



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Mihai Cotizo Sima

Help! My birthday reminder wants to brick my phone

Master Thesis MA-2014-09
March 2014 to September 2014

Tutor: Dr. Stephan Neuhaus
Co-Tutor: Brian Trammell
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

In recent years, Android has gained a lot of popularity with smartphone users, making it a target platform for malware. To protect the user, Android uses a permission system which limits an application's access to private data. However, the users have no fine-grained control over which permissions are granted to the application. In order to mitigate this problem, this project introduces a novel recommendation system which can find applications with similar functions, but which are safer. Based on the accumulated information about a significant number of apps, further research questions, such as which pairs of permissions are likely to be asked together, are explored. These analyses lay the foundation of future warning systems which have the potential of raising the user's awareness about what the applications can do.

Contents

1	Introduction	11
1.1	Problem statement	11
1.2	Android permission system	11
1.3	Recommendation systems. LDA	12
1.4	Further research questions	13
2	Methodology	15
2.1	Required data	15
2.2	System overview	16
2.3	System components	17
2.4	Data persistence	24
2.5	Further research questions	25
3	Evaluation	27
3.1	Data collection (<i>Crawler, Word extractor and Permission mapper</i>)	27
3.2	Quality of data, relevance	28
3.3	Efficiency of LDA	28
3.4	Efficiency of the recommendation system	28
4	Results	33
4.1	Topics	33
4.2	Permission distribution	35
4.3	Further research questions	36
5	Discussion	39
5.1	Recommendation system	39
5.2	Further research questions	43
6	Threats to validity	47
7	Related Work	49
8	Future Work	51
9	Conclusions	53
10	References	55
A	Running guide	59

List of Figures

2.1	Overview of the system pipeline.	16
2.2	Screen capture of the Amazon Appstore app page for “Talking James Squirrel (Free)” (store ID B00B71SDV4, visited on Sep 03, 2014). Of interest are the special Unicode tick characters.	18
2.3	Screenshot of the web interface showing the topic labels and the most relevant words.	21
2.4	Screenshot of the web interface showing the details of a particular topic: the words (including weights) and the apps (ordered by weights).	22
2.5	Screenshot of the web interface showing the details of a particular app, as recorded from the store page (price, description), or as computed (topics, cousins).	22
2.6	Screenshot of the web interface showing the search functionality using keywords.	23
2.7	UML diagram of the SQL schema.	26
3.1	Histogram illustrating the distribution of the number of permissions saved using the set-based filtering.	29
3.2	Average number of permissions of the closest cousin plotted against the number of permissions of the average app. For comparison, the dotted line has slope 1.	30
3.3	(a) Histogram illustrating the distribution of risks of the apps. (b) Histogram illustrating the distribution of app risk saved using risk-based filtering.	30
3.4	Average risk of the closest cousin plotted against the risk of the original application. For reference, the dotted line has slope 1. x-axis risks are grouped between every tenth integer, averaged, and used as a point on the graph. The blue band represents one standard deviation of the averaged group.	31
4.1	(a) Sum of weights all apps have for each topic. (b) Number of apps having the biggest weight corresponding to each topic.	34
4.2	Sum of weights produced by LDA for each word. Only words with top 20 weights are included.	34
4.3	Histograms of the distribution of number of permissions requested by apps. Figure (b) has a logarithmic y-axis.	35
4.4	Most popular 20 permissions, by percentage of apps in the data set requesting them.	35
4.5	(a) 2D histogram of the number of occurrences of each pair of permissions; only top 20 permissions by popularity are included. (b) 2D histogram of number of occurrences of each pair of permissions, normalized per permission column; only top 20 permissions by risk are included.	37
4.6	(a) 2D histogram of number of occurrences of each topic-permission pair; only top 20 permissions by popularity are included. (b) 2D histogram of number of occurrences of each topic-permission pair, normalized per permission column; only top 20 permissions by risk are included.	38

List of Tables

4.1 Listing of topics produced by LDA. "Topic name" contains the labels applied by human inspection of apps related to the topic. "Total weight" contains the sum of all the topic weights of apps for that topic. "Topic apps" contains the number of apps whose highest topic weight is associated to that topic. 33

Chapter 1

Introduction

1.1 Problem statement

The Android operating systems relies on permission enforcement in order to restrict access to sensitive data or device capabilities. The current system design forces the user to decide at install time whether she wants to grant the requested permissions or not. The user has no power to grant the app only a subset of the requested permissions, at the expense of app functionality.

In this context, this project aims to provide users with recommendations of alternative apps, which provide similar functions but which require a more favorable set of permissions.

Similarity between two apps can be evaluated in terms of a distance function defined on the app descriptions written by their authors. The descriptions are represented as a mixture of topics (i.e. app categories), a procedure known as topic modeling. The resulting model can be used to determine the weight each topic has in the description of the app. The distance function is computed based on the topic weights of every two apps, and then apps are sorted by similarity. The sorted list of possible recommendations is filtered in order to remove apps which are more dangerous.

1.2 Android permission system

Android is an operating system currently developed by Google. It is mainly targeted at mobile, touchscreen-enabled devices, and it is reported to have reached a market share of about 84.6 percent during Q2 2014 [23].

Android is based on the Linux kernel, which manages the interaction with the device hardware. Android applications are written in Java, and therefore are first compiled to bytecode for the JVM; however, in order to better manage running the apps on devices with limited memory and processing power (like a smartphone), this bytecode is translated to a special format which is run by the Dalvik Virtual Machine. In order to facilitate development, a special Java framework is provided for writing apps; this framework provides classes for different tasks, such as graphics, internet access or phone calls. App distribution is facilitated by the “app-stores” – online sites where developers publish their apps and which offer the app package to the users (e.g. Google Play Store, Amazon Android Appstore). [13]

The Linux kernel performs access control using the user and group IDs each process or file is marked with. Android takes advantage of this by creating a new user ID for each app installed on the device. Therefore, apps run in a separated environment (they are “sandboxed”); their files and memory locations are protected from the access of the other apps. Since the kernel manages access to the drivers and inter-process communication, it therefore can limit an app’s access to device components or its interaction with other apps. [13] [1]

Sensitive data or device capabilities (which do not belong to the app's sandbox) are therefore accessible only after requesting the appropriate permission. The author of the app includes all the permissions needed into the `AndroidManifest.xml` file of the app; on the other hand, the user is notified about the permissions that an app requires at install time, and can decide whether to continue the installation process or not. Therefore, the user cannot grant only a subset of the permissions at the expense of the app's functionalities. On the other hand, the user can completely disable some functions of the device (e.g. WIFI, GPS, Bluetooth), thus limiting the apps which have access to these components. At install time, after the user's approval, the system applies the permissions to the app; should the app access components it did not require permission for, an exception is thrown, and usually the app is killed. [13] [1]

Inter-process communication is possible by several mechanisms implemented in the system. One example is invoking `Intents`: they are messages which show that the app wants to perform a certain action; the system finds the appropriate message receiver, which can handle the app's `Intent` and perform the action. Since apps can provide data to other apps, or respond to the `Intents` of other apps, the developers can declare their own permissions which protect these components of their apps. [1]

1.3 Recommendation systems. LDA

In a traditional store, the owners have a finite shelving space, and therefore must decide which products will be displayed and which will be stored in the warehouse. Their decision is based on statistics made on data about consumer demand. On the other hand, a website has virtually infinite space for displaying items, by allowing the user to browse from one page to the other. However, the user still needs to be "driven" to products she might be interested in, otherwise she might never find an appropriate item. [25]

Website administrators implement recommendation systems for different items, for example products (Amazon, Google), or media (Youtube, Netflix, Spotify). Building a recommendation system faces the challenge of predicting the preference of each user, based on data about items they accessed before. These predictions can be made either by finding items with similar properties (content-based), or by finding items preferred by other similar users (collaborative filtering). Finally, the two approaches can be combined into a hybrid recommendation system. [25]

In order to reduce the complexity of the similarity measure, this project focuses only on relationships between items, not users of the system. Therefore, the items are characterized into item profiles, which can be regarded as their simplified representations. These item profiles can be regarded as vectors, where each vector element contains the value of one of the properties. Based on these vectors, one can define an appropriate distance (or similarity) measure. Popular distance measures include the Hamming distance (how many vector items differ), Euclidean distance or cosine distance (both appropriate when all vector items are numbers). [25]

In [3], Blei et al describe an algorithm (Latent Dirichlet Allocation – LDA), which computes a topic model based on the words in a document. In this model, each document is characterized by a mixture of topics (i.e. a vector of topic weights), which in turn are represented by a distribution over the set of words (i.e. each word has a weight in the profile of each topic). A word is not a "label" for a topic (i.e. a keyword); it can have an important weight within many topics.

LDA uses the text documents and the number of topics to model as input. The algorithm can be used to "train" a model over the documents (i.e. to determine the distributions of words and topics). Based on these distributions, the topic mixture can be inferred for other, new documents. LDA can be used in document classification (since it can be regarded as an effective method for dimensionality reduction), or for collaborative filtering (relationships between users and items are similar to relationships between words and documents). [3]

1.4 Further research questions

Crawling an appstore with the purpose of creating a recommendation system also has the benefit of retrieving app data which can be used in other analyses. This section contains the description of the questions which could be answered in the scope of this project.

Correlation between app cost and in-app purchases App developers have several ways of monetizing the apps they create. The straightforward alternative is making the app available for a fee, which the user has to pay before being able to access the app. However, paid apps deter users from installing, and are usually favored by communities whose trust has been built over other channels (e.g. marketing, brand power etc).

Another alternative for the developer is to offer in-app purchases to the user. These purchases can be used to unlock content (e.g. better weapons in games) or to extend access to the service (e.g. Whatsapp Messenger is free for one year, after which it requires an yearly fee). Providing in-app purchases does not necessarily entail that the app is free: a paid app might contain “pro features” which can be unlocked upon further payment.

The two variables whose correlation is studied are “is free” and “has in-app purchases”. For each app, these variables can be obtained from the price of the app, and from the presence of the `com.android.vending.BILLING` permission in the list of required permissions.

Permission-permission correlation Individual required permissions might expose the user to several threats. However, the severity of requesting different combinations of permissions is considerably higher. For example, requesting access to the list of contacts is acceptable for a contact-organizing app, but also requesting permission to access the Internet provides the app the possibility to upload the contact list to a malicious server.

Although the risk of different permission combinations has been studied in literature [9], it is also interesting to research what pairs of permissions are likely (or unlikely) to be requested together. Apart from the suspicious patterns which might be identified, answering this question might be the starting point of a warning system which can identify “unlikely” pairs of permissions requested by an app.

Permission-topic correlation Android permissions restrict access to specific data or device features. Furthermore, there are certain categories of apps which do not need access to some of the data or the features a device has (e.g. a card game app would probably not need access to the camera).

Since apps belonging to the same category have similar topic weights, finding a correlation between permissions and topics can reveal interesting patterns. Similarly to the “permission-permission” question, finding unlikely combinations of topics and permissions can help building a warning system which finds the topic model of an app and then identifies suspicious permissions which the app requires.

Chapter 2

Methodology

2.1 Required data

In order to implement an operational recommendation system, the following minimum set of app data was identified:

- **Application description:** the description text, supplied by the author of the app. The app descriptions are used as input for identifying apps which are similar. App stores display the descriptions in different sections (for example, Amazon Appstore has two sections, “Product features” and “Product description”), but a concatenation of all the sections is appropriate.
- **Application permissions:** a list of all individual permissions requested by the app. The permissions are required to filter the recommended apps, such that the user sees only apps which are “less risky”. It is expected that a majority of apps will only require default Android permissions, but third party permissions might also be found. However, the app stores show descriptions of the permissions which are easier to understand by the users; it is therefore required to map these descriptions to a canonical form, e.g. the Android permission string found in the `Manifest.permission` class.
- **Application ID:** the ID used by the store to identify the app. In an ideal case, the IDs would be mapped to a canonical form, e.g. the application’s package name, which would facilitate identifying the same app across different app stores.

The Google Play and Amazon Appstore for Android were analyzed with regards to the availability of this minimum set of data. Also, since the data would be collected automatically, the complexity of writing a web crawler was taken into account.

- **Google Play** offers the description of the app directly on the webpage, and the package name of the app can be identified in the URL. However, at the moment of writing the crawler, Google allowed users to see the permissions required by the app only if they were authenticated with a Google account, and if they clicked the “Install” button. The store was subsequently changed, and the permissions could be retrieved in a pop-up dialog by pressing the “View details” link on the page. Also, Google opted to present only a summary of the data that the app can access to the users, and not with the exact permissions that the app requires.
- **Amazon Appstore for Android** offers the description of the app and the permissions directly on the webpage. The permissions are displayed as user-friendly descriptions of what the app is allowed to perform, for the most frequently used permissions. The package name of the app is not available, but the unique Amazon product identification ID can be filtered from the URL of the page.

The Amazon Appstore was chosen as a primary data source, because it provides the required data directly on the webpage. However, the web crawler was implemented with an extensible

design, such that a special parser for Google Play webpages can be added later in the project.

Apart from the minimal set of app data described before, other information available on the webpage was stored:

- **Application name:** stored for the user interface of the recommendation system.
- **Application price:** stored for further data analysis (e.g. verifying correlations between in-app purchases and application prices).
- **Webpage URL:** stored for the user interface of the recommendation system.
- **Webpage HTML:** the raw HTML file processed by the crawler was stored to facilitate the reproduction and the debugging of the crawling procedure.

2.2 System overview

The “Unix philosophy” encourages programmers to design small, robust tools which perform a single job. Interoperability of these small tools is provided by a text-based interface and the possibility to “pipe” the output of one program to the input of the next. It is therefore preferred to compose several tools in order to perform a task, rather than write a monolithic application dedicated only to the task at hand. [29]

The design of the recommendation system is following the “Unix philosophy”, by splitting the project into several sub-components, which are implemented as standalone programs. The communication between components can be made by accessing the database which backs the project, and via Redis. Figure 2.1 provides an overview of the system pipeline, whereas a detailed description of what each component accomplishes is provided in the following sections.

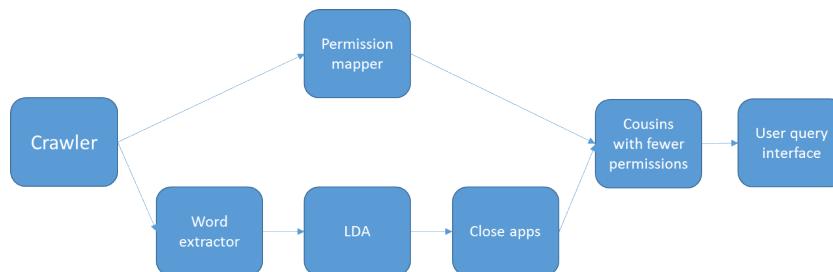


Figure 2.1: Overview of the system pipeline.

All the components load data from and store their output to the database, with the exception of the *Crawler* (which has the input online, from the appstore webpages) and the *User interface* (whose output is sent directly to the user). Each component relies on the output of the previous component, either for a single app (i.e. the *Word extractor* instances can start processing each app as soon as they are crawled) or for all the apps (i.e. the *LDA* component should run after all the apps have been processed by the *Word extractor* instances).

Redis [26] is used to satisfy this type of synchronization requirements. Redis provides blocking operations for several data structures, including (blocking) queues. For example, a redis client can issue a `BLPOP` command (“Block Left Pop”) which will block execution until an element can be popped from the left of a list. Given the fact that operations can be run atomically, redis can be used as a synchronization mechanism between the several components of the system. For example, the *Crawler* instances will fill a queue with app IDs which have been crawled; the *Word extractor* instances consume this queue, processing the apps whose IDs are retrieved.

2.3 System components

Crawler

The *Crawler* instances download webpages from the Amazon Appstore for Android and extract information about the apps. There are two types of webpages on Amazon: pages which index several apps (usually 60), and pages containing information about an app.

The *Crawler* must handle both types of pages: the "index" pages are used to enqueue "app" pages, while the "app" pages are used to extract app data. The redis queue contains URLs to Amazon pages. The queue is initially seeded by a separate script with a number of "index" pages.

In order to successfully record the app data from a store page, the *Crawler* should be able to retrieve the HTML code of the web page, emulate the DOM in memory, and possibly run the Javascript code which populates the page with dynamic data, if needed. NodeJS [21] is a Javascript interpreter built on top of Google Chrome's V8 Javascript engine. Since Javascript is already used to dynamically update webpages, being able to write Javascript code for the crawler represents an advantage (the "Javascript in the front-end, Javascript in the back-end" argument).

NodeJS is bundled with a package manager (the `npm` [22]), which has access to a repository of user-created libraries. The repository contains libraries which provide an easy way to bootstrap a webpage's DOM and run queries on it (e.g. `jsdom` [16] and `cheerio` [19]).

PhantomJS [14] provides a no-window (headless) full DOM and Javascript environment, which can be manipulated by writing Javascript code. PhantomJS is therefore a more powerful environment than using NodeJS and any of the DOM libraries; however, PhantomJS cannot run all the libraries available via `npm`, and there is no library for communicating with a database. Using PhantomJS would require a NodeJS wrapper script, which takes the output of the PhantomJS scraper and stores it in the database.

Given the extra layer of indirection required by PhantomJS, using only NodeJS is preferred. `Jsdom` provides more functionality than `cheerio`. However, a documented memory leak has been found in the library while running the *Crawler* in a prototype implementation. Since `cheerio` provides the a set of features which is sufficient for implementing the *Crawler*, it has been chosen for the final implementation.

Amazon Appstore "index" pages might contain duplicate apps. The *Crawler* can identify duplicates only by the Amazon product ID. However, it can be assumed that an app author will not submit the same app several times; apps which have a "lite" (free) version, and a "pro" (paid) version, can be safely be considered different apps altogether.

Amazon discourages automated data collection from their store pages. To limit the access of crawlers, Amazon uses CAPTCHAs whenever there are too frequent requests coming from the same client IP address. Amazon replies with a CAPTCHA page and with HTTP response code 200, so the *Crawler* is able to identify throttling and back off (delay crawling) for a period of time. Therefore, in order to avoid being throttled, a "Delayer" module was built. The Delayer is responsible for ensuring that each *Crawler* instance has a minimum waiting time between page requests (if scrapping one page takes longer than the cool-off, the next page is requested immediately). This module tries to emulate the behavior of a human user, who follows the links of the Appstore and spends some time on each page, reading its contents.

Word extractor

The LDA algorithm accepts as input documents in the bag-of-words format (i.e. each document is represented by a vector of pairs (w, c) , where w is a word appearing in the document, and c is an integer representing the number of times word w appears in the document). The *Word extractor* component is meant to process app descriptions into documents in the bag-of-word format.

Since LDA produces a model based on the frequency of each word in each document, there are some words which might lower the quality of the model. For example, short words such as “a” and “of” appear frequently in English texts, and are therefore expected to interfere with LDA. Also, words which appear frequently in the descriptions of Android apps (e.g. “facebook” or “Android”) have a negative impact on the model. Therefore, words appearing in the list of the most frequent approximately 5000 English words have been filtered, together with words shorter than 3 characters).

The Appstore is expected to also contain apps with descriptions written in another language than English. However, only a small fraction of such apps are expected to be crawled, and they would not affect the results of LDA. Moreover, users looking for a recommendation for an English app would likely disregard non-English apps. Given these reasons, the system would not treat non-English apps separately.

An important observation is that the app descriptions contain different Unicode characters, used by the app authors e.g. for creating textual bullet points or emphasis strings (see Figure 2.2). The support for Unicode strings is better in Python than in Javascript/NodeJS, and therefore the former language was preferred. Also, Python provides a regular expressions module with support for Unicode strings: using `\W+` as word separator would match any character which is not alphanumeric based on the Unicode character tables.

```

--- Features: ---
✓ High quality 3D graphics
✓ Cool voice interaction
✓ Piano with singing James
✓ Exciting touch game with 20 levels ins
✓ Great numbers game - check now the
✓ 1to60 game incl. 1 to 60, A to Z, 2 4 6
✓ Many different animations
✓ Amazing soundboard with many funny
✓ House game

```

Figure 2.2: Screen capture of the Amazon Appstore app page for “Talking James Squirrel (Free)” (store ID `B00B71SDV4`, visited on Sep 03, 2014). Of interest are the special Unicode tick characters.

Permission mapper

The Android permission system can be confusing even for app developers, as suggested in [11]. Therefore, the app stores usually present the users with intuitive descriptions of what the permissions allow an app to access or perform. Sometimes, app authors introduced spelling mistakes of the permissions in the app manifest files, which does not produce a compile error (the permission might be defined by another app), but which prevents the Appstore to provide a user-friendly description.

The *Permission mapper* component maps the different permission strings recorded from the app webpages to their canonical form (the Android permission as it appears in the manifest files, also compensating for developer errors where possible).

To facilitate the creation of the mapping table, a script which downloads the Android permissions (`android.permission.*`) was implemented. The script retrieves the manifest files from the Android source code, and stores the permissions, together with their labels and descriptions, into the database. Apart from the Android permissions, some other vendor-specific permissions (e.g. the Google Play and Google Services permissions controlling push messages or in-app purchases) were manually inserted into the database and the mapping table.

As presented in Section 7, Felt et al present in [10] a survey which measures the user’s concern regarding the actions enabled by different permissions. The actions are not specific to Android, but can be mapped to the Android permissions. The article includes a table with the results of the study, which lists the percentage of users who were “very upset” with the respective action. Therefore, the table was used as a starting point for evaluating individual permission

risks. These risks (or “severity”) are stored in the table of the Android canonical permissions (described earlier).

The study does not cover all the permissions which have been recorded from the Android manifest files. The missing severity values were estimated by comparing each permission with the other permissions (whose severity values were known from the paper). The comparison involved the data or the capabilities which the permissions protected. For example, the severity of the `WRITE_SMS` permission would be higher than `READ_SMS`, but lower than `SEND_SMS`; however, the severity of `WRITE_SMS` should be closer to the severity of `READ_SMS`, because `SEND_SMS` bears the risk of incurring costs to the user, which is more severe than the risk of deleting SMS messages (note that the SMS-related permissions mentioned already had risk values included in the cited paper, and are only used as an example).

The actual mapping between Amazon Appstore strings and canonical permissions is stored in a manually created CSV file. The *Permission mapper* loads all the permissions for each app and then looks for a match in the CSV file. If the match is found, the canonical string is added to the table containing the entry of the permission; otherwise, the entry is left `NULL`.

LDA

The LDA produces a topic model for the apps in the database, by producing association weights between apps and the topics. The input for LDA is represented by the app descriptions, processed into the bag-of-words format.

One implementation of LDA is provided by the Gensim [7] python library. Gensim contains implementation for several other topic modeling algorithms, and also support for running computations in parallel. Moreover, the LDA implementation provides access to several parameters (e.g. the number of topics, the topic prior) and does not need outputting intermediary files to disk; the internal representation of the model can be saved and training can be resumed at a later time (e.g. when information about new apps has been downloaded from the store).

Computing the topic weights for the apps is performed in two steps. First, the documents (app descriptions in the bag-of-words format) are used to train the model in several passes. Second, each document is used as parameter for the inference function of the model, which will produce an array of topic weights for the app. These arrays are stored in the database, to be used by the next component.

The quality of the model depends on the number of documents which are used for training. Since the number of apps which are available is an order of magnitude lower than the recommended threshold, the algorithm passes through the documents several times.

Another important parameter for the algorithm is the number of topics which should be modeled. The number of topics should not be too high (since similar apps might be “classified” into different topics), but also not too low (since different apps might be “classified” to the same topic). The implementation of the *LDA* component uses 20 topics.

The models built by LDA are sensible to very frequent and infrequent words, and therefore they should be removed from the bag-of-words documents. While the frequent words are already filtered by the *Word extractor* component, the infrequent words should be handled by the *LDA* component. This approach is preferred because the *Word extractor* has insufficient information (cannot know if a word which appears only once in the description of an app will or will not appear several times in the descriptions of the other apps). The *LDA* component will consider a word to be infrequent if it appears in the description of less than 10 apps.

Close apps

The *Close apps* component computes the “similarity” between any two apps, given the topics they are associated to. The *LDA* component saves for each app an array of weights, representing how correlated is the app to each topic. Any distance function based on these weight vectors can be used to compute the similarity of the apps.

Each instance of *Close apps* first loads the topic weight vectors for all the apps into memory. Then the instances consume the queue of app IDs to be processed, and for each app they compute the distance to all of the other apps in memory. Therefore, the complexity of the computation is $O(N^2)$, where N is the number of apps; the complexity of computing the distance function based on the two vectors is not included. However, the computation is highly parallelizable, and therefore having a high number of instances speeds up the computation considerably.

For two apps, given the weight vectors a_i and b_i , the distance can be computed in several ways:

- compute the Euclidean distance: $d = \sqrt{\sum_i (a_i - b_i)^2}$
- compute the angle between them: $d = \arccos \frac{a \cdot b}{\|a\| \|b\|}$
- any other vector metric $\| \cdot \|$ can be turned into a distance metric by $d = \|a - b\|$

For the *Close app* component, the angle between the two vectors has been implemented. The complexity of computing the distance function is $O(T)$, where T is the number of topics, leading the complexity of running the algorithm to $O(N^2T)$.

Loading the topic weight vectors into memory for all the apps is feasible since for each app, at most T float values are needed. The vectors can be represented in memory using pairs $\langle t, w \rangle$, where t is the topic index and w is the weight of the correlation between the app and topic t . Keeping these pairs sorted by t allows a linear computation of the distance function, using an algorithm similar to the one for merging two vectors.

Given the current implementation design, where each instance of a component is a separate process (possibly running on a different machine), the same memory amount is needed for each instance to store the weight vectors. Parallelization of the computation can be implemented also by using a shared-memory model, where several worker threads process the apps within a single process; this design is identical to the one which has been chosen, except for the possibility to run the worker threads on different machines (at least without building a framework for sharing memory over the network).

For each app, the *Close apps* component generates a list of similar apps, ordered by distance. These relations are then stored to the database, to be processed by the next component. The user expects only relevant recommendations, so the apps can be filtered by a distance upper threshold. Apps which are at the maximum distance allowed by the distance function (in the angle case, $\arccos 0 = \pi/2$) can be filtered, since they represent totally unrelated apps.

Storing the distance values in the database requires three numbers: the IDs of the two apps, and the value of the distance between the apps (a standard procedure to represent many-to-many relations). This results in $O(N^2)$ data to be stored, which for a large N is undesirable. Since the user is expected to only inspect a few recommendations that the system provides, the *Close apps* instances were programmed to store only the closest L apps for each app (L can be set via a parameter, here: $L = 200$; if parameter is left out, all the $O(N^2)$ pairs are stored).

Cousins

The recommendation system under design aims to highlight apps which are safer to use than the query app, from the point of view of the permissions required. Therefore, the list of apps sorted by distance is filtered by the *Cousins* component, to remove apps which are not safer than the query app.

There are several ways to define when an app is safer than another. The two definitions implemented in the *Cousins* component are:

- If app a requires a set of permissions S_a and app b requires set S_b , then app a “is safer” than b iff $S_a \subset S_b$; that is, a requires only some of the permissions which b requires, but not all of them.
- If app a requires a set of permissions S_a , then its risk level can be defined as $risk(a) = \sum_{p \in S_a} risk(p)$, where p is a permission in the set, and $risk(p)$ is the evaluated risk of the permission. Therefore, app a “is safer” than b iff $risk(a) < risk(b)$. The risk assessment

of each individual permission has been presented for the *Permission mapper* component (see Section 2.3). If a permission cannot be mapped to a canonical permission by the *Permission mapper*, then it will not be included in the *risk* of the app.

The *Cousins* component instances first load up the permission risk table, then start processing the apps in the queue. For each app, the list of apps produced by the *Close apps* component is loaded. Then, the cousins are filtered from the list, using the two definitions. The apps resulted from applying the two filters are stored into two different tables of the database.

User query interface

The purpose of the *User query interface* is to allow end users to retrieve the cousins of an app easier. The interface can either be a website, a mobile app or a desktop app; at the moment, a website has been implemented in NodeJS, using the Bootstrap UI library [4].

The interface allows the user to explore the available topics (see Figure 2.3). The individual page for each topic shows the word weights for the topic, and the apps ordered by topic weight (Figure 2.4). The user visits the page of an app, which shows the data retrieved from Amazon (i.e. price, description, permissions), but also data computed by the system (topic weights, cousins ordered by distance – Figure 2.5). Also, in order to facilitate finding an app, a full-text search index (on app description and app name) has been created in the database; the user can therefore search for an app using keywords (Figure 2.6).

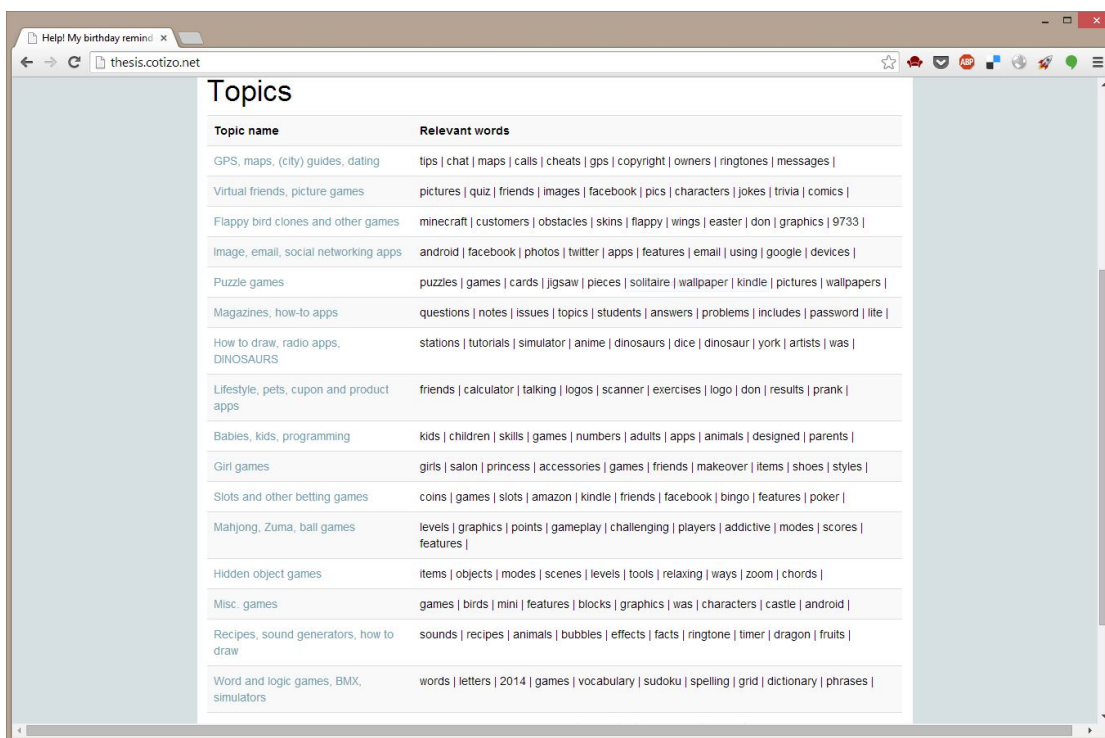
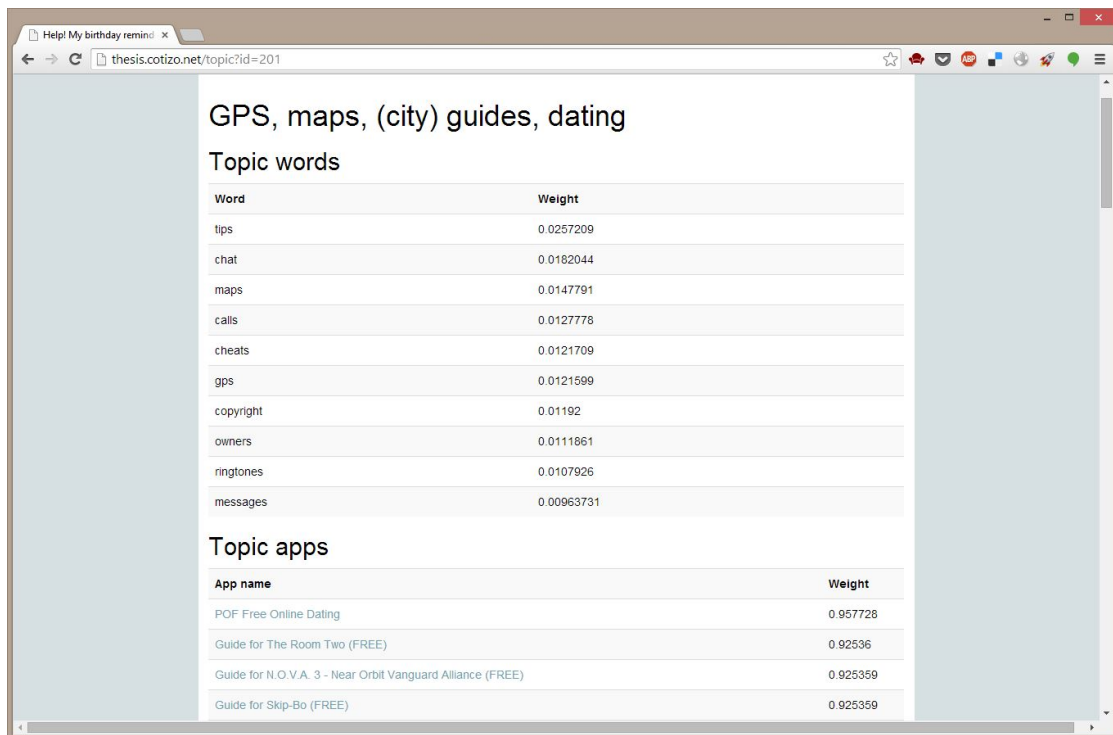


Figure 2.3: Screenshot of the web interface showing the topic labels and the most relevant words.

In order to limit burden on the database server, only the most significant results are shown on most of the pages: for example, on the individual topic page, only the top 100 apps (by topic weight) are displayed. The user can use the search function to access a particular app; if the search results do not include the desired app, using more keywords from the description will improve the results.

The interface is just a way to inspect data in the database, so there are very few computations made in its code. However, the *User query interface* is the appropriate component to convert the data into a user-friendly format. For example, the distance between two apps is converted to a



Help! My birthday remind x
thesis.cotizo.net/topic?id=201

GPS, maps, (city) guides, dating

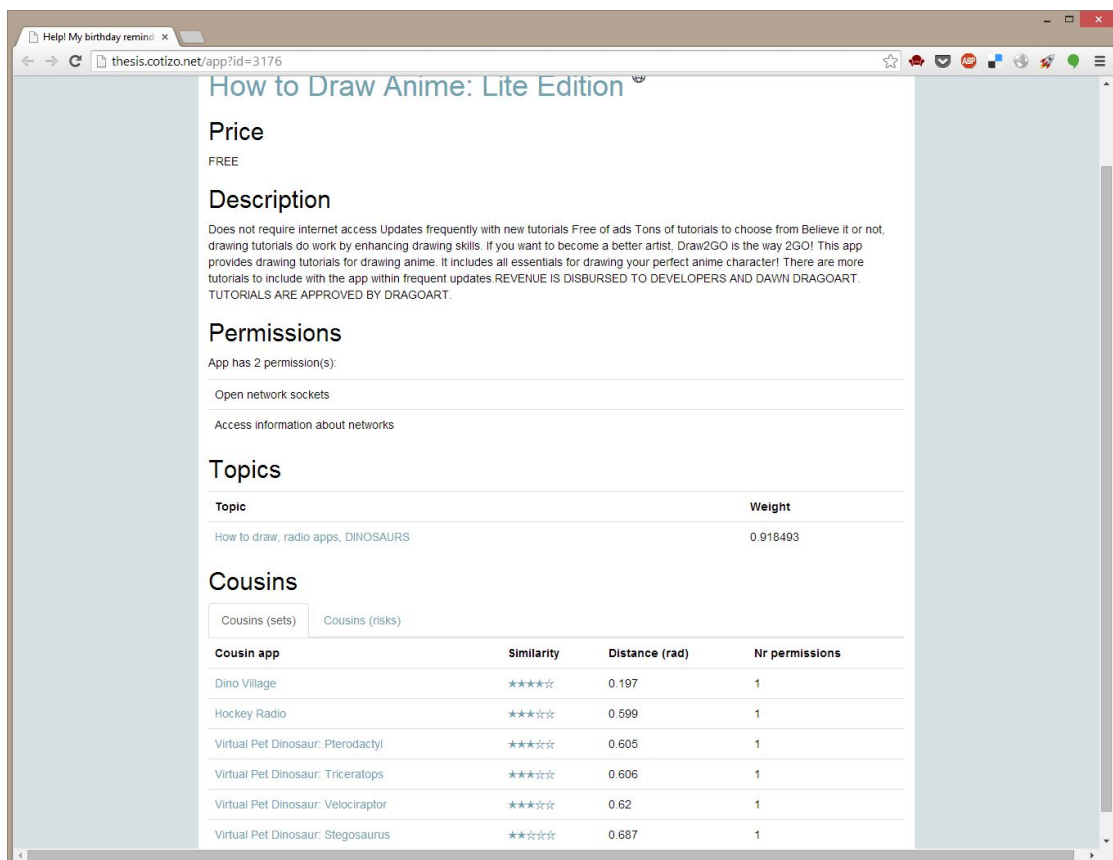
Topic words

Word	Weight
tips	0.0257209
chat	0.0182044
maps	0.0147791
calls	0.0127778
cheats	0.0121709
gps	0.0121599
copyright	0.01192
owners	0.0111861
ringtones	0.0107926
messages	0.00963731

Topic apps

App name	Weight
POF Free Online Dating	0.957728
Guide for The Room Two (FREE)	0.92536
Guide for N.O.V.A. 3 - Near Orbit Vanguard Alliance (FREE)	0.925359
Guide for Skip-Bo (FREE)	0.925359

Figure 2.4: Screenshot of the web interface showing the details of a particular topic: the words (including weights) and the apps (ordered by weights).



Help! My birthday remind x
thesis.cotizo.net/app?id=3176

How to Draw Anime: Lite Edition

Price

FREE

Description

Does not require internet access Updates frequently with new tutorials Free of ads Tons of tutorials to choose from Believe it or not, drawing tutorials do work by enhancing drawing skills. If you want to become a better artist, Draw2GO is the way 2GO! This app provides drawing tutorials for drawing anime. It includes all essentials for drawing your perfect anime character! There are more tutorials to include with the app within frequent updates. REVENUE IS DISBURSED TO DEVELOPERS AND DAWN DRAGOART. TUTORIALS ARE APPROVED BY DRAGOART.

Permissions

App has 2 permission(s):

- Open network sockets
- Access information about networks

Topics

Topic	Weight
How to draw, radio apps, DINOSAURS	0.918493

Cousins

Cousins (sets) Cousins (risks)

Cousin app	Similarity	Distance (rad)	Nr permissions
Dino Village	★★★★☆	0.197	1
Hockey Radio	★★★★☆	0.599	1
Virtual Pet Dinosaur: Pterodactyl	★★★★☆	0.605	1
Virtual Pet Dinosaur: Triceratops	★★★★☆	0.606	1
Virtual Pet Dinosaur: Velociraptor	★★★★☆	0.62	1
Virtual Pet Dinosaur: Stegosaurus	★★★★☆	0.687	1

Figure 2.5: Screenshot of the web interface showing the details of a particular app, as recorded from the store page (price, description), or as computed (topics, cousins).

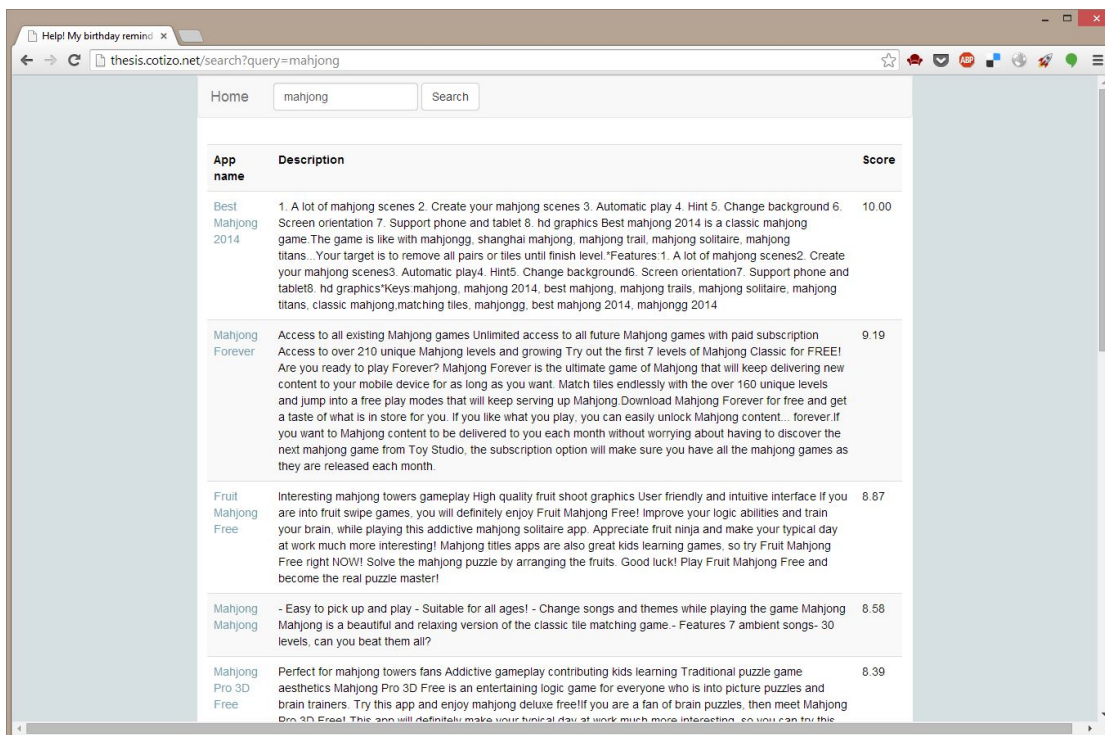


Figure 2.6: Screenshot of the web interface showing the search functionality using keywords.

5-star similarity measure using a linear transform between the $[0, \pi/2]$ angular distance interval and $\{0, 1, 2, 3, 4, 5\}$, using the formula:

$$\text{nr stars} = \left\lfloor 5 \left(1 - \frac{\text{angle}}{\frac{\pi}{2}} \right) \right\rfloor$$

In the equation, $\lfloor \cdot \rfloor$ represents the floor function.

Synchronization between components

Most of the components use the queueing primitives offered by redis (e.g. `BLPOP` and `RPUSH`). These commands enable the client to implement the producer-consumer synchronization pattern, which is used for example between the *Crawler* and the *Word extractor* and *Permission mapper* components (as soon as data becomes available, they start processing it). However, some of the components require some more advanced synchronization techniques.

The *LDA* component needs all the app descriptions to be available in the bag-of-words format in order to construct the model. Therefore, *LDA* waits for the queues of the *Crawler* and *Word extractor* to become empty. Of course, one method to implement this check is polling, but this form of busy waiting wastes resources (CPU). It is more efficient to be notified by the *Crawler* and *Word extractor*, so the *LDA* component first issues a `BLPOP` command on two queues. The *Crawler* and the *Word extractor* `RPUSH` a message once their queues are empty.

Redis ensures that the individual commands are atomic. This means that no two commands are run in parallel, but if one client issues two commands in series, there is no guarantee that there will be no other command (coming from a different client) executed between his commands. Therefore, redis offers the possibility to issue several commands in an atomic transaction using `MULTI` and `EXEC`. The *Word extractor* would then issue a `BLPOP` (to get the next app description to split) and a `LEN` (to check the length of the queue) in a transaction, and perform the logic described in the previous paragraph. However, the `BLPOP` command loses its blocking property when issued in a `MULTI` transaction, because otherwise the whole redis would block until an

element is pushed to the queue (a deadlock, since the producer never gets the chance to `RPUSH` to the blocked redis server).

Redis also has the `EVAL` command, which can be used to atomically evaluate a small script written in a dialect of Lua. The atomicity (mutual exclusion) is granted between clients trying to access redis objects which are acted upon by the script, and not all the keys in the server (i.e. the `EXEC` command knows which objects will be affected and other clients trying to touch these objects will be blocked). Therefore, redis can be used as a distributed mutex system ([30]). The mutex can be “acquired” or released like a regular mutex (e.g. Posix); if a client wants to acquire an already-locked mutex, it will be blocked (without deadlocking the whole redis server) until the mutex is released. This allows a client to issue consecutive commands (of interest: `BLPOP` and `LLEN`) without another client acting on the objects it wants to access. A similar mutex, called the “redmutex”, was implemented for both Python and NodeJS, and is used by the *Crawler* and *Word processor*.

Once the *LDA* finishes building the topic model and saving the topic weights to the database, the *Close apps* component can load the topic table in memory and start computing distances between apps. The component could re-load the table every time an app is processed, but this would not only waste resources, but also burden the MySQL server (which has an impact on the other components too). Therefore, the *Close apps* component is designed to have two threads: one which consumes the processing queue (populated by *LDA* with all the apps when it finishes processing), and one which listens for a signal from *LDA*. The signal is sent when the topic table is filled in; the receiving thread locks the processing thread (a traditional mutex is enough, since the threads share the same process) and re-loads the table into memory. Since *LDA*'s message that the topic table is filled must reach all the available *Close apps* instances, a publisher-subscriber pattern is used (as opposed to the previous case, where *LDA* had a single instance running, so it could just block on queue). Redis offers the pub-sub functionality using the `PUBLISH` and `SUBSCRIBE` commands.

Finally, the *Cousins* component needs data from two components: the *Permission mapper* and the *Close apps*. It is expected that the former finishes processing the apps well before the latter (since *Close apps* waits for *LDA*, which in turn needs all the apps to be already in the database). However, in order to be sure that the apps processed by *Cousins* have the permissions mapped, a hash is used. The $hash(app)$ is `true` if the app has been processed by the *Permission mapper*, and `false` otherwise. Redis can again be used for this synchronization problem, since it offers the `HSET` and `HGET` hash functions. In the unlikely case when the *Cousins* component must process an app whose permissions have not been mapped (i.e. $hash(app) = false$), the app is added at the end of the queue so it can be processed later.

In conclusion, redis is a versatile application which provides commands for manipulating a wide range of data structures. The atomicity of these commands can be exploited such that redis can provide several synchronization mechanisms to the clients.

2.4 Data persistence

The first implementation of the system was designed to store the data into a NoSQL database. NoSQL databases do not model data into the traditional, table-based format, but rather into documents with any number of fields. A popular NoSQL implementation is MongoDB [18]. In this implementation, the database documents were the apps (containing all the related data), and the topics. MongoDB provides several benefits, including the possibility of a flexible schema: should more app data be needed, the data could simply be added later to the documents by a second run of the *Crawler* instances.

Despite its flexibility, MongoDB makes heavy use of memory-mapped files, which require virtual memory to handle files which would not fit in RAM. The infrastructure used to test the system provided servers with no virtual memory, which limited Mongo to databases at most as large as the available RAM memory.

Since the required data for the analysis had been already defined, and the database had to

grow larger than the available RAM, the system was switched to a table-based schema, stored in a MySQL [20] database. The tables were defined as follows:

- `permissions`: this constant table is created by the SQL setup scripts. It contains data about the Android permissions recognized by the system (e.g. permission name or category).
- `apps`: stores information about the apps, such as name, description, price or store ID. To facilitate debugging and allow the study to be reproduced, a `MEDIUMBLOB` column stores the compressed HTML code of the app's store page.
- `app_words`: after the description is fetched, it is transformed in the bag-of-words format by the *Word extractor* component; the resulting pairs word-count are added to this table.
- `app_permissions`: contains the one-to-many mapping between an app and the permissions it requests. One column is used to keep the result of the mapping between the user-friendly permission description string used by Amazon and the canonical Android permission name.
- `apps_index`: is used to store the full-text search index needed in the query interface. The app name and description are automatically added by the *Crawler* to the table.
- `app_topics`: after LDA computes the topic weights for each app, they are stored in this table as tuples (`app_id`, `topic_id`, `weight`).
- `topics`: contains the mapping between topic ID and topic label (topic name).
- `topic_words`: contains the (word, weight) pairs generated by LDA for each topic.
- `close_apps`: the *Close apps* component computes the distance between every two apps, and stores the closest pairs in this table, together with the respective distance.
- `cousins_risks` and `cousins_sets`: after the close apps are sorted by distance, they are filtered either by total risk or by permissions set. The filtered lists of apps are stored in these tables. They represent the recommendations offered to the user.

A UML diagram of the database schema is shown in Figure 2.7.

Setup scripts have been written to create the MySQL table schema. Also, in order to register the changes made to the tables, the scripts have been split into migrations. It is possible to migrate backwards or forwards by either applying the “up” script (forward) or “down” script (backward).

2.5 Further research questions

Answering the research question described in Section 1.4 is possible by counting how many apps, permissions or topics satisfy the conditions of the question. For example, finding the affinity of a permission towards other permissions starts with counting how many times does an app request both permissions. These counts are then used to produce figures such as box plots (for showcasing distributions) or histograms.

Correlation questions are answered using statistical tests, such as the χ^2 test or the Fischer test.

Both plotting and performing statistical tests is facilitated by Python libraries, such as `matplotlib` [17] and `scipy` [27].

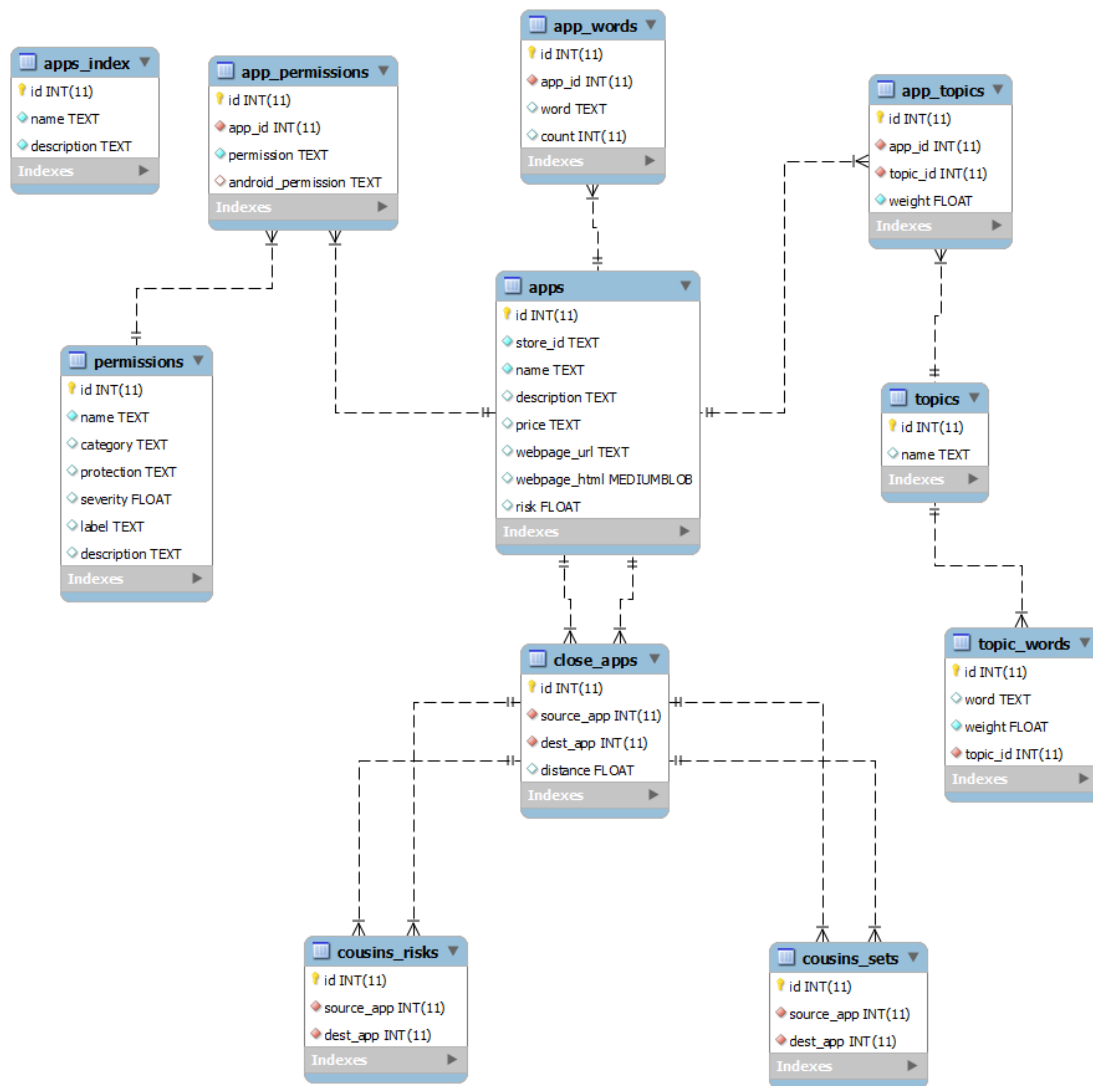


Figure 2.7: UML diagram of the SQL schema.

Chapter 3

Evaluation

3.1 Data collection (*Crawler*, *Word extractor* and *Permission mapper*)

The data collection system is organized in individual components that can be run as separate processes. Using MySQL as central data store and Redis as synchronization server enables the user to start the processes on different machines, which communicate via these two central services.

Initially, the processing queue of the *Crawler* component must be populated with Amazon “index” pages. The system is designed to require no other user intervention except the initial seeding of the *Crawler*’s processing queue and starting up the component instances. When all the data has been processed, the user can query the system using the *Query interface*.

The components log errors and other events to a local `syslog` [12] instance. In the scenario where the system is run on several different machines, it is desirable to use a centralized `syslog` server, which facilitates log analysis. `syslog` must be configured to accept data from `LOG_LOCAL0` and to store the log messages to disk. Components implemented in Python use the built-in `logging` module, whereas the NodeJS components use the `winston` [32] package from the `npm` repositories.

In the following paragraphs, the performance of each component will be evaluated, from relevant point of views.

Crawler: When Amazon is not throttling the *Crawlers*, the average crawling speed is of 1 app/s (given by 10 instances being limited at 1 request at every 10 seconds). The speed is influenced by network-related factors, but most importantly by the cool-off period used between requests (used by the “Delayer” module). Whenever the crawling of an app page fails to deliver essential data (e.g. the app name or description), the app is added at the back of the queue for reprocessing. Therefore, no apps missing essential information are added to the database. If a *Crawler* instance failed, the only lost information would be the URL to the app’s store page (i.e. the app will never appear in the database). However, the impact of this event is minimal, since the system is expected to store data for a large number of apps.

Word extractor: The *Word extractor* instances filter words which are negatively affecting the quality of the topics of *LDA*. Therefore, the filtering rules directly impact the output of *LDA*, and hence the recommendation system. The documentation of the `gensim` implementation of *LDA* recommends to trim the bags-of-words for the most frequent and most infrequent words [8]. Therefore, filtering the most common English words and other short words entails better results for the *LDA* component, as could be seen by several experiments with the filtering methods.

Permission mapper: Mapping the permissions revealed that approximately 6.3% of the permission descriptions cannot be identified. Some of the permissions are Amazon-specific (e.g. `com.amazon.STORE_ACCESS`), while others are defined by individual apps. However, these permissions should not have a significant impact on the other results of this study, given the fact that they represent such a small fraction of the total number of permissions.

3.2 Quality of data, relevance

The crawling run resulted in nearly 24,000 apps being recorded in the database, out of nearly a quarter of a million apps available in the Amazon Android Appstore. Although the crawled apps represent about 10% of the available data, the crawling was stopped because most of the apps served by Amazon were duplicates (apps already existing in the database).

Amazon does not provide information regarding the number of downloads per app. However, the index pages are by default sorted by the “New and popular” criterion. Therefore, crawling the list from the beginning should provide the most popular apps, and thus the data set is relevant for the analysis.

Amazon Android Appstore is mainly used by Amazon Kindle devices, which benefit from less popularity than other Android devices [15]. Using the Google Play store would also provide access to more apps, although the store might also be subject to repeating apps on index pages. The only information regarding apps which Google Play provides and Amazon Android Appstore does not is the (approximate) number of downloads; however, this data is not used in answering the research questions of this thesis.

3.3 Efficiency of LDA

LDA is designed to analyze and model efficiently a number of documents in the order of millions. Gensim provides the opportunity to process the same document corpus in several passes, to emulate a higher number of documents [6]. Therefore, 200 passes were performed through the dataset, resulting in nearly five million (virtual) documents being modeled.

The recommendation system defines the similarity (or “distance”) between app as a function of the topic weights for the app. Therefore, the LDA algorithm has a direct impact on the quality of the recommendations (i.e. how related are the recommended apps to the query app).

In order to measure the efficiency of the *LDA* component and of the distance function employed in the *Close apps* component, the following experiment was performed:

1. A sample of 50 pairs of apps was taken from the pool of 24,000 available apps. The pairs consisted of a random app and the closest app that could be found in the database, given the distance function and the topic weights.
2. The apps were listed as pairs of descriptions. A user read the descriptions and rated the similarity of these descriptions on a scale from 1 to 4 (where 1 is “not similar”, and 4 is “very similar”).

The results show that 48% of the pairs received a score of at least 3, while 30% of the apps received a perfect score of 4.

3.4 Efficiency of the recommendation system

The close apps are filtered such that the user observes only more secure recommendations. The apps which are not filtered away are called “cousins”. The filtered list of apps should be presented to the user ordered by distance, and therefore the first recommended app is called the “closest cousin”.

The filtering based on permission sets is straightforward, since it does not depend on the users' personal assessment of risks delivered by each permission. The efficiency of this filtering method can be quantified by the number of permissions that the user can “save”, which is defined as the difference between the number of permissions that the query app and its closest cousin have. Figure 3.1 provides a histogram of the distribution of the number of “saved” permissions. The distribution has a mean of 3.78, and a mode of 1.

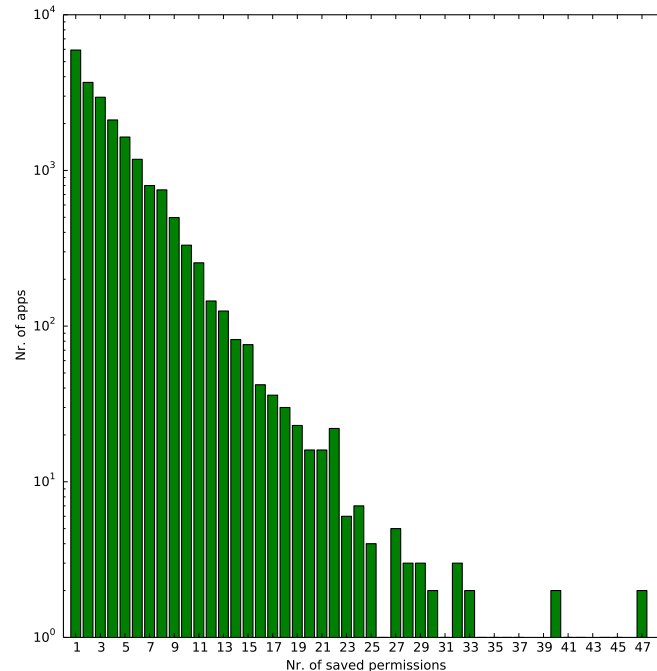


Figure 3.1: Histogram illustrating the distribution of the number of permissions saved using the set-based filtering.

A different way to visualize how many permissions can be saved is by plotting the average number of permissions of the closest cousin against the number of permissions of the original app (see Figure 3.2).

The “risk” of an app, as defined in Section 2.3, can be used in defining how much safer is a recommended app than the query app. For this purpose, the difference between the two risk values is called “saved risk”, and its distribution is depicted in Figure 3.3b. The distribution has a mean of 154, and a mode of 5. For comparison, Figure 3.3a depicts a histogram of app risks (mean 316, mode 88, 870 apps with 0 risk).

Similarly to Figure 3.2, the average risk of the closest cousin can be plotted against the risk of the original app (see Figure 3.4). Since risk is not a discrete quantity, a line plot with standard deviation bands is used.

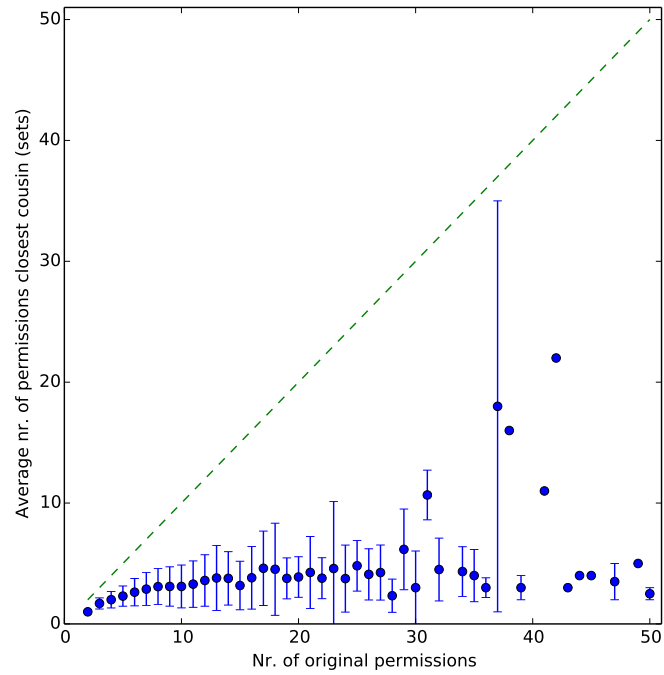


Figure 3.2: Average number of permissions of the closest cousin plotted against the number of permissions of the average app. For comparison, the dotted line has slope 1.

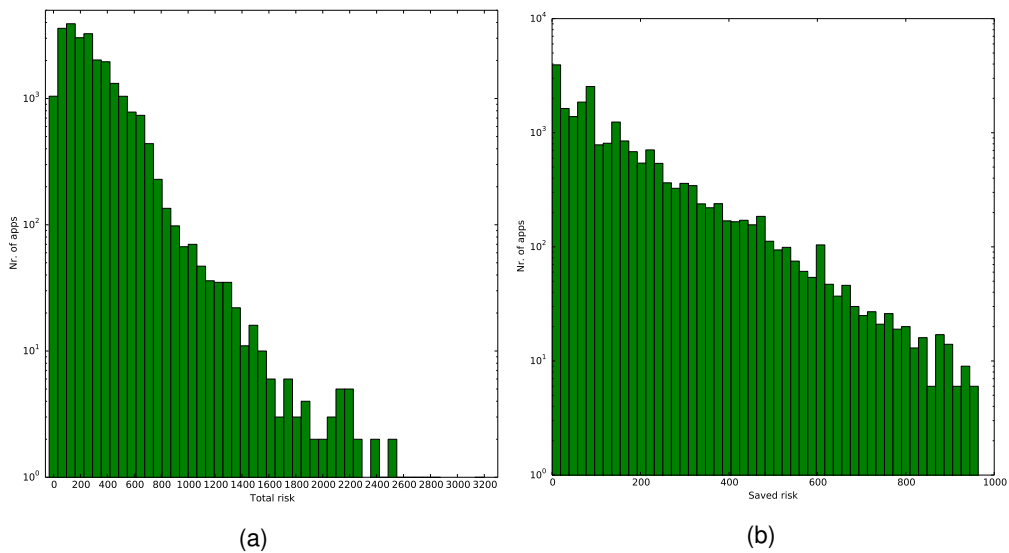


Figure 3.3: (a) Histogram illustrating the distribution of risks of the apps. (b) Histogram illustrating the distribution of app risk saved using risk-based filtering.

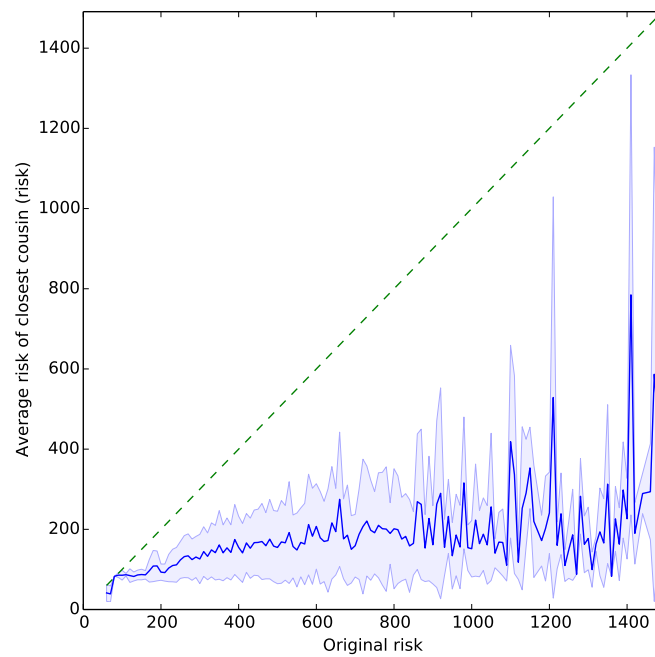


Figure 3.4: Average risk of the closest cousin plotted against the risk of the original application. For reference, the dotted line has slope 1. x-axis risks are grouped between every tenth integer, averaged, and used as a point on the graph. The blue band represents one standard deviation of the averaged group.

Chapter 4

Results

4.1 Topics

The LDA algorithm outputs both topic weights (indicating the strength of association between an app and a topic), and word weights (indicating the strength of association between a word and a topic). The number of topics that LDA models is therefore influencing the word and topic weights, since a lower number of topics lump different apps together, and a higher number of topics might split similar apps apart.

Identifying the real-world class of apps which is modeled by a topic (for example, “productivity apps” or “sport simulator games”) is hard without human intervention. Upon inspecting the results that the LDA component produced, the topics were re-labeled as depicted in Table 4.1. For an app, the topic with the highest weight is called the “topic of the app”; for each topic, the number of such apps is counted and listed in the column “Topic apps” of the table.

Topic id	Topic name	Total weight	Topic apps
201	GPS, maps, (city) guides, dating	1048.92	996
202	Virtual friends, picture games	773.66	656
203	Flappy bird clones and other games	651.72	579
204	Image, email, social networking apps	2166.66	2499
205	Puzzle games	780.77	696
206	Magazines, how-to apps	872.97	755
207	How to draw, radio apps, DINOSAURS	545.73	377
208	Lifestyle, pets, coupon and product apps	758.07	711
209	Babies, kids, programming	1167.29	1396
210	Girl games	812.81	947
211	Slots and other betting games	820.06	793
212	Mahjong, Zuma, ball games	2492.00	3056
213	Hidden object games	668.51	418
214	Misc. games	1121.33	990
215	Recipes, sound generators, how to draw	887.55	850
216	Word and logic games, BMX, simulators	590.11	522
217	Movies, lyrics, news	1747.70	2102
218	Children books, coloring	711.02	719
219	Bible apps, language apps, file explorers	500.18	364
220	Racing, zombie and combat games	1798.47	2503

Table 4.1: Listing of topics produced by LDA. “Topic name” contains the labels applied by human inspection of apps related to the topic. “Total weight” contains the sum of all the topic weights of apps for that topic. “Topic apps” contains the number of apps whose highest topic weight is associated to that topic.

The total weights are plotted in Figure 4.1a. Also, the number of topic apps for each topic is

plotted in Figure 4.1b. Finally, the total weights for the most important 20 words is plotted in Figure 4.2.

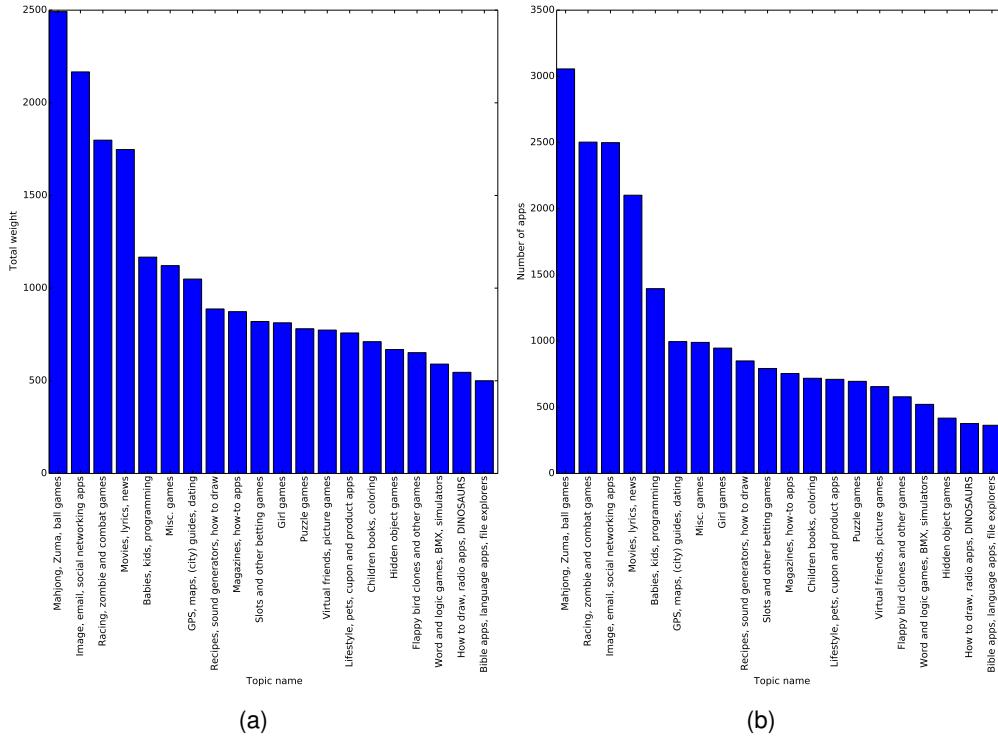


Figure 4.1: (a) Sum of weights all apps have for each topic. (b) Number of apps having the biggest weight corresponding to each topic.

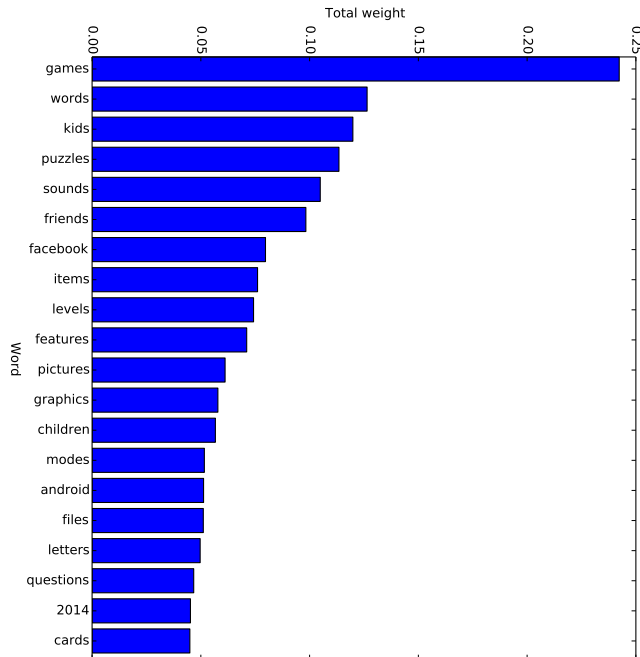


Figure 4.2: Sum of weights produced by LDA for each word. Only words with top 20 weights are included.

4.2 Permission distribution

Figure 4.3 displays the distribution of number of permissions per app. The mean number of permissions required is 5.7, whereas the mode of the distribution is 3. There are 870 apps which require no permissions.

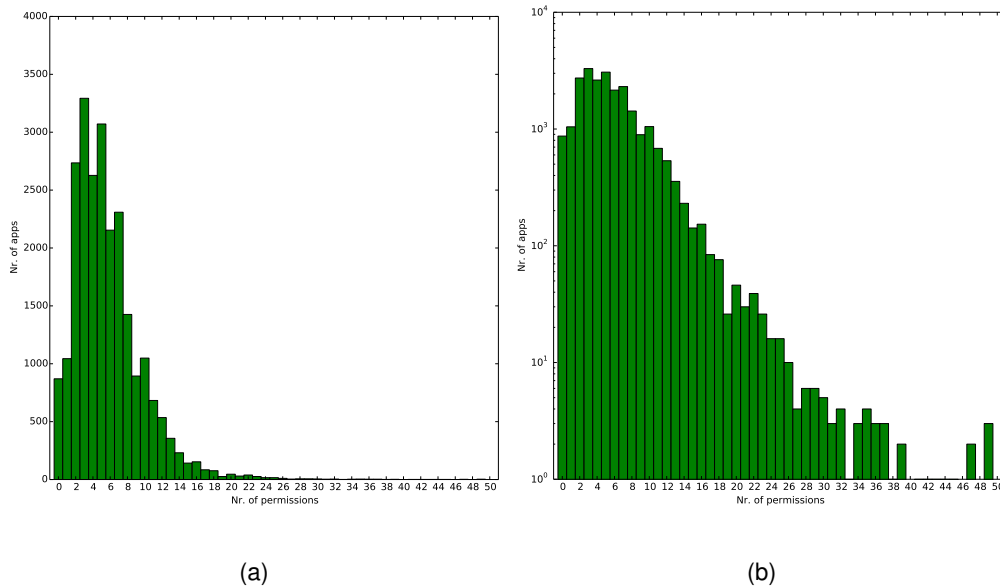


Figure 4.3: Histograms of the distribution of number of permissions requested by apps. Figure (b) has a logarithmic y-axis.

The “popularity” of a permission can be measured by the number of apps which request it, or by the percentage of apps which request it. Therefore, the most popular permission is `android.permission.INTERNET`, which allows an app to open network sockets. The most popular 20 permissions are shown in Figure 4.4.

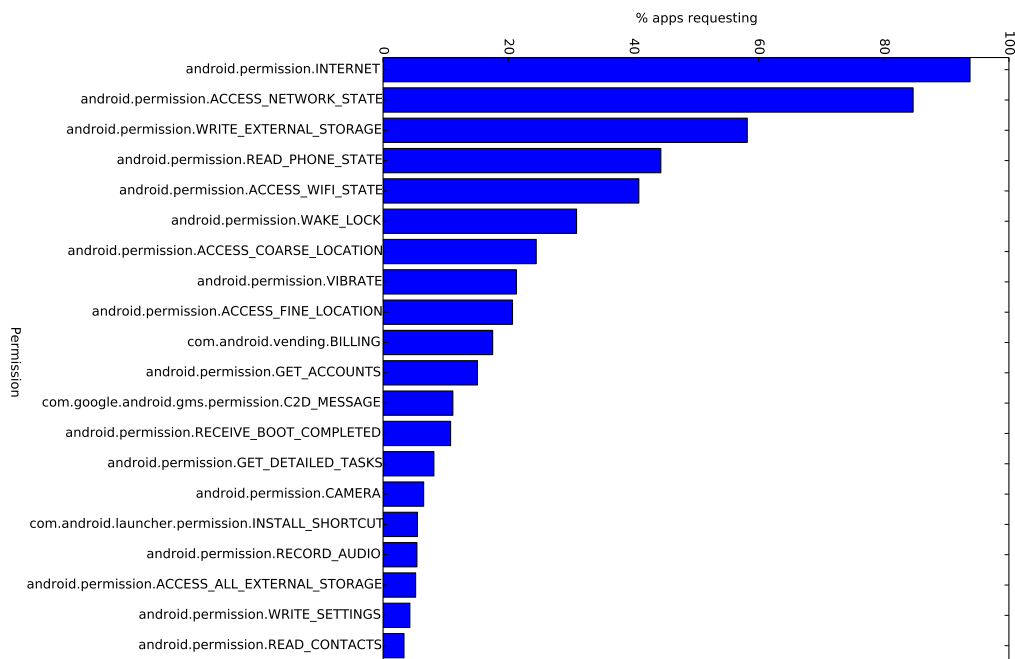


Figure 4.4: Most popular 20 permissions, by percentage of apps in the data set requesting them.

4.3 Further research questions

Correlation between app cost and in-app purchases The χ^2 test is commonly used for independence tests. The test resulted in several topics yielding p -values lower than the threshold of 0.05. However, since the data set was split into 20 separate “groups” (topics), the threshold should also be divided by 20 (i.e. the probability of observing the effect by mere chance should include the probability of randomly picking a sample of our “group”, which is $\frac{1}{20}$).

A particular topic in this case is “Bible apps, language apps and file explorers” which returned $p < 5 \times 10^{-4}$, and a negative correlation (i.e. free apps are not correlated to in-app purchases).

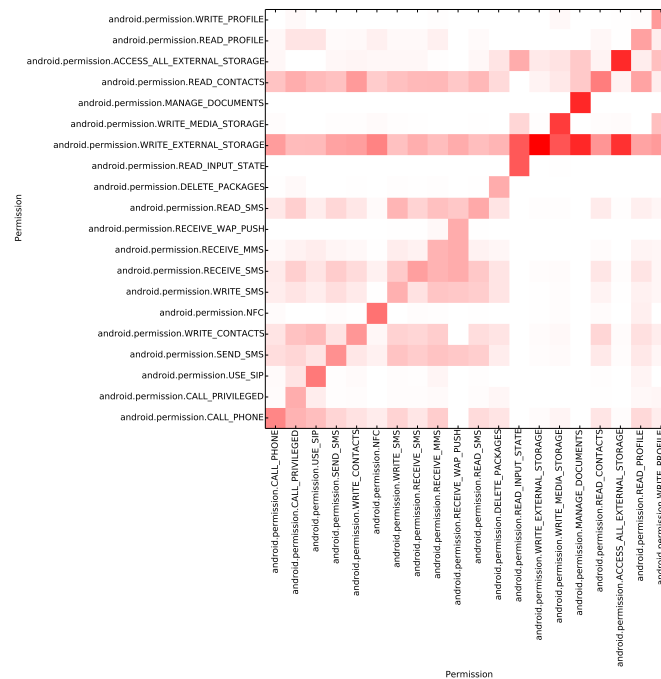
In order to address the issue of too few samples, an F-test was also performed on the data. The results were similar to the χ^2 test: only “Bible apps, language apps and file explorers” ($p < 5 \times 10^{-4}$) and “Movies, lyrics, news” ($p < 2 \times 10^{-3}$) topics scored below the threshold, both suggesting a negative correlation.

Permission-permission correlation The number of occurrences of a pair of permissions for an app is depicted in Figure 4.5a. The permissions are sorted in descending order of popularity, as presented in Section 4.2. Figure 4.5b presents the same correlation, where the occurrences are normalized for each permission on the x-axis; however, the permissions are sorted by associated risk. In both cases, only the top 20 permissions are included.

Permission-topic correlation Similarly to the permission-permission correlation, Figure 4.6a presents the number of occurrences of a permission-topic pair, whereas Figure 4.6b presents the counts normalized per-permission. The former figure contains the top 20 most popular permissions, whereas the latter contains the top 20 most risky permissions.

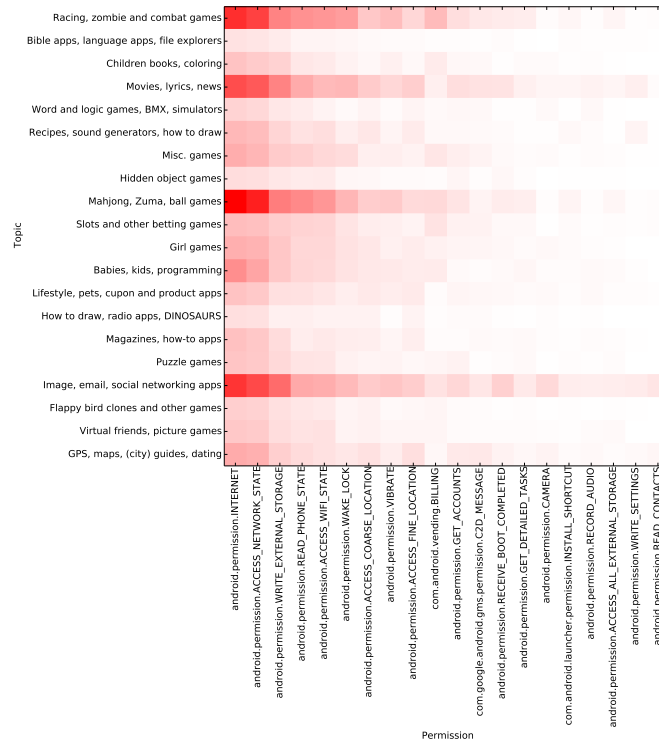


(a)



(b)

Figure 4.5: (a) 2D histogram of the number of occurrences of each pair of permissions; only top 20 permissions by popularity are included. (b) 2D histogram of number of occurrences of each pair of permissions, normalized per permission column; only top 20 permissions by risk are included.



(a)



(b)

Figure 4.6: (a) 2D histogram of number of occurrences of each topic-permission pair; only top 20 permissions by popularity are included. (b) 2D histogram of number of occurrences of each topic-permission pair, normalized per permission column; only top 20 permissions by risk are included.

Chapter 5

Discussion

5.1 Recommendation system

An important observation regarding the reported results is that they characterize a snapshot of the data gathered from Amazon. More exactly, should the user want to reproduce this study, several of the results would be different, mainly because the Amazon Appstore is a dynamic website, with new apps being published or app descriptions being updated. On the other hand, the user could decide to use the compressed HTML source-codes of the webpages; while this method eliminates the changes which appeared online, the routine for storing this data contained a bug at the moment of data collection, and this study does not benefit from the feature. The bug has been fixed, and a subsequent run of the whole system would make it possible to reproduce the study based on the new HTML data present in the database.

Speed of the system: The *Crawler* component can gather app data at an average speed of 1 app per second. This speed is forced by the "Delayer" module, and hence it is under the control of the user. However, the "Delayer" was implemented to circumvent access limitations imposed by Amazon. An improved speed should be recorded if the *Crawler* instances were run on different machines (such that Amazon observes different IPs); however, a better understanding of the throttling algorithms used by Amazon would reveal the best access strategy. The other speed bottleneck is the *LDA* component, which needs all the app descriptions in order to produce a topic model; however, *gensim* provides the possibility to add documents to the model online, which could be translated in adding app descriptions to the model as they are processed by the *Word extractor* component. Making use of this online functionality of *gensim* also requires implementing special logic for creating snapshots of the model periodically in order to address the possibility that in a distributed environment nodes can fail. Furthermore, periodic snapshots require a way of keeping track which apps have been added to the most recent saved model, because otherwise any apps crawled after making a snapshot would be ignored if the *LDA* node failed and a snapshot were restored.

Amount of data: Crawling the Appstore for a longer period should increase the number of apps which are stored in the database. However, due to the way Amazon implemented its "index pages", fewer new apps are displayed as the user browses the Appstore. A different crawling strategy, which visits the recommended apps for each app, might solve the problem of the "index pages" method (this strategy is similar to performing BFS using the "recommended apps" as links). However, this strategy depends on the initial seed (the app or apps where the crawling starts), because some apps might never be visited since they do not appear as recommended for any app. Therefore, a hybrid strategy is likely to provide the best results.

Relevance of data: Even if the whole catalog of Amazon were crawled, the number of apps published on Amazon is much lower than Google's Play Store. While more data would trans-

late in both better quality of the LDA model and in a higher probability that the recommended apps are satisfactory, the 24,000 apps currently analyzed represent the most popular apps on Amazon, which are therefore more likely to be downloaded by the user.

The relative popularity of apps varies over time. Crawling the Appstore once provides the opportunity to research phenomena between apps and permissions, but the system should periodically crawl for new, fresh data, such it stays relevant to the users.

LDA model: Choosing 20 topics to be modeled by LDA provides a good balance between fragmentation of similar apps and creating large topics which contain different apps. Evaluating the adequacy of this number is made by sampling apps from each topic and judging if they are similar. This strategy, although subjective, provides the most reliable decision without knowledge of the ground truth (i.e. what category each app belongs to). Also, the experiment described in Section 3.3 resulted in a high fraction of the pairs being similar. This result is a strong proof that the LDA model is facilitating good recommendations.

Several of the topics produced by LDA are labeled as games (e.g. “Mahjong, Zuma, ball games” or “Racing, zombie and combat games”; see Table 4.1). However, this can also be observed in the way app stores organize the apps: for example, Google Play Store treats “apps” and “games” as top-level categories, with both of them further split into functional sub-categories. Furthermore, splitting the games into several topics allows the user to find an app which is related to her query, not totally unrelated games. On the other hand, Table 4.1 contains topics which apparently group together unrelated apps (e.g. “How to draw, radio apps, dinosaurs”); however, the advantage of LDA is that an app (document) is not characterized by a single topic-weight, but by weights for all the topics. In combination with the angular distance function used in this implementation, apps which have the largest weight in the same topic are not necessarily close together – the distance depends on the complete mixture of topics, and therefore the recommendations are expected to be reliable.

It is interesting to observe that the most popular topics contain app types including Mahjong and Zuma clones, zombie and racing games, and social apps. This finding can be correlated to the popularity of some representative apps (e.g. Facebook and Twitter, the original Mahjong and Zuma games), which encourage developers to implement similar apps, expecting users to prefer familiar themes. The fact that the distributions of total weights, and of the number of apps which have highest weight for a certain topic (Figure 4.1), are not uniform is also normal, and attributed to the different popularity each different type of app has.

Distance function: The similarity between two apps, and consequentially the recommendations, are depending heavily on the choice of the distance function. However, whether one app is “more similar” than another app depends on the personal perception of each user. It is expected that the user of the recommendation system would browse through the first couple of suggestions and pick the app which best suits her needs. The experiment described in Section 3.3 proves also that the distance function implemented in this system is adequate. Also, the distance function can be analyzed for the following two cases:

- Two apps have a high weight for the same topic: in this case, the high weight produce a high dot product between the two vectors, which results in a low \arccos (angle).
- Two apps have a high weight two different topics: in this case, the angle between the two apps will be high, since the dot product is small.

Since the two apps belonging to the same topic are more likely to be similar than the two apps belonging to different topics, the distance measure is satisfactory. Furthermore, one can also observe that this distance function allows apps belonging to the same topic (i.e. with highest weight for the same topic) to be functionally segmented, based on the mixture of weights they have for other topics: for example, two apps with the highest weight for topic t_1 , but with the second-to-largest weight for different topics t_{2a} and t_{2b} (with $t_{2a} \neq t_{2b}$) would not be as close together as two apps having similar highest-weight topics ($t_{2c} = t_{2d}$).

Filtering method: It is hard to decide which filtering method (set-based or risk-based) produces better results. For example, one user might use the recommendation system to find similar apps which do not use a particular permission (in which case the set-based filtering is more appropriate), whereas another user might look for an overall less risky app (so she should use the risk-based filtering). However, since the user is considered a subjective agent whose risk-adversity is not known, the risk evaluation of an app might be perceived as wrong; therefore, the risk-based filtering might exclude similar apps which would suit some users.

Permission risk evaluation: As stated in the previous paragraph, the risk-adversity function is unknown for every user of the system. As presented in [10], the users can be surveyed in order to construct a model of the average user and his sensitivity toward different risks. The cited work stays as a basis for evaluating each individual permission's risk value. The estimation procedure for unknown values by evaluating each permission against permissions which control similar data provides adequate results, with a plausible risk hierarchy of the permissions, where cost-incurring permissions (e.g. `CALL_PRIVILEGED`) stay at the top, and permissions which could only annoy the user (e.g. `VIBRATE`) stay at the bottom.

Users might also challenge the definition of the risk of an app (see Section 2.3). Summing the individual permission risks might not capture the risks at which the users are exposed by some apps: for example, a hypothetical app which requires `BLUETOOTH` and `BLUETOOTH_ADMIN` (which are usually requested together anyway) and also `FLASHLIGHT` would have a risk higher than an app which can send SMS messages; however such an app clearly can do less damage than the SMS-sending app. An alternative way to define the risk of app a with set of permissions S_a is $risk(a) = \max_{p \in S_a} risk(p)$. This definition only takes into account the maximum impact that the app can have. However, this definition also fails to identify “permission synergies” (combinations of permissions which have little impact individually, but which pose a higher risk when requested together) [9].

Saved permissions and risk: The “number of permissions saved” and “risk saved” were described in Section 3.4. They can be used as measures of the efficiency of the overall system. However, these measures also depend on the way they are defined: for example, the definition from Section 3.4 involved the closest cousin, but the measures can also be defined in such a way that they are maximized (e.g. the “saved risk” becomes the difference between the risk of the query app and the lowest risk of the recommended apps). These alternative definitions actually hide the fact that the recommended apps might not be similar to the query (i.e. more importance is given to maximization of the “saving” rather than to the similarity of the apps), and therefore should not be used in characterizing the system.

Figure 3.1 reveals a long tail, with some recommended apps saving up to 47 permissions. For example, app “ooVoo Video Call, Text & Voice” (store ID `B007BSPOG4`) requests 36 permissions, whereas its closest cousin “ChatZee” (store ID `B00F560MP2`) requests only 4; both apps implement chatting functions, and therefore the recommendation is appropriate. On the other hand, the authors of the former app advertise more functions than the latter (e.g. video and photo messaging), which is a good reason for requesting more permissions.

Another relevant example is the pair “AVG Antivirus Security - FREE” (`B0089XH38M`, 49 permissions) and “Android Trust Antivirus RW” (`B00I1H8BIC`, 5 permissions). Even if the apps seem to be related judging by their description (they advertise standard antivirus features), the recommended app is most probably disguised malware: the app does not require any permission which allows inspection of other apps, but it does require writing and reading SMS messages, connecting to the internet and reading the list of accounts, a combination of permissions likely to be used for nefarious purposes. In this case, even if the query app is “more dangerous” from both the risk, and set-based definitions, it is in fact the more appropriate app to choose. However, given the design of the recommendation system, it is impossible to avoid such corner cases using automated computation; curating the database of apps, for example by marking them as “dangerous” and/or excluding them from the cousin lists, is a possible solution. All antivirus apps represent a special category of apps: it is expected for them to require several

permissions, since their purpose is exactly to inspect other apps and prevent them from performing malicious jobs; therefore, many of the “permission-hungry” antiviruses contribute to the long tail of Figure 3.1 (other examples are “Kaspersky Internet Security for Android”, store ID B00HYZBX9Q, and “Lookout Security & Antivirus”, store ID B00AQ398AY).

It can also be observed that apps which are unrelated appear on the list of apps with most saved permissions. For example, “Security - Free” (B004U85D2S) and “Fast Facebook” (B00IJ70T48); however, even though these two apps provide different functions, their descriptions contain common words (e.g. “memory”, “network”, “install”), which influence the allocation of topic weights. It is therefore reasonable to find functionally unrelated apps as close cousins after the automated processing of the dataset; users’ input could be used to curate the recommendations, which would help mitigating the problem.

In Figure 3.2, the point with x-coordinate 37 features a high standard deviation. The point corresponds to apps which require 37 permissions, and to their closest cousins filtered by permission sets. There are 3 apps in the list (“textPlus Gold Free Text + Calls for Android Phones, Tablets + Kindle Fire”, store ID B004ZFK600, “Security - Complete”, store ID B007CJTIO6, and “Lookout Security & Antivirus”, store ID B004QEFQIM), but only 2 of these apps have a close cousin with fewer permissions. Therefore, the point with x-coordinate 37 corresponds to a set of two close cousins which require 1 and 35 permissions; this set has a standard deviation of 17 and a mean of 18. Therefore, the apparent glitch is related to a low number of samples.

Figures 4.3 and 3.3a reveal that the number of apps which require many permissions or which have a high total risk decreases rapidly (nearly exponentially, with a long tail). This fact reduces the likelihood that a high number of permissions are saved, or that a high risk value is saved by using the recommendation system (these values were plotted in Figures 3.1 and 3.3b): given the fact that the minimum number of permissions / risk of an app is 0, an upper bound of the “saved permissions” or “saved risk” is the number of permissions or risk of the query app; therefore, having a low number of apps with a lot of permissions or with high total risk translates into a lower likelihood to “save” a high number of permissions or risk.

Figures 3.2 and 3.4 might seem to contradict Figures 3.1 and 3.3b (their histogram counterparts): it seems that the average number of permissions / average risk of the closest cousin deviates much from the $x = y$ line. However, the fact that fewer apps require many permissions / have high risk results in averaging of smaller lists, and therefore a higher probability that the mean values are low: for example, there are many apps which require few permissions, so the few apps requiring many permissions are more likely to have their closest-cousins among the former set.

The distribution of “saved permissions” has a mean of 3.78, which means that a user can expect to get a recommendation with 3 or 4 permissions less in average. Furthermore, the distribution of “saved risks” has a mean of 154 (for comparison, the risk of an individual permission is a value between 0 and 100). It is worth re-emphasizing that these values describe the case when the user considers only the first recommendation provided by the system (most similar app); it is however likely to find an even better alternative simply by considering more recommendations from the list. On the other hand, one could argue that apps which do not require more permissions than strictly needed are likely to have no similar apps, which require the same set of functionalities and require less permissions. Therefore, the possibility of saving in average 3 to 4 permissions, and up to 47 permissions in extreme cases, offers a significant benefit to the user.

Permission distribution: Figure 4.3 shows the distribution of the number of permissions requested by the apps, which is in concordance with the literature [31]. It is interesting to note that about 3.5% of the apps do not require any permission at all. The list of apps contains several toddler games (e.g. “Trains, Planes & Sea Vehicles - Puzzle for Toddlers”, store ID B008QMMEOE) or calculators (e.g. “NeoCal Advanced Calculator”, B004S3MB2U). While it is expected that calculators would not require any permissions, the authors of young children games could take advantage of the fact that in-app purchasing libraries offered by an app store are tied to the user’s account, and therefore a young child could be tricked into making purchases without the need of physical access to a credit card.

Other apps requiring no permissions might fail to provide the advertised functionality: for example, “Onion Comic Viewer” (B0055184B6), should open comic files in different formats; it is however unclear how could such an app access the files since it does not require the permissions needed to access external storage or the internet (such that it can download the files for the comics). By default, any app has access to the “internal storage”, which is a folder created specially for the app, and not accessible from other apps, and to the app cache, which is also inaccessible from another app. Installing the app revealed that there is no easy way to explore the files, and that the app has no access to the places where files are usually placed by users (for example, the `sdcard` folder). The app could list the Android filesystem and access some folders, but the user would need other apps with special permissions to place files in these folders. To further analyze the behavior of the app, the utility `APK Tool` [5] was used to decompile the APK file of the app. The tool produces source code similar to assembler code, called “smali”; relevant smali commands include `invoke-direct`, `invoke-virtual` and `invoke-interface`, which represent function calls. Listing all “`invoke-*`” commands revealed no suspicious function calls.

The Trendmicro blog describes how an app which requires no permissions can still perform malicious actions using different techniques [24]. An interesting technique mentioned in the article involves creating Android `Intents` (one of the platform’s inter-process communication channels) to invoke the default browser and navigate to a webpage. The app does not need any permission to invoke the `Intent`, and the browser already has the permission to access the internet. Decompiling the “Onion Comic Viewer” app revealed that the app is only invoking `Intents` that allow the user to switch between the two `Activities` of the app, which pose no security threat. In conclusion, no hidden, malicious behavior could be found implemented in the app.

The most popular permission is `android.permission.INTERNET` (Figure 4.4), requested by 93.8% of the apps. This permission allows opening network sockets, which in turn allows the app to retrieve updated content to the user. However, the internet connection can be used also for malicious purposes, such as uploading sensitive user data to the attacker’s servers. Therefore, the presence of this permission in the set of requested permissions of an app which does not need content from the internet might be a warning sign. It should be noted that apps require an internet connection also for serving ads.

The second most popular permission is `ACCESS_NETWORK_STATE` (84.7% of the apps), which allows the app to retrieve information about the network (e.g. if the device is connected). This information is usually used prior to making a request to a server. Since `INTERNET` is the most popular permission, it is reasonable to have such a high number of apps requesting `ACCESS_NETWORK_STATE`. The third permission on the list is `WRITE_EXTERNAL_STORAGE` (58.2% of the apps), which allows the app to write files to the external storage.

It is also interesting to see how many apps require the most dangerous permissions. For example, the `BRICK` permission (which would allow the app to disable the device, rendering it inoperable), is not required by any app. The `CALL_PRIVILEGED` permission (which allows the apps to place a phone call without going through the Dialer app, so the user can confirm the call) is required only by 12 apps, all of which advertise functions such as internet calling, but also placing regular phone calls if no internet connection is available (hence justifying the need for the permission). Since there are not many apps who require the permission, human curation of the database is possible, maybe even preferred: periodically, an administrator of the system could test the apps (e.g. using static analysis) and determine whether the apps contain malware or not. However, as the number of apps to manually analyze rises, the solution becomes less feasible and an automated approach should be implemented.

5.2 Further research questions

Correlation between app cost and in-app purchases Since an app developer faces the problem of a low (or lack of) reputation within the user-base, publishing an app for free and subsequently providing restricted content for a fee (i.e. in-app purchases) would incentivize

users to install the app. Another popular alternative is splitting the app functionality between a base (“lite”) version, and a more advanced (“pro”) version. While the former method is popular with game developers, the latter is more frequently found in utility apps (e.g. file explorers, music players).

The tests prove that there is no correlation between app cost and in-app purchases for most of the app categories (topics). The evidence of negative correlation for “Bible, language and file explorers” is expected: the “Bible apps” are meant to have a massive reach (transfer information to many users), and therefore they need not incur any costs to the users; the “language apps” and the “file explorers” might find the “lite/pro” monetization model more appropriate, since in-app purchases are usually used to unlock content, not features.

The second topic with a significant proof of negative correlation is “Movies, lyrics, news”. The apps belonging to this category do consume content, but the nature of the content is different. Movies and lyrics apps resemble digital encyclopedias on the respective themes, and therefore partially restricting access to the data is unlikely. News apps are usually meant to attract users to a particular information source (e.g. a news agency) or to drive traffic to a website (e.g. Google’s “News and weather” app, which opens web articles in the default browser); these apps are also unsuitable for the in-app purchasing model.

The lack of correlation is unsurprising also for the apps belonging to game-themed topics. On one hand, the “in-app purchasing” variable is defined in terms of requesting the `com.android.vending.BILLING` permission, which is provided by the Google Play Services SDK; it is expected that apps published in the Amazon Appstore are specifically recompiled for the Amazon SDK. On the other hand, although the in-app purchasing method is effective, it might not fit the theme of the game, or even the developer’s desire to monetize its apps.

Permission-permission correlation Figure 4.5a offers supporting evidence for the argument made in Section 5.1, regarding the popularity of `android.permission.INTERNET` and `ACCESS_NETWORK_STATE`. It can be observed from the histogram that the two permissions are most frequently found together, since it is a good practice to test for the availability of an internet connection before making requests to a server.

The cells corresponding to `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION` show an increased likelihood to find these two permissions together. This result is expected, since using cell-based localization is a reasonable fallback to GPS for apps which require the user’s location. Similarly, the `ACCESS_WIFI_STATE` (which allows the app to get the ids of the WIFI networks in range, among others) is likely to be requested together with the location permissions, since WIFI networks have a fixed and usually known location and therefore can be used as “landmarks” in determining where is the device located.

The `CAMERA` permission is likely to be requested together with `RECORD_AUDIO`, which can be used in audio-video recording. `CAMERA` also is likely to appear in combination with the fine and coarse location permissions, since the photo-shooting apps usually offer the option to tag pictures with GPS coordinates, or general location.

Another likely pair of permissions is `com.google.android.gms.permission.C2D_MESSAGE` (which allows an app to receive notifications using Google Cloud Messaging – Google’s push notifications service) and `GET_ACCOUNTS` (which controls access to the accounts created by the apps installed on the device). These two permissions are likely to be used in social-networking apps, which usually implement chat functions. Furthermore, `C2D_MESSAGE` is also frequently requested in combination with `VIBRATE` and `WAKE_LOCK` (which allows the app to prevent the device to sleep), which are features usually provided by the chat apps.

Figure 4.5b offers the same type of likelihood analysis as Figure 4.5a, but for a different set of permissions (the most risky); also, the histogram is normalized by column: for each x-axis permission, the vector of occurrences of the y-axis permissions is normalized and then added to the plot. It therefore reveals not the pairs of permissions which are likely to be generally found together, but the permissions which are likely to be found together with each x-axis permission. For example, `ACCESS_ALL_EXTERNAL_STORAGE` is most likely to be found together with

`WRITE_EXTERNAL_STORAGE`, which is reasonable, given the fact that both permissions restrict access to the same functions of the device (namely, writing data to the SD card).

Some pairs of permissions expose the user to a great security risk. For example, the SMS-related permissions (`WRITE_SMS`, `RECEIVE_SMS` and `READ_SMS`) are likely to be found in pairs. Allowing an app to both write and receive SMS messages is usually used in device-authentication procedures (e.g. the app receives an SMS with a token, to which it replies with a computed message, completing the authentication); on the other hand, a malicious app with both write and receive SMS permissions could exploit the connection at the expense of the user (without her consent, or even knowledge).

Apart from the insights about which permissions are likely to be found together in the app manifest files, the histograms also provide insights on permissions which usually do not appear together. Of course, some pairs can be explained intuitively, for example `com.android.vending.BILLING` (which provides in-app purchasing capabilities) is not often requested by apps which also need permission to access location (Figure 4.5a). Another example would be the fact that `CALL_PHONE` does not necessarily require `CALL_PRIVILEGED`, whereas `CALL_PRIVILEGED` is likely to require `CALL_PHONE` (Figure 4.5b), which also reflects the difference between the two histograms. These insights can be used to identify unlikely patterns occurring in apps, and warn the user that the app might not be trusted.

Permission-topic correlation Figures 4.6a and 4.6b show the likelihood for a permission to be required by an app belonging to one of the topics. The histograms can also be interpreted as a per-topic segmentation of the popularity of the permissions (i.e. in which topics is the permission more popular).

It is interesting to observe that, generally, the “Image, email, social networking apps”, “Mahjong, Zuma, ball games”, “Movies, lyrics, news” and “Racing, zombie and combat games” have the highest probabilities (Figure 4.6a), which can be explained by the fact that the topics are more popular (Figure 4.1a).

The `BILLING` permission is most frequently requested by games (topic ids 211, 212, 214, 220; see Table 4.1), but also by apps targeted to children (topic ids 209, 218). The games are a reasonable target for in-app purchases, but the apps for children can become target to incur hidden costs to the users (usually, the parents).

The location-related permissions are distributed quite evenly among the topics. This could happen due to the fact that ad-serving services require the approximate location of the user in order to serve effective ads. Furthermore, one could expect that the GPS and mapping apps (topic 201) have the highest probability to request access to the user’s location.

Figure 4.6b displays the probability to find each permission in the topics, normalized across topics (not overall, as Figure 4.6a). This way to depict the data reveals several cells of interest.

First, the `READ_INPUT_STATE` permission appears to be requested most by the “Racing, zombie and combat games”. The permissions allows the app to listen for key presses and touch events, even when the user is using a different app. An attacker could use this permission for harvesting passwords. It is unclear why apps belonging to this gaming category would require the permission.

Second, the `WRITE_PROFILE` permission is strongly correlated to “Babies, kids and programming” topic. The permission enables editing the personal profile of the user of the device. The personal profile contains details such as the user’s name or photo. Furthermore, the `READ_PROFILE` permission appears to have a weak correlation to the topic. It is also unclear why this category of apps would require the permission. However, investigating the actions which an app performs, which require the permission, would be possible only by static analysis of the apps, which is outside the scope of this work.

Other combinations can be intuitively explained. For example, `CALL_PRIVILEGED` is required by GPS apps and city guides (topic 201), which can be explained by the fact that the apps provide the option to call the emergency services of the visited city (for city guides) or in case of accident (GPS apps).

The row corresponding to “Image, email, social networking apps” appears to have the highest weight overall in Figure 4.6b, but this can be explained by the fact that apps belonging to this category usually are permission hungry, asking for access to a lot of information. For example, the Facebook app (store ID `B0094BB4TW`) requires 31 permissions, including reading SMS messages.

It is troubling to discover that not only the apps belonging to topic 201 require the `CALL_PRIVILEGED` permission, but also apps belonging to the “Mahjong, Zuma, ball games” topic. The same can be said about the `DELETE_PACKAGES` permission (which allows an app to uninstall other apps) and the “Slots and other betting games” topic. Apps requesting these permissions and belonging to these topics should be avoided.

These histograms can not only be used in creating a warning system (which highlights unlikely combinations of permissions and topics in an installed app), but also to help a curator of the database to find strange combinations (as exposed above), and highlight the corresponding apps as not trusted.

Chapter 6

Threats to validity

The LDA algorithm is designed to process massive datasets, with millions of documents. The lack of data is compensated in this work by repeatedly processing the set of 24,000 apps in the database. This technique could have a negative impact on the quality of the topics produced by LDA.

The risk evaluation of each individual permission might be erroneous. However, it is clear that the notion of risk is different for each user, which makes an objective risk assessment impossible.

Some Amazon-specific permissions are not handled by the *Permission mapper* component. This also happens for some Google-specific permissions, and for any other permission which is unknown (e.g. permissions defined by apps). While these permissions might have a small impact due to their low frequency, some measures might be affected (e.g. the risk of an app does not include the risk of the permissions which are not mapped correctly, and therefore is lower than in reality).

The system is implemented to only crawl the Amazon Android Appstore. It was also presented how the data is analyzed for patterns of likely or unlikely combinations. Using a single data source might establish a dependency between the findings and the source (e.g. the frequency of Amazon-specific permissions has an exacerbated impact). The system should crawl several app stores, and consider an app published to different stores as two different apps (because they might request different permissions). This will mitigate the hidden effects specific to individual app stores and will provide a better overview of the Android app ecosystem.

Chapter 7

Related Work

Barrera et al present in [2] a method to employ Self-Organizing Maps (a way to produce 2D representations of a higher-dimensional space) in order to highlight app similarities with respect to the permissions they used. The authors discover that although Android provides many permissions to the developer, only a small number of them are actually used. In concordance with the findings resulted from this project, they reveal that `android.permission.INTERNET` is requested by a majority of apps, and suggest that some permissions (especially `INTERNET`) should be split into different permissions which provide finer-grained control to resources. Finally, they reveal that social and communication apps are requesting the most number of permissions in average (between 4.5 and 6.7), which also corresponds to the findings presented in this study.

Kirin is an app certification service introduced by Enck et al in [9]. The system uses security rules to identify potentially dangerous behavior in an app. The rules are defined on Android permissions. Alone, some of the permissions pose no security risk; however, the combinations of requested permissions might enable malware to exfiltrate sensitive data. The article uses as example the combination of starting the app on boot (`RECEIVE_BOOT_COMPLETE`), access to the user location and ability to access the internet, which signal a location-tracking app. The system was used to test apps, and successfully signaled potentially dangerous behavior which was not explained by the descriptions of the respective apps.

Over-privileged apps are those apps which request more permissions than they actually use to provide their functionality. Felt et al study this type of apps in [11], where they present “Stowaway” – a static analysis tool which maps the set of API calls made by an app to the permissions restricting access to these calls. Their system reveals that about one third of the dataset used in the study (940 apps) are actually overprivileged. While constructing “Stowaway” it was also found out that the `BRICK` permission is actually used in the Android 2.2 source code only in an unreachable area of code. The paper highlights common developer errors, such as requesting the permission needed to perform an action when in fact the app only issues an `Intent` which is handled by a different app (e.g. requesting `INTERNET` when opening an URL in the default browser). The authors attribute these errors on developer confusion due to the poor documentation of the Android APIs.

Felt et al also published in [10] the results of a survey they conducted on smartphone users. The survey measured the users’ concern (“how upset would they be”) regarding the risks associated to granting different permissions to the apps. Their findings reveal that the lowest-ranked risks include phone vibration or turning the flash on (correlated to Android permissions `VIBRATE`, respectively `FLASHLIGHT`), while the highest-ranked risks were related to deleting contacts (`WRITE_CONTACTS`), sending SMS (`SEND_SMS`) or make phone calls (`CALL_PHONE`, `CALL_PRIVILEGED`) to costly numbers. As recommended in the article, the results were used as a guide to evaluating the severity of different Android permissions.

A quantitative risk assessment of permissions was performed by Wang et al in [31], which resulted in the definition of “risk of an app” as the sum of individual risks of the permissions it requests. The authors split their data-set into benign apps and malign apps (mal-

ware). The article reveals that the most requested permission is `INTERNET` (for both sets); malware apps have a preference towards changing device settings (e.g. `WRITE_APN_SETTINGS`, `CHANGE_WIFI_STATE`) and accessing cost-incurring services (SMS and call related permissions). It is also revealed that the distribution of number of requested permissions has an affinity for higher values in malware. The results of the quantitative risk assessment show that the highest-risk permissions include `WRITE_SMS`, `SEND_SMS` and `DELETE_PACKAGE`. Their software, “DroidRisk”, can order the apps by risk and for each app identify the most risky permissions.

In [28], Shabtai et al provide a risk analysis of the Android framework. They identify the threats with highest risk: abusing cost-incurring services and functions, such as sending SMS/MMS or placing phone calls. The authors propose a set of risk-mitigating strategies, such as installing anti-malware or firewall software. They also propose solutions connected to the Android permission system: allowing the user to selectively grant permissions to the app (as opposed to granting all requested permissions), or the implementation of an app which scans the to-be-installed app for requested permissions and present a detailed report of what the user would agree to install.

Chapter 8

Future Work

There are several ways to improve the accuracy of a recommendation system. Probably the most often implemented strategy is to integrate the user's feedback into the system. For example, the similarity of two apps can be evaluated using any scale (e.g. the 5-star scale); the user's input is used by the system to update the distance between the two apps. This strategy keeps the automated nature of the system, and depends on the efficiency of the update procedure (e.g. a one-star review should not change the distance between the apps completely). Unfortunately, this strategy is threatened by the hostility of the reviewers: for example, the author of a malicious app could try to remove any similarity between his app and all the recommendations the system generates.

In order to compensate for all the drawbacks that a fully automated system possesses, human curation should be implemented. For example, the administrators of the website could flag apps as "dangerous", which would remove them from the lists of recommended apps. Deciding which apps to inspect can be facilitated by the topic-permission or permission-permission histograms, as presented in Section 5.2. However, flagging an app as dangerous is prone to errors (legit apps could be flagged, or malicious apps could be missed).

The histograms can also be used to identify unlikely combinations of permissions or permissions requested by apps unlikely to belong to a topic. This information can be used to implement a warning system, running on the devices. Upon installation, the program could inspect the new app, classify it using the LDA model, and retrieve its permissions; this data would then be checked, and should the app require some intriguing permissions, the user would be notified.

The recommendation system can be improved by extending the app database. More data would not only allow LDA to produce better results, but would also increase the chances that a similar and more secure app is available to be recommended. Therefore, the *Crawler* component should be extended to also process Google Play Store. Also, to circumvent the problem that Amazon (and possibly also Google) start repeating apps on the index pages, an extended crawling strategy should be implemented. For example, the *Crawler* instances could follow the "related apps" links which the stores provide, in an algorithm resembling breadth-first search.

The stores also provide some more information which could be analyzed. For example, Google Play Store provides the approximate number of downloads of the app, which can be used to measure the app's popularity, and consequently in ordering the recommendations. Both Google's and Amazon's app stores provide user reviews for the apps, employing a 5-star scale; this information can be integrated in the "popularity" attribute of the app. Also, the "related apps" links on their websites are the result of the recommendation systems built by Amazon and Google; these links can be used as valuable hints of which apps are similar, and could be integrated in the distance function (e.g. if two apps are recommended on the app store, the distance should be low, even if the angle between the topic vectors suggests otherwise).

Finally, the system can be optimized, such that it processes the dataset faster. This improvement would become valuable as the *Crawler* instances retrieve more data. For example, Twitter describes in [33] a technique which computes all-pairs similarities using MapReduce. The tech-

nique can be implemented in the *Cousins* and *Close apps* components. To generalize, the distributed design of the system suggests that it can be modified to take advantage of MapReduce even further.

Chapter 9

Conclusions

A recommendation system has been built based on app data from the Amazon Android App-store. The system processes app descriptions and creates a topic model, which is used to compute the similarity between apps using the cosine distance function. Recommended apps are filtered based on the permission sets they request, or on the risk they pose to the user.

Given the distributed design of the system, several synchronization schemes have been explored. The versatility of redis has been exploited, by implementing different patterns including distributed mutexes, pub-sub systems and producer-consumer queues.

The system can provide recommendations which require in average 3 permissions less. It also helps lower the risk of the apps by about 154 risk-points (where the maximum risk a single permission can bear is 100). The assessment of these measures is based on the “most similar app” among the recommendations, but in practice the user can browse through the whole list of recommended apps and find a more suitable and safer app.

Using the data available, several research questions can be answered. No correlation could be found between the price of the app and the availability of in-app purchases overall. If each topic is treated separately, some significant negative correlations appear; however, they are not extremely relevant in the context of the respective topics.

Investigating the relationships between permission pairs requested by the same app revealed several interesting couples. None of these pairs could not be explained by common sense; however, this does not rule out the possibility of malicious use of the respective combinations. Furthermore, some risky pairs could be found (e.g. writing and reading SMS messages); apps falling in this category should be avoided.

The same investigation was performed for the frequency of all the topic-permission pairs. In this case, some unexplainable combinations were found (e.g. the permission allowing an app to listen to input which is given even to other apps is most frequently asked by a certain category of action games). The capability to reveal such suspicious cases stands proof to the benefits brought by the system. Furthermore, should human curation be added to the database, the histograms presented would provide a starting point for the curation process.

The system is designed to run on several machines, with little human interaction needed. The implementation is production-ready, and the database can be queried via the website interface. However, the project exposes some future work which could benefit from the current results, which would result in a warning system: upon installation, any app is be checked, and should an unlikely combination of permissions be met, the user would be warned and allowed to stop the installation process.

Chapter 10

References

- [1] *Android Security Overview*. Google. URL: <https://source.android.com/devices/tech/security/> (visited on Aug. 29, 2014).
- [2] David Barrera et al. “A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 73–84. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866317. URL: <http://doi.acm.org/10.1145/1866307.1866317>.
- [3] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. “Latent Dirichlet Allocation”. In: *J. Mach. Learn. Res.* 3 (Mar. 2003), pp. 993–1022. ISSN: 1532-4435. URL: <http://dl.acm.org/citation.cfm?id=944919.944937>.
- [6] Radim Řehůřek. *Creating a Corpus, Dictionary and running LDA etc.* 2012. URL: https://groups.google.com/d/msg/gensim/9ZwaITP_WfI/tTUXfpPkR04J (visited on Sept. 7, 2014).
- [8] Radim Řehůřek. *Is it a must to remove tokens once when building a LDA model?* 2013. URL: <https://groups.google.com/d/msg/gensim/cjgJottKGSi/vtX-PmtuSRQJ> (visited on Sept. 7, 2014).
- [9] William Enck, Machigar Ongtang, and Patrick McDaniel. “On Lightweight Mobile Phone Application Certification”. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: ACM, 2009, pp. 235–245. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653691. URL: <http://doi.acm.org/10.1145/1653662.1653691>.
- [10] Adrienne Porter Felt, Serge Egelman, and David Wagner. “I’ve Got 99 Problems, but Vibration Ain’t One: A Survey of Smartphone Users’ Concerns”. In: *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 33–44. ISBN: 978-1-4503-1666-8. DOI: 10.1145/2381934.2381943. URL: <http://doi.acm.org/10.1145/2381934.2381943>.
- [11] Adrienne Porter Felt et al. “Android Permissions Demystified”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: ACM, 2011, pp. 627–638. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046779. URL: <http://doi.acm.org/10.1145/2046707.2046779>.
- [13] Sheran Gunasekera. *Android Apps Security*. 1st. Berkely, CA, USA: Apress, 2012. ISBN: 1430240628, 9781430240624.
- [15] Mark Hoelzel and Emily Adler. *Amazon’s Kindle Fire Is Losing Share To Android Tablets, But The Device Is Still Succeeding At Earning Money From Users*. Business Insider. 2014. URL: <http://www.businessinsider.com/a-look-at-amazons-kindle-ecosystem-3-2014-2> (visited on Sept. 7, 2014).

- [23] Josh Ong. *Report: Android reached record 85Q2 2014, Xiaomi now fifth-largest vendor*. The Next Web. 2014. URL: <http://thenextweb.com/google/2014/07/31/android-reached-record-85-smartphone-market-share-q2-2014-report> (visited on Aug. 29, 2014).
- [24] Yinfeng Qiu. *Bypassing Android Permissions: What You Need to Know*. Trend Micro Inc. 2012. URL: <http://blog.trendmicro.com/trendlabs-security-intelligence/bypassing-android-permissions-what-you-need-to-know/> (visited on Sept. 7, 2014).
- [25] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011. ISBN: 1107015359, 9781107015357.
- [28] Asaf Shabtai et al. "Google Android: A State-of-the-Art Review of Security Mechanisms". In: *CoRR* abs/0912.5101 (2009).
- [29] *The Unix Philosophy: A Brief Introduction*. The Linux Information Project. 2006. URL: http://www.linfo.org/unix_philosophy.html (visited on Sept. 3, 2014).
- [30] David Verhasselt. *A Distributed Mutex and Semaphore using Redis*. 2011. URL: <http://www.davidverhasselt.com/2011/08/06/a-distributed-mutex-and-semaphore-using-redis/> (visited on Sept. 10, 2014).
- [31] Yang Wang et al. "Quantitative Security Risk Assessment of Android Permissions and Applications". In: *Proceedings of the 27th International Conference on Data and Applications Security and Privacy XXVII*. DBSec'13. Newark, NJ: Springer-Verlag, 2013, pp. 226–241. ISBN: 978-3-642-39255-9. DOI: 10.1007/978-3-642-39256-6_15. URL: http://dx.doi.org/10.1007/978-3-642-39256-6_15.
- [33] Reza Zadeh. *All-pairs similarity via DIMSUM*. Twitter. 2014. URL: <https://blog.twitter.com/2014/all-pairs-similarity-via-dimsum> (visited on Aug. 29, 2014).

Software

- [4] *Bootstrap*. Twitter. URL: <http://getbootstrap.com/> (visited on Sept. 4, 2014).
- [5] Brut.all. *android-apktool*. URL: <https://code.google.com/p/android-apktool/> (visited on Sept. 7, 2014).
- [7] Radim Řehůřek. *gensim*. URL: <http://radimrehurek.com/gensim/> (visited on Sept. 4, 2014).
- [12] Rainer Gerhards. *rsyslog*. URL: <http://www.rsyslog.com/> (visited on Sept. 7, 2014).
- [14] Ariya Hidayat. *PhantomJS*. URL: <http://phantomjs.org/> (visited on Sept. 3, 2014).
- [16] Elijah Insua. *jsdom*. URL: <https://github.com/tmpvar/jsdom> (visited on Sept. 3, 2014).
- [17] Eric Firing Michael Droettboom John Hunter Darren Dale. *matplotlib*. URL: <http://matplotlib.org/> (visited on Sept. 5, 2014).
- [18] *MongoDB*. MongoDB, Inc. URL: <http://www.mongodb.org/> (visited on Sept. 4, 2014).
- [19] Matthew Mueller. *cheerio*. URL: <http://cheeriojs.github.io/cheerio/> (visited on Sept. 3, 2014).
- [20] *MySQL*. Oracle Corporation. URL: <http://www.mysql.com/> (visited on Sept. 4, 2014).
- [21] *node.js*. Joyent, Inc. URL: <http://nodejs.org> (visited on Sept. 3, 2014).
- [22] *npm*. npm, Inc. URL: <http://www.npmjs.com/> (visited on Sept. 3, 2014).
- [26] *redis*. Redis. URL: <http://redis.io/> (visited on Sept. 3, 2014).
- [27] *SciPy*. SciPy. URL: <http://www.scipy.org/index.html> (visited on Sept. 5, 2014).
- [32] *winston*. flatiron. URL: <https://github.com/flatiron/winston> (visited on Sept. 7, 2014).

Appendix A

Running guide

The following README file is also available in the source file package. It describes the steps necessary to setup the system and to run it. It also contains suggestions for the case where the system gets stuck.

```
1 0. Introduction
2 =====
3
4 The system is designed to run in a completely decentralized setup: a
5 database machine, a synchronization machine (running redis), and one machine
6 for each component instance. Of course, all the software can be run from
7 the same machine without any special changes.
8
9 In order to ensure that the components can reach the MySQL and redis servers,
10 the source files must be edited to use the appropriate IP addresses.
11 The default configuration connects to localhost, on the default ports of the
12 servers.
13
14
15
16 1. Installing the prerequisites
17 =====
18
19 * MySQL (>= 5.5.32)
20 * redis (>= 2.8.7)
21 * python (2.7.5)
22 * node.js (>= 0.10.26)
23 * rsyslog (>= 5.8.11, configure it to save LOG_LOCAL0 to a file)
24
25 Python packages
26 -----
27
28 * numpy (>= 1.8.1)
29 * scipy (>= 0.13.3)
30 * mysql-connector-python (>= 0.3.2-devel)
31 * gensim (0.9.1)
32 * redis (>= 2.1.10)
33 * matplotlib (>= 1.3.1 - easier via package manager on Ubuntu)
34
35 The packages can be easily installed using easy_install.
36
37 Note: numpy requires python-dev on Ubuntu (and its forks).
38 Note: scipy requires libatlas-dev, liblapack-dev and libblas-dev and a Fortran
39 compiler (e.g. gfortran) to be installed on Ubuntu (and its forks).
40
41 Node.JS packages
42 -----
43
44 The standard node.js installation comes with the npm (package manager).
45 Therefore, just navigate to the Crawler or Interface root dir and run:
46
47 $ npm install
48
49 The latest version of NodeJS (0.10.31) comes packaged with an implementation of
```

```

50 npm which is slow, and buggy. Please run "npm install" several times if you
51 encounter errors. Also, it might help to remove the "node_modules" folder if
52 installation or running the scripts fails repeatedly, and run "npm install"
53 again.
54
55 Getting the sources
56 -----
57
58 Although at this point you already have the archive with the source code, it is
59 still worth mentioning that you can download it from:
60
61 http://thesis.cotizo.net/source.zip
62
63 2. Preparing the database
64 =====
65
66 * Navigate to the folder called "sql"
67 * Run the setup.sh file
68 * When prompted, insert the password for the root user of MySQL (enter if empty)
69
70 The SQL database is called "thesis", with password "thesis". Although the name
71 and password can be changed (in sql/migrations/create.mysql), remember to also
72 change them in all the source files and the sql/setup.sh file.
73
74 You can move from one migration to the next/previous one by applying "up" or
75 "down" scripts found in sql/migrations/.
76
77 3. Running the components
78 =====
79
80 The source files of all the components should be edited to point to the machines
81 running MySQL and redis. By default, the scripts point to localhost and default
82 ports of the servers.
83
84 Some of the scripts use libraries placed in the util/ folder. It is therefore
85 mandatory to set up the library paths for node.js and python:
86
87 * NODE_PATH=/path/to/project/util/redmutex node app.js
88 * PYTHONPATH=/path/to/project/util/redmutex:/path/to/project/util/logging python app.py
89
90 You can find a bash script (util/mps.sh) which can be used to start several
91 instances of the same script. For example, for starting 4 instances:
92
93 * NODE_PATH=[...] /path/to/project/util/mps.sh "node app.js" 4
94 * PYTHONPATH=[...] /path/to/project/util/mps.sh "python app.py" 4
95
96 The bash script runs the processes in the background, and saves their PIDs. When
97 you stop the bash script (e.g. by CTRL-C), it will kill all the PIDs first.
98
99 Crawler
100 -----
101
102 * navigate to crawler/
103 * install the node.js dependencies using `npm install`
104 * start the crawlers using `node app.js`
105 * edit the file called "init.js" to set the number of Amazon pages to be crawled
106   (i.e. change the limit of the for loop) and run it with `node init.js`. Wait a
107   bit and then you can kill the node process.
108
109 Note: you can run "init.js" after all the components are up and running.
110
111 Note: if the crawler does not start processing after init, it means that the
112 redis is locked. Please stop all the instances, and run:
113
114 $ redis-cli FLUSHALL
115
116 Then, restart the instances, and run init again.
117
118 Word extractor, LDA, Close apps, Permission mapper, Cousins
119 -----
120
121 * navigate to the folder of each of the components: "word" (Word extractor),
122   "lda" (LDA), "similarity" (Close apps), "permissions" (Permission mapper),

```

```

123   or "cousins" (Cousins)
124 * run the components using `python app.py` (or `python mapper.py` for the
125   permission mapper)
126
127 Note: Close apps component accepts a "--limit=N" parameter, which instructs it
128 to store only the closest N recommendations for each app. It is strongly
129 recommended to run the component with "--limit=200", otherwise the performance
130 of the User interface will have to suffer from JOINS over  $O(N^2)$  tables.
131
132 Note: If the LDA component is stuck at "Waiting for crawler and word to finish
133 processing...", even if the other two components have finished, then you can
134 force trigger LDA by:
135
136 $ redis-cli RPUSH /queue/lda-crawler 1
137 $ redis-cli RPUSH /queue/lda-word 1
138
139 The executable redis-cli can be found in the "src/" folder (if redis is compiled
140 from sources).
141
142 Note: if Close apps (similarity) does not produce any app distances, run from the ↔
143   component's folder:
144
145 $ python redis-inserter.py
146
147 Then re-run the component. The component can be kept alive while the previous
148 command is run.
149
150 Note: If there are no Cousins produced by the component, run the following
151 command from the folder of the component:
152
153 $ python redis-inserter.py
154
155 Then, re-run the Cousin instances.
156
157 User interface
158 -----
159
160 * navigate to webpage/
161 * install node.js dependencies using `npm install`
162 * run the server using `node app.js`
163 * navigate to localhost:8022
164
165 It is recommended to use install `forever` and use it to run the interface in
166 the background:
167
168 $ npm install -g forever
169 $ forever start app.js
170
171 The server's port can be changed by setting the environment variable PORT or
172 editing the "app.js" file.
173
174 4. Experiments
175 =====
176
177 Small plotting and data processing files, called "experiments", can be found in
178 the respective folder. Most of the scripts are controlled by different flags.
179 Please consult the headers of the scripts to see an explanation of what they do
180 and what flags are accepted.
181
182 5. Help, I'm stuck!
183 =====
184
185 The system might get stuck (deadlock) because of the distributed mutexes, or
186 because a pubsub message is missed. Normally, re-running the whole system should
187 fix the problem. If it is not the case, try to remove everything stored in redis
188 using:
189
190 $ redis-cli FLUSHALL
191
192 Now all the synchronization primitives (e.g. processing queues, mutexes) are
193 removed, and you can start the components from scratch. Removing the data from
194 the database is not necessary, since the crawler will not process a page which
195 is already in the database, so you can speed up the crawling this way.

```

195

196 If nothing works anymore, and the system deadlocks always, write an email to
197 csima@student.ethz.ch.