Self-Reported Activities of Android Developers

Luca Pascarella¹, Franz-Xaver Geiger², Fabio Palomba³
Dario Di Nucci⁴, Ivano Malavolta², Alberto Bacchelli³

¹Delft University of Technology, The Netherlands — ²Vrije Universiteit Amsterdam, The Netherlands
³University of Zurich, Switzerland — ⁴Vrije Universiteit Brussel, Belgium
lpascarella@tudelft.nl, f.geiger@student.vu.nl, palomba@ifi.uzh.ch
ddinucci@vub.ac.be, i.malavolta@vu.nl, bacchelli@ifi.uzh.ch

ABSTRACT
To gain a deeper empirical understanding of how developers work on Android apps, we investigate self-reported activities of Android developers and to what extent these activities can be classified with machine learning techniques. To this aim, we firstly create a taxonomy of self-reported activities coming from the manual analysis of 5,000 commit messages from 8,280 Android apps. Then, we study the frequency of each category of self-reported activities identified in the taxonomy, and investigate the feasibility of an automated classification approach. Our findings can inform be used by both practitioners and researchers to take informed decisions or support other software engineering activities.

CCS CONCEPTS
• Software and its engineering → Maintaining software;

KEYWORDS
Android, Empirical Study, Mining Software Repositories

1 INTRODUCTION
Developing Android apps is fundamentally different from developing other types of software [12, 22, 51, 61]: On the one hand, even the smallest error may have quick and large effects (such as negative user reviews, with subsequent loss of future users [41]); on the other hand, Android apps have to deal with potential interaction with other apps, heavy usage of sensors like accelerometer and GPS, limited battery life, limited display size, and so forth. This inherent difference in the development of Android apps limits the possibility to use results, off-the-shelf, from software engineering research done on other kind of software systems. Instead, to use and guide our research to support the engineering of Android apps, first we need to gain a novel, deeper empirical understanding of how developers work on these apps.

Our goal, in line with this need, is to investigate and understand the various types of activities performed by Android developers in the context of real projects. We focus on self-reported activities, which represent one of the most valid ways to comprehend and analyze the development process [31]. As done in previous studies, we tap into the commit messages left by Android developers in GitHub repositories, as a way to study these self-reported activities. Past research has focused on specific aspects of Android apps such as performance [9] and energy consumption [4, 38], here we continue on this line, but broaden the scope to any type of activity, as done for open-source software [48].

Our research method follows that of an exploratory investigation, i.e., we started without hypotheses about the contents of the GitHub commit messages and made the types of development activities emerge from the extracted data [62]. To this purpose, we firstly built a dataset of 8,280 Android apps (which are both open-source in GitHub and distributed through the Google Play store) and randomly selected 5,000 commits from their repositories. Then, we (i) manually inspected and categorized all the commits by conducting independent content analysis sessions involving 5 researchers, (ii) collaboratively merged the independently-identified categories into a single taxonomy, (iii) validated the obtained taxonomy with external mobile app developers, (iv) analyzed the frequency of each category in the taxonomy across the 5,000 commits, and (v) investigated how effectively these commits can be automatically classified via standard machine learning techniques.

Our results show that Android developers reportedly perform a wide variety of different activities at different levels of abstraction, ranging from bug fixes, release management, access to sensors, etc. The most prominent category of activities is app enhancement (new and updated features), followed by bug fixing (mostly in an app-specific manner) and project management (mostly by merging/branching of the repository and by preparing a new app release). Those results confirm the importance of research related to feature management and release planning of Android apps, Android-specific program analyses, and software repository mining. Finally, the automated classification reaches promising initial results.

The main contributions of this study are the following:
(1) A taxonomy of self-reported activities performed by Android developers when developing their apps;
(2) An empirical analysis of the frequency of the self-reported activities performed by Android developers aimed at understanding their main concerns when developing their apps;
(3) An automated approach for classifying commit messages according to the defined taxonomy.
(4) A comprehensive replication package containing the raw data, analysis scripts, and the automatic classifier produced in our research.¹

¹https://figshare.com/articles/Self-Reported_Activities_of_Android_Developers/5802909
2 METHODOLOGY

The goal of the study is to empirically investigate and classify the activities performed by Android developers reported within commit messages, with the purpose of understanding the typical actions they perform and easing a variety of decision making mechanisms (e.g., code review triaging or monitoring of the development process). The perspective is of both researchers and practitioners, interested in an empirical understanding of the activities performed during the development process.

The choice of considering self-reported activities is driven by the recent advances in program comprehension [31], which demonstrated that the analysis of commit messages represents one of the most valid strategies to comprehend and analyze the development process of a software system. Our study revolves around three research questions and follows well-established guidelines on empirical software engineering [56, 62].

In the first place, we aim at categorizing the developers’ self-reported activities through the analysis of commit messages that accompany the changes performed while developing Android apps:

RQ1. How can self-reported activities of Android developers be categorized?

After having categorized the self-reported activities, we analyze the frequency of each category to quantify the different developers’ concerns when developing Android apps:

RQ2. How often does each category of self-reported activities occur?

Finally, we investigate how effectively self-reported activities can be automatically classified from commits via standard machine learning techniques, so that developers and project managers can be automatically supported during their decision making processes:

RQ3. How effective is an automated approach, based on machine learning, in classifying self-reported activities?

In the following subsections, we detail the design choices that allow us to answer our research questions.

2.1 Context Selection and Dataset Creation

We study self-reported activities based on commit messages authored by developers, thus we need real-world Android applications for which commit history is available. To ensure the analysis of a proper set of mobile apps having different size and scope as well as being published on the Google Play store, we design and conduct the selection process shown in Figure 1.

In step 1 we identify the GitHub repositories containing the source code of Android applications. Then, to properly link a GitHub repository to its corresponding app in Google Play, we exploit the Android manifest file (step 2). In fact, every Android app must have an AndroidManifest.xml file that includes a package name that identifies the application and serves as an identifier of the app on Google Play. The data concerning all open-source repositories on GitHub are available in databases on Google BigQuery.2 BigQuery list 378,610 AndroidManifest.xml files on GitHub with 112,153 unique package names. Duplication of package names may occur because of forked projects, the frequent usage of example names, or inclusion of manifest files from third-party code [23].

In step 3 we remove unpublished and non-existent apps by checking the existence of the corresponding page on Google Play. As a result, 9,478 package names are listed as apps in Google Play. For some of these apps, one or more repositories contain a matching AndroidManifest.xml file, as detailed above. In step 4 we match the repositories to Google Play entries with an heuristic approach:

(1) if only one repository contains a manifest for a package name, we assume it hosts the code for the app;

(2) if more than one repository with the manifest file exists, we search for links from Google Play meta-data of the app to any of the GitHub repositories. If we find a distinct repository the app entry linked to, we assume the repository to be the canonical source for the app;

(3) if no such unique link exists, we select the most popular repository based on number of (i) forks, (ii) watchers, and (iii) subscribers, as listed by GitHub.

During step 4, we remove the apps (718) for which we cannot determine a canonical repository, thus reducing the total number of apps we investigate (8,432). In step 5, we exclude repositories with fewer than 2 commits (152) to exclude unmaintained, toy, or demo projects [23]. Our final dataset consists of a total 8,280 mobile apps covering all 34 categories of the Google Play store, for a total of 1,727,930 commits belonging to the main branch of these apps’ GitHub repositories (step 6). The implementation of the dataset and further details about its creation process are available in [15].

In step 7, we randomly select a sample of 5,000 commits that cover 30 categories of Google Play, since the manual analysis of all the collected commits is infeasible. By considering 5,000 commits, we achieve a 99% statistically significant sample of the total number of commits of our dataset with a 1.82% confidence interval (assuming a 50% population proportion).

2.2 RQ1. Self-reported activities categorization

To answer our first research question, we conduct three iterative content analysis sessions [25] involving five software engineering
researchers, all authors of this paper, (2 graduate students, 2 research associates, and 1 faculty member) with at least five years of programming experience. From now on, we refer to them as inspectors. We describe the methodology for these three iterative sessions, followed by the validation method.

**Taxonomy Building.** Starting from the set of 5,000 commits composing our dataset, overall each inspector independently analyzes 1,000 commits.

**Iteration 1:** The inspectors analyze an initial set of 300 commit messages. Then, they open a discussion on the labels assigned so far and try to reach a consensus on the names and types of the categories assigned. The output of this step is a draft taxonomy that contains some obvious categories (e.g., changes to the Graphical User Interface), while others remain undecided.

**Iteration 2:** The inspectors first re-categorize the 300 initial commits according to the decisions taken during the discussion, then they use the draft taxonomy as basis for categorizing another set of 500. This phase is for both assessing the validity of the codes coming from the first step—by confirming some of them and redefining others—and for discovering new codes. After the completion, the inspectors open a new discussion aimed at refining the draft taxonomy, merging overlapping categories or characterizing better the existing codes. A second version of the taxonomy is produced.

**Iteration 3:** The inspectors re-categorize the 800 commits previously analyzed. Afterwards, they complete the final draft of the taxonomy verifying that each kind of commit message encountered in the final 200 commits is covered by the taxonomy.

Following this iterative process, we defined a hierarchical taxonomy composed of two layers. The top layer consists of 9 categories, while the inner layer contains of 49 subcategories.

**Taxonomy Validation.** In addition to the iterative content analysis process, we also externally validate the defined taxonomy. To this aim, we involved 2 professional developers having 4 and 5 years of Android programming experience, respectively. They were contacted via e-mail by one of the authors of this paper, who selected them from her personal contacts.

We provided them with a spreadsheet containing a list of 200 commit messages randomly selected from the total 5,000 in the dataset and asked to categorize the commits according to the taxonomy we previously built. During this step, the developers were allowed to either consult the taxonomy (provided in PDF format and containing a description of the commit categories in our taxonomy similar to the one we discuss in Section 3.1) or assign new codes if needed.

Once the task was completed, the developers sent back the spreadsheet file annotated with their categorization. Moreover, we gathered comments on the taxonomy and the classification task. As a result, both the participants found the taxonomy clear and complete. As a proof of that, the tags they assigned were exactly the same as the ones assigned during the phase of taxonomy building.

### 2.3 RQ2. Frequency of self-reported activities

In this research question, we aim at analyzing how frequently each category of our taxonomy appears. To this aim, we compute the frequency each category of activities was assigned to a commit message during the iterative content analysis.

In this way, we can overview the main developers’ concerns when developing mobile apps and identify the most popular self-reported activities. In Section 3 we present and discuss bar plots showing the frequency of each category in the taxonomy.

### 2.4 RQ3. Automated classification of activities

With our final research question we test standard machine learning techniques to automatically classify self-reported activities. As a side effect, the output of this research question poses a baseline against which future approaches aimed at more accurately classifying commit messages can be tested.

While several techniques can classify text of self-reported activities (e.g., keyword-based approaches [59]), we use machine learning since this type of approach can automatically learn the features discriminating a certain category, thus simulating the behavior of a human expert [45]. Overall, machine learning is a method (supervised, in our case) where a set of independent variables (the predictors) are used to predict the value of a dependent variable (in our case, the commit classification) using a machine learning classifier (e.g., Logistic Regression [40]). The following subsections detail the design decisions taken to build and validate our approach.

**Independent Variables.** Our goal is to classify the nature of self-reported activities based on commit messages: the basic information for the classification is therefore given by the words characterizing the commit message. However, not all the words in a commit can be actually representative for the classification of the self-reported activity. For this reason, we need to properly preprocess them [7].

In the context of our work, we use the widespread Term Frequency - Inverse Document Frequency (TF-IDF) model [52], which is a weighting mechanism that determines the relative frequency of words in a specific document (i.e., a commit message) compared to the inverse proportion of that word over the entire document corpus (i.e., the whole set of commit messages in our dataset). This approach measures how characterizing a given word is in a commit message: For instance, articles and prepositions tend to have a lower TF-IDF since they generally appear in more documents than words used to describe specific actions [52]. More formally, let C be the collection of all the commit messages in our dataset, let w be a word, and let c ∈ C be a single commit message, the TF-IDF algorithm computes the relevance of w in c as:

\[
    \text{relevance}(w, c) = f_{w,c} \cdot \log(|C|/f_{w,C})
\]

where \(f_{w,c}\) equals the number of times w appears in c, \(|C|\) is the size of the corpus, and \(f_{w,C}\) is equal to the number of documents in which w appears. The weighted words given as output from TF-IDF represent the independent variables for the machine learner.

**Dependent Variables.** The category of a self-reported activity is the variable to predict. We set the granularity of the dependent variable to the top layer of the taxonomy, i.e., the one reporting the 9 main categories of self-reported activities in our taxonomy.

**Machine Learners.** In our context, a certain self-reported activity might refer to more than one category: For instance, suppose that in a commit a developer performs both an enhancement and a
bug fix. This is a target for multi-label classifiers [60]. These can be of two types: (i) **problem transformation methods**, which transform the multi-label classification into a more single-label classifications, and (ii) **algorithm adaptation methods**, which extend specific classifiers in order to handle multi-label data [60].

Since the two types of multi-label algorithms have similar performance [49], we adopt a problem transformation strategy and use the OneVsRest classifier [21]. Taking as input a standard single-label algorithm, OneVsRest wraps up the process of training a classifier for each possible class. As a result, the input classifier assigns a probability that a certain commit message belongs to each of the categories of our top layer taxonomy: If the probability is higher than 0.5, then the commit message is considered as belonging to it.

With the aim of providing a wider overview of the performance achievable by different single-label classifiers when adopted in combination with OneVsRest, we consider (i) **Naive Bayes**, (ii) **Support Vector Machines** (SVM), (iii) **Logistic Regression**, and (iv) **Random Forest**. These classifiers make different assumptions on the underlying data, as well as have different advantages and drawbacks in terms of execution speed and overfitting [40]. We rely on the SciKit-Learn and NLTK Python libraries to implement our model. Before running the models, we identify their best configuration using the Grid Search algorithm [5].

**Evaluation Strategy and Metrics.** To assess the performance of the proposed machine learning approach, we adopt the **10-Fold Cross Validation** [57]. This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling (*i.e.*, each fold has the same proportion of self-reported activity categories). A single fold is used as test set, while the remaining ones are used as training set. The process is repeated 10 times, using each time a different fold as test set. Then, the model performance is reported using the mean achieved over the ten runs.

The performance of the experimented models are reported using widespread classification metrics such as **precision**, **recall**, and **F-Measure** (the harmonic mean between precision and recall) [3].

### 2.5 Threats to Validity

**Taxonomy validity.** To ensure that the correctness and completeness of the categories of self-reported activities identified, we iteratively built the taxonomy by merging and splitting categories if needed. As an additional validation, we asked 2 professional developers to classify a set of 200 commits according to the proposed taxonomy. They assigned to the sampled commits the same categories as the ones assigned during the phase of taxonomy building, also reporting the completeness and clarity of the categories we identified. We cannot exclude the missing analysis of specific commit types out of the categories identified, however the validation session gives us confidence of the reliability of the taxonomy.

In this study we consider GitHub commit messages as indicators of the actual activities performed by developers. This assumption may not hold for all projects, as many active projects do not use GitHub exclusively [23]. We partly mitigated this potential threat to validity by carefully selecting projects related to real apps published in the Google Play store (*e.g.*, no toy examples) and with at least two commits. The inclusion of additional data sources other than GitHub repositories (*e.g.*, project activities and issues on JIRA, developers’ discussions on message boards, etc.) is left for future work.

**Automated approach validity.** To build a multi-label classification technique, we exploited the OneVsRest method [21], which has been shown to have similar performance than other approaches [49]. To provide an overview of the performance achieved when using OneVsRest in combination with different single-label classifiers, we tested four categories of machine learners.

**External validity.** As for the generalizability of the results, we conducted this study on a statistically significant sample of 5,000 commits belonging to 8,280 open-source mobile apps that are published on the Google Play store. The proposed taxonomy may differ when considering closed-source apps; at the same time, the performance of the experimented automatic approach might be lower/higher than the one reported herein.

### 3 RESULTS

We report the results of our study by research question.

#### 3.1 RQ1. Categories of self-reported activities

The manual analysis of the 5,000 commits led to the creation of the taxonomy of Android developers activities shown in Figure 2. The taxonomy is composed of two layers: The top layer (9 items) groups together activities with similar overall purpose (*e.g.*, app enhancement, bug fixing), whereas the subcategories (49 items) in the lower level provide a finer-grained categorization. In the following we describe each category with the corresponding subcategories.

**A. App enhancement.** This category represents the activities aimed at adding or improving existing features of the mobile app. This is clearly at the core of mobile apps development and, as we will see in Section 3.2, its related commits involve a large number of changed source code files.

**Example commit.** “[Wear] Implemented Favourites feature for wearable companion app.” This commit is an example of a bug fix. This is a target for multi-label classifiers [60]. These can be of two types: (i) **problem transformation methods**, which transform the multi-label classification into a more single-label classifications, and (ii) **algorithm adaptation methods**, which extend specific classifiers in order to handle multi-label data [60].

Since the two types of multi-label algorithms have similar performance [49], we adopt a problem transformation strategy and use the OneVsRest classifier [21]. Taking as input a standard single-label algorithm, OneVsRest wraps up the process of training a classifier for each possible class. As a result, the input classifier assigns a probability that a certain commit message belongs to each of the categories of our top layer taxonomy: If the probability is higher than 0.5, then the commit message is considered as belonging to it.

With the aim of providing a wider overview of the performance achievable by different single-label classifiers when adopted in combination with OneVsRest, we consider (i) **Naive Bayes**, (ii) **Support Vector Machines** (SVM), (iii) **Logistic Regression**, and (iv) **Random Forest**. These classifiers make different assumptions on the underlying data, as well as have different advantages and drawbacks in terms of execution speed and overfitting [40]. We rely on the SciKit-Learn and NLTK Python libraries to implement our model. Before running the models, we identify their best configuration using the Grid Search algorithm [5].

**Evaluation Strategy and Metrics.** To assess the performance of the proposed machine learning approach, we adopt the **10-Fold Cross Validation** [57]. This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling (*i.e.*, each fold has the same proportion of self-reported activity categories). A single fold is used as test set, while the remaining ones are used as training set. The process is repeated 10 times, using each time a different fold as test set. Then, the model performance is reported using the mean achieved over the ten runs.

The performance of the experimented models are reported using widespread classification metrics such as **precision**, **recall**, and **F-Measure** (the harmonic mean between precision and recall) [3].

**2.5 Threats to Validity**

**Taxonomy validity.** To ensure that the correctness and completeness of the categories of self-reported activities identified, we iteratively built the taxonomy by merging and splitting categories if needed. As an additional validation, we asked 2 professional developers to classify a set of 200 commits according to the proposed taxonomy. They assigned to the sampled commits the same categories as the ones assigned during the phase of taxonomy building, also reporting the completeness and clarity of the categories we identified. We cannot exclude the missing analysis of specific commit types out of the categories identified, however the validation session gives us confidence of the reliability of the taxonomy.

In this study we consider GitHub commit messages as indicators of the actual activities performed by developers. This assumption may not hold for all projects, as many active projects do not use GitHub exclusively [23]. We partly mitigated this potential threat to validity by carefully selecting projects related to real apps published in the Google Play store (*e.g.*, no toy examples) and with at least two commits. The inclusion of additional data sources other than GitHub repositories (*e.g.*, project activities and issues on JIRA, etc.) is left for future work.

**Automated approach validity.** To build a multi-label classification technique, we exploited the OneVsRest method [21], which has been shown to have similar performance than other approaches [49]. To provide an overview of the performance achieved when using OneVsRest in combination with different single-label classifiers, we tested four categories of machine learners.

**External validity.** As for the generalizability of the results, we conducted this study on a statistically significant sample of 5,000 commits belonging to 8,280 open-source mobile apps that are published on the Google Play store. The proposed taxonomy may differ when considering closed-source apps; at the same time, the performance of the experimented automatic approach might be lower/higher than the one reported herein.

### 3 RESULTS

We report the results of our study by research question.

#### 3.1 RQ1. Categories of self-reported activities

The manual analysis of the 5,000 commits led to the creation of the taxonomy of Android developers activities shown in Figure 2. The taxonomy is composed of two layers: The top layer (9 items) groups together activities with similar overall purpose (*e.g.*, app enhancement, bug fixing), whereas the subcategories (49 items) in the lower level provide a finer-grained categorization. In the following we describe each category with the corresponding subcategories.

**A. App enhancement.** This category represents the activities aimed at adding or improving existing features of the mobile app. This is clearly at the core of mobile apps development and, as we will see in Section 3.2, its related commits involve a large number of changed source code files.

**Example commit.** “[Wear] Implemented Favourites feature for wearable companion app.” - thecosmicfrog/LuasataGlance (commit: 57c92a8784d5b5ac003af82b91aaee2135f41c3c4)
A.5 - Android lifecycle: Activities about the management of Android components lifecycle events and transitions. In the commit messages developers refer to technical aspects related to the Android programming model, such as the onCreate method.

A.6 - Profitability: Developers add/improve profitability aspects of the app. In the commit messages developers refer to activities such as adding ways to receive donations and displaying ads.

A.7 - Utility: Developers mention utility classes or methods, potentially used across the whole app, such as those for serializing/deserializing dates, strings manipulation, and app-specific exception handlers.

B. Bug fixing. This category represents development activities where app issues that appear in the mobile app are fixed.

Example commit. "v2.2 Testing new fix for massive battery usage caused by GPS not being disabled during application pause." - GrahamBlanshard/WiseRadar (commit: 7c35abb8512bb899b65750175e4ab07c26a813677)

B.1 - App specific: Bug fixing specific to the domain of the app. Commits belonging to this category do not relate to generic software qualities (e.g., performance), yet they have been marked as bug fixes.

B.2 - Performance: Activities aiming at improving the performance. Examples of commits in this category are Android wake locks, memory leaks and optimization of string operations.

B.3 - Security: Fixing security issues in the app. Example commits belonging to this category include sanitizing the input provided by users and removal of unused permissions.

B.4 - Crash: Fixing crash problems. Commits in this category are at different levels of abstraction, ranging from fixing null pointer exceptions to correcting issues for specific Android devices.

B.5 - Energy: Optimizing battery consumption optimization and managing potential energy leaks. Commits included in this category mostly include refactoring of the code in terms of better use of sensors (e.g., GPS) and WiFi as well as Bluetooth scanning.

C. Project management. In this category developers manage app releases, documentation, the build process, the GitHub repository itself (e.g., merges), IDE-related issues (e.g., Android Lint configuration).

Example commit. "Refactored the PlayerService by moving parts of the code into smaller classes" - bottiger/SoundWaves (commit: a1911b5229ce1d1c3b5ca11066c8c32e14c5cf68)

C.1 - GitHub-related: GitHub-specific aspects of the project. The commits in this category mostly mention the creation/merging of branches and the execution of the first commit in the repository.

C.2 - Release management: Activities to prepare a new app release. The commit messages deal, for example, with changing the app version number and preparing a new entry in the changelog.

C.3 - TODO: Activities on future actions to be done as potential enhancements or fixes. In this cases, commit messages deal both with low-level items (e.g., removal of a code smell) and higher-level concerns (e.g., implementation of a new feature).

C.4 - Documentation: Activities on the documentation of the app. Commit messages in this category mainly deal with adding/refining comments in the source code and the documentation of the app (e.g., description of the app functionalities, its requirements, UI mockups).

C.5 - Build: Activities on improving project compilation. Commit messages related to these activities usually relate to the creation of the app binaries (i.e., its APK file), rules for building the app, and migration to/from building systems.

C.6 - Manifest: Changes in the Android manifest of the app. Usually, commits belonging to this category concern updating the target SDK of the app, cleanup of default unused tags in the manifest, and adding/changing views definitions in the manifest.

C.7 - IDE: Activities related to the configuration of the IDE (e.g., Android Studio, Eclipse). Mainly commit messages include the definition of a new Eclipse project for the app and upgrade of the latest version of the IDE.

D. Code re-organization. These activities are aimed at improving the structure, size, and readability of the code (e.g., refactoring, cleaning, improvement of the code) or of the project organization, without changing behavior of the app.

Example commit. "Merge pull request #4 from RyDroid/readme, Update of README" - uberspot/AnagramSolver (commit: 322ca43654065ca00d1a8757059154cd1c5d1155)

Figure 2: Taxonomy of self-reported activities of Android developers
D.1 - Refactoring: Refactoring of the source code. Messages usually refer to moving code to specific methods of the lifecycle of Android activities, import statements reorganization and extracting methods from classes.

D.2 - Code cleanup: Source code cleaning activities (lighter weight than refactoring). Commits include the removal of unused API keys or unused string resources and deletion of dummy objects.

D.3 - Feature removal: Activities in which some features of the app are removed. Messages are mostly about what has been removed, not about why.

D.4 - Reduce app size: Activities aimed at reducing the app size to make it a more lightweight download. Commit messages belonging to this category mostly regard the removal of unused files or media resources.

D.5 - Dead code elimination: Elimination of source code never executed at run-time. Commit messages refer mostly to removing legacy Android activities, unused layouts and variable assignments, as well as redundant initialization code.

E. User experience improvement. This category represents the activities related to the user experience of the app, including screen layouts, elements colors and padding, text boxes appearance, buttons, messages shown to the user, as well as gestures support.

E.1 - GUI: Changes to the graphical user interface aimed at improving the user experience. Commit messages in this category are mostly about color schemes, buttons and UI layout.

E.2 - Strings: Activities related to the management of strings in the app. In the commit messages in this category usually developers discuss about static strings shown to the user.

E.3 - Images: Activities related to graphic elements such as icons, images, and graphics shown to the user. Commit messages are usually about changes of icon sets, addition or change of fixed images shown to the user, and logos.

E.4 - Gesture: Management of gestures of the users. Messages are mostly about features such as scroll to refresh, swipe for performing some action, and disabling scroll in some specific parts of the app.

E.5 - Orientation: Management of device orientation. Messages are mostly about the detection of orientation change (e.g., for playing a video), the creation of a dedicated layout for landscape orientation, and the margins when in landscape mode.

E.6 - Dialog: Activities related to dialogs, toasts, and pop-ups used to show notifications to the user. Messages usually concern adding/removing confirmation dialogs, adding toasts for giving feedback to the user, and fixing the style of pop-ups.

E.7 - Menu: Activities related to menus and navigation bars in the UI. Commit messages usually regard adding/removing menu items, reordering items in navigation bars as well as menus, and adding contextual menus where needed.

F. Storage management. This category of activities concerns changes involving archives, access to the file system, files storage, local settings, and persisting data via local database.

F.1 - Settings: Activities on locally-stored user settings and preferences. Commits related to this category are mostly about adding/removing specific items in user preferences and the integration with the Android system settings or with its Preference API.

F.2 - Local database: Activities related to data management via local databases (e.g., SQLite). Commit messages are usually about improving the queries to a SQLite database defined locally in the app and the addition of a local SQLite database in the app.

F.3 - File system: Activities related to the management of files in the local file system of the mobile device. Messages are usually about storing files in the SD card, cleaning up old temporary files, and checking if some locally stored configuration files exist.

G. Sensing & Communication Activities belonging to this category are related to (i) access to the device sensors (e.g., camera), recording and playing media streams (e.g., making a video), and (ii) communication features of the device (e.g., access to the WiFi/4G networks, making calls, messaging).

G.1 - Network: Activities related to the usage of the network. Commit messages usually include management of different levels of available bandwidth, switching to secure protocols, management of network errors, and management of TCP sockets leaks.

G.2 - Audio: Activities related to audio playback. Commit messages are usually about the management of the Android audio focus for playing sounds when in background, and the management of audio playback sessions.

G.3 - Image: Activities about the management of images in the app. Commit messages in this category are usually about backup, elaboration, and download of images.

G.4 - Sensor: Activities aimed at accessing device sensors. Commit messages regard mostly the interaction with the GPS sensor, the accelerometer, and the gyroscope.

G.5 - Camera: Activities related to the usage of the camera. Commit messages in this category are usually about taking a picture when using the app, when and how to show the preview of a taken picture, usage of the flash light, switching between front and rear camera.

G.6 - Messaging: Activities related to SMS/MMS messages. Commit messages in this category are usually about sending/receiving messages and developing fallbacks when SMS/MMS messages cannot be handled.

2http://developer.android.com/guide/topics/media-apps/audio-focus.html
G.7 - Call: Activities related to making and receiving phone calls. Messages usually regard making a call to a specific phone number, receiving calls, and silencing calls.

G.8 - Microphone: Activities related to the usage of the device microphone. Commit messages in this category are about recording audio and controlling the microphone settings.

H. API management: This category regards the activities related to the interaction of the app with external APIs. In this context, by external API we mean the software used by the app, but not owned/developed by the app developers themselves (e.g., the APIs of the Android platform or REST APIs).

Example commit. "Upgrade to broken-out Google Play Services v8.4.0" - zulip/zulip-android (commit: ae299267defec003e11cd10738b6e1f71849fd235)

H.1 - Library: Activities related to used Android libraries. Commit messages in this category are mostly about library substitution and usage of a new library.

H.2 - Android API: Activities related to the interaction of the app with the Android platform APIs. Messages refer to code changes for supporting new Android versions, retrofitting the code for supporting older Android versions, and fallbacks for fixing bugs manifesting only when the app is running on one specific version of the Android platform.

H.3 - REST API: Activities related to the interaction with REST APIs. Commit messages in this category regard changing URLs and ports of REST endpoints, adapting to new formats of the payloads of HTTP responses produced by REST endpoints, and management of authentication as well as sessions.

H.4 - Deprecation: Activities regarding reaction to deprecation, e.g., by moving to supported versions. Commit messages are about removing or changing calls to deprecated code.

I. Testing & Debugging: This category covers the activities related to logging information about the app at run-time, testing (e.g., test cases implementation, tests execution), and debugging.

Example commit. "test: ensure tests for retrieval of all persons in local database and repository passes" - chikecodes/Debt-Manager (commit: a4bc070540c2b2726f42a78d0afa86d13d6c333f)

I.1 - Testing: Activities related to testing. Commit messages in this category are about adding, fixing, or updating test cases, and ensuring that all tests are passing.

I.2 - Logging: Activities related to logging information at run-time and to reporting crashes. Commit messages in this category mention removing logging messages before publishing the app, adding logging statements for inspecting app behaviour at development time, logging errors in the IDE console, and integrating third-party logging as well as crash reporting libraries (e.g., Crashlytics, Timber).

I.3 - Debugging: Activities related to the debugging of the app. Commit messages refer to finding not-yet-localized bugs, manually checking test results, and raising the need for debugging a specific feature.

Discarded commits. During our manual analysis, we identified 115 commits with non-informative commit messages, which we discarded when building the taxonomy. There are three types of discarded commits: (i) 105 commits without any informative commit message (e.g., just one single character, three dots, one generic word), (ii) 9 commits with funny but non-informative commit message, and (iii) 1 commit reporting about an easter egg in the app. This low number of discarded commits (i.e., 2% over the total) gives more credibility to the completeness of the proposed taxonomy.

Result 1: Our taxonomy comprises 9 top layer and 49 subcategories reporting a large variety of developers’ self-reported activities.

3.2 RQ2. Frequency of self-reported activities

After having categorized and described the diversity of activities that Android app developers report to do while developing their apps, we now focus on determining how each of these activities is prevalent in our dataset.

Figure 3 shows the distribution of the commit messages across the categories of self-reported development activities. Each block in the figure reports the cumulative value for its corresponding top level category (e.g., 1,690 commits are in the category A - App enhancement) and the absolute value for its subcategories (e.g., of the 1,690 commits belonging to category A, 623 belong to the A.1 - New feature category and 581 to the A.2 - Feature changes category).

App enhancement is the most frequent among the high-level self-reported activities. This result can be explained by the highly dynamic ecosystem like the Google Play store, where developers are involved in very rapid release cycles [33], which are mainly driven by user ratings and reviews [18, 32, 43, 55]. In fact, the two most frequent subcategories are the development of new features (New feature) and their improvement (Feature changes). Other quite recurring types of app enhancement include the improvement of usability and internationalization of the apps. Specially the latter is a likely consequence of the global nature of the Google Play store, which imposes to take the language spoken by the app users in consideration.

Bug fixing is the second most frequent category of self-reported activities of Android developers. We conjecture that this high frequency is linked to how the app quality can have a dramatic impact on the success of an Android app [41], thus forcing developers to pay special attention to continuously correct bugs [26, 33]. Also, this frequency may be explained by Android bug reports being of high quality [6], thus easing the bug fixing process, mainly via long textual descriptions of the bug, the steps to reproduce the bug, and explanations of the difference between expected and the actual outputs. In the majority of the cases, fixed bugs are about aspects specific to the app domain (e.g., fixing the value shown in a specific card), whereas in other cases they are related to well-known key dimensions of the quality of a mobile app, such as performance [47], security [36], presence of crashes [16], and energy efficiency [12].
### Project management aspects of the app cover almost a fifth of all self-reported activities. In those cases, developers are mostly referring to GitHub-related activities (e.g., merging a branch) or about a new releases of the app (e.g., changing the app version number, changing app-store-related metadata). Developers also use GitHub for leaving todo items for keeping note of what should be done/ixed in future iterations. Interestingly, documentation seems to be not really a prominent activity of Android developers (only 56 activities reported in our study). This result may be due to the fact that developers do not use GitHub for storing and managing the documentation of their apps, maybe in favour of more flexible, easy-to-use, and designer-friendly document sharing platforms.

**Code re-organization** activities are reported in 560 cases (11% on the total) by developers, with a strong predominance of refactoring and code cleanup. Those activities seem to be regarded as important by developers, despite the noticeable lack of refactoring approaches working in the context of Android applications [35]. This may also be a result of the need for quick release cycles for Android apps, where it may be the case that maintainability-related activities like code refactoring and cleanup might overlap with more functional evolutions of the app [39]. Other less-recurrent activities are feature removal, app size reduction, and dead code elimination. All of them aim at making the app more lightweight both at run-time and during the initial download of the app binary (i.e., the APK file) from the Google Play store.

**User experience improvement** activities are almost as prevalent as code re-organization and this is aligned with previous research findings. In fact, past research has provided evidence that Android developers are aware of the importance of the user experience they are providing with their apps and are putting a huge emphasis on it [22]. In this area, according to our study, developers activities are mostly dedicated to the GUI of the app (e.g., layout, animations, views), followed by a proper formatting and phrasing of textual feedback shown to the user (i.e., strings), and the proper management of images (e.g., images size, asynchronous loading).

Other less recurrent activities are about users gestures, the device orientation, dialogs and toasts, and (navigation) menus. The difference of the frequencies of the above described activities may be due to the fact that the GUI, strings, and images are in the vast majority of Android apps and can strongly vary across apps and projects. Differently, (i) gestures, dialogs and menus are quite standard today, both from a design and Android APIs perspectives, and (ii) the explicit management of device orientation is not widespread.

Developers store and manage their data locally in the app, mainly for keeping app’s functionalities reliable and responsive even when the mobile device does not have a reliable connection [28]. Android settings and access to local databases (e.g., SQLite) are the most recurrent subcategories, followed by access to the file system. This result is quite surprising since the Android settings system is based on a single class, Preference, that provides a relatively basic API to developers. Intuitively, Android developers can store settings as key-value pairs, where (i) the value of each setting can be only a primitive type (e.g., boolean, integer, string) and (ii) the graphical representation of each setting is managed by the Android platform. It will be interesting to (empirically) assess how Android developers interact with the Android settings system and why such a relatively high number of settings-related activities are performed.

**Sensing & communication** activities are reported in 148 cases. Among them, developers mostly interact with the network (e.g., by making HTTP requests, managing cached results, or managing situations where the device does not have an Internet connection). Other less common activities are related to multimedia features of

#### Figure 3: Frequencies of self-reported activities.
mobile devices (e.g., audio recording, camera) and other sensors (e.g., GPS, accelerometer). Sending and receiving messages and making phone calls are in the lower part of our ranking of activities, suggesting that they are becoming less and less used by developers in favour of their Internet-based alternatives (e.g., VoIP, push notifications, etc.). Surprisingly, the usage of the microphone is reported in only 2 cases and this is in contrast with the current trend of voice-operated apps, such as Amazon Alexa, Apple Siri, and various Google products (e.g., Google Translate, Google Maps).

API management activities are predominated by access to third-party libraries and the interaction with official Android APIs. This results is a confirmation that using third-party libraries is a common practice for Android developers [24, 37]; moreover, it is reasonable to find a non-negligible number of commit messages referring to the interaction with the Android API, since Android apps are by their nature tightly integrated with the Android platform (e.g., for managing activities’ lifecycle events, accessing sensors, and showing views in the device display). The interaction with REST APIs is less prominent as it is the management of deprecated methods. The latter shows that app developers are little influenced by deprecation, similarly to developers of other systems [53, 54].

Testing and debugging are the least reported activities (only 124). Among them, testing is leading with 58 activities, followed by logging (43) and debugging (23). We suspect that those activities are so infrequent in our dataset, because developers may have embedded them into other self-reported activities (that is, when a developer implements a new feature, testing and debugging may also be present, but are not referred to). Future studies can investigate whether this is confirmed and it has an impact on developers’ perception of the importance of these tasks.

**Result 2**: Enhancement and bug fixing operations are the most popular self-reported activities, followed by project management and code re-organization ones. Interestingly, the least reported activity is related to testing and debugging.

### 3.3 RQ3. Automated classification of activities

Our third research question seeks to understand to what extent it is possible to use traditional machine learning approaches to automatically classify commit messages into our taxonomy.

Table 1 reports the results achieved by the four different multi-label classification approaches we experimented. The models relying on SVM and Logistical Regression provide the best balance between precision and recall (on average, the F-Measure is 68%). This is possibly due to the use of Grid Search as technique for setting the parameters of the classifiers: as shown by recent work [10, 58], a proper configuration of these algorithms strongly improve their performance. The other classifiers, i.e., Naive Bayes and Random Forest, have a lower ability to correctly classify self-reported activities. Their average F-Measure is 13 and 7 percentage points lower than SVM, respectively. Thus, in our scenario, the choice of the machine learning algorithm has an impact on the classification performance.

Considering the classification for the single categories, self-reported activities related to Bug Fixing are better classified by all the classifiers. A possible explanation is related to the characteristics words used by developers when reporting this type of activities. In fact, in the commit messages in our dataset, we often found the use of specific words like ‘fix’ and ‘bug’, or references to issue reports (e.g., ‘#19823’), which give a strong signal that the classifiers are able to capture.

Similarly, the Project Management and Enhancement categories are classified with a similar accuracy by SVM and Logistical Regression, possibly because of the specificity of the activities performed by developers during these tasks.

Other categories have a higher variability, thus showing that there is no set of words that can be easily used as features to discriminate them. For instance, considering the cases of Storage, API Management, SVM is 14 and 26 percentage points more effective than Logistical Regression, respectively. At the same time, in the classification of UI activities, Logistical Regression has an F-Measure 19 percentage points higher than SVM. This indicates that for some particular categories the underlying classification algorithm makes some difference and allows an improved categorization of self-reported activities. As part of our future research agenda, we aim at further investigating how the classifiers can be used as an ensemble to improve the results [44] (e.g., by means of a dynamic switching based on the characteristics of the commit messages [11]).

Finally, the investigated classifiers are not able to identify any of the commit messages related to Sensing & Communication. We further looked at the prediction results to investigate the reasons behind this result: we found that the misclassification is mainly due to the overlap between the terms used in Sensing & Communication and Enhancement. In other words, discriminating the two categories represents an arduous task for a machine learning algorithm since it cannot properly learn the words characterizing the two types of self-reported activities. This final result highlights a limitation of our approach. The machine learning algorithm is based on the implicit assumption that commit messages are representative of the action performed by developers, because as humans we have been able to classify them. However, a human analysis—as the one conducted in RQ1—may often be able to correctly characterize commits because of external factors that are often implicit (e.g., experience or information contextualization [17]); these external implicit factors are not available to the machine learning approach, hence it may fail in cases where the overlap between terms in two categories is high [2, 42, 43, 46].

**Result 3**: While for categories like Bug Fixing and Project Management the classification performance is up to 80%, machine learning approaches can classify developers’ self-reported activities with an average F-Measure of 68%. However, our analysis revealed some possible points of improvement (e.g., exploiting the complementarity among classifiers).

### 4 RELATED WORK

Classifying commits by intent with machine learners has been a popular research area [13, 27, 30, 34]. Robles et al. [50] conducted a
preliminary investigation in that direction confirming the importance to consider in software projects all kind of files tracked by versioning systems other than source files. These files may bring evidence of the self-reported activities documenting technical and human-related aspects. Later, Hattori et al. [19] conducted a study aimed at enhancing the knowledge of the nature of commits in terms of size and meaning. Similarly, Alali et al. [1] and Maaële et al. [29] proposed two studies aimed at characterizing commit messages by intents. Moreover, Hindle et al. [20] applied a machine learning technique to classify commits by considering only metadata and commit messages without inspecting the source code. Oppositely, Fu et al. [14] adopted Latent Dirichlet Allocation (LDA) to classify change messages by semi-supervised topic semantic analysis.

The analysis of self-reported activities represents one of the most valid ways to comprehend and analyze the development process of a software system [31]. Despite this, so far self-reported developers’ activities have been investigated only by targeting different type of systems, e.g., generic open-source software [48], or by focusing on specific aspect of Android apps, such as performance [9] and energy consumption [4, 38].

Ray et al. [48] analyzed a large dataset from GitHub to understand the effect of programming languages on software quality, finding that language design has modest effect on software quality.

Das et al. [9] investigate to what extent developers take care of performance issue analyzing commit messages. The analysis, conducted on 2,443 open source Android apps, showed that most of the commits that lead to performance issues are related to GUI, code smell fixing, network related code, and memory management. Moura et al. [38] conducted a study similar to ours on an initial sample of 2,189 commits from the GitHub repository to analyze energy-aware commits. Analyzing a final dataset of 371 commits from 317 real world apps, they found that software developers heavily rely on low-level energy management approaches, such as frequency scaling and multiple levels of idleness. Moreover, energy saving techniques can impact the application correctness. With the same aim, Bao et al. [4] extended Moura et al. by analyzing 468 commits from 154 Android apps. They discovered six power management activities for Android apps and discovered that power management activities vary with respect to the Android store category of the app. With respect to these works, we analyzed commits related not only on performance issues and energy management.

5 CONCLUSIONS AND IMPLICATIONS

Our work aimed at understanding and classifying self-reported activities of Android developers.

Our results showed that changes applied by developers are mostly related to enhancement or bug fixing operations: These categories are clearly the ones for which more automatic support would be needed. A very few commits are instead related to the management of APIs and testing, possibly highlighting the lack of specific tools supporting developers during these operations. Finally, a machine learning approach can correctly classify self-reported activities with an average F-Measure of 68%.

Our findings have a number of implications for both Android developers and researchers. Android developers can use our taxonomy of development activities for taking more informed decisions when assigning code reviews to team members. For example, commits related to the REST API category may be assigned to those members who are also involved in the development of the back-end of the mobile app (who potentially are more knowledgeable of the interaction between the app and its back-end). Also, categorized commits can be used (i) for getting a clear idea about which activities are being performed by developers during the whole project lifecycle, (ii) for identifying potential blocking activities where developers are spending the majority of their working time, or (iii) as decision support system when allocating resources to the project. Finally, developers can use our classifier for automatically categorizing code commits according to our taxonomy of activities.

We support researchers by increasing our empirical understanding of the types of (self-reported) activities performed by Android developers in real projects. This is a key step to guide future research in the area. Specifically, the most recurrent activities may be a good indicator for future research on Android apps development. Moreover, both the taxonomy and our automatic classifier have the potential to strengthen the reliability of other mining approaches that use commit messages as input (e.g., [4, 8, 9, 38, 41]). It is our hope that our results and the shared dataset will help and guide future research on support the engineering of Android apps.

ACKNOWLEDGMENTS

Bacchelli and Palomba gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529. Di Nucci is funded by the Belgian Innoviris TeamUp project INTiMALS.


