Automated App Categorization using API analysis

JI ZHOU

MSc Business Analytics

Department of Computer Science

University College London

Sep, 2016

Abstract

App store is a center location for users to discover, to download and to install apps with only a few clicks. Categorizing mobile apps according to their functionalities has many benefits to both users and developers. Placing apps into the proper categories can help users to quickly discover and explore the desired apps. It can also help app developers to analysis the potential features and technical trends within the same domain.

Unfortunately, existing categorizations are ineffective because apps are assigned to some categories arbitrarily by developers. To help developers choose the right categories, we propose an API-based app categorization approach in this project. This approach automatically clusters apps based on the API calls extracted from the APK files.

We empirically evaluated this approach against the state of the art description-based categorization approach using 10,020 real apps downloaded from the Google Play store. We use k-means to cluster 9,980 apps and unify parameters for attribute and model selections. We evaluated the quality of the results based on the Silhouette Coefficient, Adjusted Rand Index and Adjusted Mutual Information metrics.

The result shows that our approach achieved two times higher Silhouette scores and as well as improved the other quality metrics. As far as we know, this is the first paper to prove that APIs are a good attribute for app clustering by comparing it against app descriptions, which provides a much lower granularity way for automatically app categorization.

Acknowledgements

I would like to thank my academic supervisor Dr. Jia Yue for offering consistent guidance and support throughout my work. A special thanks goes to research student Afnan A. Al-Subaihin for technical support and research advices. It would be impossible to complete this thesis without the help from above two.

Content

List of Figures

List of Tables

List of Examples

1. Introduction

Smartphones are now closely tied up with our life, in 2015, smartphones are checked 8 million times in the US every day and on average, 46 times contributed by each person [20], which is an indicator of the extensive mobile usage. According to GSMA, total $3.1 trillion revenue was generated by the mobile industry in 2015, which accounts for 4.2% of the word's global GDP.

Due to the increasing demand for smartphones, mobile apps generated a great fortune for the developers, shown in the Figure 1, the global gross revenue reached to $41.1 billion in 2015. App economy will see continuing growth and mobile app market is projected to expand 24 percent by the end of 2016. The expected annual gross revenue will exceed $101.1 billion world widely in 2020.

With emerging market of mobile apps, the total number of apps and developers grew rapidly for the past five years as well. As shown in Figure 2, the Google Play Store, for example, doubled the number of apps in 2014 compared with 2013. Google Play ended the year 2014 with more than 100k developers compared with Apple's App Store.

App Store Categorization

An automated app categorizing approach according to their functionalities has many benefits to both users, developers as well as the app store.

- Placing apps into the proper categories can help users to quickly discover and explore the desired apps

- A well chosen category maximizes app's discoverability and hence more profit for developers. Developers could also study feature patterns among similar application for improvement and maintenance.

- Automatic app categorization enables efficient regulations within App stores, for example, flagging any abnormal behaviors or mismatch between assigned categories and true categories.

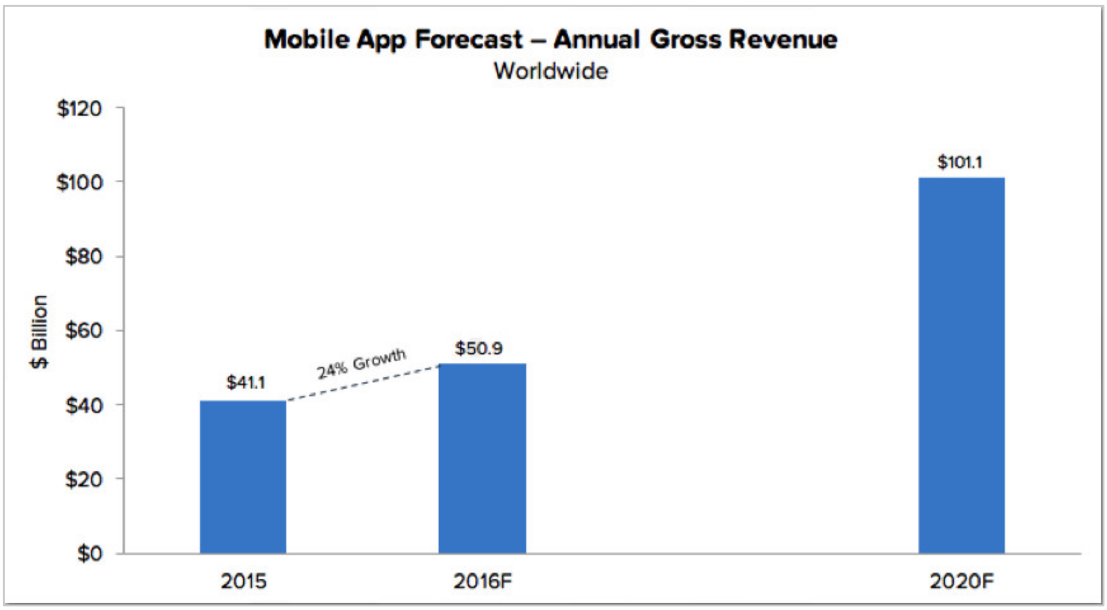


Figure 1: Mobile App Forecast – Annual Gross Revenue [1]

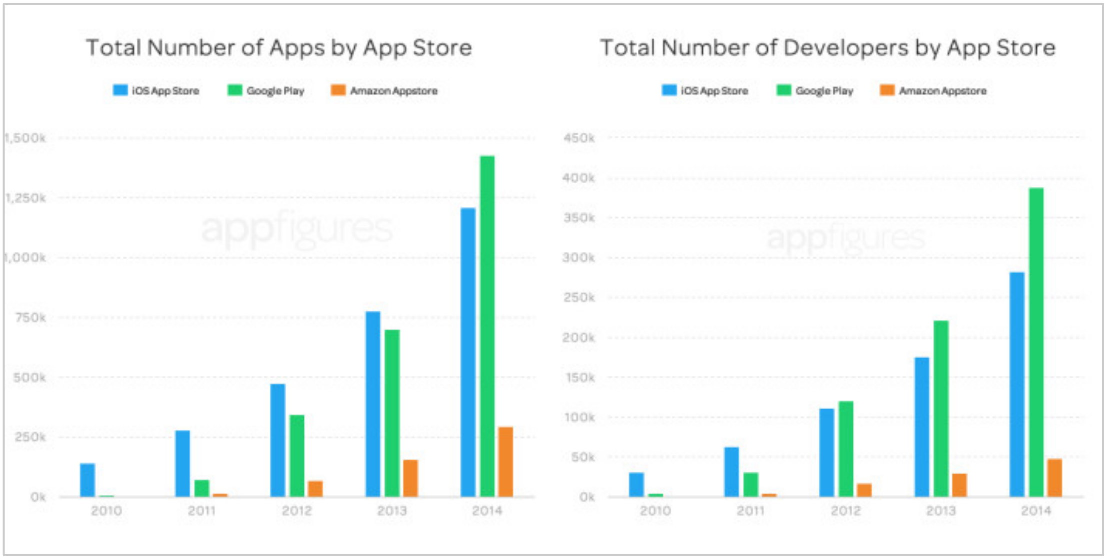


Figure 2: (LHS) Total Number of Apps by App Store, (RHS) Total Number of Developers by App Store (Source: Business of Apps) [2]

Unfortunately, existing categorizations are ineffective because apps are assigned to some categories arbitrarily by developers. Current app store categorization approach seeks to cluster apps based on the mining textual features extracted from app descriptions [3]. The text descriptions are one of the main resources for users to understand the features of one app. They can be easily accessed from app stores without the need to download the app. However, this approach may only provide a coarse granularity to some limitations of the app descriptions:

1. Claimed features in app descriptions are not always truly implemented in source code. App descriptions are usually manipulated for increasing the visibility of apps and attracting users.

2. App descriptions are natural languages and it is still hard to process them accurately. Sometimes descriptions could be written in other languages rather than English, which makes automatically categorizing apps even harder.

Researchers have also tried to analyze app categories using the terms (i.e. identifiers, comments) extracted from source code [14]. Identifiers and comments are much more reliable than descriptions while there are still some problems associated with them. First of all, source codes of apps are not always available for public access. Furthermore, suggesting by Tan et al., mismatch happens between source code and comments.

To overcome these limitations, we purpose an in-between solution, API-based approach to categorize applications. This approach automatically clusters apps based on the API calls extracted for the APK files. We believe API calls are good proxies representing app functionalities than claimed features extracted from the text descriptions and easier accessing without acquiring source code.

In this project, we investigated three types of API usages, term frequency, binary frequency and sensitive APIs. Sensitive APIs are a small subset of APIs, which are permissions authorized by users and listed in *AndroidManifest.xml* file. We empirical evaluated this approach against the state of the art description-based categorization approach using 10,020 real apps downloaded from the Google Play store and evaluated the quality of the results based on the Silhouette Coefficient, Adjusted Rand Index and Adjusted Mutual Information metrics.

Goal and Objective

Since clustering based on API usage can be a good alternative to overcome the limitations of both text descriptions-based and source code-based approaches. The general aim of the project is to develop an effective API-based app categorization approach. The detailed aims and objectives of this thesis are as follows:

1. To mine App descriptions and extract API calls from APK files
2. To apply machine learning techniques for cluster apps based on API usages and minded features
3. To carry out an empirical study on the proposed API approach and state of the art description-based approach

Contributions

The contributions of this thesis are:

1. We proposed a practical API-based app categorization approach using machine learning.

2. We implemented a framework automatically clusters Apps supporting both proposed API-based approach and traditional feature-based approach.

3. We created a benchmark, sampling 10,020 real Android apps from the PlayDrone data set, including metadata and compressed APK files.

4. We built several evaluation matrices (1) app-topic through topic modeling (2) app-token from textual descriptions (3) app-api extracted from APK files (4) app-sensitive_api for finally applying k-means clustering.

5. We carried out an empirical study and found that API usages, as a proxy for what is truly implemented by the apps, is much more powerful in clustering apps than text descriptions, having average Silhouette Coefficient 0.508 and 0.236 accordingly.

6. We reported a case study on YouTube app, showing an insight what properties making apps a cluster, in aspect of descriptions, topics and APIs. We also explained our experiment in details with YouTube as an example.

Outlines

The remainder of this paper is structured as follows: Section 2 surveys the related research work in the app store analytic area and describes the main categorization approaches in details. Section 3 introduces our API-based approach, applied machine learning algorithms and implementation details. Section 4 discusses the research question followed by the descriptions of the empirical study. Section 5 presents parameters settings including how we tuned the parameters. Section 6 presents the answer to research questions and results discussion. Section 7 concludes and provides possible future directions.

2. Background & Literature Review

With rapid growth in smart phone applications, more and more research now has focused on analyzing app features over app stores. This section reviews work in the field of app store analytics. It begins with an introduction of the general research directions in the filed, followed by discussion on App Store mining, API analysis and App categorization.

2.1. General Directions

App stores provide rich information mainly on app details, reviews and related apps. App details include app interface, descriptions, new release, technical support, developer information and etc.. Reviews are in two forms, stars rating and verbal review. Related apps will automatically recommend similar apps in three ways, apps from single developers, 'customer also bought' and top apps in single category.

Therefore, analysis on app features is beneficial in many ways, for examples:

McMillan et al presented a system for **rapid prototyping** by mining feature descriptions and source code from open source repositories as well as recommend them to the software under development. Prototyping is initial phase of software development but typically thrown away and therefore, prototypes need to be created quickly with little cost and effort. The system offers excellent solutions and facilitates software reuse. [4]

One more paper working on **recommendation system** is to support domain analysis, which is the process of identifying, organizing, analyzing, and modeling features common and variable to a particular domain. [5] Taken product descriptions as input, association rule mining is used to discover affinities among features across products and k-nearest neighbor approach is applied upon the product profile to identify new features.

There are many ultimate goals associated with **detecting similar apps**, including categorization, recommendation, maintenance and etc.. Holtzhauer et al proposed an approach for automatically detecting Closely reLated applications in ANdroid (CLANdroid) by relying on advanced Information Retrieval techniques and five semantic anchors: identifiers, Android APIs, intents, permissions, and sensors. [6]

From the aspect of requirements elicitation, Sarro et al believes that "requirements for the masses; requirements from the masses" and hence to study the **lifecycle of features**. Sarro et al introduced a simple set-theoretic characterization of the lifecycle and migratory behaviors of app features across product categories, such as migration, exodus, extinction, intransitive, birth and death. Further more, correlation analysis also highlights differences between trends relating price, rating, and popularity. [11]

2.2. Mining App Description

App store is a platform, which allows users to discover, purchase, download, and install software with only a few clicks. Meanwhile, it is also a mechanism for developers to advertise, sell, and distribute their applications. It is in fact that there are rich source of information in app stores on all aspects of business, customers and technology. The paper *App Store Analysis: Mining App Stores for Relationships between Customer, Business and Technical Characteristics* explores the relationships among customer-, business- and technically- focused attributes, in more details, including customer ratings, number of downloads, price of apps and technical details in app descriptions.

App description plays a big role in app marketing and in Google Play, it directly affects Apple Store Optimization (ASO). ASO is an app optimization process for search results in app stores. Therefore, a good app description must include relevant keywords and search terms, which makes your app ranking higher and hence more visible to potential customers.

One of the most indispensable factors in app descriptions is the app feature. “A feature is a claimed functionality offered by an app, captured by a set of collocated words in the app description and shared by a set of apps in the same category.” Mining app description for feature extraction is very popular in last few years and it becomes fundament of many further researches on app stores.

Since app descriptions are written in natural language, Harman et al. developed a simple four-step Natural Language Processing (NLP) algorithm to extract feature information and implemented it using the Natural Language Toolkit (NLTK), a comprehensive natural language processing package in python. [8]

The first step of the algorithm is to define ‘feature patterns’ from the raw app descriptions. Feature pattern is a list composed of some sentences and each sentence contains a feature. Feature list is then tokenized into a lower case token stream and then apply filtering. The result of the third step is a set of ‘featurelets’, Harman et al perform a collocation analysis to find words that associate frequently from the refined feature pattern, built on top of NLTK’s N-gramCollocationFinder package. Step four applies a greedy hierarchical clustering algorithm to aggregate similar featurelets together.

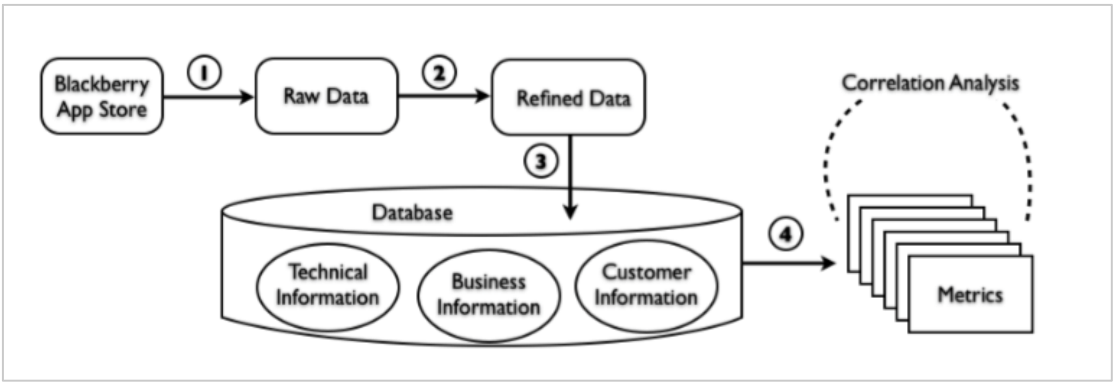


Figure 3: App Analysis Flows for paper 'App Store Analysis: Mining App Stores for Relationships between *Customer, Business and Technical Characteristics'*

Figure 3 shows a general workflow for the traditional app store analytic work.

Phase 1 (Data extraction): A customized web crawler is used to firstly collect category information from Blackberry app store and secondly download raw app data through the URLs of all apps within each category.

Phase 2 (Parsing): A set of attributes, including category, description, price and so on, is extracted by parsing the raw data based on some search rules.

Phase 3 (Data Mining Features): Features are mined from app descriptions using approach proposed by Harman (details in section 2.2).

Phase 4 (Analysis): This involves the analysis of the mined information.

In Harman et al.'s study, they found there is strong correlation between ratings given to apps and their popularity, that is highly rated apps are more frequently downloaded, for both free and non-free apps. Free apps are rated significantly higher than their chargeable counterparties. Despite a mild correlation between price and the number of features provided, limited evidence has emerged to establish any correlation between the price and either the rating or popularity for non-free apps. The number of features per app (and the number of shared features between apps) follow a power law. Despite the fact that a large number of apps are zero-rated, non-zero rated apps generally receive positive ratings. The power law is evidenced as more ratings are presented towards the higher and more favorable end of the rating spectrum.

Mining app description benefits both customers and developers in many ways, for examples, to investigate behavior/description mismatches, to detect malicious applications and so on. Pandita et al. present a framework, WHYPER, to tell if the need for sensitive permissions (mainly on address book, calendar and record audio) is motivated in the application description by using Natural Language Processing (NLP) techniques. WHYPER achieves an average precision and recall both over 80% for above three permissions. These results demonstrate great promise in using NLP techniques to bridge the semantic gap between user expectations and application functionality, further aiding the risk assessment of mobile applications. [7]

2.3. API/Description Mismatches

Programmers typically build software by calling Application Programming Interface (API) to perform basic functions, for example:

Without API:	With API:
An app finds the current weather in London by opening *http://www.weather.com/* and reading the webpage like a human does, interpreting the content.	An app finds the current weather in London by sending a message to the *weather.com* API (in a structured format like JSON). The *weather.com* API then replies with a structured response. [9]

API is released by software companies and usually publicly available. Therefore, as a proxy for real value of the app, digging into API usage helps investigate app features and app categorization.

Detecting malware has been the focus of recent research, conducted through the comparison of static code and dynamic behaviour with predefined patterns of malicious behaviour. However, as increasing emphasis has been placed on privacy recently, program behaviour is more difficult to be simply classified as beneficial or malicious beforehand and instead the nature will depend on the current context. An example to illustrate this debatable nature would be WhatsApp, one of the world's most popular mobile messaging applications. Despite the app takes all of users' contacts and transmits them to some server, it would be difficult to consider the app as malicious.

'Malware' is defined as acting against the interests of its users in the paper *Checking App Behaviour against App Descriptions.* The paper compares implemented app behaviour to advertised app behaviour, using the natural language description from the Google Play Store and the set of Android APIs in the app binary as proxies for the advertised behaviour and implemented behaviour respectively, The association of descriptions and API usage enables anomaly detection and drives observations such as "Unusual for the 'weather' category, this application accesses the messaging API." [13]

The approach presented in the paper named as CHABADA. CHABADA identifies main topics (at most four) for each application by mining its descriptions. API information is extracted and checked for outliers or abnormal behavior given respective topic cluster using One-Class Support Vector Machine learning (OC-SVM) technique.

Mismatch happens between source codes and comments as well [10]. It has been conventional to comment source code in software development. Not only do comments help enhance the readability of code, explain code segments and data structures, they also state assumptions and reveal the programmers' intention. Checking the former type of comments adds limited value due to its consistency with source code in the majority of cases. In contrast, the latter type typically provides more insights and usually contains many imperative words including "must", "should", "need", "ought to" etc.

Requirements, also referring as rules in the paper, is extracted from comments and used to automatically detect inconsistencies between comments and source code, indicating either (1) **bugs**, the source code does not follow the assumptions and requirements specified by correct program comments or (2) **bad comments**, comments that are inconsistent with correct code, which can confuse and mislead programmers to introduce bugs in subsequent versions.

The results of the experiment show that iComment produces 90.8%-100% accuracy in the automatic extraction of 1832 rules from comments in the latest versions of the four programs. In total it has detected 60 comment-code inconsistencies, 33 new bugs and 27 bad comments, of which nineteen, including 12 bugs and 7 bad comments, have been confirmed correspondingly by their developers.

2.4. Categorization

The traditional way of clustering mobile apps is based on **mining textual features**. Al-Subaihin et al proposed a way to do so explained in details as following [3]:

Feature Extracting: The result of mining raw app descriptions is a collection of featurelets and each featurelet is a set of terms $f = \{t_1, t_2, .., t_k\}$. The set of all unique terms in the corpus is constructed as $T = \{t_1, t_2, .., t_N\}$.

Feature Representation: Each featurelet is converting to a vector in which 1 for present of a term in the set T and 0 for absent. Standard information retrieval technique, Term Frequency-Inverse Document Frequency (TF-IDF) is applied to the value of vector, by giving less weight to common terms and more importance for powerful (ability of distinguish) terms.

The final vector space F is taking into account of similarity between each word in the featurelet and each term. Each element of F is Wordnet similarity score. Featuring clustering: After selecting optimal number of clusters K, skmeans is used to cluster the resulting Feature-Term Matrix.

App Representation: An app-feature matrix is constructed from resulting feature clusters, where the rows are vectors corresponding to apps. Each element in the AFM is a Boolean value to indicate whether the app exhibits a feature within the corresponding feature cluster. App Clustering: Final app clustering is conducted using agglomerative hierarchical clustering technique.

While in consideration of behavior and description mismatches, once the terms extracted from source code of the applications are used for categorization. For examples, words in identifiers and comments, which also called attributes of the application, serving as input of machine learning algorithm for automatic categorization.

An explicit assumption of above approach is that the source code for applications is always available, however, it is not true in many cases, as only executable forms of commercial application would be released for various legal and organizational reasons.

The paper *Categorizing Software Applications for Maintenance* mentioned that stakeholders would benefit from properly classifying software applications for several reasons and the most important one is, grouping applications into categories by features or functionalities allow stakeholders to decide what features would be included in the prototype and hence, more easily to predict common bugs.

The work attempt to categories traditional software using Application Programming Interface (API) calls as a set of attributes with following reasons:

- Unlike the terms ie. names of identifiers, which are selected arbitrarily in most cases by programmers, the set of APIs is predefined as API calls used to develop software are extracted from well-defined and widely used libraries. Therefore APIs used by apps are less likely to be arbitrarily distributed in comparison to terms

- APIs have already been grouped in packages and libraries according to their respective functionalities. The existing grouping can be leveraged alongside with machine-learning approaches to assist the application categorization process.

In order to test this approach, Java applications collected from two software repositories are supporting the whole project, including 745 closed-source applications from Sharejar and 3,286 open-source applications from SourceForge. API information is extracted in two forms referring to different levels of granularity, API packages and API classes. Along with project terms for each application, the top 100 attributes that best distinguish each category were used as input for three different machine learning algorithms.

API packages are fund to be more effective attributes than API classes, on average 20% better predictions for categorization of the applications. The paper also points out that Support Vector Machines (SVM) is the best-performing algorithm compared with the other two, Decision Tree and Naïve Bayes, in both repositories used as a dataset in the evaluation. API packages are a good alternative to terms (identifiers, comments extracted from source code) in the case when the terms are not available, moreover, the number of API packages is much smaller than the number of terms.

3. Approach

Figure 4 shows the overall framework of the data post process. Our framework takes the app downloaded from PlayDrone as input. PlayDrone is a scalable Google Play store crawler, developed by Columbia University Department of Computer Science.

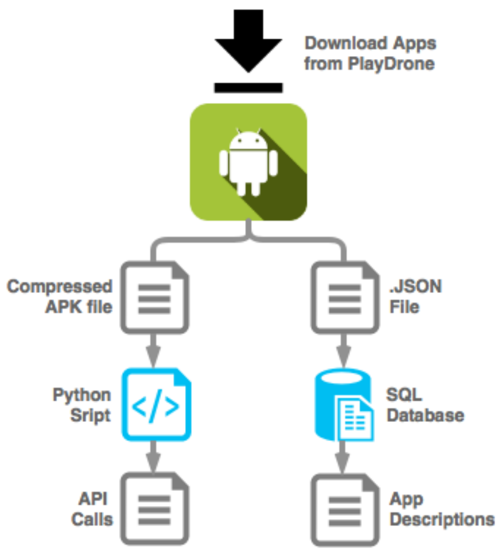


Figure 4: Flow chart for data post process

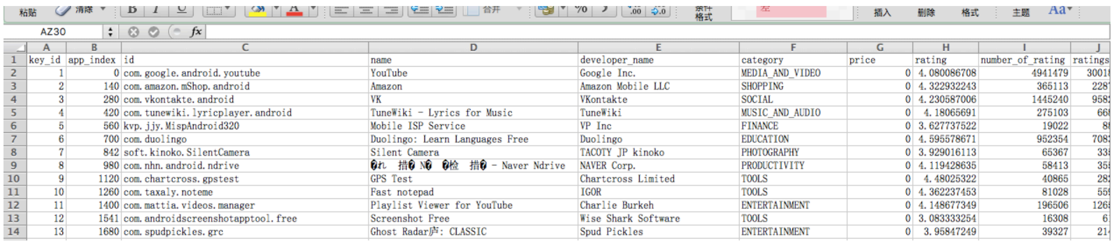
API extraction: we have written a python script to extract API usages directly from APK files. The implementation firstly decompressed APK file and then get *AndroidManifest.xml* file, search for smali file, extract APIs from them, clean duplicated APIs and store API information in text file under the name of APK files.

Smali file is an assembler for the dex format used by dalvik, Android's Java Virtual Machine implementation and hence smali files are obtained from decompiling dex file. Dex files are executables included in Android apps (APK file).

APIs inside of double quotation marks are separately stored as permissions: <uses-permission android:name="android.permission.**" />. Furthermore, the python script is revised without clean duplicated APIs, which is counting API frequencies (binary = False).

However, we had some problems extracting APIs with 40 apps, which might be caused by the APK files downloading from the PlayDrone. Therefore we exclude those 40 apps from our sample.

Extracting App Metadata: All the metadata of apps is stored in JSON format. JSON is short for JavaScript Object Notation, which contains essential information about an application in logic manner, such as application name, category, price, rating, number of downloads and so on. We implemented a utility to extract all the information we need and exporting into SQL database in a structured table, shown by Figure 5. The column names are listed in Appendix 1.



	key_id	app_index	id	name	developer_name	category	price	rating	number_of_rating	ratings
2	1	0	com.google.android.youtube	YouTube	Google Inc.	MEDIA_AND_VIDEO	0	4.080086708	4941479	3001
3	2	140	com.amazon.mShop.android	Amazon	Amazon Mobile LLC	SHOPPING	0	4.322932243	365113	228
4	3	280	com.vkontakte.android	VK	VKontakte	SOCIAL	0	4.230587006	1445240	958
5	4	420	com.tunewiki.lyricplayer.android	TuneWiki - Lyrics for Music	TuneWiki	MUSIC_AND_AUDIO	0	4.18065691	275103	66
6	5	560	kvp.jjy.MispAndroid320	Mobile ISP Service	VP Inc	FINANCE	0	3.627737522	19022	8
7	6	700	com.duolingo	Duolingo: Learn Languages Free	Duolingo	EDUCATION	0	4.595578671	952354	708
8	7	842	soft.kinoko.SilentCamera	Silent Camera	TACOTY JP kinoko	PHOTOGRAPHY	0	3.929016113	65367	33
9	8	980	com.nhn.android.ndrive	�れ 措� N� �検 措� - Naver Ndrive	NAVER Corp.	PRODUCTIVITY	0	4.119428635	58413	35
10	9	1120	com.chartcross.gpstest	GPS Test	Chartcross Limited	TOOLS	0	4.48025322	40865	28
11	10	1260	com.taxaly.noteme	Fast notepad	IGOR	TOOLS	0	4.362237453	81028	55
12	11	1400	com.mattia.videos.manager	Playlist Viewer for YouTube	Charlie Burkeh	ENTERTAINMENT	0	4.148677349	196506	126
13	12	1541	com.androidscreenshotapptool.free	Screenshot Free	Wise Shark Software	TOOLS	0	3.083333254	16308	6
14	13	1680	com.spudpickles.grc	Ghost Radar庐: CLASSIC	Spud Pickles	ENTERTAINMENT	0	3.95847249	39327	21

Figure 5: Screenshot for App Metadata

3.1. Machine Learning Techniques

Two distinct inputs are taken for clustering app descriptions: (1) App-topic matrix: LDA discovers topics that occur in the collection of app descriptions and hence we have every app belongs to several topics with some possibilities. (2) App-token matrix: this method directly takes every word in the collection of app descriptions and counting token occurrences along with optional dimensionality reduction.

Figure 6 shows various matrices built from structured data.

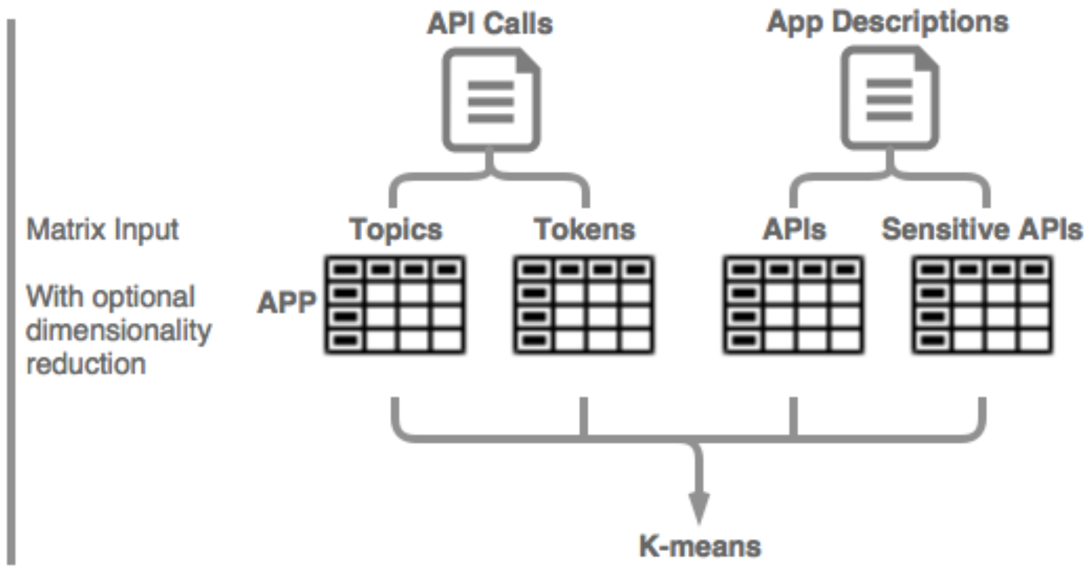


Figure 6: Matrix Input for K-means

Latent Dirichlet Allocation (LDA) is a topic model that generates topics based on word frequency from a set of documents.

LDA assumes documents are produced from a mixture of topics. Those topics then generate words based on their probability distribution, like the ones in our walkthrough model. In other words, LDA assumes a document is made from the following steps: (1) determine the number of words in a document. Let's say our document has 6 words (2) determine the mixture of topics in that document. For example, the document might contain 1/2 the topic "health" and 1/2 the topic "vegetables" (3) using each topic's multinomial distribution, output words to fill the document's word slots. In our example, the "health" topic is 1/2 our document, or 3 words. The "health" topic might have the word "diet" at 20% probability or "exercise" at 15%, so it will fill the document word slots based on those probabilities.

Given this assumption of how documents are created, LDA backtracks and tries to figure out what topics would create those documents in the first place. [15]

Hashing Vectorizer is a python class under *sklearn.feature_extraction.text,* which turns a collection of text documents into a sparse matrix holding token occurrence counts. This text vectorizer implementation uses the hashing trick to find the token string name to feature integer index mapping.

It takes parameters such as *stop_words, lowercase, norm* and etc. while IDF weighting is not applicable. Sometime, Hashing Vectorizer has collisions when number of features is small, for example, 24 in our case.

TF–IDF Vectorizer: Term frequency counts how many times that a word appears in the document. Inverse document frequency measures that if the word is rare or common across the collection of documents. Tf-idf, namely, is the product of tf and idf, which is often used as a weight factor in text mining. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

Latent Semantic Analysis: the class *sklearn.decomposition.TruncatedSVD* performs linear dimensionality reduction by means of truncated singular value decomposition (SVD). It operates on sample vectors directly. In particular, truncated SVD works on term count/tf-idf matrices as returned by the vectorizers, which is known as latent semantic analysis (LSA).

The intuition behind our experiment is that default categories are not precise enough to reveal app functionalities. Therefore we applied unsupervised machine learning techniques, k-means, to cluster apps. Document clustering is usually suffering from big volume, high dimensionality and complex semantics. Suggested by [6][17][18], k-means is proved to be efficient for document clustering by various researchers, along with bag-of-words approach and dimensionality reduction. K-means is a simple algorithm and works well with large-scale datasets. Comparing with hierarchical clustering, K-means is computationally faster and producing tighter clusters.

K-means is an unsupervised clustering technique to partition a set of n samples into a set of k clusters. K-means assign every sample to exactly one cluster using Euclidean distance by default, the square root of sum of squared. Clustering tasks are targeting to minimizing intra-cluster distances and maximizing inter-cluster distances, shown in Figure 7.

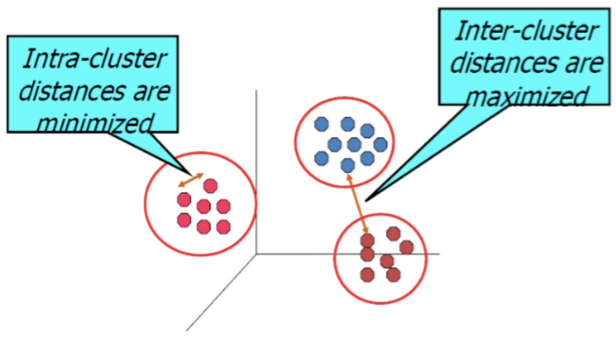

Intra-cluster distances are minimized
Inter-cluster distances are maximized

Figure 7: Intra-cluster and Inter-cluster Distance Explanation for K-means [16]

Samples are randomly assigned to K clusters initially and each cluster roughly has the same number of data points. Data point will be moved into the closest cluster in terms of distance to cluster centroid. It will be remained inside of its own cluster if the distance is already the closest one. Repeat the above step until a complete pass through all the data points results in no data point moving from one cluster to another.

However, k-means is sensitive to noises and often stuck in local optimum. The choice of initial partition can greatly affect the quality of final clusters.

3.2. Implementation

This section details the settings of our experiment and implementation process.

3.2.1. Clustering through LDA

Here we have a series of descriptions for each application and common steps of natural language processing are applied, given an example of descriptions for YouTube:

Example 1: Text Descriptions for YouTube

```
* Personalized &#39;What to Watch&#39; recommendations<p>* Launch a never-ending YouTube Mix from your favorite music videos<p>* Find music faster by playing albums or artists right from search<p>* Cast videos straight from your phone to Chromecast and other connected TV devices and game consoles<p>* Search using text or voice<p>* Quickly find your favorite channels using the guide
```

Tokenizing: descriptions are firstly converted into lowercase and then a stream of tokens, which could be words, phrases, symbols, or any meaningful elements. Here I remove single letter, digits, punctuations and reserve words only.

Stopping: stop words like conjunctions ('to', 'on') or personal pronouns ('it', 'we') are meaningless to topic modeling and need to be removed. I use existing stop words package imported from nltk.corpus.

Stemming: Porter stemming algorithm is commonly used for stemming words to their roots. For example, "stemming," "stemmer," "stemmed," all have similar meanings; stemming reduces those terms to "stem." This is important for topic modeling, which would otherwise view those terms as separate entities and reduce their importance in the model. [15]

The result of above clean processing is a list of words for each application. The descriptions for YouTube after Natural Language Processing look like:

Example 2: Tokenized Text Descriptions for YouTube

```
'person', 'watch', 'recommend', 'launch', 'never', 'end', 'youtub', 'mix', 'favorit', 'music', 'video', 'find', 'music', 'faster', 'play', 'album', 'artist', 'right', 'search', 'cast', 'video', 'straight', 'phone', 'chromecast', 'connect', 'tv', 'devic', 'game', 'consol', 'search', 'use', 'text', 'voic', 'quickli', 'find', 'favorit', 'channel', 'use', 'guid'
```

Constructing a document-term matrix: a document-term matrix helps us to understand how frequently a specific term occurs within the document, which is a necessary to generate an LDA model.

The Dictionary function under the package gensim traverses texts, assigning a unique integer id to each unique token while also collecting word counts and relevant statistics. The result is a series of tuples for each app descriptions. The tuples are (term ID, term frequency) pairs and now descriptions for YouTube become:

Example 3: Text Descriptions for YouTube in Dictionary Tuples

```
[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10,
1), (11, 1), (12, 2), (13, 1), (14, 1), (15, 1), (16, 1), (17, 1), (18, 2), (19,
1), (20, 1), (21, 1), (22, 1), (23, 2), (24, 2), (25, 2), (26, 1), (27, 1), (28,
1), (29, 1), (30, 1), (31, 2), (32, 1)]
```

Applying the LDA model: given document-term matrix, Ldamodel is now used to generate topics. Here we set number of topics returned to be 24, covering 24 default categories. Topics are printed with the top seven possible words contained and their possibilities (see Appendix [1]). Each application is assigned to some topics with possibilities greater than 0.05.

Document topics retuned for YouTube is as following, shown in Example 4. Descriptions of YouTube belong to topic 9 with a probability of 56.4%, topic 5 with a probability of 25.9% and topic 2 and 4 with some small extend.

Example 4: Document Topics with Probabilities for YouTube

```
[(2, 0.05188831966799428), (4, 0.10417479124956981), (5,
0.2585344832137057), (9, 0.56456907253038335)]
```

Creating an app-topic matrix: each application is assigned to some topics with certain possibilities at the last step and we then are able to generate a matrix with 10020 rows for each application and columns for 24 categories. The elements of matrix are the possibilities and most of them are zeros.

Grouping applications with similar descriptions: instead of grouping applications with plain descriptions directly, here we use topics assigned (app-topics matrix) as input for k-means, because few features allowing k-means works better. [6]

Performing metrics evaluation: three metrics are used to evaluate clustering performance, which are Adjusted Rand-Index, Silhouette Coefficient and Adjusted Mutual Information.

3.2.2. Clustering through Vectorizers

In Python, there are many classes developed by scikit-learn can be used for automatically converting text documents into term-document matrix rather than hand-processing like I did for applying LDA.

Table below summaries kinds of technique combinations tried to get best metric results:

Table 1: Experiments on Vectorizer Combinations

	Techniques	**1**	**2**	**3**	**4**	**5**	**6**
Vectorizer	Hashing	✓	✓		✓	✓	
	tf-idf	✓		✓	✓		✓
Dimensionality Reduction	LSA				✓	✓	✓

Hashing Vectorizer: the number of features set to 4k; stop words and lower case are applied; taking token frequency counts rather than binary occurrences; applying least square normalization if tf-idf not used. Experiment 1 performed IDF normalization on the output of HashingVectorizer.

Tf-idf Vectorizer: the maximum number of features set to 4k; stop words and lower case are applied; set maximum document frequency as 0.5, which means tokens appear over half proportion of documents are ignored; set minimum document frequency as 2 and value 2 represents absolute counts.

Performing LSA: the desired dimensionality of output data set as 24 temporarily; redo the normalization on the LSA results, which makes k-means behave as spherical k-means for better results; taking the sum of explained_variance_ratio for each of the selected components.

K-means is applied for 6 combinations. The number of clusters set as 24; using default method for initialization, 'k-means++'; max_iter and n_init are set as 200 and 10 respectively.

3.2.3. Clustering by APIs

API information is extracted directly from APK files and hence we have 9,980 individual text files, each of them contains APIs called by corresponding app.

Post process: we loop through all text files and built a list of lists storing APIs. Before getting to analysis, we exclude some meaningless APIs, for example, shown in the following table, which is part of API usages extracted from

YouTube,

Example 5: Part of Extracted API Usages for YouTube

```
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer.m',
'yv.e', 'qq.d', 'android.content.Intent.getAction', 'fnq', 'cbd.a', 'wf.d',
'java.lang.Object.getClass', 'java.util.Set.size',
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer.T',
'java.lang.String.equals',
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer.U',
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer.S',
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer.ab'
```

We get rid of any individual API (1) less than five characters in length; (2) less than three-string-combine structure (string.string.string); (3) remove the rightmost substring, which is just before the end of whole API string, containing only one or two characters after dot. Above snapshot would become:

Example 6: Part of Processed API Usages for YouTube

```
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer',
'android.content.Intent.getAction', 'java.lang.Object.getClass',
'java.util.Set.size',
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer',
'java.lang.String.equals',
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer',
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer',
'com.google.android.youtube.api.jar.client.RemoteEmbeddedPlayer'
```

To unify the clustering technique for metrics comparison, we applied tf-idf vectorizer along with dimensionality reduction to APIs. While, here, we used two ways of presenting term frequency, binary=True and binary=False (absolute term counts). Binary weighting enters 1 if an API presents and 0 if an API absents.

Parameters: Tfidf Vectorizer (max_df=0.5, max_features=4000, min_df=2); TruncatedSVD (n_components=24); KMeans (n_clusters=24, init='k-means++', max_iter=200, n_init=10)

Clustering by Sensitive APIs: sensitive permissions are gathered from file

AndroidManifest.xml for each application. Taken YouTube as an example, it totally requires 20 permissions to access users' sensitive information (see Appendix [2] for full list of permissions), listed below, which is extracted by matching regular expression patterns and only API package names are reserved.

Example 7: Sensitive APIs used by YouTube

```
['INTERNET', 'ACCESS_NETWORK_STATE',
'CHANGE_NETWORK_STATE', 'ACCESS_WIFI_STATE',
'WRITE_EXTERNAL_STORAGE', 'RECEIVE_BOOT_COMPLETED',
'MANAGE_DOCUMENTS', 'GET_ACCOUNTS', 'MANAGE_ACCOUNTS',
'USE_CREDENTIALS', 'READ_GSERVICES', 'GOOGLE_AUTH',
'GOOGLE_AUTH', 'GOOGLE_AUTH', 'RECEIVE', 'WAKE_LOCK', 'NFC',
'CAMERA', 'C2D_MESSAGE', 'READ_EXTERNAL_STORAGE']
```

Creating app-permission matrix: similarly, an app-permission matrix is created by firstly assigning each unique permission token a number, secondly representing permissions by a list of tuples (id, frequency), and lastly creating a matrix with 9338 rows and 334 columns. Some applications do not request any permission and therefore excluded. There are total 334 unique permission tokens found for all 9,338 applications.

K-means: identical parameters are used for unifying clustering technique and comparing evaluation metrics.

4. Empirical Study

This section presents the main research question followed by the description of our data collection process and the experimentation environment.

Researchers so far are suggesting various approaches while no comparison between either techniques or attributes. To investigate and compare categorization abilities of textual descriptions and APIs, we propose to answer the following research question.

(RQ) How effective is the API-based approach comparing to the textual based approach on app categorization?

We designed a series of experiments, summarized in the table below. As the common techniques used for automatically categorizing applications are based on textual mining, we only compared our approach to this type of approach. Table 1 shows that we have implemented multiple machine learning techniques for both approaches. Before answering this research, we fist carried out a parameter tuning experiments and find the best methods for each technique in our empirical study to answer this research question.

Table 2: Experiment Details

	Method	Clustering	Evaluation
Default Category			
App Descriptions	Latent Dirichlet Allocation	K-means	Adjusted Rand Index, Silhouette Coefficient, Adjusted Mutual Information
	Vectorizers (Hashing, TF-IDF), Latent Semantic Analysis		
API Usages	TF-IDF Vectorizer (binary=T or F or Sensitive APIs), Latent Semantic Analysis		

We select three evaluation metrics to access the quality of the category output, comparing with the default category from the app store.

(1) Adjusted Rand Index is an evaluation metric that measures the similarity between two assignments, ignoring permutations and with chance normalization. Adjusted_rand_score is symmetric: swapping the argument does not change the score. ARI is range from -1 to +1. Negative values indicate independent labelings and positive values mean similar clustering. Identical labeling is scored 1.

(2) Silhouette Coefficient is a metric to measure the goodness of the clustering method. The Silhouette Coefficient for a sample is $S = (b - a)/max(a, b)$, where a is mean intra-cluster distance with all other samples within the same cluster and b is the mean nearest-cluster distance, more specifically, to the next best fit cluster for the sample.

The Silhouette score return by package sklearn is an average over all samples, which should be in range -1 to +1 and value 0 indicate overlapping clusters.

(3) Adjusted Mutual Information is a variation from mutual information and is normalized against chance. AMI calculates the agreement between two assignments, true labels against predicted labels. AMI is ranging from 0 to +1 and for two clusterings U and V, AMI is given in formula:

$$AMI(U,V) = \frac{MI(U,V) - E[MI(U,V)]}{\max(H(U),H(V)) - E[MI(U,V)]}$$

Perfectly matching scores 1 and random labeling scores 0.

Data Collection

All applications used in this experiment are download from PlayDrone, a scalable Google Play store crawler, developed by Columbia University Department of Computer Science. To craw the Google Play store, PlayDrone employed Turkers (the workers of Amazon's Mechanical Turk) to register for Google accounts from a diverse set of IPs. With the harvest amount of Google accounts and four different APIs used for interact with Google Play servers, PlayDrone clawer discovers and downloads applications along with their metadata. There are six components of clawer architecture.

PlayDrone is able to download over 1.1 million Android apps and decompile over 880,000 free applications on a daily basis. To build our database, we extract the first one of every 140 applications over total 1.4 million apps to avoid biased sampling. For every application, we have a compressed APK file and a JSON file (metadata).

Totally we have 9,980 apps assigned to 24 default categories. Each app belongs to one category and the top frequent category is Business. The date of snapshot is 31st October 2014. Our framework was implemented in python and all our experiments were undertaken on a MacBook Air laptop running OS X EL Capitan 10.11.6 (15G31) (CPU: 1.3 GHz Intel Core i5, Memory: 8 GB 1600 MHz DDR3).

5. Parameter Tuning

To carry out a thorough evaluation, we designed an initial experiment to tune the machine learning algorithms and find the best combination of machine learning methods for both description-based and API-base approach.

We totally have 6 results for different combinations among the usages of vectorizers and dimensionality reduction technique. To unify the experiment results for comparison, we set the maximum number of features as 4,000 and number of components output by LSA as 24.

Table 3: Results in term of Silhouette Coefficient for Vectorizer Selection

	Techniques	Result 1	Result 2	Result 3
Vectorizer	Hashing	✓	✓	
	tf-idf	✓		✓
Dimensionality Reduction	LSA	✗	✗	✗
Clustering	K-means	0.009	0.012	0.019
		Result 4	**Result 5**	**Result 6**
Dimensionality Reduction	LSA	✓	✓	✓
Clustering	K-means	0.169	0.190	0.206

The combination of tf-idf vectorizer along with LSA gives the best result in Silhouette Coefficient, which are highlighted in red in the table. This combination is referred as the best combination in the following.

From the table we can tell, tf-idf vectorizer is much more powerful than hashing vectorizer; dimensionality reduction improves clustering quality a lot.

Hashing vectorizer is computational efficient as there is no need to store a vocabulary dictionary in memory. It also makes a problem, as there is no way to compute the inverse transform, from feature indices to string feature name.

Collisions happen sometimes, especially when the number of features is small, set as 4k in our case, which is not large enough for text classification problems. Distinct tokens can be mapped to the same feature index. If we set n_features as 2^18, model 4 performs slightly better but no big change.

K-means is very sensitive to feature scaling and that in this case the IDF weighting helps improve the quality of the clustering by quite a lot as measured against the "ground truth".

This improvement is not visible in the Silhouette Coefficient, as this measure seem to suffer from the phenomenon called "Concentration of Measure" or "Curse of Dimensionality" for high dimensional datasets such as text data. Other measures such as V-measure and Adjusted Rand Index are information theoretic based evaluation scores: as they are only based on cluster assignments rather than distances, hence not affected by the curse of dimensionality. [17]

Curse of Dimensionality: there is a trade-off between explained variance and Silhouette coefficient by tuning number of desired dimensionality of output data.

The default number of components is 2 and the recommended number of components for LSA is 100. The table below is generated by the best combination with maximum features set as 4k. It is obviously that with the increasing number of components, explained variance is increasing while silhouette coefficient is decreasing.

Table 4: 'Curse of Dimensionality' of Silhouette Coefficient

n_components	**2**	**10**	**24**	**50**	**100**	**200**	**500**
Explained variance of the SVD step	1%	6%	10%	16%	22%	32%	51%
Silhouette Coefficient	0.522	0.213	0.206	0.144	0.092	0.062	0.038

Parameter tuning for tf-idf vectorizer: there are three important parameters we are experimenting with, max_features, min_df and max_df.

Keeping min_df and max_df as default and averaging the results of 10 repetitions, we have the following table:

Table 5: Parameter Tuning for TF-IDF Vectorizer in term of max_features

max_features	**2000**	**4000**	**6000**	**10000**	**12000**
Explained variance of the SVD step	13%	10%	9%	9%	8%
Silhouette Coefficient	0.20373	0.2033	0.2045	0.20578	0.20423

It seems that maximum number of features has little influence on goodness of clustering. While, running time is proportional to the maximum allowance of features.

Keeping max_features as 4k and averaging the results of 10 repetitions, we have Table 6:

Table 6: Parameter Tuning for TF-IDF Vectorizer in term of max_df and min_df

TfidfVectorizer	**min_df = 1**	**min_df = 2**	**min_df = 3**
max_df=0.3	0.20053	0.20393	0.20310
max_df=0.4	0.20391	0.20449	0.20239
max_df=0.5	0.20244	0.20509	0.20442
max_df=0.6	0.17930	0.17311	0.17887
max_df=0.7	0.18118	0.17861	0.17241
max_df=0.8	0.18065	0.17631	0.17613
max_df=0.9	0.17638	0.17829	0.17860

It is obviously that average Silhouette Coefficient decreasing as the max_df increasing. Terms have a document frequency strictly higher than the given threshold are ignored when building the vocabulary, which is a form of corpus-specific stop word. A smaller max_df helps to filter the terms that are common in the corpus and hence improve the clustering quality.

We group 9,980 apps into 24 topics through LSA with some probabilities and then cluster them with app-topic matrix. Comparing with Tf-idf Vectorizer (max_feature=4000, max_df=0.5, min_df=2) along with dimensionality reduction technique, we conclude that topic modeling is slightly outperforming the tf-idf vectorizer in term of clustering quality, which has average Silhouette Coefficient around 0.236 and 0.203 respectively.

Followed best-performance parameters combination experimenting from clustering text descriptions, for tf-idf vectorizer, we have a huge improvement on goodness of clustering, boosting Silhouette Coefficient to around 0.508 (averaging result for 10 repetitions). Taken 'binary=False' also helps.

If we follow exactly the same procedure but replace all APIs by the subset, sensitive APIs, we get poorer clustering quality.

6. Results and Discussion

This section reports our results and provides answers to the research question. To access the effectiveness of the app classification approach, we applied the best combinations of the machine learning methods and settings found in the parameter-tuning phase to our data sample.

We ordered the topics and clusters by the number of apps in Table 7, because it is hard to say that these clustering results on every line are matching each other, for example, apps from cluster 6 by LDA, are not necessary representing all the BUSINESS apps. Clustering 6 might be a group of TOOL apps with size 906. We will present examples of topics and clusters in the case study later.

Table 7: Comparison between three Assignments

def_categories	category_size	cluster_index_LDA	cluster_size_LDA	cluster_index_API	cluster_size_API
BUSINESS	1043	**6**	906	**2**	2249
TOOLS	939	**2**	837	**4**	795
ENTERTAINMENT	852	**13**	833	**1**	663
EDUCATION	825	**4**	725	**3**	613
LIFESTYLE	815	**22**	700	**21**	584
PERSONALIZATION	731	**11**	607	**13**	434
MUSIC_AND_AUDIO	487	**8**	542	**0**	428
TRAVEL_AND_LOCAL	486	**19**	464	**9**	370
BOOKS_AND_REFERENCE	436	**5**	441	**6**	354
PRODUCTIVITY	427	**7**	417	**14**	330
HEALTH_AND_FITNESS	384	**1**	406	**22**	326
FINANCE	351	**17**	382	**18**	323
SPORTS	338	**14**	364	**7**	314
SOCIAL	313	**21**	358	**8**	296
NEWS_AND_MAGAZINES	297	**9**	284	**5**	279
COMMUNICATION	275	**18**	266	**12**	269
PHOTOGRAPHY	189	**15**	253	**11**	245
MEDIA_AND_VIDEO	181	**16**	211	**17**	209
SHOPPING	180	**10**	194	**15**	191
MEDICAL	158	**12**	168	**10**	186
TRANSPORTATION	140	**20**	165	**20**	185
WEATHER	48	**23**	162	**19**	134
LIBRARIES_AND_DEMO	48	**0**	158	**23**	109
COMICS	37	**3**	137	**16**	94

In general, the cluster sizes are quite different by three assignments if we group total 9,980 apps into fixed size of categories, 24 in our case. We observe that the size distribution of the description-based approach is closer to the real category, while the size distribution of the API-based approach is very different. The top frequent cluster returned through learning apps' API usages has size 2,249, compared with 1043 apps in BUSINESS category and 906 apps returned through clustering topics.

We now turn into the similarity metrics of the approaches. Adjusted Rand Index is measuring the similarity between two assignments and Adjusted Mutual Information is calculating the agreement between two assignments. Both ARI and AMI are comparing labels from two clustering ignoring permutations. The table below summaries averaging AMI and ARI from 10 repetitions, top right three figures are ARI and bottom left three figures are AMI, differentiated in black and blue. Each pair out of total three is compared.

From the table, we can tell that all three assignments are similar to each other since all ARIs are positives, but in small extent. Comparing default labels against labels generated by agreement between those two are slightly higher than other pairs and the similarity between those two clustering says that way as well.

Table 8: ARI and AMI across each pair of three assignments

AMI\ARI	def_labels	LDA_labels	API_labels
def_labels		0.07850	0.02244
LDA_labels	0.17732		0.01712
API_labels	0.06646	0.06970	

There is a huge improvement in terms of Silhouette Coefficient by analyzing API usages, comparing taking app-topics as input for k-means. Clustering quality through analyzing API usages doubled the clustering quality through topic modeling, which proves that API calls are far more effective for automatically app categorizing.

Table 9: Silhouette Coefficient comparison for LDA and API

	LDA	API
Silhouette Coefficient	0.236	0.508

Analyzing API usages provides App store a more accurate way to categorize apps as APIs truly reveal functions implemented by the apps rather than descriptions. Collecting API calls also can be used for outlier detections, for example, if any abnormal usages.

Case Study

Let us take a deep look into a particular app, category and cluster. The LDA model believes that the words in descriptions of YouTube are generated from topic 9 with possibility 56.5%, topic 5 with possibility 25.9% and topic 4 and 2 with possibilities around 10%. The left side of the following table is YouTube description and right side lists top frequent topic words. It is very reasonable that topic 9 has highest possibility. It is also understandable that topic 5 is matching a new version of YouTube launched with more device–end support. Some key words from topic 4 and 2, such as ‘use’, ‘time’, ‘servic’ are matching that the benefit mention in the description is saving time.

Example 8: (LHS) YouTube Text Descriptions (RHS) Frequent tokens from Topics assigned to YouTube

'* personalized &#39;what to watch&#39; recommendations<p>* launch a never-ending youtube mix from your favorite music videos<p>* find music faster by playing albums or artists right from search<p>* cast videos straight from your phone to chromecast and other connected tv devices and game consoles<p>* search using text or voice<p>* quickly find your favorite channels using the guide'	(9, '0.035*video + 0.035*music + 0.030*radio + 0.020*play + 0.020*game + 0.017*app + 0.016*player') (5, '0.020*app + 0.020*use + 0.015*devic + 0.015*applic + 0.013*android + 0.012*version + 0.012*support') (4, '0.019*map + 0.018*app + 0.018*locat + 0.015*find + 0.013*inform + 0.012*citi + 0.010*use') (2, '0.024*app + 0.016*time + 0.016*messag + 0.015*use + 0.014*contact + 0.012*servic + 0.011*track')

Default category for YouTube is MEDIA_AND_VIDEO, cluster assigned through LDA is 19 and cluster assigned through API is 13. If we choose YouTube as benchmark and we shall believe that all apps from MEDIA_AND_VIDEO should be assigned to cluster 19 and cluster 13. What is really happening?

The following Venn diagram shows the overlap among three clusterings. The size of default category MEDIA_AND_VIDEO is 181, which has 30 overlaps with cluster 19 through LDA and only 5 overlaps with cluster 13 by API. That means there are 30 apps assigned to cluster 19 through LDA and original belonging to MEDIA_AND_VIDEO. Overlaps between each pair of assignments are really small, which supports conclusion that these three assignments are indifferent.

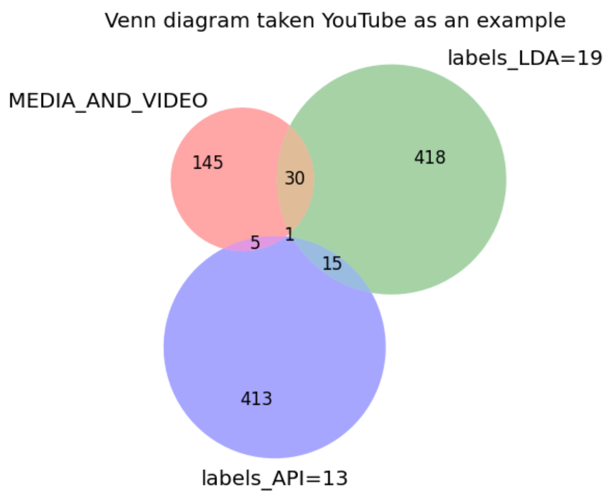


Figure 8: Agreement among three assignments taken YouTube as benchmark

What apps are assigned to cluster 19? What properties they got?

Analyzing 464 apps from cluster 19 and top five frequent categories are summarized on table 10. There are 182 apps originally from MUSIC_AND_AUDIO, 97 apps from SPORTS, 83 apps from ENTERTAINMENT, 46 apps from MEDIA_AND_VIDEO and 13 apps from LIFESTYLE. Why those apps are grouped into single cluster?

Table 10: Top Five Categories within cluster 19

def_categories	category_size
MUSIC_AND_AUDIO	182
SPORTS	97
ENTERTAINMENT	83
MEDIA_AND_VIDEO	46
LIFESTYLE	13

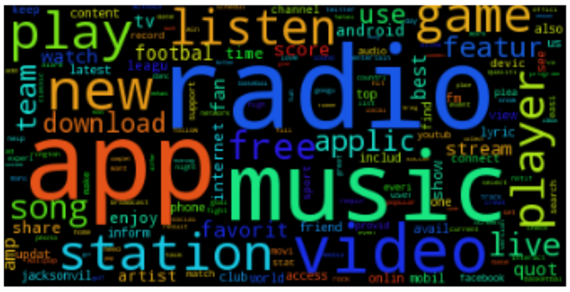


Figure 9: Visualized frequent tokens within cluster 19 through LDA

The most common three topics are topic 9, 14 and 5 with average probabilities 0.470, 0.198, and 0.185 respectively. Figure 6 is word cloud showing frequent tokens from descriptions within cluster 19. The bigger size of the tokens, the more frequent it appears in the collection of corpus.

Apps descriptions from cluster 19 are sharing tokens such as ‘radio’, ‘music’, ‘app’, ‘video’, ‘play’, ‘listen’, ‘game’, ‘player’ and so on. It is highly possible that apps are grouped into cluster 19 if descriptions of apps containing variations of those tokens.

What APIs the apps are using within the same cluster? The most common 60 APIs used within cluster 13 are listed in Appendix 4. Since tf-idf weighting is used for excluding common APIs in the collection, APIs with medium level of occurrences matters in fact, neither top nor bottom APIs. Looking into those APIs, we can tell that apps are grouped into cluster 13 because of the behavior that invokes classes for Google Mobile Ads, ‘com.google.android.gms.ads’.

7. Conclusions & Future Work

Conclusions

Mobile App Store is a unique platform mixing activities between developers and users. Both of them will benefit from the automated categorization approaches we investigated in this paper. Previous work has concentrated on the text description based approach, which is known for low granulated categorization. We implemented two ways of analyzing text descriptions of apps. One of them relies on topic modeling technique.

We firstly study the corpus and generate 24 topics from them. Secondly, each app is assigned to several topics with some probabilities and lastly, we cluster apps taking app-topic matrix as input. The other method cluster apps using processed tokens directly along with dimensionality reduction. Under this method, we study the clustering power of two vectorizers, hashing vectorizer and tf-idf vectorizer. Later is better. Internal metric, Silhouette Coefficient, is the main measurement we used for evaluating goodness of clustering. We have the experiment repeated for 10 times and take the average. Clustering app text descriptions with the help of topic modeling seems outperform tf-idf vectorizer slightly.

To overcome the limitations, we proposed an API-based categorization approach. We explore three ways of counting APIs, which are absolute API counts, binary occurrence and considering only sensitive APIs. All of them are then applied to tf-idf vectorizer and dimensional reduction. When comparing to text-based approach, it turns out that absolute API counts contribute to the highest clustering quality. Comparing the clustering quality between through topics modeling and through absolute count of APIs, we have 0.236 against 0.508 respectively in terms of averaging Silhouette Coefficient for 10 repetitions. It is obviously that API usages have more power in clustering apps. We then look into other two metrics, Adjusted Rand Index and Adjusted Mutual Information, across each pair among three assignments. We find out that all three assignments are really different from each other.

It is beneficial to include API checking into review processes before apps are available to the public. In general, checking API usages (1) strengths current review processes (2) prevents malwares spreading and accessing users' privacy (3) detects pirate apps for copyright protection and in particular, checking API usages helps placing apps into proper categories and flagging any abnormal API usages.

Threats to Validity and Future Work

Sample data: we are trying to get real world data and build our database unbiased, while due to limited time and resource, the database have only 9,980 apps excluding 40 apps that unable to decompress APK files. Total size is too small compared with existing researches out there and the source of dataset is too simply. The dataset used by [6] doubles ours and PlayDrone indexed 887,220 applications, which limits reliability of our results.

Moreover, the dataset we used for our experiment has free apps only and also exclude the category Game. Not mention that free apps make our dataset biased. Free apps generate profit through advertisements, in-app purchases or donations, which would be identified as undesired behavior. It believes that paid apps are generally better in functionalities and more regulations on API usages. Our experiment results hence are conservative and could be improved by adding more paid apps.

Terms from source code: Suggested by McMillan et al [14], words from comments and identifiers outperformed API packages. Only single words are considered as attributes rather than bigrams because of better performance of single words for software categorization. Further work on automatically categorizing Apps could consider terms as attributes if source code is available.

APIs: API calls are explored for automatically app categorizations because we believe that API calls are practical attributes. Not limiting to categorizations, API calls also could facilitate many other researches, for example, malware detections, study app features and etc.. API calls are much more stable over terms and app descriptions, because APIs are pre-defined and named in structured ways by functions accordingly. Further study could extend API usages for many other studies.

Appendix

[1] List of column names of CSV file:

['key_id', 'app_index', 'id', 'name', 'developer_name', 'category', 'price', 'rating', 'number_of_rating', 'ratings_5', 'ratings_4', 'ratings_3', 'ratings_2', 'ratings_1', 'description', 'whats_new', 'date_published', 'size', 'downloads', 'comments', 'version', 'rank_of_downloads', 'permissions']

[2] Topics index with top seven tokens return:

[(0, '0.051*app + 0.015*free + 0.015*use + 0.014*make + 0.014*us + 0.013*get + 0.012*like'), (1, '0.117*com + 0.104*http + 0.080*amp + 0.071*www + 0.046*googl + 0.045*href + 0.036*target'), (2, '0.024*app + 0.016*time + 0.016*messag + 0.015*use + 0.014*contact + 0.012*servic + 0.011*track'), (3, '0.011*world + 0.011*year + 0.008*product + 0.007*countri + 0.007*servic + 0.007*offer + 0.007*one'), (4, '0.019*map + 0.018*app + 0.018*locat + 0.015*find + 0.013*inform + 0.012*citi + 0.010*use'), (5, '0.020*app + 0.020*use + 0.015*devic + 0.015*applic + 0.013*android + 0.012*version + 0.012*support'), (6, '0.036*english + 0.033*word + 0.029*languag + 0.020*translat + 0.014*dictionari + 0.013*learn + 0.009*applic'), (7, '0.019*makeup + 0.016*hair + 0.016*beauti + 0.013*water + 0.012*fashion + 0.011*natur + 0.009*cloth'), (8, '0.121*quot + 0.083*applic + 0.032*use + 0.025*list + 0.022*agreement + 0.019*licens + 0.017*inform'), (9, '0.035*video + 0.035*music + 0.030*radio + 0.020*play + 0.020*game + 0.017*app + 0.016*player'), (10, '0.028*cours + 0.020*light + 0.018*golf + 0.018*measur + 0.015*flashlight + 0.014*flash + 0.013*use'), (11, '0.061*wallpap + 0.032*live + 0.019*set + 0.018*screen + 0.015*background + 0.013*free + 0.012*phone'), (12, '0.046*car + 0.036*vehicl + 0.022*part + 0.013*driver + 0.012*drive + 0.012*talent + 0.011*li'), (13, '0.066*recip + 0.024*cat + 0.018*step + 0.017*weight + 0.015*kitchen + 0.014*food + 0.013*easi'), (14, '0.032*app + 0.026*news + 0.022*event + 0.017*inform + 0.014*mobil + 0.013*latest + 0.013*school'), (15, '0.035*mobil + 0.020*account + 0.020*call + 0.018*phone + 0.016*use + 0.016*bank + 0.015*card'), (16, '0.026*learn + 0.017*kid + 0.015*game + 0.013*number + 0.012*fun + 0.012*children + 0.011*train'), (17, '0.026*busi + 0.012*issu + 0.011*subscript + 0.009*current + 0.009*money + 0.009*user + 0.009*purchas'), (18, '0.050*sound + 0.040*estat + 0.029*properti + 0.028*real + 0.022*home + 0.021*ml + 0.020*rington'), (19, '0.058*theme + 0.029*quot + 0.023*font + 0.021*widget + 0.021*instal + 0.019*go + 0.017*icon'), (20, '0.024*licensor + 0.018*church + 0.015*bibl + 0.014*stori + 0.013*book + 0.013*read + 0.013*life'), (21, '0.065*photo + 0.034*share + 0.031*pictur + 0.027*imag + 0.022*app + 0.020*friend + 0.018*facebook'), (22, '0.034*calcul + 0.019*inform + 0.018*mortgag + 0.015*medic + 0.012*complet + 0.011*profession +

0.011*help'), (23, '0.017*attende + 0.015*de + 0.013*exhibit + 0.011*grill + 0.010*bread + 0.008*deem + 0.007*atom')]

[3] Sensitive Permissions for YouTube:

'android.permission.INTERNET, android.permission.ACCESS_NETWORK_STATE, android.permission.CHANGE_NETWORK_STATE,
android.permission.ACCESS_WIFI_STATE,
android.permission.WRITE_EXTERNAL_STORAGE,
android.permission.RECEIVE_BOOT_COMPLETED,
android.permission.MANAGE_DOCUMENTS, android.permission.GET_ACCOUNTS,
android.permission.MANAGE_ACCOUNTS,
android.permission.USE_CREDENTIALS,
com.google.android.providers.gsf.permission.READ_GSERVICES,
com.google.android.googleapps.permission.GOOGLE_AUTH,
com.google.android.googleapps.permission.GOOGLE_AUTH.youtube,
com.google.android.googleapps.permission.GOOGLE_AUTH.YouTubeUser,
com.google.android.c2dm.permission.RECEIVE, android.permission.WAKE_LOCK,
android.permission.NFC,android.permission.CAMERA,
com.google.android.youtube.permission.C2D_MESSAGE,
android.permission.READ_EXTERNAL_STORAGE'

[4] Most Common APIs used with cluster 13:

```
[('android.util.Log', 437),
 ('java.lang.Object', 420),
 ('android.app.Activity', 345),
 ('android.content.Intent', 344),
 ('android.app.Activity.onCreate', 341),
 ('java.lang.StringBuilder.toString', 320),
 ('java.lang.StringBuilder', 320),
 ('java.lang.StringBuilder.append', 318),
 ('com.google.android.gms.ads.AdRequest$Builder', 269),
 ('com.google.android.gms.ads.AdRequest$Builder.build', 269),
 ('com.google.android.gms.ads.AdView.loadAd', 255),
 ('android.widget.TextView.setText', 244),
 ('android.content.Intent.putExtra', 234),
 ('android.widget.Toast.makeText', 230),
 ('android.widget.Toast.show', 228),
 ('java.lang.String.valueOf', 222),
 ('android.net.Uri.parse', 217),
 ('java.lang.String.equals', 209),
 ('android.widget.Button.setOnClickListener', 201),
 ('java.util.ArrayList', 184),
 ('com.google.android.gms.ads.InterstitialAd.show', 179),
 ('com.google.android.gms.ads.InterstitialAd.setAdUnitId', 179),
 ('com.google.android.gms.ads.InterstitialAd', 179),
 ('com.google.android.gms.ads.InterstitialAd.loadAd', 178),
 ('android.app.AlertDialog$Builder', 169),
 ('android.app.Activity.onDestroy', 164),
 ('android.app.Activity.onResume', 163),
 ('android.widget.LinearLayout.addView', 160),
 ('java.lang.Integer.valueOf', 158),
 ('android.app.Activity.onPause', 157),
 ('android.view.LayoutInflater.inflate', 152),
 ('com.google.android.gms.ads.InterstitialAd.isLoaded', 151),
 ('com.google.android.gms.ads.AdListener', 149),
 ('android.view.View.findViewById', 148),
 ('android.os.Handler', 146),
 ('java.io.IOException.printStackTrace', 144),
 ('android.content.Context.getSystemService', 144),
 ('android.view.MenuItem.getItemId', 142),
 ('com.google.android.gms.ads.AdView', 140),
 ('com.google.android.gms.ads.InterstitialAd.setAdListener', 138),
 ('java.util.ArrayList.add', 138),
 ('android.view.MenuInflater.inflate', 136),
 ('java.lang.Exception.printStackTrace', 134),
 ('android.content.SharedPreferences$Editor.commit', 133),
 ('android.content.SharedPreferences.edit', 132),
 ('android.app.AlertDialog$Builder.setTitle', 132),
 ('com.google.android.gms.ads.AdView.setAdUnitId', 129),
 ('com.google.android.gms.ads.AdView.setAdSize', 129),
 ('android.app.AlertDialog$Builder.setPositiveButton', 127),
 ('java.util.ArrayList.get', 127),
 ('android.app.AlertDialog$Builder.setMessage', 123),
 ('android.app.AlertDialog$Builder.create', 120),
 ('android.content.Intent.setType', 119),
 ('java.lang.Integer.intValue', 114),
 ('java.lang.String.length', 112),
 ('java.util.Iterator.next', 110),
 ('android.widget.ListView.setAdapter', 110),
 ('java.util.Iterator.hasNext', 109),
 ('java.util.List.add', 108),
 ('android.content.res.Resources.getString', 107)]
```

Reference

[1] D. Levitas, *App Forecast: Over $100 Billion In Revenue by 2020.* Available Online: [http://blog.appannie.com/app-annie-releases-inaugural-mobile-app-forecast/] App Annie, 2016

[2] A. Boxall, *Google Play outgrew Apple's App Store in 2014, in number of apps and developers*, Available Online: [http://www.businessofapps.com/google-play-outgrew-apples-app-store-2014-number-apps-developers/] Business of Apps, 2016

[3] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia and Y. Zhang. *Clustering Mobile Apps Based on Mined Textual Features.* CREST, Department of Computer Science, UCL, UK

[4] C. McMillan, N. Hariri, D. Poshyvanyk and J. C. Huang. *Recommending Source Code for Use in Rapid Software Prototypes*

[5] M. L. Va´squez, A. Holtzhauer and D. Poshyvanyk. *On Automatically Detecting Similar Android Apps*

[6] N. Hariri, C. C. Herrera, M. Mirakhorli, J. C. Huang and B. Mobasher. *Supporting Domain Analysis through Mining and Recommending Features from Online Product Listings,* IEEE Transactions on Software Engineering, VOL. 39, NO. 12, pp. 1736-1752, December 2013

[7] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. *WHYPER: Towards automating risk assessment of mobile applications.* USENIX Security Symposium, pages 527–542, 2013.

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

[9] C. Beach. *What is an API?* Available Online: [https://www.quora.com/What-is-an-API-4] Quora, April 2011

[10] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. *iComment: Bugs or bad comments?* ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 145–158, 2007.

[11] F. Sarro, A. A. Al-Subaihin, M. Harman, Y. Jia, W. Martin and Y. Zhang. *Feature Lifecycles as They Spread, Migrate, Remain, and Die in App Stores*, CREST, Department of Computer Science, UCL, London

[12] A. Finkelstein, M. Harman, Y. Jia, W. Martin, F. Sarro and Y. Zhang. *App Store Analysis: Mining App Stores for Relationships between Customer, Business and Technical Characteristics,* CREST, Department of Computer Science, UCL, London,

September 2014

[13] A. Gorla, I. Tavecchia, F. Gross and A. Zeller. *Checking App Behavior Against App Descriptions*. Saarland University, Germany

[14] C. McMillan, M. L. Vásquez, D. Poshyvanyk and M. Grechanik. *Categorizing Software Applications for Maintenance.* 27th IEEE International Conference on Software Maintenance (ICSM), 2011

[15] J. Barber, *Latent Dirichlet Allocation (LDA) with Python*, Available Online: [https://rstudio-pubs-static.s3.amazonaws.com/79360_850b2a69980c4488b1db95987a24867a.html#so-what-does-lda-actually-do] 2016

[16] Slide Share, Available Online: [http://www.slideshare.net/NontawatB/08-clustering], 2014

[17] P. Prettenhofer and L. Buitinck, *Clustering text documents using k-means*, Available Online: [http://scikit-learn.org/stable/auto_examples/text/document_clustering.html#example-text-document-clustering-py] Scikit-learn, 2016

[18] R. C. Balabantaray, C. Sarma and M. Jha, Document Clustering using K-Means and K-Medoids, Available Online: [http://arxiv.org/pdf/1502.07938.pdf], PublishingIndia, 2016

[19] K. Hornik, I. Feinerer, M. Kober and C. Buchta, *Spherical k-Means Clustering*, Journal of Statistical Software, Volume 50, Issue 10, September 2012

[20] L. Eadicicco, *Americans Check Their Phones 8 Billion Times a Day*. Available Online: [http://time.com/4147614/smartphone-usage-us-2015/], Time, Dec 15, 2015