# DroidVisor: An Android Secure Application Recommendation System

Pulkit Rustgi Carol Fung Bahman Rashidi Bridget McInnes
Department of Computer Science, Virginia Commonwealth University, Richmond, VA, USA
{rustgip, rashidib, cfung, btmcinnes}@vcu.edu

***Abstract*—In current Android systems, the application recommendation function is an important feature that users can use to find a similar application to replace a targeted one. The current recommendation system provided through Google and the Google Play store presumably recommends applications similar to a target application while accounting for the popularity of each application. However, it does not take the security features of each application or users preferences into consideration when doing so. In this paper, we propose DroidVisor, an Android tool that provides users with fine-grained and customizable application recommendations. Compared to the Google store recommendation function, DroidVisor does not only use the similarity to a preselected target application, but also considers other metrics such as popularity, security, and usability. More specifically, DroidVisor provides an interface for users to configure the weight of each metric and a recommendation algorithm that generates a list of recommended applications based on the combined scores. We evaluate our proposed criteria and the quality of recommendation through use case studies. Finally, we present our findings through a discussion of accuracy as well as possible ways to improve our recommendation results.**

## I. Introduction

The proliferation of mobile applications, commonly shorted to apps, is the primary driving force for the rapid increase of the number of smartphone users. It is predicted that the number of smartphone users world-wide will double from 2.7 billion in 2015 to 6 billion by 2020 [8]. The number of mobile apps has also been growing exponentially in the past few years. According to the official report by Android Google Play Store, the number of apps in the store has reached 1.8 billion, surpassed its major competitor Apple Apps Store, in 2015 [17]. As the number of smartphone apps increases, the privacy and security problem of users has become a serious concern [13], especially due to the nature of the tasks smartphones are used in when doing business. A malicious third-party app can not only steal private information, such as contact lists, text messages, and GPS location from the user, but can also cause the user to take a financial loss by making secretive premium-rate phone calls or text messages [16].

In current Android systems, users have to decide whether an app is safe to use or not based on intuition, which causes potential security concerns since most users are not technical savvy [14], [15]. Estimating the security risks [12], [5] of apps became an important concept that could help inexperienced users avoid installing malicious apps that may steal private information. For example, a flashlight app may request the identity and contact list from the user, which is considered malicious as that application may not necessarily require those permissions to function. In this case, users may be inclined to look for another similar app that does not require providing private information.

Currently, the Google play market provides the "*you may also like*" feature to recommend alternative apps to users. However, its recommendations only consider the similarity and popularity of apps and not their security features. For example, users may often see applications that were in the same category of their selected app, created by the same developer, or some highly downloaded popular apps. In order to provide a recommendation that protects the security and privacy of users, we propose *DroidVisor*, a secure app recommendation mechanism. The key idea of DroidVisor is to provide a customizable, secure, and high-quality list of application recommendations based on a target application.

More specifically, DroidVisor integrates the security aspect into the application recommendation system that can help users select apps with fewer security risks. To do so, we first provide an interface to allow users to configure their preferences on our recommendation metrics. We then propose corresponding methodologies to compute the app scores on all metrics, including similarity score and security score. For example, we use the Natural Language Processing (NLP) Lesk Algorithm [1] to compute the similarity score between apps, and we compute the security scores of apps based on their requested permissions. Finally, we integrate the preferences of the users to produce a final recommendation score which will be utilized to determine the list of recommended apps the user receives.

The contribution of this work can be summarized as follows: (1) we propose a novel Android app recommendation system that takes the security features as well as three other app metrics into consideration; (2) we propose multiple methodologies to score the apps on all corresponding metrics; (3) we evaluate the effectiveness of the system and compare it to the existing Google recommendation using real data.

The rest of the paper is organized as follows: Section II introduce background of this work; Section III briefly overviews the related work of this paper. Section IV describes the design of DroidVisor; We present our evaluation results and example outputs in Section V; and finally, Section VI concludes this paper.

## II. Background

To compute the similarity of two applications, we use a NLP approach. In this section, we discuss the Latent Dirichlet Allocation method and Lesk algorithm that are used to find similarity scores between apps.

978-3-901882-89-0 @2017 IFIP 

### A. Latent Dirichlet Allocation

Latent Dirichlet Allocation, or LDA [3], is described as a generative probabilistic model that can be used to model the topics of a document. LDA is a three-level hierarchical Bayesian model, in which each item of a collection is modeled as a finite mixture over an underlying set of topics. Each topic is, in turn, modeled as an infinite mixture over an underlying set of topic probabilities [3]. The primary goal of LDA is to define a general list of topics that represent key terms behind a document or set of documents. After each topic is defined, different word types are classified and placed into each topic.

*1) Machine Learning for Language Toolkit:* The MAchine Learning for LanguagE Toolkit, referred to as MALLET, is a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications to text [10]. The MALLET topic modeling toolkit contains efficient, sampling-based implementations of Latent Dirichlet Allocation that we will be utilizing later on in order to visualize outputs in a clear, textual format.

MALLET has multiple optional input parameters, but we need not concern ourselves with the utilization of most of them. We are however concerned with training MALLET, as it involves machine learning, and having it acclimate to different application genres and classifications. For the required inputs, we need the location of our applications descriptions, the number of ”topics” we want, and the destination the output should be placed in.

### B. Word Sense Disambiguation

In the context of Natural Language Processing, Word Sense Disambiguation (WSD) is the process of identifying the “sense” of a word, or there meanings in the context of which they reside, that may possibly have multiple meanings. It’s an approach that uses a words relative proximity to another in text and then attempts to analyze the context of which it resides in to find common themes.

The practice involves use of a predefined dictionary with the inclusion of homonyms, or words that sound the same. As every dictionary has a different interpretation of what a word can mean contextually, alternatively known as the ”sense” of a word, problems arise when different dictionaries or thesaurus are available. One solution to this is the use of a common data bank known as WordNet. WordNet is a computational lexicon that encodes concepts as synonym sets (e.g. the concept of car is encoded as $\{car, auto, automobile, machine, motorcar\}$).

*1) Lesk:* The Lesk algorithm, introduced by Michael E. Lesk in 1986, is based on the assumption that words in a given text file or description will tend to share a common topic or meaning. A simplification of Algorithm 1, which is a modified version of the original Lesk algorithm written by Satanjeev Banerjee and Ted Pederson [1], would typically be used to compare the dictionary definition of an ambiguous word with the terms contained in its neighborhood or surrounding section.

We later modify our own version of Algorithm 1 and use it to form our matrix visualization of textual similarity between applications.

**Algorithm 1** LESK Algorithm Pseudo-Code

```
1:  for every word w[i] in in the phrase do
2:      let BEST_SCORE = 0
3:      let BEST_SENSE = null
4:      for every sense sense[j] of w[i] do
5:          let SCORE = 0
6:          for every other word w[k] in the phrase, k != i do
7:              for every sense sense[l] of w[k] do
8:                  SCORE = SCORE + num of words that occur
9:                  occur in the gloss of both sense[j] and sense[l]
10:             end for
11:         end for
12:         if SCORE > BEST_SCORE then
13:             BEST_SCORE = SCORE
14:             BEST_SENSE = w[i]
15:         end if
16:     end for
17:     if BEST_SCORE > 0
18:        output BEST_SENSE
19:     else
20:        output “Could not disambiguate w[i]”
21:     end if
22: end for
```

## III. Related Work

Many works and studies have been done in the past few years on mobile application recommendation [18], [11], [4], [7], [9]. However, those works focus primarily only on applications with similar descriptions or functionalities, meaning none of the works take into consideration the security aspect of the recommended apps. In the rest of this section we discuss some of these selected works in more detail.

Bu et al. [4] proposed an algorithm to find similar apps using the *Transfer Learning* method. Their similarity function is based on apps’ descriptions, user reviews and other meta-information. Due to the use of Transfer Learning, their approach can cover apps from different markets. Furthermore, they combine different similarity functions to achieve improved accuracy. As a results, apps from markets where no similar ones are presented can also be recommended through their algorithm.

McMillan et al. [11] proposed an approach called *CLAN* that helps find similar apps based on each applications source code. CLAN can detect similar apps if the source code is written only in the Java language. In addition, they proposed an algorithm that calculates a similarity index value between the given Java app and other apps using the notion of semantic layers that correspond to class hierarchies. More specifically, CLAN computes the similarity based on an applications API calls. Since most apps make similar API calls, they assign weights to APIs and use a combination of calls to achieve a higher accuracy. In their work, they do not consider application descriptions and metadata which can result in false positives.

Linares-Vasquez et al. [7] proposed *CLANdroid*, an approach to automatically detect similar Android apps. CLANdroid is based on an information retrieval technique. CLANdroid is an extension of CLAN and does not only consider API calls, but also four more semantic anchors: identifiers, intents, permissions, and sensors. CLANdroid combines these features to achieve a higher overall accuracy. Combining different features can also help to find similar apps across different app categories.

Finally, Ma et al. [9] proposed *App2Vec* which models apps using vectors. One of App2Vec’s functionalities is to find the

top similar apps based on a word vectorization model called *word2vec*. App2Vec is mostly based on visualizing the time series of how users use apps. In their model, they consider every individual app as a ”document”. After collecting the apps’ usage data, they use word2vec to model the documents. As a result of this approach, they are able to find the top similar apps. App2Vec can be considered as a behavioral-based model which means that it finds similar applications based on the behavior of the users who use other similar apps.

Compared to the discussed works in this section, our proposed approach does not only find similar apps (in terms of functionality), but also considers the security aspect of the recommended apps.

There are also some existing works which focus on the analysis of the necessity of the requested permissions from apps [2], [6]. For example, Latifa et al. proposed PermisSecure to find the maximum set of Android permissions that an application may need. Bao et al. use a collaborative filtering technique to identify the necessary permissions that an app should request. However, our work evaluates the security risk levels of apps based on the permissions the apps request. Taking the security aspect into the model provides a more comprehensive and useful app recommendation system.

## IV. DroidVisor Design

DroidVisor is an Android app recommendation system that takes the security aspect of apps into consideration through a novel security scoring method based on requested permissions. In this section, we discuss the design of DroidVisor, including the scoring metric and algorithms, and our graphical user interface for weighing metrics and recommendations.

### A. Metrics Selection

When providing app recommendations to users, the first step is to identify several metrics that users may be interested in, but which data are also available. The four metrics that we adopted are described as follows:

*1) Similarity:* Similarity is the core metric since the purpose of recommendation is to find replacement apps that are both similar and secure. The similarity of two apps can be measured through comparing the textual descriptions of the apps. The description typically contains the app’s purpose, what users may be able to achieve through usage, the device requirements, or what a developer may want to mention.

*2) Security:* The security risks of apps is also a critical concern for smartphone users. An important novelty of DroidVisor is taking the security risk of apps into account in the recommendation. The security risk level of apps can be estimated through the permissions an app requests.

*3) Popularity:* Popularity can be also taken into consideration for the recommendation algorithm because users tend to pick more popular apps to install. The popularity can be measured using the number of downloads by users.

*4) Usability:* Usability measures how well the apps are designed and function, which is another typical metric that users care about. Usability can be measured by user ratings given to each individual app on a scale from 0.0 to 5.0.

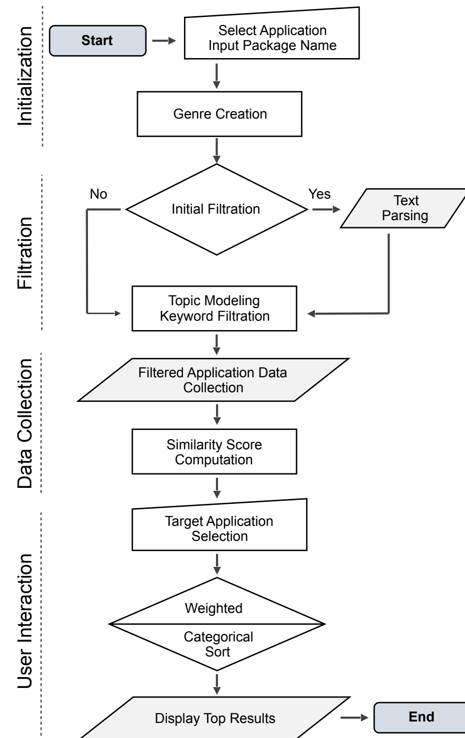


**Figure 1:** DroidVisor’s process flowchart.

### B. DroidVisor Design

DroidVisor uses the process shown in Figure 1 to compute the recommendation scores in order to find the recommend apps. The process can be divided into four key steps: *initial filtration*, *keyword filtration*, *metric scoring computation*, and *normalized sorting*. Once the final similarity score, $\sigma$, is calculated, DroidVisor then displays the highest scored apps to its users as shown in Figure 2(a). When a user chooses to see the details of an app, it will be displayed in a fashion identical to Figure 2(b).

(a) (b)

**Figure 2:** User Interfaces: (a) illustrates Google Chrome’s related apps using our proposed model; (b) shows more details of popular messenger application WhatsApp.

*1) Initial filtration:* In order to access the available apps, we did some primary filtration based on category (genre) to narrow down the scope of the initial search. We did this through a simple process that reads the category information of each app from its description in the dataset. We start with the set of apps Google Play lists as similar to the target app and recursively obtain similar apps in regards to our target app until the set is of sufficient size. Finally, we only keep the apps if they are in the same category of the target app and abandon the rest as they are not relevant enough to analyze.

*2) Keyword filtration:* After initial filtration, the list of apps that we have descriptions of is still relatively large. We then further filter the apps based on keywords. This is done by assigning each genre a list of non-generic keywords and setting a chosen threshold in which the minimum number of keywords in an apps description must be met for it to be added to our final pool. It can also be done by placing a cap on how many apps are allowed score calculation. If we choose to employ the app capping method, we simply analyze $N$ number of apps with the highest number of keywords in common with our target app. This is done through a process known as *Topic Modeling*, or *Latent Dirichlet Allocation* (see Section II-A), using the *MAchine Learning for LanguagE Toolkit* (MALLET).

Once our app specific keywords have been defined, we then go through the process of checking each text description in the remaining target apps pool and assigning them scores corresponding to the number of keywords each of their respective texts contain. For example, if we had the textual description of “A dog jumps over a log” with the keyword pool $\{dog, wolf, rabbit, log, boat\}$ that particular app and description would receive a matching score of 2 due to containing “dog” and “log”.

We are now given the choice of placing apps into our final pool based on whether they meet a threshold for a minimum number of keywords, or our capping method by selecting $N$ amount of apps with the highest amount of matching keywords. In order to keep consistent sizes for our final app pools amongst various different genres and target apps we chose to employ our capping method setting $N$, our final app pool size, to 75. If in the anomalous case that our target app did not make it into its own final app pool we choose to remove the app with the lowest keyword matching score in our pool and replace it with our target app.

*3) Metric scores computation:* At this stage we have established a pool of related apps to the target app. By utilizing *Natural Language Processing* (NLP) overlap measures, we create a matrix visualization of app *similarity scores* for all apps inside of our pool using the *Word Sense Disambiguation* (WSD) algorithm known as Lesk (see Section II-B1). We then visualized and mapped each score in an adjacency matrix format, where each column and row represented a separate app and the matrices elements representing how related one apps description is to another.

To calculate the *security scores* we first compute the *risk score* by counting the number of permissions requested by each application. We analyze each application and its individual permissions and assign each of the permissions a value of ”L”, ”M”, or ”H”, corresponding with low, medium, and high risk when accessed by an application in an isolated manner (meaning not in concurrence with any other permissions). The examples of permission risks are shown in Table I. We assign different weights to different levels, where low/medium/high correspond to weight $0.33/0.67/1.0$ respectively. Once we find the score corresponding to each permission we then sum them together. An applications score for the popularity and user rating categories are taken by looking at the downloads and user rating metric provided by Google Play.

In addition, the *usability scores* can simply be the rating scores, and the *popularity scores* can be computed using the logarithm of the number of downloads an app has. All four scores of each app are stored in a matrix for easy access.

| Permissions | Risk |
|---|---|
| Install shortcuts | L |
| Set an alarm | L |
| Control vibration | L |
| Network connection status | L |
| Device app history | M |
| Close other apps | M |
| Make app always run | M |
| Retrieve running apps | M |
| Photos/Media/Files | H |
| Read contacts | H |
| Get location | H |
| Use microphone | H |

**Table I:** Examples of permissions and their security risk levels.

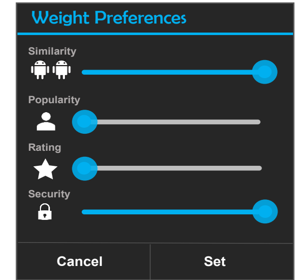


**Figure 3:** GUI design of weight tuner for DroidVisor.

*4) Normalized scoring and sorting:* The scoring methods on each metric produce metrics scores with different range scope. For example, the scores of usability is within $[0, 5]$ while the scores of popularity is within $[0, \infty]$. Our next step is to normalize them into the range of $[0, 1]$. This can be done by dividing each individual apps score with the highest value found in that metric. Note that $\bar{\beta}_j^i$ is the normalized score for metric $i$ and app $j$ and $\beta_j^i$ is the unnormalized score.

$$\bar{\beta}_j^i = \beta_j^i / \max_{\forall j}(\beta_j^i) \tag{1}$$

To tune and customize results for each user we incorporated a weighted sort or slider to be placed into the graphical user interface (GUI), which can be seen in Figure 3. Each individual user is able to select how much is the importance of each metric, with each section of the slider ranging from 0.0 to 1.0. Each normalized score is then multiplied out by the chosen weight each user has picked. These totals per app are then summed together to simply see which apps receive the highest score, as shown in the equation below:

$$\sigma_j = \sum_{i=1}^{3} \alpha^i * \bar{\beta}_j^i + \alpha^4 * (1 - \bar{\beta}_j^4) \tag{2}$$

where $\sigma_j$ refers to the total score for app $j$; $\alpha_i$ is the weight of metric $i$ set by the user. Note that the *security score* is the complement of the *risk score* due to their opposite meanings.

## V. Evaluation

### A. Experimental Setup

To evaluate our work on a real-life app, we use the Google Chrome app, with the associated package name `com.android.chrome`, as an example. We collected the

initial pool of apps, as described in Section IV, to populate our target app genre, which is“Communication”.

In our sample, the actual size of the initial pool before any form of filtration was 471 unique apps. After the initial filtration it is diminished to 251 apps. We then collect the textual descriptions of each app, provided their respective developer, and stored them for more reliable access.

While the previous filtration narrowed our pool size, we are still relying on genre classification to further reduce the pool. As described in Section IV-B2, we did this through a process called Topic Modeling utilizing Latent Dirichlet Allocation (see Section II-A) and a tool developed by Andrew McCallum called MALLET (see Section II-A1).

After carefully analyzing the results and accounting for pool degradation, we came to consensus that 2 is a suitable number of topics for MALLET in which we have a sufficient number of keywords for various description lengths, yet repetition of words remains minimized to avoid any possible topic overlap (see Table II). We continued on with our final filtration step, as described in Section IV-B2, which generated our complete app pool of 75 on which we perform our comparisons as shown in Figure 4.

**Table II:** Evaluation of number of topics.

| Topics \ Keywords | Total | Unique | Non-unique | Overlap |
|---|---|---|---|---|
| 1 | 19 | 19 | 0 | 0.00% |
| 2 | 38 | 37 | 1 | 2.63% |
| 3 | 57 | 55 | 2 | 3.51% |
| 4 | 76 | 73 | 3 | 3.95% |

After completing the app pool we moved on to evaluating the relevance and metrics of each of the apps that remain using our four primary categories (see Section IV-A). Obtaining our popularity and rating scores and then normalizing them was straight forward (see Section IV-B4), however we had to compute our similarity and security parameters.

To compute our remaining apps similarity scores we needed to compare the textual description in the final pool of apps to one another. We did so using a Perl implementation of the Lesk algorithm (see Section II-B1). To get our permission scores we used our permission calculation method as shown in Section IV-B3.

Once all of the four categorical scores were calculated, and normalized, we multiplied them with each of their corresponding $\alpha$ weights, or slider weights given by the user, as seen in Table III, to get our $\sigma$ scores. We recorded each response and visualized the top apps for “Chrome Browser - Google” in a fashion similar to Figure 2(a).

**Table III:** Trial category weights.

| Trial \ Category | $\alpha$ Similarity | $\alpha$ Popularity | $\alpha$ Usability | $\alpha$ Security |
|---|---|---|---|---|
| Trial 1 | 1.0 | 0.0 | 0.0 | 0.0 |
| Trial 2 | 1.0 | 0.5 | 0.25 | 0.0 |
| Trial 3 | 1.0 | 0.0 | 0.0 | 1.0 |

Our experiment environment is the IDE Eclipse Mars on a Windows 10 machine with a 2.6Ghz Intel i7 Core and 12G RAM. All experiments were run numerous times to confirm accuracy and are represented graphically as well as tabularly.

## B. *Trial 1*

The results in our first trial were completed in order to emulate how we imagine the Google Play store currently displays top apps. We tuned the weight values of each category to its respective value shown in Table III.

These values were then multiplied by DroidVisor’s four categorical scores calculated for each measure to result in the final $\sigma$ each app receives for that particular category. Table IV shows our evaluation for the most relevant apps with highest scoring criteria and overall $\sigma$ value when we used “Chrome Browser - Google” as our target app. These apps represent the top-most relevant apps out of our final pool of 75 apps relating to our target app.

**Table IV:** Evaluation of Trial 1.

| App \ Category | $\beta$ Similarity | $\beta$ Popularity | $\beta$ Rating | $\beta$ Security | $\sigma$ Total |
|---|---|---|---|---|---|
| Aon Browser | 1.00 | .444 | .851 | .278 | 1.00 |
| Opera Mini - fast web... | .409 | .836 | .936 | .139 | .409 |
| CM Browser - Fast & Light | .333 | .818 | .957 | .147 | .333 |
| Ninesky Browser | .278 | .602 | .894 | .156 | .278 |
| Dolphin - Best Web... | .275 | .830 | .957 | .102 | .275 |

## C. *Trial 2*

The results in our second trial were completed, in an identical fashion to our first trial, to emphasize the addition of the popularity and usability categories in conjunction with the similarity category previously tuned in Section V-B. In order to see if we could recommend a list of apps which are slightly more well known amongst app users, we put medium emphasis on the popularity category and slight emphasis on usability (see Table III) when tuning our $\alpha$ values in order to see if we could recommend a list of apps which are slightly more well known amongst app users.

**Table V:** Evaluation of Trial 2.

| App \ Category | $\beta$ Similarity | $\beta$ Popularity | $\beta$ Rating | $\beta$ Security | $\sigma$ Total |
|---|---|---|---|---|---|
| Aon Browser | 1.00 | .444 | .851 | .278 | 1.43 |
| Opera Mini - fast web... | .409 | .836 | .936 | .139 | 1.06 |
| CM Browser - Fast & Light | .333 | .818 | .957 | .147 | .982 |
| WhatsApp Messenger | .199 | 1.00 | .936 | .075 | .933 |
| Dolphin - Best Web... | .275 | .830 | .957 | .102 | .929 |

## D. *Trial 3*

We again follow the previous trials for our third trial in order to display the core purpose behind DroidVisor. We put heavy emphasis on the similarity and security categories (see Table III) when tuning our $\alpha$ values in order to see if we could recommend a list of apps which focus on the users safety, privacy, and security.

## E. *Results Discussion*

We can see that with respects to the similarity feature, which is prioritized in Section V-B, DroidVisor’s quality of recommendations seems much higher than those of the current Google Play store. This is seen by simply examining the amount of browsers that appear in the top list of apps similar to “Chrome Browser - Google” (see Table VII). DroidVisor’s top 5 apps all consist of browsers while only 2 of the 5

**Table VI:** Evaluation of Trial 3.

| App \ Category | $\beta$ Similarity | $\beta$ Popularity | $\beta$ Rating | $\beta$ Security | $\sigma$ Total |
|---|---|---|---|---|---|
| Aon Browser | 1.00 | .444 | .851 | .278 | 1.28 |
| Viber Messages & Ca... | .230 | .686 | .872 | 1.00 | 1.23 |
| Opera Mini - fast web... | .409 | .836 | .936 | .139 | .548 |
| Ghostery Privacy Browser | .196 | .523 | .872 | .294 | .490 |
| CM Browser - Fast & Light | .333 | .818 | .957 | .147 | .480 |

apps Google Play recommend, Firefox and Opera Mini, are browsers.

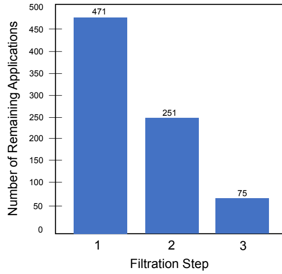


**Figure 4:** Progression of app filtration steps for the "Chrome Browser - Google" app.

| DroidVisor | Google Play |
|---|---|
| Aon Browser | Gmail |
| Opera Mini - fast web... | WhatsApp Messenger |
| CM Browser - Fast & Light | Firefox. Browse Freely |
| Ninesky Browser | Messenger |
| Dolphin - Best Web... | Opera Mini - fast web... |

**Table VII:** DroidVisor's versus Google Play's similar app recommendation for "Chrome Browser - Google".

While some of the apps that appear in our similarity list may not be well known, we adjust for this in Section V-C, by increasing popularity and rating weights, and the outputs again look promising. However, our observed results for DroidVisor start to appear more closely related to the recommendations by the Google Play Store with the addition of the popular messaging app "WhatsApp Messenger", which currently has over 49 million downloads, confirming our hypothesis of Google accounting for downloads when recommending apps.

The first drastic change we see in DroidVisor's recommendations is when we tune to adjust for the addition of the security feature. We see the addition of two new apps to our top list. Immediately we notice one of these apps, "Viber Messages & Calls Guide", does not match the criteria of actual similarity to our app in the sense of its intended purpose. When analyzing why this app was added to our top list we can immediately see the reason was because it requests very few permissions from the user. However, the second new app, "Ghostery Privacy Browser", matches our tuned criteria weights almost perfectly. It is both a browser and places heavy emphasis on security and privacy, as indicated by it's name and permission usage.

Overall results look promising with minor refinement being needed in regards to textual analysis, which is completely dependent on the developers part, and the possible inclusion of handling permission usage and requests in conjunction with one another to assess the true risk level of an app.

## VI. Conclusion

DroidVisor is an Android application recommendation system which allows users to select a target application, define preference to weight recommendation metrics, and then display a list of applications a user may find more usable than their selected target application. We propose the four metrics a user may find relevant are textual similarity, number of downloads, user rating, and security. Our evaluation based on real data has shown that in its current form the outputs are satisfying at recommending highly similar and secure apps. As our future work, we plan to apply other categories into our summation/score measure in order to create a more accurate representation of what applications may fit the needs of a user.

## References


[1] S. Banerjee and T. Pedersen. An adapted lesk algorithm for word sense disambiguation using wordnet. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 136–145. Springer, 2002.

[2] L. Bao, D. Lo, X. Xia, and S. Li. What permissions should this android app request? In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 36–41, Nov 2016.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[4] N. Bu, L. Yu, W. Ma, C. Du, S. Niu, and G. Long. Detect similar mobile applications with transfer learning. In *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pages 856–859, Dec 2015.

[5] Y. Chen, M. Ghorbanzadeh, K. Ma, C. Clancy, and R. McGwier. A hidden markov model detection of malicious android applications at runtime. In *2014 23rd Wireless and Optical Communication Conference (WOCC)*, pages 1–6. IEEE, 2014.

[6] E.-R. Latifa and E. K. M. Ahmed. A new protection for android applications. *International Journal of Interactive Multimedia and Artificial Inteligence*, 3(Regular Issue), 2016.

[7] M. Linares-Vsquez, A. Holtzhauer, and D. Poshyvanyk. On automatically detecting similar android apps. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, May.

[8] I. Lunden. 6.1b smartphone users globally by 2020, overtaking basic fixed phone subscriptions. http://techcrunch.com/2015/06/02/6-1b-smartphone-users-globally-by-2020-overtaking-basic-fixed-phone-subscriptions.

[9] Q. Ma, S. Muthukrishnan, and W. Simpson. App2vec: Vector modeling of mobile apps and applications. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 599–606, Aug 2016.

[10] A. K. McCallum. Mallet: A machine learning for language toolkit. http://www.cs.umass.edu/ mccallum/mallet, 2002.

[11] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 364–374, June 2012.

[12] B. Rashidi and C. Fung. Xdroid: An android permission control using hidden markov chain and online learning.

[13] B. Rashidi and C. Fung. A survey of android security threats and defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 6(3):3–35, September 2015.

[14] B. Rashidi, C. Fung, and T. Vu. Dude, Ask The Experts!: Resource Access Permission Recommendation with RecDroid. In *IFIP/IEEE International Symposium on Integrated Network Management (IM 2015)*.

[15] B. Rashidi, C. Fung, and T. Vu. Recdroid: A resource access permission control portal and recommendation service for smartphone users. In *Proc. of the ACM MobiCom Workshop on Security and Privacy in Mobile Environments (SPME '14), Maui, Hawaii, USA*, pages 13–18. ACM, September.

[16] W. Rothman. Smart phone malware: The six worst offenders. http://www.nbcnews.com/tech/mobile/smart-phone-malware-six-worst-offenders-f125248.

[17] Statista. Number of available applications in the google play store from december 2009 to november 2015. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[18] F. Thung, D. Lo, and L. Jiang. Detecting similar applications with collaborative tagging. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 600–603, Sept 2012.