candidate covariates before training the prediction model.

By obtaining the feature values from the training set, we calculate the pair-wise *Spearman coefficient* between every single pair of features, and cluster the features according to the coefficient values. The clustering dendrogram is shown in Figure 3. It is observed that some features are indeed highly correlated and clustered. Most of the highly-correlated features are from the same category. For example, one cluster consists of the features *num_proClass*, *num_proByteCodeIns*, and *num_proMethod*, where all features belong to **D1**, i.e., the size of the app. Another noticeable cluster consists of the features *num_class*, *num_method*, and *num_byteCodeIns* in **D1**, and surprisingly the feature *num_callAndroidLib* in **D4**. One probable reason is that the implementation of functionalities relies on Android APIs, and can somehow explain why *num_callAndroidLib* appears in this cluster. Interestingly, the metrics about the total code base (*num_class*, *num_method*, and *num_byteCodeIns*) and those about the project code base (*num_proClass*, *num_proByteCodeIns*, and *num_proMethod*) are placed into different clusters, demonstrating the necessity of including both perspectives of code size. Another interesting finding is from the clustering of **D2** metrics measuring code complexity. Not all the maximum, mean, and standard deviation of a class-level metric are highly-correlated with one another. For example, *max_ca* and *std_ca* are in the same cluster while *mean_ca* is in another, and *max_lcom* is in the same cluster with *max_wmc* instead of *std_lcom*. Such results demonstrate the advantage of comprehensively measuring the class-level metrics in contrast to simply using the mean value of the metrics. Some indications can also be derived from other clustered features. For example, the cluster of *num_catchBlock* and *num_noAction* indicates that developers tend to be passive in exception handling after claiming `try-catch` blocks.

We then cut the dendrogram using a coefficient threshold of 0.8, resulting in 79 clusters of highly-correlated features (every pair of features in the same cluster has a correlation coefficient of at least 0.8). We preserve only one feature in each cluster, assuming that such a feature represents the whole cluster. Note that a high threshold of correlation is used here to balance the independence and expressive power of the features. As a result, 79 of the 108 features are preserved, none of which has a correlation above 0.8 with any other feature. Accordingly, we use these 79 features (also labeled in Table 5) for validation and testing.

6.2 Regressions

With the selected features and indicators, we explore state-of-the-art machine learning algorithms for the prediction task, i.e., to predict the user adoption of a new app with certain features. In addition to the prediction accuracy, we are also concerned about the interpretability of the model, i.e., how much a feature contributes to the prediction of user adoption. Since the indicators are numerical, the prediction task is naturally cast as a regression problem.

We choose two standard linear-regression models, i.e., Ridge Regression (*Ridge*) [25] and Lasso Regression (*Lasso*) [52]. The two regression models use the same loss function but different regularization terms, where *Ridge* uses L2 regularization and *Lasso* uses L1. λ is a parameter that controls the overall level (intensity) of regularization.

Intuitively, the coefficients of variables learned by either of the regression models indicate the importance of the corresponding features. These coefficients, however, have to be interpreted with caution. First, both linear regression models may suffer from the problem of collinearity, which compromises the stability of the learned coefficients. We aim to alleviate this problem by filtering highly correlated features (Section 6.1). Second, *Lasso* is able to eliminate some input variables (as a type of feature selection) while *Ridge* is not. As a result, the coefficients learned by *Lasso* and *Ridge* are usually not identical. Moreover, different scales of the same variables also affect the numerical values of the coefficient. Compared to the numerical values, the *signs* of the coefficients are more reliable, which should be mostly consistent in *Lasso* and *Ridge*. To deal with the scaling issue, we first standardize every variable of regression following

$$x' = \frac{x - \bar{x}}{\sigma}, \tag{6}$$

where x' is the transformed value of the variable, x is the original value of the variable, and $\bar{x}$ and σ are the mean and standard deviation of the variable in training set, respectively.

Finally, not all features are linearly correlated with the output variable (indicators of user adoption); some may present a non-linear relation and would probably be eliminated by *Lasso*. We include an additional regression model, the *Random Forest* [11] (abbreviated as *RF* in the rest of this paper), which is able to handle non-linear regressions while the importance of each variable is still interpretable. Note that we decide not to include deep neural networks: although they may perform even better in prediction tasks, the models are less interpretable and are normally considered as black boxes.

• **Evaluation metrics**. Treating the observed indicators as the ground truth, we use the *Mean Squared Error* (*MSE*) to measure the accuracy of the predicted results. Hyperparameters of the learning algorithms (e.g., λ in *Lasso* and *Ridge*) are selected by minimizing the *Mean Squared Error* on the validation set. The prediction algorithms are then trained using these parameters on the training set. The effectiveness of each algorithm is measured by making prediction on the test set.

In many real scenarios, a ranking is more practical than absolute values: predicting which apps are more likely to be adopted than others is more important than predicting the actual adoption rate. In addition to *MSE*, we may also rank the apps in the test set by the predicted outcomes, and measure how much it differs from the ground-truth ranking (by observed adoption rates). Statistically, this difference can be measured by a metric called *Kendall's Tau* (τ), which is widely used to compare two ranking lists [39]. It measures the difference between the number of concordant pairs in two ranked lists and the number of discordant pairs, normalized by the total number of ordered pairs:

$$\tau = \frac{\sum_{i,j}(\mathbb{I}[(x_i - x_j)\cdot(y_i - y_j) > 0] - \mathbb{I}[(x_i - x_j)\cdot(y_i - y_j) < 0])}{\sum_{i,j}(\mathbb{I}[(x_i - x_j)\cdot(y_i - y_j) > 0] + \mathbb{I}[(x_i - x_j)\cdot(y_i - y_j) < 0])}, \tag{7}$$

where i, j are two apps, x_i, y_i are the orders of the two apps in two ranking lists, and $\mathbb{I}[\cdot]$ is an indicator function whose value equals to 1 if the expression is true and 0 otherwise. The value of τ is 1 if the two rankings are identical, -1 if one is completely opposite of the other, and 0 if they are independent.

Compared to other ranking-based metrics such as mean average precision and NDCG, Kendall's τ considers the entire ranking list equally while others typically focus on the top-ranked items. Such characteristic is particularly useful in situations where underrepresented or new apps are considered to be important [30]. Note that the

Table 1. Model effectiveness. *MSE*: the lower the better; τ: the higher the better.

Model	U-I Ratio		Like Ratio		Download Percentile	
	τ	*MSE*	τ	*MSE*	τ	*MSE*
Lasso	0.280	0.028	0.150	0.054	0.236	0.080
Ridge	0.314	0.027	0.135	0.061	0.220	0.084
RF	0.331	0.025	0.252	0.047	0.316	0.070
Baseline	-0.016	0.075	0.010	0.075	-0.011	0.091

numerical values of the τ metric may appear rather small in practice. Indeed, having 20% items completely out of order may have reduced τ to somewhere below 0.3 instead of 0.8.

For comparison purposes, there are also naive baselines to consider such as uniform guesses. For example, we may simply guess the *U-I Ratio* as 0 for each app in the test set and calculate the *MSE*. We also guess it as 0.5 and 1, respectively, and choose the lowest *MSE* in the three conditions as the baseline result. For the ranking effectiveness, we adopt random ranking of apps in the test set, and calculate τ with comparison to the ground truth. The results are shown in Table 1.

6.3 Results

We next report the prediction results for three indicators of user adoption, *U-I Ratio, Rating*, and *Number of Downloads*, respectively.

6.3.1 U-I Ratio. To predict *U-I Ratio* for a new app, we first use the validation set of 324 apps to tune the hyperparameter λ of the *Lasso* model through cross validation (using R package `cv.glmnet`). We choose the $\lambda.min$ as 0.012, which minimizes the out-of-fold deviance. We then train the *Lasso* model with the generated λ using the training set of 9,000 apps. On the test set, we calculate τ and *MSE* of the predicted result of the trained model. As shown in Table 1, *Lasso* achieves a τ of 0.280 and an *MSE* of 0.028.

Training *Ridge* is similar, and the effectiveness turns out to be better: $\tau = 0.314$ and $MSE = 0.027$. The τ of *RF* is 0.331 and the *MSE* is 0.025, which is the best among the three regressors. Such result indicates that the correlation between the features and *U-I Ratio* may not be linear, and it is beneficial to use a non-linear regression model.

All three regression models trained with our selected features outperform the baseline results for both τ and *MSE*. The lower *MSE* means that our models yield more accurate predictions than the uniform guess baseline. The higher τ indicates that the app list ranked by *Lasso*/*Ridge*/*RF* predicted *U-I Ratio* is more consistent with the ground-truth ranking, compared to the random ranking.

6.3.2 Rating. Compared to Google Play, which uses a starring model for app rating, the Wandoujia app store employs a simple binary model, i.e., "*Like*" or "*Dislike*". In this way we employ the metric of *Like Ratio*, which is calculated as *the number of likes* divided by the number of (*likes+dislikes*) as the indicator of app rating. Such a metric has been shown to be consistent with ratings in Google Play [30].

It was reported that using ratings to evaluate overall user attitude toward an app can lead to biases [35, 43]. Some researchers leverage the criteria of no fewer than 10 raters [43] to address such a bias. However, even when an app has enough ratings, bias can still exist. Consider when the positive and negative ratings have no significant difference, one single rating would have flipped the sign of the gold standard, even when there are an enormous number of raters. Thus threshold criteria cannot well address such a problem.

To avoid possible bias, we filter the apps in the validation set and test set with a **statistical test** as we have done to the whole data set in our previous work [30]. We use a two-tail proportion test with confidence level of

Table 2. Significant features for user-adoption indicators.[1]

U-I Ratio		Number of Downloads (Download Percentile)	
rank	**feature**	**rank**	**feature**
1	****len_description(D8)+**	1	****official(D9)+**
2	num_proClass(D1)	2	****num_callAndroidLib(D4)+**
3	num_noAction(D3)	3	****num_service(D1)+**
4	size_apk(D1)	4	len_description(D8)
5	max_lcom(D2)	5	****num_layout(D7)+**
6	****official(D9)–**	6	****num_noAction(D3)+**
7	num_resPic(D1)	7	****num_userPermission(D5)+**
8	num_callAndroidLib(D4)	8	****num_tag(D8)+**
9	****num_img(D8)+**	9	****size_apk(D1)+**
10	num_layout(D7)	10	targetSDK(D5)
11	max_rfc(D2)	11	****max_cbo(D2)+**
12	num_userPermission(D5)	12	num_resPic(D1)
13	max_npm(D2)	13	max_rfc(D2)
14	max_cbo(D2)	14	****num_proClass(D1)+**
15	max_paraPerMethod(D2)	15	mean_AndroidUpdateFre(D4)
16	std_ca(D2)	16	num_activity(D1)
17	std_lcom(D2)		
18	mean_thirdPartyLibPopu(D4)		
19	max_para(D2)		
Rating (Like Ratio)			
rank	**feature**	**rank**	**feature**
1	****num_resPic(D1)+**	21	num_dangerousPermission(D5)
2	****num_layout(D7)+**	22	****max_viewsInAnXML(D7)+**
3	****len_description(D8)+**	23	max_rfc(D2)
4	****num_callAndroidLib(D4)+**	24	mean_thirdPartyLibPopu(D4)
5	****mean_AndroidUpdateFre(D4)–**	25	std_rfc(D2)
6	num_noAction(D3)	26	mean_pubVarDefPer(D2)
7	std_ca(D2)	27	mean_rfc(D2)
8	num_activity(D1)	28	num_outputUI(D7)
9	size_resPic(D1)	29	num_viewsInController(D7)
10	num_proClass(D1)	30	std_lcom(D2)
11	size_apk(D1)	31	std_wmc(D2)
12	num_userPermission(D5)	32	max_lcom(D2)
13	****targetSDK(D5)+**	33	std_npm(D2)
14	mean_cbo(D2)	34	num_exePath(D2)
15	mean_ca(D2)	35	max_paraPerMethod(D2)
16	std_noc(D2)	36	num_connectionTimeout(D6)
17	mean_noc(D2)	37	mean_para(D2)
18	mean_pubVarAccPer(D2)	38	max_para(D2)
19	std_cbo(D2)	39	std_dit(D2)
20	max_npm(D2)		

[1] Each of the listed features is with a $\%IncMSE$ value larger than 10^{-3}, which is generated by `rf$importance` instruction in R. Features in bold are also selected by *Lasso* and the +/- indicates the sign of correlation.

0.05. Apps that pass such a test have abundant ratings and a significant difference between positive and negative ratings. Finally, we have 131 apps remaining in the validation set and 178 apps in the test set.

Similarly to *U-I Ratio*, we train the *Lasso*, *Ridge*, and *RF* models for the *Like Ratio* prediction. The τ and *MSE* values are shown in Table 1. Consistently, all three models outperform the baselines, and *RF* performs the best among the three models. *Lasso* outperforms *Ridge* this time, yielding a higher τ and a lower *MSE*.

6.3.3 Number of Downloads. We then investigate the indicator *Number of Downloads*, which is traditionally the most adopted indicator for app popularity. As reported in our previous work [30], the distribution of *# downloads* follows a *Power Law*. The long tail distribution suggests that the model evaluation is challenging, as *MSE* uses a normal assumption and is not applicable. To address this issue, we transform the *Number of Downloads* into *Download Percentile against Rankings* and use the latter as the target of prediction. That is, we rank the apps according to *their number of downloads* and calculate the percentile position that each app holds against the least downloaded app. For example, in a set of n apps, the *Download Percentile* value of the app with the least downloads (ranks n) is $(n - n)/n = 0$, and the value of the app with the most downloads (ranks 1) is $(n - 1)/n$. A higher *Download Percentile* indicates more downloads. The evaluation metrics are shown in Table 1. Conclusions are similar to those of *Like Ratio*.

7 INTERPRETATIONS

So far, we have performed the regression models and demonstrated their validity and effectiveness. As mentioned earlier, another important goal of our work is to interpret the learned models, or to identify the influences of individual features on every indicator.

7.1 Approach

As introduced earlier, the coefficients learned by *Lasso* and *Ridge* after variable standardization can be interpreted as the importance of the corresponding features. *Lasso* further eliminates unimportant features. Interpreting feature importance learned by *RF* is less straightforward. In Table 2, we rank the importance of features ranked by the *%IncMSE* values generated by the instruction `rf$importance` after *RF* regression, which measures the increase rate of *MSE* if a feature is permuted in the model, with the threshold of 10^{-3} to obtain statistical significance. A higher *%IncMSE* indicates that the variable is more significant to the outcome (i.e., the indicator) of the model.

One limitation of this approach is that it can select only the statistically significant features, but does not report whether such a feature is positively or negatively correlated with the outcome. Based on the significant features ranked by *RF*, we further check whether the features are also selected by *Lasso* and the **signs** of these features. Such features in Table 2 are labeled bold and tagged with the sign +/−. Note that some features are significant in *RF* but not in *Lasso*, since *Lasso* is a linear model, and variables that are not linearly correlated with the indicators can be filtered out. Such a synthesis can improve the interpretability of results.

7.2 Results

Interestingly, we first find that the significant features are quite different for the three indicators. Specifically, *RF* selects 19 features for *U-I Ratio*, 39 features for *Rating*, and 16 features for *Number of Downloads*. There are 44 features as the union set for all the three indicators. Such a finding indicates that developers should have more comprehensive considerations rather than relying on the features that are indicator-specific.

As mentioned earlier, the features selected by *RF* cannot explicitly indicate the positive/negative correlations to the indicators. To better interpret the results, we focus on the significant features selected by *Lasso*, and make pair-wise comparisons accordingly.

For the indicator *Rating*, *Lasso* selects 6 features as positive and 1 feature as negative. In other words, apps in our data set, which have more resource pictures (*num_resPic*), layout elements (*num_layout*, implying richer UI), and longer textual description (*len_description*, implying more informative description), rely on more Android APIs (*num_callAndroidLib*) but less-frequently changed APIs (*mean_AndroidUpdateFre*), require higher version of SDK (*targetSDK*), more views in an XML file (*max_viewsInAnXML*), are more likely to gain higher ratings. Similarly, apps that are official ($official = 1$), rely on more Android APIs (*num_callAndroidLib*), require more permissions (*num_userPermission*), and have more services (*num_service*), layout elements (*num_layout*), empty `try-catch blocks` (*num_noAction*), more tags (*num_tag*), larger size of `.apk` files (*size_apk*), more classes in project code (*num_proClass*), and more coupling (*max_cbo*), are more likely to gain more downloads.

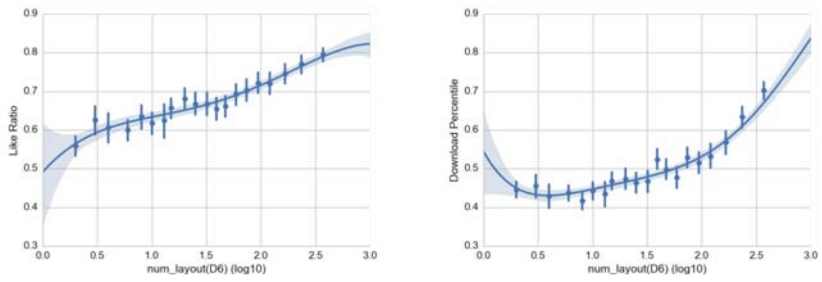
Indeed, some of the preceding features can be interpreted and also quite actionable by developers in practice. For example, a higher number of images (*num_img*) displayed at app stores can imply that users' first impression is significant; longer textual descriptions (*len_description*) and more descriptive information (*num_tag*) can make an app more informative, and thus may lead to users' higher interest in downloading the app. In this way, developers can use more images and enrich the descriptions to illustrate/translate the app's functionalities and features. Other positive examples are the number of calls to Android APIs (*number_callAndroidLib*) and the version of target SDK (*targetSDK*), encouraging developers to more rely on Android native APIs and build their apps over a newer version of Android SDK. Additionally, using those frequently updated Android APIs (*mean_AndroidUpdateFre*) indicates the negative correlation, since these APIs may not be stable enough and even lead to failures. As a result, developers shall pay more attention when using such APIs.

Note that some features could be interpreted, but may not be immediately or simply addressed. For example, the feature *size_apk* seems to be positive to those more-downloaded apps. Indeed, the larger size of an app usually implies more functionalities and features. However, it does not mean that the developers should intentionally increase the size of their apps. A similar observation is also held for the feature *num_proClass*.

Interestingly, some results even seem to be a bit **counter-intuitive**. For example, the higher *max_cbo* seems to be a positive feature to the downloads; such result is a bit contradictory to the suggested "lower-coupling-between-class" principle in traditional software engineering practice. Although we currently cannot derive the underlying reasons, such observations could be very interesting to software engineering researchers and practitioners, and even make them rethink some specific characteristics to app development.

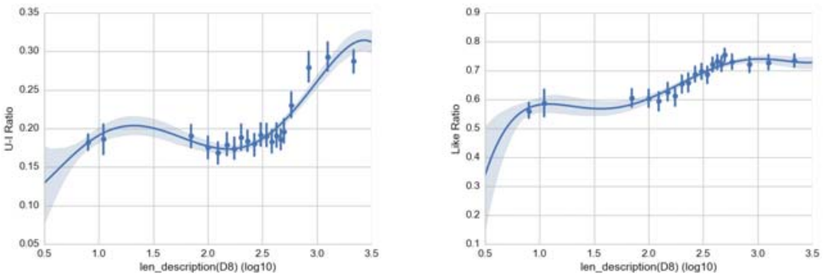
Some features are important to more than one indicator. For example, $official$ is negatively correlated with *U-I Ratio* and positively correlated with *Download Percentile*. In other words, authentication can help the app to reach more users and to be less likely uninstalled. The feature *num_layout*, as shown in Figure 4, is positively correlated with both *Download Percentile* and *Like Ratio*. This result is within expectations because more layout files may indicate more functionalities and features, and thus may gain more downloads and higher rating.

Interestingly, we also find some contradictory features for different indicators. For the indicator *U-I Ratio*, we can find that apps with non-official versions (i.e., $official$=0), longer textual descriptions (*len_description*), and more promotional images (*num_img*) are more likely to be uninstalled. The first selected feature ($official$) complies with common knowledge, since the quality of non-official apps may not be assured, and the infection rate of non-official apps is reported to be higher than official apps [56]. However, it is a bit surprising to find that an app having longer textual description can possibly lead to higher *U-I Ratio*. In contrast, for the indicator *Like Ratio*, the feature *len_description* is positively correlated with it, indicating that apps with longer descriptions may have higher ratings. We further investigate how this feature is correlated with the two indicators. We employ the bin-bin plot to illustrate such a correlation (we set 20 bins for each plot and fit the data with 5-degree polynomial regression), as shown in Figure 5. We find that the increase of *U-I Ratio* is quite flat when the description is shorter than the average length. Hence, it is possible to find an equilibrium where the *U-I Ratio* is relatively low and the *Like Ratio* is relatively high, i.e., the length of textual description is not necessarily too long. The underlying



(a) Rating with number of layout files. (b) Download percentile with number of layout files.

Fig. 4. Consistence between different indicators.



(a) U-I Ratio with length of textual description. (b) Rating with length of textual description.

Fig. 5. Divergence between different indicators.

reason could be that too long descriptions may require more time and more scroll-down actions, but users may feel impatient to read them through.

7.3 Correlation, NOT Causality

Although the preceding machine learning process identifies some statistically significant features, we should clarify that the current main goal of our PRADO approach is to explore whether there are tangible features for possibly affecting the user adoption of an app. PRADO currently performs only **correlation analysis** by employing machine learning algorithms. Although we synthesize various features and user-adoption indicators, the findings derived from our correlation analysis may not be fully generalized to imply the underlying causality. Indeed, such a limitation also exists in previous studies that analyze relationships between app characteristics and success (or popularity) by mining code, software repositories, and app stores [10, 12, 21, 23, 50, 51]. These existing studies along with PRADO indeed identify the possible significant features of these successful apps. The derived features may not be **thoroughly** interpreted or actionable, but can still provide suggestions to developers. In future work, we plan to combine advanced causality analysis techniques [48] and developer interviews [53] to validate the findings. In practice, our findings provide guidance for the developers to design and prioritize A/B tests to validate the causal effect of particular features.

Table 3. Model effectiveness on *GAME* apps. *MSE*: the lower the better; τ: the higher the better.

Model		U-I Ratio		Like Ratio		Download Percentile	
		τ	MSE	τ	MSE	τ	MSE
Game Model	Lasso	0.455	0.035	–	0.059	0.363	0.094
	Ridge	0.410	0.039	0.085	0.053	0.293	0.095
	RF	0.510	0.030	0.183	0.047	0.429	0.061
General Model	Lasso	0.437	0.040	0.147	0.048	0.305	0.087
	Ridge	0.450	0.039	0.108	0.048	0.273	0.093
	RF	0.491	0.032	0.242	0.046	0.415	0.068

8 APPLYING PRADO ON SPECIFIC APPS: A CASE STUDY

In the aforementioned demonstration of PRADO, apps in all categories are mixed together. However, it is possible that some factors may work only on some certain types of apps. For example, apps with different functionalities, such as a game app and a financial app, can have different user groups that have potentially different user preferences. To validate whether the preceding results of PRADO are generalizable and robust across categories, we narrow the scope of the apps to a specific domain, i.e., *GAME*, and explore the effectiveness of PRADO on this particular category of apps.

In our data set, the *GAME* domain includes 2,658 apps, which constitute the data set in the category-specific validation. The data set is divided into three subsets, i.e., a **training** set (2,438 apps), a **validation** set (84 apps), and a **test** set (136 apps). The partition of the sets is consistent with that described in Section 6. In particular, the 2,438 *GAME* apps in the training set are also part of the training set in Section 6. The situation is similar to the validation set and the test set.

8.1 Regressions

Similarly, we train the *Ridge*, *Lasso*, and *RF* models on the *GAME* set (abbreviated as *Game Model*) to predict the three adoption indicators. The evaluation metrics, τ and *MSE*, are calculated and shown in Table 3. As shown in the table, *RF* outperforms *Lasso* and *Ridge* with a higher τ and a lower *MSE* for each indicator. Note that we fail to generate τ of *Lasso* for *Like Ratio* because the predicted *Like Ratio*s are a constant.

To conclude the effectiveness of the *Game Model* of PRADO in comparison to the models trained on the entire set (abbreviated as *General Model*), we need to perform the *General Model* on the same test set of games. The values of the adoption indicators are also shown in Table 3. Again, *RF* outperforms the two linear models.

For *U-I Ratio* and *Download Percentile*, the *Game Model* and *General Model* have quite similar performance, with slightly higher τ and lower *MSE* of the *Game Model* using *RF*. However, the *General Model* outperforms the *Game Model* in *Like Ratio*. In particular, *Lasso* in *Game Model* fails to generate a τ because of constant outputs, and the value of τ for *RF* in the *Game Model* is 0.183 while that in the *General Model* is as high as 0.242. As observed, the *General Model* is more robust and is generalizable to a specific category of apps.

8.2 Interpretations

Given that the *Game Model* achieves satisfactory results for the *U-I Ratio* and the *Download Percentile* in comparison to those of the *General Model*, it is motivated that the important features selected by the *Game Model* have quite similar prediction power while being more specific to those *GAME* apps. The *RF*-selected features are shown in Table 4, and the *Lasso*-selected features are labeled bold and tagged with the sign +/−. Note that *Lasso* fails to select any feature for *Like Ratio* because the output coefficients of the features are all 0.

We interpret only from the two perspectives of *U-I Ratio* and *Download Percentile*. In comparison to Table 2, the number of important features is reduced. For *U-I Ratio*, 4 out of the 7 features are also selected by the *General Model*, and the signs are consistent. The other 3 features, i.e., *num_exePath*, *max_paraPerMethod*, and *num_adLib*, are

Table 4. Significant features for user-adoption indicators in *GAME* apps.[1]

U-I Ratio		Number of Downloads (Download Percentile)	
rank	**feature**	**rank**	**feature**
1	****len_description(D8)+**	1	****num_tag(D8)+**
2	****official(D9)–**	2	****official(D9)+**
3	num_exePath(D2)	3	len_description(D8)
4	****num_img(D8)+**	4	num_noAction(D3)
5	****max_paraPerMethod(D2)+**	5	size_apk(D1)
6	num_callAndroidLib(D4)	6	std_ca(D2)
7	****num_adLib(D4)+**	7	max_viewsInAnXML(D7)
		8	num_userPermission(D5)
		9	num_callAndroidLib(D4)
Rating (Like Ratio)			
rank	**feature**	**rank**	**feature**
1	size_apk(D1)	15	num_outputUI(D7)
2	num_callAndroidLib(D4)	16	num_dangerousPermission(D5)
3	len_description(D8)	17	mean_pubVarDefPer(D2)
4	mean_AndroidUpdateFre(D4)	18	std_dit(D2)
5	std_noc(D2)	19	max_cbo(D2)
6	num_resPic(D1)	20	size_resPic(D1)
7	mean_thirdPartyLibPopu(D4)	21	mean_rfc(D2)
8	mean_noc(D2)	22	mean_cbo(D2)
9	std_ca(D2)	23	mean_wmc(D2)
10	num_noAction(D3)	24	std_para(D2)
11	mean_ca(D2)	25	max_npm(D2)
12	num_userPermission(D5)	26	num_thirdPartyLib(D4)
13	max_rfc(D2)	27	mean_para(D2)
14	mean_pubVarAccPer(D2)	28	max_para(D2)

[1] Each of the listed features is with a $\%IncMSE$ value larger than 10^{-3}, which is generated by `rf$importance` instruction in R. Features in bold are also selected by *Lasso* and the +/- indicates the sign of correlation.

considered to be more important for the *GAME* apps than for all the apps. Such a result is reasonable because *num_exePath* and *max_paraPerMethod* can be related to the fluency of running a game app, and *num_adLib* can influence user experiences when using a game app. A higher *num_adLib* is correlated with a higher *U-I Ratio*, which indicates a higher tendency to uninstall an app. For *Download Percentile*, most of the selected features are also selected by the *General Model*.

8.3 Discussion

By performing the different models on the test set of *GAME* apps, we find that the *General Model* trained with all apps can achieve competitive results with the *Game Model* for two indicators (*U-I Ratio* and *Download Percentile*), and is observed to outperform in *Like Ratio*. One potential reason for the unsatisfactory results of the *Game Model* (by the *Like Ratio*) may be the reduction of the size of the data set, which would be more serious for smaller app categories (see Figure 2). In summary, we believe that the *General Model* of PRADO is robust enough and can be generalized to specific categories of apps.

9 LIMITATIONS AND DISCUSSION

As a measurement and correlation-analysis study, considerable care and attention should be given to ensure the rigor. However, as with any chosen research methodology, it is unlikely free of limitations. We next discuss some limitations or threats to validity in our evaluations.

Generalization of findings. A major threat to validity is using apps from only Wandoujia (primarily in the China market). All apps along with their user behaviors, including installation, uninstallation, and ratings, are collected from Wandoujia. The large scale of user-behavioral data (from over 12 million users) could make the results statistically meaningful and thus help developers when they publish their apps on Wandoujia, in terms of optimizing release planning, app development, and ranking. We believe that the current results are informative, since there are more than 1 million apps and 300 million users on Wandoujia. It is worth mentioning that developers who publish their apps on other stores such as Google Play may not be able to immediately generalize all knowledge derived in this paper as reference to predict their app adoption, or identify those features that are potentially significant. However, the idea and correlation-analysis process of PRADO are quite general and can be applied to all apps. On one hand, most of the extracted features such as app size, code complexity, employed libraries, marketing efforts, are standard and general to all Android apps, no matter where they are published. On the other hand, the user behaviors such as the number of downloads, ratings, and uninstallation (despite not always available) are also not specific to any app store. With similar data collected by other app stores, we believe that one can easily reproduce PRADO and derive the desired results. In future work, we plan to apply PRADO on the apps published on Google Play along with their public user-adoption information. It would be interesting to validate whether the results and significant features are consistent with the ones inferred in this paper.

Practical applicability. PRADO is principally designed for two goals: helping app developers predict the possible user adoption and identifying the possible significant features. In other words, PRADO aims to provide some insightful references to developers for them to better allocate their resources and optimize their apps. The features in the feature taxonomy are selected from various aspects, but are general to all Android apps and can be fully controlled by their developers. Some of the selected features (such as those in D8) can be immediately actionable by developers while others can be performed only in a long-term way. In practice, developers are able to perform A/B tests to validate whether their apps can win better adoption with these PRADO-selected features addressed.

Feature taxonomy. All the features in PRADO are defined by classical software engineering metrics and extracted based on static analysis. However, the static-analysis tool used in this work may be insufficient to evaluate the behavior and performance of apps at runtime. For example, the results indicate that the energy drain features are not sensitive to *U-I Ratio*. Such a finding is a bit surprising since energy drain is considered to be a significant factor affecting the experience of smartphone users [15]. To further explain such observations, we plan to adopt more comprehensive measurements, such as more advanced static analysis or dynamic analysis techniques, or even human-intervention cooperative analysis, in future work.

Time and app versions. Another threat to validity in our evaluations is the time and the versions of apps. We evaluate PRADO with a five-month-long data set and the first-released version of apps during this time span. However, the data set does not record the detailed information of management activities performed against a specific app version. In practice, developers can frequently update and release their apps. If a user downloads an older app version released before the starting time of our data set, and uninstalls or rates the app during the five months, there can be mismatches. In the future, the Wandoujia app store will consider recording the precise version of an app in the management activities. In addition, it would be an interesting topic to explore the change of user adoption along with the evolution of app versions.

Free apps vs. paid apps. The apps on Wandoujia are all free. Certainly, it is possible that user behaviors on paid apps are different [44]. Such inclusion of only free apps is an inherent limitation of our data set. Given that free apps account for a large percentage of apps, our results are still meaningful.

Feature extraction tools. Indeed, PRADO relies on existing tools such as `Dex2Jar` and `CKJM` to extract features. These tools failed on some apps (such as certain anonymized apps like WeChat and QQ), which are excluded from the study, and thus our evaluations contain only 9,000+ apps. To alleviate such a limitation, we need to explore alternative effective tools in future work.

Cross-platform issues. The features used in this paper are quite standard for all OO applications, but extracted from Android apps that are implemented in the Java language. These features may require different measurement tools and metrics in iOS and Windows Mobile applications. However, the idea of PRADO is still generalizable if these features are available.

10 CONCLUSION AND FUTURE WORK

We have presented the PRADO approach, which predicts user adoption of an Android app using app information and large-scale user behaviors. Based on a well-established feature taxonomy, we apply advanced machine learning algorithms to derive the correlations between the features and the adoption behaviors from large-scale users. The derived models of PRADO can accurately predict how a new app would be managed, rated, and downloaded by users. We are also able to synthesize and suggest those potentially influential features for affecting user adoption. In future work, we plan to focus on causality analysis of the suggested features and the user adoption. Such insightful knowledge can be used to help Android developers take direct actions for improving user adoption of their apps.

APPENDIX

In this appendix, we summarize the developer-controllable features used in this paper, so that the readers can fast trace and understand them. Features in **bold** and *Italic* are those preserved after the feature correlation analysis (see Section 6.1).

Table 5. Summary of features.

Feature Name	**Description**
	App Size (D1), 11 features
size_apk	The size of `.apk` file, measured in kilo byte (KB). A larger size can imply richer functionalities.
num_class	The number of all classes in the app. More classes can imply more functionalities.
num_proClass	The number of classes in the project code (code excluding libraries).
num_method	The number of all methods in the app.
num_proMethod	The number of methods in the project code.
num_byteCodeIns	The number of byte code instructions decompiled from the `.apk` file, which implies the volume of source code.
num_proByteCodeIns	The number of byte code instructions in the project code, decompiled from the `.apk` file.
num_activity	The number of activities. Each *activity* represents a single screen with a user interface.
num_service	The number of services. A service runs at the background and keeps working even if the app is destroyed [7].

num_resPic	The number of pictures in the resource file, implying the richness of UI.
size_resPic	The volume of pictures in the resource file.
Code Complexity (D2), 38 features	
mean_wmc max_wmc ***std_wmc***	The weighted methods per class, originally defined as the sum of complexity of all methods in the class [49].
mean_dit ***max_dit*** ***std_dit***	The depth of inheritance tree. A larger depth implies greater complexity [49].
mean_noc max_noc ***std_noc***	The number of children: the number of immediate subclasses subordinated to a class in the class hierarchy [49].
mean_cbo ***max_cbo*** ***std_cbo***	The degree of coupling between object classes. A less degree is encouraged [49].
mean_rfc ***max_rfc*** ***std_rfc***	The response for a class: a set of methods potentially executable in response to a message received by an object of that class [49].
mean_lcom ***max_lcom*** std_lcom	The lack of cohesion in methods: the number of method pairs whose similarity is zero minus the number of method pairs whose similarity is not zero [49].
mean_ca max_ca ***std_ca***	The afferent coupling: the number of classes (outside a category of classes) that depend upon classes within this category [49].
mean_npm ***max_npm*** ***std_npm***	The number of public methods in a class [9]. A lower number is desired.
mean_pubVarDefPer ***max_pubVarDefPer*** std_pubVarDefPer	The percentage of public/protected fields to all the fields defined in a class. A lower number is desired [9].
mean_pubVarAccPer ***max_pubVarAccPer*** std_pubVarAccPer	The percentage of access to public/protected fields.
mean_para ***max_para*** ***std_para***	The number of parameters of methods per class. A lower number is desired [9].
mean_paraPerMethod ***max_paraPerMethod*** ***std_paraPerMethod***	The number of parameters per method.
num_ipm	The number of byte-code instructions per method.
num_exePath	Cyclomatic complexity, or possible paths of execution, counted as the number of conditional statements plus one.
Code Quality (D3), 17 features	

num_catchBlock	The number of `try-catch blocks`. A larger number indicates better consideration of exception handling.
num_logOnly	The number of `try-catch blocks` with only logging instructions.
num_noAction	The number of empty `try-catch blocks` without explicit exception handling. Intuitively, such a programming style is not encouraged.
num_networkUIThread	The number of API calls that were reported to be possibly leading to Application-Not-Responding (ANR) [55] issues, i.e., *network access*, *flash storage access*, *database access*, and *bitmap processing* in a UI thread.
num_fileIOUIThread	
num_SQLiteUIThread	
num_bitmapUIThread	
per_ucDimiss	The number of the top-10 methods reported by Kechagia and Spinellis [29], i.e., the methods *dismiss*, *show*, *setContentView*, *createScaledBitmap*, *onKeyDown*, *isPlaying*, *unregisterReceiver*, *onBackPressed*, *showDialog*, and *create*. Here, we compute the percentage of calls to such a method without explicit `try-catch blocks` relative to total calls to the method. Intuitively, a higher percentage can imply more probability to lead to exceptions.
per_ucShow	
per_ucSetContentView	
per_ucCreateScaleBitmap	
per_ucOnKeyDown	
per_ucIsPlaying	
per_ucUnregisterReceiver	
per_ucOnBackPressed	
per_ucShowDialog	
per_ucCreate	
Libraries (D4), 15 features	
num_callAndroidLib	The number of calls to Android APIs in libraries starting with “`android`”. Intuitively, using Android APIs is encouraged.
mean_AndroidUpdateFre	The average of update frequencies of involved Android APIs. The more-frequently changed APIs imply more error-proneness, and is discouraged [42].
mean_AndroidBug	The average number of bug-fixing actions of involved Android APIs by mining the updating logs. The more bug-fixing actions of an API can indicate more error-proneness.
num_thirdPartyLib	The number of third-party libraries identified with the LibRadar tool [38]. More usage of third-party libraries is reported to enrich functionalities [51].
num_adLib	The number of imported libraries in the top-10 popular categories defined by Ma *et al.* [38], i.e., *advertisement*, *social network*, *mobile analytics*, *development aid*, *app market*, *payment*, *UI component*, *game engine*, *utility*, and *map*.
num_socialNetworkLib	
num_mobileAnalyticLib	
num_devAidLib	
num_appMarketLib	
num_paymentLib	
num_UIComponentLib	
num_gameEngineLib	
num_utilityLib	
num_mapLib	
mean_thirdPartyLibPopu	The average popularity of third-party libraries used. A higher value indicates better usage of third-party libraries.
Requirements on Devices/Users (D5), 7 features	

num_deviceFeature	The number of required device features. A larger value can narrow the functionality or user experiences, and is discouraged.
minSDK	The minimum SDK level required for the app to run.
targetSDK	The target SDK level. Intuitively, a higher SDK version is encouraged.
num_userPermission	The number of required permissions. More permissions required indicate not only more functionalities but also more threats [19].
num_normalPermission	The number of required permissions at the normal level.
num_dangerousPermission	The number of required permissions at the dangerous level. Intuitively, such permissions should be avoided [19].
num_ssPermission	The number of required permissions at the signature or signatureOrSystem level.
Energy Drain (D6), 8 features	
num_noTimeoutWakeLock	The number of WakeLocks without timeout. A larger number indicates more battery consumption [46].
num_LocationListener	The number of `LocationListener` uses. A larger number indicates more battery consumption [46].
num_GPS	The number of GPS uses. A larger number indicates more battery consumption [46].
num_DOMParser	The number of `DOMParsers`. A larger number indicates better performance and less battery consumption [46].
num_XMLPullParser	The number of `XMLPullParsers`. A larger number indicates better performance and less battery consumption [46].
num_SAXParser	The number of `SAXParsers`. A larger number indicates better performance and less battery consumption [46].
num_connectionTimeout	The number of connection timeouts. A larger number indicates more carefulness in preventing network hanging, and maybe less battery consumption [46].
num_socketTimeout	The number of socket timeouts. A larger number indicates less battery consumption [46].
Complexity of Graphical User Interface (D7), 7 features	
num_inputUI	The number of input elements per layout. A larger number indicates more confusion and frustration of users for too many input elements displayed at the same time [26, 28].
num_outputUI	The number of output elements per layout. Users are likely to be confused and frustrated with a large number of output elements displayed at the same time [26, 28].
max_viewsInAnXML	The maximum number of views in a layout. A large number (usually over 80) of views in a layout is considered to be negative for app performance [5].
num_layout	The number of layout files. A larger number may indicate more functionalities and more complex GUI.
size_layout	The size of layout files. A larger number may indicate more functionalities and more complex GUI.

num_viewsInController	The number of views defined in controllers. A larger number indicates further deviation from the MVC pattern.
per_viewsInController	Percent of views defined in controllers. A larger number indicates further deviation from MVC pattern.
Marketing Effort (D8), 3 features	
len_description	Length of textual description on the profile page. A longer description indicates more adequate introduction to the app.
num_img	The number of promotion images on the profile page. A larger number indicates more adequate introduction to the app.
num_tag	The number of categorization tags on the profile page. A larger number indicates more adequate introduction of the app, and more functionalities.
App Store Evaluation (D9), 2 features	
has_ad	Whether the app has ads, assigned as a binary value (0 or 1). Having adds may disturb users.
official	Whether the app is official: “1” means official and “0” otherwise. An official version is less likely to be infected or malicious [56].

REFERENCES

[1] 2015. Android SDK. https://developer.android.com/guide/topics/manifest/uses-sdk-element.html. (2015).
[2] 2015. Keeping Your App Responsive. https://developer.android.com/training/articles/perf-anr.html. (2015).
[3] 2015. Layout. https://developer.android.com/guide/topics/ui/declaring-layout.html. (2015).
[4] 2015. UI Overview. https://developer.android.com/guide/topics/ui/overview.html. (2015).
[5] 2016. Android Lint Checks. http://tools.android.com/tips/lint-checks. (2016).
[6] 2017. Android-Activities. http://www.tutorialspoint.com/android/android_acitivities.htm. (2017).
[7] 2017. Android-Services. http://www.tutorialspoint.com/android/android_services.htm. (2017).
[8] 2017. App Annie. https://www.appannie.com. (2017).
[9] Jagdish Bansiya and Carl G. Davis. 2002. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on software engineering* 28, 1 (2002), 4–17.
[10] Gabriele Bavota, Mario Linaresvasquez, Carlos Bernalcardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2015. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering* 41, 4 (2015), 384–407.
[11] L Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.
[12] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. 2014. AR-Miner: Mining Informative Reviews for Developers from Mobile App Marketplace. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*. 767–778.
[13] Pern Hui Chia, Yusuke Yamamoto, and N Asokan. 2012. 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, WWW 2012*. 311–320.
[14] 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.
[15] Jürgen Cito, Julia Rubin, Phillip Stanley-Marbell, and Martin Rinard. 2016. Battery-Aware Transformations in Mobile Applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. 702–707.
[16] Paul Coulton and Will Bamford. 2011. Experimenting Through Mobile ’Apps’ and ’App Stores’. *International Journal of Mobile Human Computer Interaction* 3, 4 (2011), 55–70.
[17] William H. DeLone and Ephraim R. McLean. 2003. The DeLone and McLean Model of Information Systems Success: A Ten-Year Update. *Journal of Management Information Systems* 19, 4 (2003), 9–30.
[18] Neil F. Doherty, Malcolm King, and Omar Al-Mushayt. 2003. The Impact of Inadequacies in The Treatment of Organizational Issues on Information Systems Development Projects. *Information & Management* 41, 1 (2003), 49–62.
[19] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*. 627–638.
[20] Brad Fitzpatrick. 2010. Writing Zippy Android Apps. In *Google I/O Developers Conference*.
[21] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. 2013. Why People Hate Your App: Making Sense of User Feedback in a Mobile App Store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013*. 1276–1284.
[22] Robert D. Galliers and Jacky Swan. 2000. There’s More to Information Systems Development than Structured Approaches: Information Requirements Analysis as a Socially Mediated Process. *Requirements Engineering* 5, 2 (2000), 74–82.

[23] Latifa Guerrouj, Shams Azad, and Peter C. Rigby. 2015. The Influence of App Churn on App Success and StackOverflow Discussions. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015*. 321–330.
[24] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2012. App Store Mining and Analysis: MSR for App Stores. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, MSR 2012*. 108–111.
[25] Arthur E Hoerl and Robert W Kennard. 1970. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics* 12, 1 (1970), 55–67.
[26] Shaun K Kane, Amy K Karlson, Brian R Meyers, Paul Johns, Andy Jacobs, and Greg Smith. 2009. Exploring Cross-Device Web Use on PCs and Mobile Devices. In *Proceedings of the IFIP Conference on Human-Computer Interaction*. 722–735.
[27] Jan Terje Karlsen, Jeanette Andersen, Live S. Birkely, and Elise Ødegård. 2005. What Characterizes Successful IT Projects. *International Journal of Information Technology and Decision Making* 4, 4 (2005), 525–540.
[28] Amy K Karlson, Shamsi T Iqbal, Brian Meyers, Gonzalo Ramos, Kathy Lee, and John C Tang. 2010. Mobile Taskflow in Context: A Screenshot Study of Smartphone Usage. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2010*. 2009–2018.
[29] Maria Kechagia and Diomidis Spinellis. 2014. Undocumented and Unchecked: Exceptions That Spell Trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*. 312–315.
[30] Huoran Li, Wei Ai, Xuanzhe Liu, Jian Tang, Gang Huang, Feng Feng, and Qiaozhu Mei. 2016. Voting with Their Feet: Inferring User Preferences from App Management Activities. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016*. 1351–1362.
[31] Huoran Li, Xuan Lu, Xuanzhe Liu, Tao Xie, Kaigui Bian, Flex Xiaozhu Lin, Qiaozhu Mei, and Feng Feng. 2015. Characterizing Smartphone Usage Patterns from Millions of Android Users. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement, IMC 2015*. 459–472.
[32] Soo Ling Lim, Peter J. Bentley, Natalie Kanakam, Fuyuki Ishikawa, and Shinichi Honiden. 2015. Investigating Country Differences in Mobile App User Behavior and Challenges for Software Engineering. *IEEE Transactions on Software Engineering* 41, 1 (2015), 40–64.
[33] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*. 242–251.
[34] Xuanzhe Liu, Xuan Lu, Huoran Li, Tao Xie, Qiaozhu Mei, Hong Mei, and Feng Feng. 2017. Understanding Diverse Usage Patterns from Large-Scale Appstore-Service Profiles. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1.
[35] Xuan Lu, Xuanzhe Liu, Huoran Li, Tao Xie, Qiaozhu Mei, Gang Huang, and Feng Feng. 2016. PRADA: Prioritizing Android Devices for Apps by Mining Large-Scale Usage Data. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*. 3–13.
[36] Teresa Lynch and Shirley Gregor. 2004. User Participation in Decision Support Systems Development: Influencing System Outcomes. *European Journal of Information Systems* 13, 4 (2004), 286–301.
[37] Kalle Lyytinen and Rudy Hirschheim. 1987. Information Systems Failures: A Survey and Classification of the Empirical Literature. *Oxford surveys in information technology* 4, 1 (1987), 257–309.
[38] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering Companion, ICSE 2016 Companion*. 653–656.
[39] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Vol. 1.
[40] Robert Martin. 1994. OO Design Quality Metrics. *An analysis of dependencies* 12 (1994), 151–170.
[41] Thomas J McCabe. 1976. A Complexity Measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.
[42] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 29th IEEE International Conference on Software Maintenance, ICSM 2013*. 70–79.
[43] Israel Jesus Mojica Ruiz. 2013. Large-scale Empirical Studies of Mobile Apps. (2013).
[44] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. 2016. Release Practices for Mobile Apps – What do Users and Developers Think?. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015*. 552–562.
[45] Everett M Rogers. 2010. *Diffusion of Innovations*. Simon and Schuster.
[46] Jeff Sharkey. 2009. Coding for Life–Battery life, That is. In *Google IO Developer Conference*.
[47] Eric Shaw. 2014. *A Survey of Android App Quality Using Third Party Markets*. Ph.D. Dissertation. Auburn University.
[48] Peter Spirtes, Clark N Glymour, and Richard Scheines. 2000. *Causation, Prediction, and Search*. MIT press.
[49] Ramanath Subramanyam and Mayuram S. Krishnan. 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering* 29, 4 (2003), 297–310.
[50] Seyyed Ehsan Salamati Taba, Iman Keivanloo, Ying Zou, Joanna W. Ng, and Tinny Ng. 2014. An Exploratory Study on the Relation between User Interface Complexity and the Perceived Quality. In *Proceedings of the 14th International Conference on Web Engineering, ICWE 2014*. 370–379.
[51] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E Hassan. 2015. What are the Characteristics of High-Rated Apps? A Case Study on Free Android Applications. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution, ICSME 2015*. 301–310.
[52] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* 58, 1 (1996), 267–288.
[53] Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. 2016. Release Planning of Mobile Apps Based on User Reviews. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*. 14–24.
[54] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. 226–237.
[55] Shengqian Yang, Dacong Yan, and Atanas Rountev. 2013. Testing for Poor Responsiveness in Android Applications. In *Proceedings of the 1st International Workshop on the Engineering of Mobile-Enabled Systems, MOBS 2013*. 1–6.
[56] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets.. In *Proceedings of Annual Network & Distributed System Security Symposium, NDSS 2012*, Vol. 25. 50–52.

[57] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *International Workshop on Predictor MODELS in Engineering, PROMISE 2007*. 9.

Received February 2017; revised May 2017; accepted July 2017