Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Software y Sistemas

CHECKING ANDROID APPLICATIONS BEHAVIOUR AGAINST GOOGLE PLAY DESCRIPTIONS AT SCALE

Autor: Daniel Domínguez Álvarez

Director: Julio Mariño Carballo

Co-director: Alessandra Gorla

Madrid, January 17, 2019

CHECKING ANDROID APPLICATIONS BEHAVIOUR AGAINST GOOGLE PLAY DESCRIPTIONS AT SCALE

Autor: Daniel Domínguez Álvarez
Director: Julio Mariño Carballo
Co-director: Alessandra Gorla

IMDEA Software Institute
DLSIIS
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

January 17, 2019

Thanks

I would like to thank Alessandra for directing my internship at IMDEA, its a great place for starting my research career. And thank you Julio for your understanding and patience with the logistical nightmare behind the submission of this work.

Resumen

Resumen —

En este trabajo fin de máster presento una nueva versión de la técnica conocida como CHABADA. El objetivo de la técnica es la búsqueda de correlaciones entre implementation de aplicaciones de Android y lo que anuncian sobre ellas en la tienda de aplicaciones. Esta nueva versión se centra en mejorar la escalabilidad de la anterior implementation con el objetivo de ser capaz de ejecutar la técnica con un dataset proveniente de la base de datos Tacyt. Tacyt es una plataforma de aplicaciones de Android obtenidas por ElevenPaths, una subsidiaria de Telefónica. Dentro de la conferencia JNIC investigadores de España pueden acceder a la base de datos y realizar experimentos sobre ella. A partir de la plataforma generé un dataset de 700000+ aplicaciones para analizar. La técnica usa una combinación de procesamiento de lenguaje natural, clustering y análisis estático para crear clusters de aplicaciones relacionadas entre sí. Dentro de los clusters la técnica extrae el comportamiento común para las aplicaciones del mismo y puede detectar malware dentro del cluster. Un aplicación se considera maliciosa si presenta comportamiento que no es común dentro del cluster. Por ejemplo, dentro de un cluster de aplicaciones de fotografía el comportamiento habitual sería el de acceder a la camara del telefono y al sistema de ficheros. Si una aplicación accediera a la API de SMS tendría un comportamiento fuera de lo común dentro del cluster y por tanto se clasificaría como una anomalía. Junto con las mejoras a la escalabilidad también he obtenido información directamente de la tienda Google Play Store para construir una verdad cercana al mundo real y usar esa verdad para evaluar los algoritmos de aprendizaje que se usan en la técnica.

Palabras clave — Android, detección de malware, análisis de descripciones, clustering

Abstract

Abstract —

In this master thesis I present a new version of an existing technique known as CHABADA. The goal of the technique is to search for a correlation between Android applications implementation and what is advertised in the store description of the Application. This new version is focused in improving the scalability of the old implementation in order to be able to run the technique with a dataset coming from the Tacyt database. Tacyt is a platform of scraped Android applications obtained by ElevenPaths, a Telefonica company, that is accessible to researchers in the context of the JNIC conference. From this platform I generated a dataset of 700000+ applications to be analyzed. The technique uses a combination of natural language processing, clustering and static analysis for creating clusters of related applications by the topics found in their descriptions. From this clusters the technique extracts the common behavior of the applications and can detect malware in a dataset of applications. A malicious application is classified as such if they show a behavior that is not common for the cluster of applications. For example, in a cluster of photography applications the usual behavior would be to access the camera and file system APIs. If an application accessed the SMS API it would be different from what is common in that cluster and therefore tagged as an anomaly. Along with the scalability improvements, I also obtained information directly from the Google Play Store for building a ground truth similar to the real world that can be used to evaluate the machine learning algorithms used in the technique.

Key words — Android, malware detection, description analysis, clustering

Contents

List of Figures

List of Tables

1 Introduction

In this work I have applied an existing technique known as CHABADA to a large dataset of Android applications. This technique, which is explained thoroughly later in this book, searches for correlations between the API usage in the application binary and the natural language description that is shown to the users in the Google Play Store. In order to be able to apply the technique to a large dataset of Android application I have had to refactor several parts of the implementation and add new modules to the system.

1.1 The Android application ecosystem

Android is the dominant operating system in the mobile devices market and had roughly 88% of market share in 2016 [1]. Users of the Android operating system can install third party software in app stores. This app stores serve as a marketplace where developers can publish their software and users can install it. Most marketplaces allow the developers publish the applications for free or behind a paywall. Google Play, Google's marketplace for Android applications, is the default marketplace installed in most of today's Android devices. In this marketplace the developers can write a description of the application capabilities and other information. There is, however, no automated way to know if this descriptions are accurate or honest; developers could be accessing information that is not related to the functionality of the application or the description could promise some features that aren't actually implemented. The manual way of inspecting an application also requires high technical expertise in Android applications, reverse engineering and analysis. In summary, there's no simple way for the average user to actually know if the software they are installing in their devices is a trustworthy software.

1.2 Scope

The goal of this project is to apply the existing technique of CHABADA to the Tacyt dataset. However, the existing implementation have scalability limitations that make running the original implementation as is very complicated, if not impossible. The size of the dataset is two orders of magnitude larger that the size of the dataset used in the first iteration of CHABADA back in 2014.

1.2.1 Tacyt

Tacyt[2] is a large dataset of Android applications made by ElevenPaths, a Telefonica company. They have scraped the Google Play store among others to create this dataset and have approximately 5 million applications. Tacyt has a web panel where users can query the system to obtain information about applications. Some applications can also be downloaded from the web panel. ElevenPaths also offers a free software library to connect to Tacyt written in the Python programming language.

The query language that Tacyt offers to query the system has a lot of parameters available and is composable using the three basic boolean operators ($\wedge$, $\vee$ and $\neg$). However, the results that can be obtained from the queries is limited. The query language only admits up to 40 predicates in a query, and returns a maximum of one thousand results. This limitations are probably implemented to avoid abuse and large queries that can degrade the performance of the system or consume too much resources during transmission.

In the JNIC conference of 2017 ElevenPaths proposed to the research community to use this dataset and further expand the meaning of the dataset.

1.2.2 CHABADA

CHABADA[3] is a technique developed by Gorla et al in 2014. The idea behind CHABADA is that is difficult to know whenever a certain application carries malicious code or not. For example, one could think that the ability to send SMS is out of place in a photo edition application, but is perfectly normal for a messaging application. By using a combination of natural language processing and clustering we can know what API usage is normal for a group of related applications. Once we have this information we can use outlier detection algorithms to find applications that use API calls that are not normal in the group. This applications could have a malicious intent or be poorly coded. To further know what's the actual intention of that abnormal API usage manual inspection is required.

1.3 Document structure

The rest of the document is structured as follows; first I explain the current state of the art related to the Android application ecosystem analysis. Following the state of the art I present an explanation of the CHABADA technique. After this I detail the changes that I have made to the implementation of CHABADA, and I finish with the obtained results and some conclusions.

2

State of the art

This thesis is a major extension of the CHABADA technique, which I explain in more details in the next chapter. While the CHABADA technique has been the first to generally check app descriptions against app behavior, it builds on a history of previous work combining natural language processing and software development.

2.1 Mining App Descriptions

Most related to CHABADA is the WHYPER framework of Pandita et al. [4]. Just like CHABADA, WHYPER attempts to automate the risk assessment of Android apps, and applies natural language processing to app descriptions. The aim of WHYPER is to tell whether the need for *sensitive permissions* (such as accesses to contacts or calendar) is motivated in the application description. In contrast to CHABADA, which fully automatically learns which topics are associated with which APIs (and by extension, which permissions), WHYPER requires manual annotation of sentences describing the need for permissions. Also, CHABADA goes beyond permissions in two ways: first, it focuses on APIs, which provide a more detailed view, and it aims for general mismatches between expectations and implementations.

The very idea of app store mining was introduced one year earlier when Harman et al. mined the Blackberry app store [5]. They focused on app meta-data to find patterns such as a correlation consumer rating and the rank of app downloads, but would not download or analyze the apps themselves.

The characterization of "normal" behavior comes from mining related applications; in general, CHABADA assumes what most applications in a well-maintained store do is also

what most users would expect to be legitimate. In contrast, recent work by Lin et al. [6] suggests *crowdsourcing* to infer what users expect from specific privacy settings; Lin et al. also highlight that privacy expectations vary between app categories. Such information from users can well complement what I infer from app descriptions.

2.1.1 Behavior/Description Mismatches

The CHABADA approach is also related to techniques that apply natural language processing to infer specifications from comments and documentation. Lin Tan et al. [7] extract implicit program rules from program corpora and use these rules to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments. Rules apply to ordering and nesting of calls and resource accesses ("f_a must not be called from f_b").

Høst and Østvold [8] learn from program corpora which verbs and phrases would normally be associated with specific method calls, and used these to identify misnamed methods.

Pandita et al. [9] identify sentences that describe code contracts from more than 2,500 sentences of API documents; these contracts can be checked either through tests or static analysis.

All these approaches compare program code against formal program documentation, whose semi-formal nature makes it easier to extract requirements. In contrast, CHABADA works on end-user documentation, which is decoupled from the program structure.

2.1.2 Detecting Malicious Apps

There is a large body of industrial products and research prototypes that focus on identifying known malicious behavior. Most influential for the CHABADA technique was the paper by Zhou and Jiang [10], who use the permissions requested by applications as a filter to identify potentially malicious applications; the actual detection uses static analysis to compare sequences of API calls against those of *known* malware. In contrast to all these approaches, CHABADA identifies outliers even without knowing what makes malicious behavior.

The TAINTDROID system [11] tracks dynamic information flow within Android apps and thus can detect usages of sensitive information. Using such dynamic flow information would yield far more precise behavior insights than static API usage; similarly, profilers such as ProfileDroid [12] would provide better information; however, both TAINTDROID and ProfileDroid require a representative set of executions. Integrating such techniques in CHABADA, combined with automated test generation [13, 14, 15, 16], would allow to learn normal and abnormal patterns of information flow, which has been implemented and published by the same research group of CHABADA [17].

3

The CHABADA technique

In this chapter I explain the CHABADA technique in detail. This technique along with its already existing implementation is the starting point for the rest of the work presented in this thesis.

The rationale behind this technique is based on the idea that what is considered a malicious feature for an application is actually a benign feature for other type of applications. In the Android ecosystem, which is where this technique has been developed on and evaluated against, applications are encapsulated in a sandbox where certain features of the operating system are unavailable. To access this features the application must ask for permission of the user. From this idea of requesting permissions we can get an idea of what certain type of applications should access and what not. However, this idea is not practical, because one would have to think of all possible features and properties that a program must have and decide if said features and properties fall into that category of programs or in another.

The Android application ecosystem is large, with millions of applications in the most common market, Google Play. With this applications we could take a group of related applications, for example, map and GPS applications, and find what makes them related. Once this relationship is made, we can search for applications that fall out of what is normal for this type of applications. Following the map and GPS applications example, it would be normal for this kind of application to access features like the GPS system and internet connection. It would be weird for these types of applications, instead, to request the capability of reading the phone contacts.

The CHABADA technique goes a step further, and tries to identify if a program acts as advertised or not. In our previous example we said that an uncommon feature of GPS apps is to read the contacts, but, what if a certain app wants to wed out its

competitors by offering some social features that require those contacts to work properly. This application, although is a legitimate one, could be erroneously classified as malicious, because it offers more features than what is considered common.

To identify what is advertised in an application, the CHABADA technique takes what the users are going to receive as an advertisement of the application. More precisely, it uses the description that accompanies the app in the store page where the user can download said application. From the descriptions the technique finds the different topics that the descriptions contain. GPS application's description will talk about maps, GPS and so on. While mobile games, for example, will talk about the features of the game.

From a set of applications, the technique finds the topics and relates each application to one or more topics. From this relationship we can inspect what features are common, not for the group of applications, but for that topic. That GPS application that included social features and accessed the contacts list would not be considered malicious if those features were advertised in the application's description.

3.1 The steps of the technique

The CHABADA technique is a sequence of steps that starts with a collection of applications and finishes with a set of clusters of applications that can be used to query for outliers of the clusters. As already mentioned, an outlier is an application that has some features that are not common for the cluster.

The first step of the technique consists in a cleanup of the application's descriptions. This cleanup improves the following step; in this second step the technique applies a NLP technique known as topic modeling. This technique takes a collection of documents and extracts what topics are discussed in those documents. The topics are usually represented by the most representative words in the topic. From the topics and the data that say to what topics a document belongs to, the technique generates clusters of applications. The next step is to augment the information in this clusters by analyzing the features of the application. With this information in the clusters the technique can detect the "interesting" applications by using outlier detection algorithms.

3.1.1 Input preprocessing

The descriptions in the Google Play store can have a lot of variation on how they are written, what metadata, such as URLs, they contain, they can be written in several languages, etc. In order to improve the subsequent machine learning steps there must be some normalization of the data. The first thing is that NLP algorithms must be tuned for the nuances and quirkiness of each language. A model trained for the Spanish language will not work very well on Italian, and will surely not work at all on languages like Chinese. For this reason, the technique only works with one language at a time.

In the original paper and in the following chapters, we chose to work with English since

is the most common language for writing descriptions. All the applications that have a description in a language that is not English are removed from the dataset. Google Play provides a translated description. It is responsibility of the implementation to properly choose the correct version of the translation if there's the case that the English version is available through this translated description.

In the Google Play store descriptions are interpreted as HTML files with the possibility of including limited HTML features like text formatting or anchors. This HTML tags need to be removed from the description because they don't include any relevant information from the point of view of the NLP algorithms. The next information that is removed from the description is numbers, punctuation and stop words (*the*, *is*, *at*, *which*, *on*, ...).

The final step of the cleanup is to reduce the words to their root forms. This way, similar words are reduced to the same root. To achieve this a common NLP technique for this is Stemming. This technique takes words like *playing*, *player* and *play* and reduces them to the root *plai*. After Stemming, the words that where reduced to one character words are removed because they have lost their semantics and make no sense to maintain them.

In the original paper the application ***London Restaurants Bars & Pubs +*** was used as an example. Figure 3.3 shows the result of applying the steps to the description of this application.

3.1.2 Topic modeling

For extracting the topics from the collection of descriptions, the technique uses *Latent Dirichlet Allocation* (LDA) [18]. LDA uses the statistical model created by *Dirichlet* for discovering the topics that appear in the collection of descriptions. It is a non supervised algorithm since it doesn't require labeled data to work. The idea is that a topic is a collection of words that frequently appear together.

As an example, if we feed LDA with documents talking about traveling and navigation it will group together words like *map*, *traffic*, *route* and *position*, and will group together words like *city*, *attraction*, *tour* and *visit*. If we then provide as input the description of the ***London Restaurants Bars & Pubs +*** application, it will be classified as belonging to both groups, since the description contains words of both groups. It is important to emphasize that applications can belong to several topics. The output of LDA, beside the topics themselves, is the probability of a certain document belonging to a certain topic (see Table 3.1). The maximum number of topics a document can belong to is a parameter of the algorithm. In the original paper this number was 4, in this work I have used 5. Also, in the original paper the number of topics was 30, the same number of categories that are in the Google Play store.

The original implementation of CHABADA used the Mallet framework [19], and I kept it for the new version as well.

Looking for a restaurant, a bar, a pub or just to have fun in London? Search no more! This application has all the information you need:

- You can search for every type of food you want: french, british, chinese, indian, etc.
- You can use it if you are in a car, on a bicycle or walking
- You can view all objectives on the map
- You can search objectives
- You can view objectives near you
- You can view directions (visual route, distance and duration)
- You can use it with Street View
- You can use it with Navigation

Keywords: london, restaurants, bars, pubs, food, breakfast, lunch, dinner, meal, eat, supper, street view, navigation

Figure 3.1: The description as is shown in the Google Play.

look restaur bar pub just fun london search applic inform need can search everi type food want french british chines indian etc can un car bicycl walk can view object map can search object can view object near can view direct visual rout distanc durat can us street view can us navig keyword london restaur bar pub food breakfast lunch dinner meal eat supper street view navig

Figure 3.2: The description as the topic modeling algorithm will see it.

Figure 3.3: The description of ***London Restaurants Bars & Pubs*** + before and after the cleaning step.

Application	$topic_1$	$topic_2$	$topic_3$	$topic_4$
app_1	0.55	0.45	–	–
app_2	–	–	0.67	0.33
app_3	0.5	0.3	–	0.2
app_4	–	–	0.4	0.6

Table 3.1: An example with 4 application and 4 topics with the probabilities of belonging to each.

3.1.3 Clustering

The result of topic modeling is a vector of affinity values to each topic for each application. The CHABADA technique needs groups of applications. For this reason, the applications need to be clustered in groups. The technique uses K-means for this task. K-means is

one of the most widely used clustering algorithms [20].

K-means works by selecting a centroid for each cluster, and associates each element of the space with it's nearest centroid. The centroids can be moved by a small amount in a series of iterations for improving the clusters. The numbers of clusters is a parameter of the algorithm and must be fine-tuned. The are several ways of fine-tuning the number of clusters, the CHABADA technique uses a technique known as the silhouette [21]. The silhouette is a measure of how closely related is an element to the elements of its cluster and how loosely related is to the elements of the other clusters. The value of the silhouette can vary from -1 to 1. -1 means that the number of clusters was poorly chosen, and 1 that the number of clusters is adequate to the shape of the data.

3.1.4 API analysis

Once the clusters of applications are identified, the technique can search for outliers in the clusters based on the behavior of the application implementation. For extracting this behavior information the technique relies on static analysis. This kind of analysis has weaknesses like obfuscation, but in the case of the API calls to the Android framework, those need to be explicitly declared.

The static analysis iterates over the bytecode of the application and extracts the list of API calls. This list of API calls includes all the calls to the methods to the Android framework. There are certain parts of the API that are key to the normal functionality of all apps, like the application life cycle or the UI subsystem. Including these API calls in the features of the machine learning algorithms would introduce noise and cause overfitting. CHABADA disregards any API call that is not considered sensitive. Sensitive means that it is associated to a certain permission. It relies on the work of Felt et al to extract the mapping between permissions and API calls [22]. Since the permission related to the API calls is available, the analysis ignores also the API calls which associated permission is not declared. If the permission is not declared in the manifest of the application, the API call will fail to execute, so is not relevant to the analysis of the behavior of the application.

3.1.5 Outlier detection

In the original paper the outlier detection was performed using *One-Class Support Vector Machine* (OC-SVM) [23]. This technique is used to learn the features of *one class* of elements. This kind of machine learning algorithm performs well when the dataset has many elements of one class at not many of other classes. One can provide only samples of one class and the classifier will be able to identify samples belonging to the same class. The sensitive API calls are considered as a set of binary features, and each cluster is trained using a subset of the elements of said cluster. This way, the cluster has a model of what API calls are common in the cluster. This model can then be used to identify applications that are outliers. The outliers are identified by the distance from the hyperplane created by the OC-SVM as the model. The bigger the distance the more different is an element

from the common behavior. A rank of outliers can be obtained by ranking the distances of the elements to the hyperplane.

In further work of CHABADA [24] the outlier detection system was changed to a distance based approach. This technique is based on calculating the distance between elements in a vector space. One way of for classifying the outliers is taking the elements with a greater distance to it's neighbors as outliers. This method is known as *k-Nearest Neighbors* (kNN). This algorithm needs a parameter k that determines how many neighbors are checked. In [24] the number used as parameter was 5, that was chosen as a good trade-off between the two extremes. A very small number of neighbors is too sensitive to noise and a very high number will be useless as it will practically classify everything as an outlier.

The outlier score is taken by the average of the distances. An application can be considered an outlier because it uses API calls that have never been used in the cluster or that are rarely used. Since the decision of being an outlier or not is a binary decision, the score needs a threshold that determines from what score value an application is considered an outlier within the cluster or not. An easy approach to do this is to sort the dataset by score and return a certain percentage of it with the highest score. Other approach is to use quartile statistics. The potential outliers using this statistical method would be those applications that exceed the third quartile by at least 1.5 times the *interquartile range* $(Q_3 - Q_1)$.

The approach used in the paper was originally proposed by Kriegel et al. [25] based on transforming the scores into probabilities using *Gaussian scaling*. Given the mean μ and the standard deviation σ, it uses the Gaussian error function $erf()$ to turn the score into a probability:

$$P(s) = \max\left\{0, \operatorname{erf}\left(\frac{s-\mu}{\sqrt{2}\sigma}\right)\right\} \tag{3.1}$$

All the applications with a probability not equal to 0 is considered an outlier.

4

Running CHABADA against the Tacyt dataset

This chapter presents first the limitations of the existing implementation of the CHABADA technique. Once the limitations have been stated, it presents an explanation of the changes that are needed in the implementation in order to improve the scalability of the implementation and a brief description of the technologies stack. The next section would then explain in detail the implementation, once both the design and the technologies have been stated. This chapter concludes with a description of the known limitations of this new design and architecture, either old limitations that could not be addressed with the new design or new limitations introduced in the new architecture.

4.1 Limitations of the old implementation

The starting point of the design and development is an existing implementation of a data pipeline using Luigi (4.3.1). This pipeline implements each step as a Task. Each Task has a set of requirements and a set of outputs. In order to run a certain Task, the outputs must be unsatisfied and the requirements satisfied.

The first step in the old pipeline is the preprocessing of the plain text descriptions. This needs as input a set of natural language descriptions in plan text format. The Task reads each description, process it and store the result in one file per application. From that point the pipeline executes a topic modeling tool called Mallet. This tool extracts the topics from the input text and assigns scores for each topic to each application. From this point, the results are processed in a Task that implements the Kmeans clustering algorithm. Once the clusters are created, the pipeline expects a mapping between the

applications and its permissions. With this information it will apply the KNN algorithm to find outliers from the clusters.

This implementation has several flaws that have been addressed. The first problem is that storing everything in files is costly when the data is in the order of hundreds of thousands. Along with those levels of input size, the implementation is not prepared to scale horizontally in order to be able to ingest all the input at a reasonable peace.

The old code base also used the permission list for applying KNN, however, using only the permissions list is not enough because there isn't always a one-to-one mapping of the declared permissions to actually used permissions [26].

4.2 The new architecture

In the legacy implementation the data was stored in files in the file system. Each piece of data of an application was in a separate file identified by the package name in the name of the file. While this approach is quick and simple; reading and writing files in the order of hundreds of thousands would be stressful to the file system and could even help reduce the lifetime of the drive.

The old implementation already had a pipeline style architecture that has been adapted to the new architecture. The main idea is to implement the tasks as atomic as possible to allow fine control of the execution. This will allow the system to control the parallelism more easily. In the old design each task took a piece of data as input and produced some other piece of data.

The pipeline and files approach is interesting to be maintained, but the persistence must be implemented in something that is designed for scalability. This performant persistence component must be able to handle several operations simultaneously since the tasks can potentially be executed in parallel to each other. The component that could fit best would be an eventually persistent database with locking at the row level.

The tasks would be chained to its dependencies, like in the old design. In this iteration however, each task operates over N rows of data, being N a configurable number. In the old implementation each task would operate over all the elements of the dataset in one thread. With those N-ary subsets of the input, each task would declare its dependencies to operate over the same subset. With this the tasks chains are completely independent from one another. Figure 4.1 explains graphically the new design and its main components.

For the permissions usage part, the goal would be to obtain information from the application binary. A static analyzer can obtain a list of android related API calls. This list of API calls can then be filtered to have only the API calls related to permissions. It's not interesting to have in the set API calls that are common to all android applications, like the application life cycle API calls, that only introduce noise in the dataset.

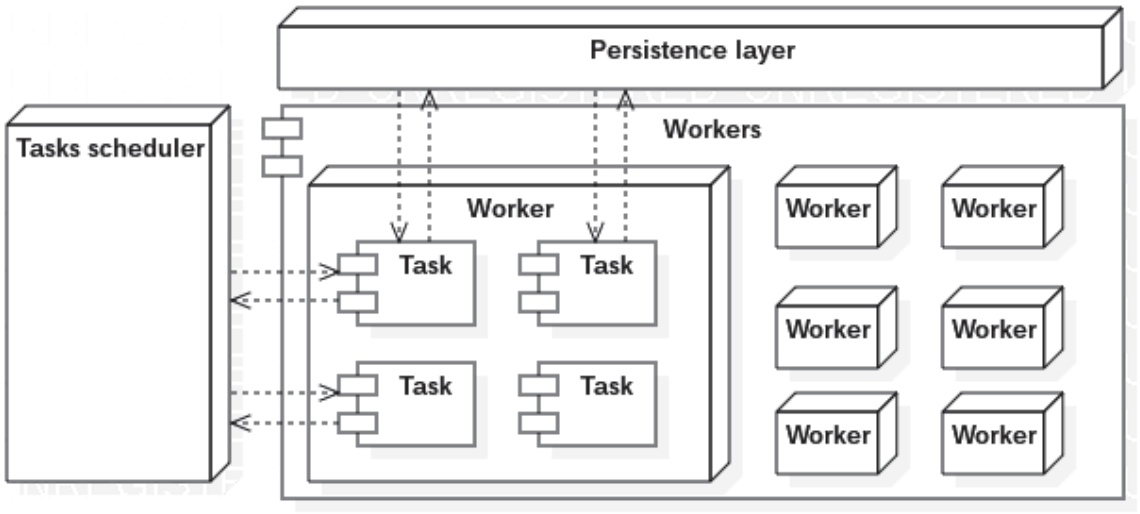

Figure 4.1: The new design and its logical components

4.3 Technologies Stack

4.3.1 The Luigi framework

Luigi is a free software framework for creating pipelines of dependencies, and it is implemented by Spotify. The programmer builds small tasks that perform a certain action. If a task depends of the output of another one, the programmer simply declares this dependency, and Luigi automatically runs the dependencies prior to the task to ensure that the dependencies are met before running the task. The framework is built for the Python programming language.

The framework has batteries included. The framework includes a lot of modules for running tasks in certain execution environments like PySpark or Hadoop. It also includes a lot of modules for specifying dependencies on external systems like HDFS or several databases. Although most of the database modules are maintained by the community. Creating custom modules for this kind of operations is also easy thanks to a simple class model. To create a custom element, no matter if its a task or a dependency module the programmer only has to inherit the base class for that kind of thing and implement in the template methods the custom behavior of the component. Figure 4.2 (from the official documentation) explains how this works.

Luigi also support concurrent execution of tasks given that the dependency graph continues to be sound (no task is run with unmet dependencies). For example, if one task has several other dependencies, Luigi will fill all the available workers with dependent tasks to be run and will be run in parallel. Once all this dependencies have been run, the initial task will be run. Luigi will also avoid rerunning tasks during execution. The output of a task is first check and the output is found then the task is not run. There's

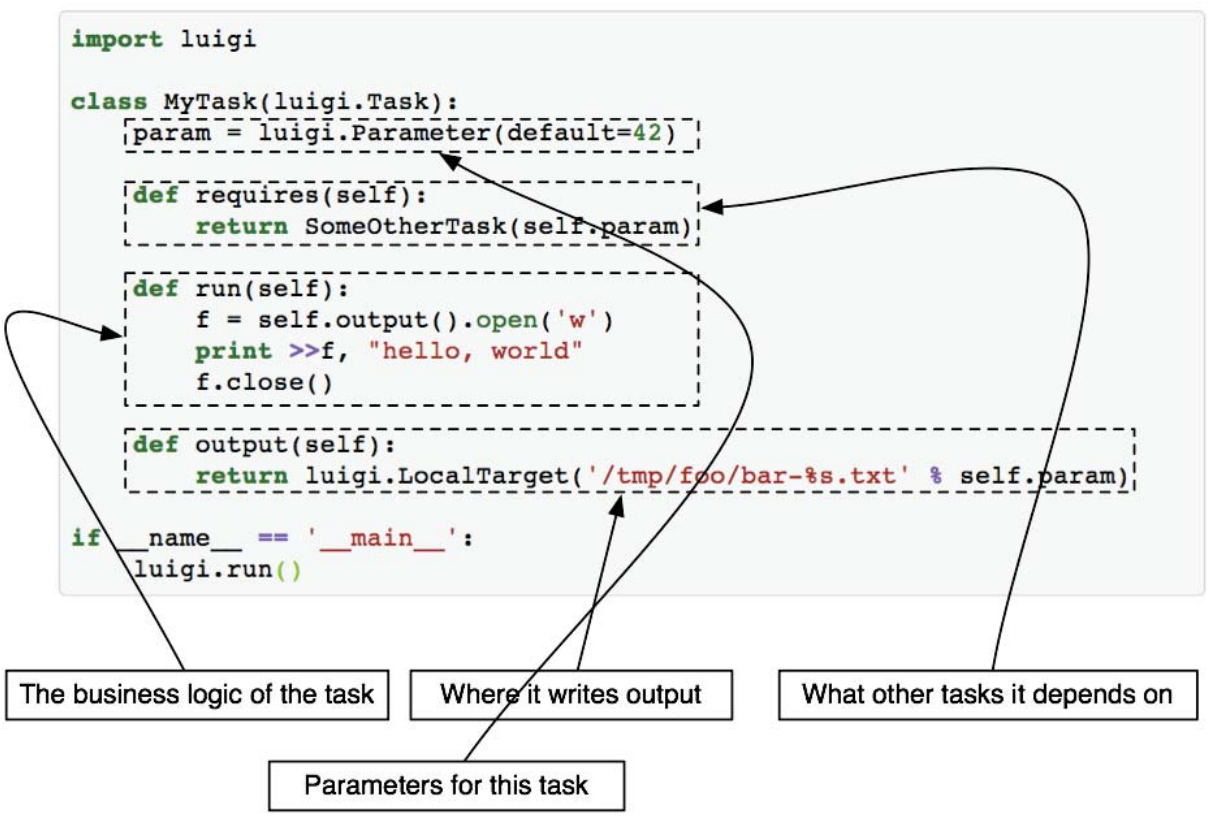

Figure 4.2: How a Luigi task is implemented. Taken from the Luigi official documentation [27]

also the scenario in which two tasks an have the same exact dependency with the same parameters and configuration, instead of running it twice, Luigi will cache the semantic information of the task and will run only one instance of that particular task.

To handle all the task management Luigi has a scheduler that is designed to be distributed by default. The remote scheduler (Figure 4.3) is a web application where workers submit tasks that need to be run. This scheduler only tracks the executions, but the actual running of the code is done by the worker that submitted the task. However, once a task is submitted to the scheduler it will be enqueued. When the task is ready to be run, it will be sent to the worker in order to be executed. Luigi also has dumb workers that will take tasks from other workers when the scheduler says so. This load balancing is primitive, but this is because a good load balancer is not in the project goals.

4.3.2 MongoDB

MongoDB is a No-SQL document database. The main features of MongoDB is that it is schema free and that is designed to be fast. It easily scales by creating clusters and it is able to handle a lot of simultaneous connections at the same time. While writing, the locking mechanism locks the document, contrary to traditional SQL databases that lock the entire table by default. One downside of MongoDB is that the whole database is

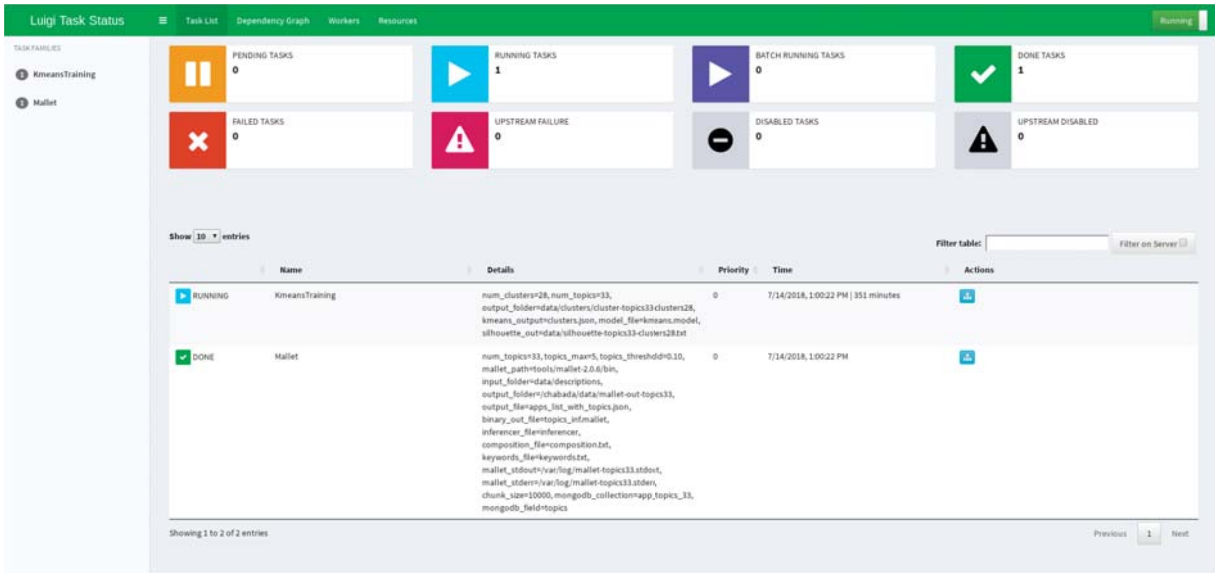

Figure 4.3: The status of the running tasks can be tracked with the remote scheduler web panel

locked during maintenance, but this is not an issue for my use case, because the system only needs to run during experiments and evaluations. A downtime for maintenance is permitted. Other interesting feature of MongoDB is indexing, which dramatically increases the reading times.

4.3.3 Docker

Docker is a container system that allows the creation of containerized applications. Usually this is meant for distribution in cloud environments. The usage of Docker is to create a container with a certain application fully installed. Then, this container can be instantiated all the times necessary and each instance of the application will be isolated from the rest. The containers can be connected to other containers, the host system and the Internet to allow distributed capabilities in the encapsulated applications. The most typical use case is to deploy a complete stack of a certain web application and to submit the container to cloud platforms that run the application. This application can then be instantiated several times to allow load balancing and other distributed features transparently.

The new architecture has several elements that need to be communicated with each other. By using Docker containers the deployment of the whole system is simplified.

4.4 Implemented changes

4.4.1 Data storage

MongoDB has the features described in the architecture section, so I have used it to implement the persistence layer in the pipeline. The type of data that I am dealing with exploits also the schema free capabilities of MongoDB because not all fields are available for each application inside Tacyt.

In the MongoDB database each document is identified by an unique identifier. By default this identifier is automatically generated by the system. However, this behavior is not interesting for my need because the input for the system is a list of android application packages. This can be changed though, but there are certain rules. The most significant one for my use case is that dots are not allowed in the name. This means that Java style package names, with dots separating each namespace can't be used. The system internally should have to modify the names of the applications to an alternative that is unique too and is allowed in the id rules of MongoDB. In the Java style naming convention slashes are not allowed, however, they are in MongoDB identifier rules. This means that I will transform each dot in the package name with a slash. For example, `com.whatsapp` would become `com/whatsapp`. Is easy to see intuitively that this transformation is bijective in both directions and both transformations satisfy $(f \circ f)(x) = f(x)$. This will allow to uniquely identify an application in any part of the system by simply applying the necessary transformation.

In the MongoDB database I would need to store several fields related to a certain application. This fields are the following:

- `versions`: This field will store all the available versions found in the Tacyt dataset.
- `latest`: This field will store the latest version that is in the `versions` list.
- `filtered_descr`: This field will store the cleaned up natural language description of the application. The source of this description is the description field found in the `latest` field.
- `topics`: This field stores the results of the topic modeling operation. Each application has a set of probabilities of belonging to a certain topic.

4.4.2 Refactoring

The old implementation of the filtering step was a set of functions that where connected manually by a sequence of call-n-store steps as seen in Figure 4.4.

A cleaner approach is to implement the filtering steps as an instance of the decorator pattern. This pattern takes an interface and implements some code that complements or

```
text = "..."
text = step1(text)
text = step2(text)
...
text = stepN(text)
```

Figure 4.4: Pseudo-code of the old implementation approach

overrides the implementation of a delegate member of the same interface. This pattern is great for creating this kind of sequential operations that progressively modify some piece of information.

With this implementation the steps are connected to the other ones by the interface and by being encapsulated in classes that can maintain an internal state between invocations. This is interesting because some steps require resources like dictionaries or algorithms that can be expensive to instantiate each time a piece of text is passed through.

To aid in the creating of the decorator chain, a builder class serves as a public API that internally connects the decorators by some methods. Figure 4.5 describes in UML the design of the classes.

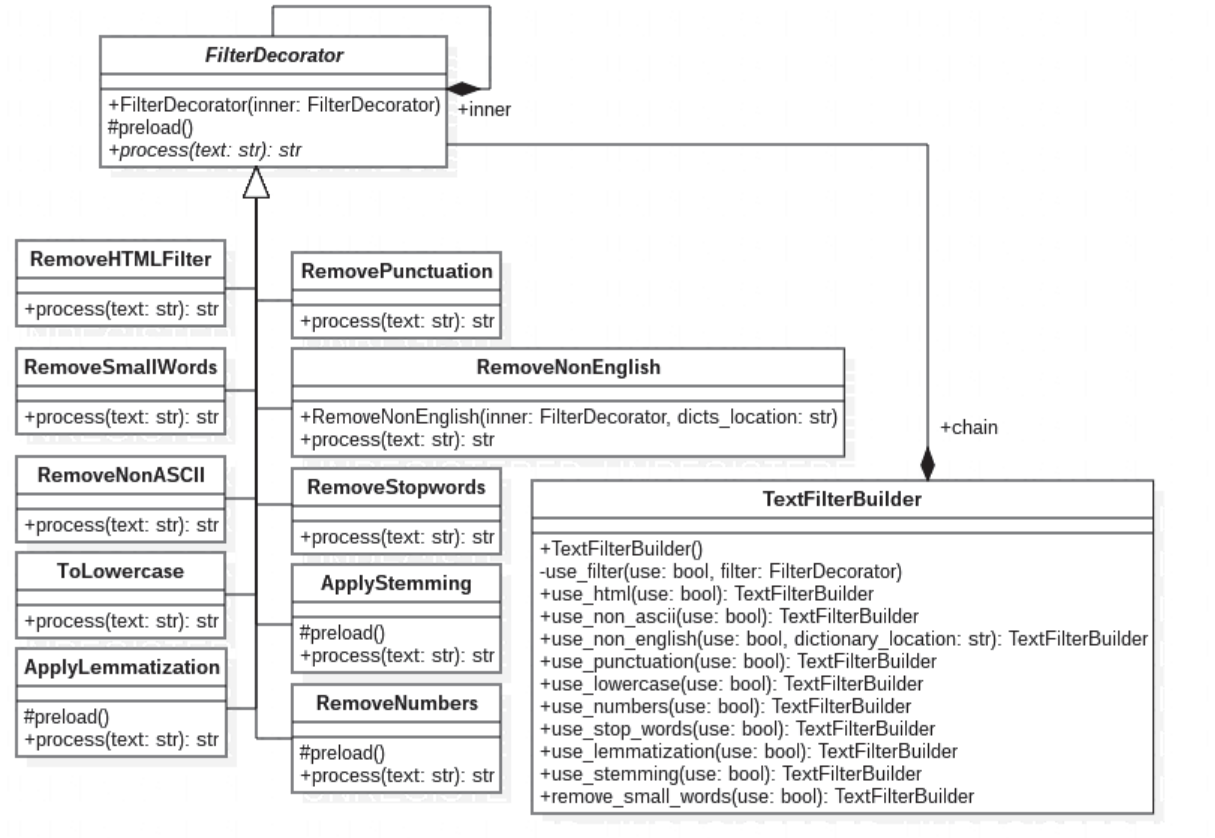

Figure 4.5: This UML diagram shows the design used during the refactoring of the filtering task

4.4.3 Scalability and load balancing

As mentioned in the Luigi section (4.3.1), Luigi doesn't implement automatic load balancing but allows the parallel execution of tasks. In order to be able to ingest the whole dataset in a timely manner the legacy implementation would need to be changed. In the old implementation each task took all the input and processed it sequentially.

Most of the operations in the system could be parallelized because those operation where handling data at the row level and could be parallelized to that level. However, the cost of switching context in the tasks could lead to a performance loss if the cost of loading a task is greater that the cost of processing a row. To avoid this costs if they occur, the parallelism was designed with chunks of data in mind. Using the old implementation of a processing loop as base, the tasks could be reimplemented to use a piece of data, represented in a list. With this model the load can be balanced by setting the chunk size.

4.4.4 Communication with Tacyt

In order to be able to obtain data from Tacyt and feed it to the pipeline, a module must be implemented that takes data from Tacyt and stores it in the database. However, not all applications in Tacyt are valid. Some have natural language descriptions in languages different than English or mixed with it. Also, there wasn't a list of packages of the applications that where in the dataset and the application binary wasn't easily available for downloading from the API. To overcome this problem, I decided to use the Androzoo dataset for downloading the application binaries.

Before running the pipeline with the dataset I need to clean it up. As a starting point I used the CISPA dataset, which is a list of 1.6 million app package names. To create a cleaned dataset using this list I would need to filter out the applications that don't follow this criteria:

- Tacyt has the application in the dataset.
- The application record has all the necessary information.
- Either the description or the translated description found in Tacyt is in English.
- There was an available application binary hash that could be used to reference it in Androzoo.

To filter the applications, I implemented a small program that downloaded application metadata and applied all the filtering steps. If one application was filtered out by some of the filters it was stored in the database with a `status` field that contains information about what filtering step decided to throw it. If an application survives all the filtering steps it means that it is eligible for the final dataset. In this case it was saved in the database with the corresponding `status` field stating it.

To description language filtering step was done with a pretrained dataset found in the Python package `langid`. This package exported a function that returns the probability of belonging to a certain language. If the description was above a certain threshold and the language was English, then the application was considered good for the next filtering step. In the execution the threshold value was 60%.

4.4.5 Static analysis

The old implementation used *smali* and *apktool* to do the static analysis. This tools generate an intermediate representation of the bytecode that can be parsed. In this version of CHABADA I used *androguard* to do the analysis. *androguard* parses in memory the application and exposes a Python API that can be used to query the apk file. With this framework I avoid two things; writing to much intermediate files to the filesystem and writing parsers. The analysis program is fairly simple, it looks for each class in the application code and searches for all method invocations, then filters out the methods that are not method invocations to the Android framework and that are not sensitive API calls. This leave me with the sensitive API calls made by the application code. The static analyzer has been developed as a separate module and can be found in Github [28].

4.4.6 Modules

The pipeline is composed by a set of tasks that each implement a certain action in the pipeline. In Figure 4.6 can be seen the dependencies of each task.

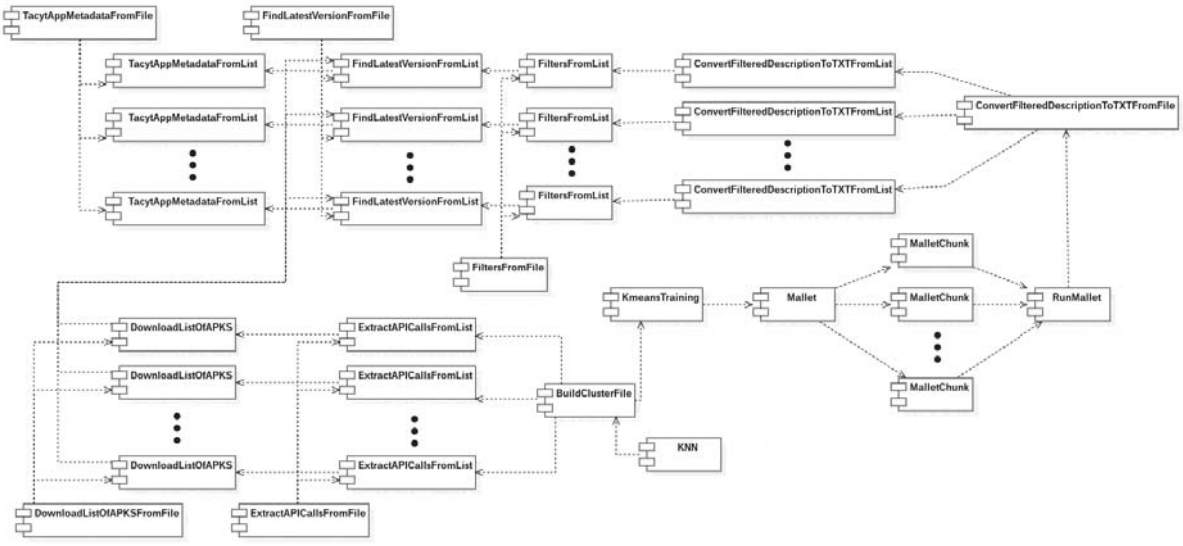

Figure 4.6: The concrete architecture of the pipeline with each task and its dependencies

`TacytAppMetadataFromList`: This task takes a list of package names and downloads from Tacyt the metadata found for each version of the application. All the data is

stored as is in the `versions` field of the app's document.

`TacytAppMetadataFromFile`: This task allows the easy running of `TacytAppMetadataFromList` by reading a file with package names and creating tasks that actually download the metadata. It serves nothing but as a facade.

`FindLatestVersionFromList`: This task will take the result of `TacytAppMetadataFromList` and will search for the latest version the application has. The latest version will be copied as is in the `latest` field in the app's document.

`FindLatestVersionFromFile`: Similar to `TacytAppMetadataFromFile`, this task will read a file with package names and will invoke `FindLatestVersionFromList` with subsets of the input depending on the chunk size.

`FiltersFromList`: This task will take the natural language description of the application from the `latest` field in the app's document. This text is processes by a series of filters and then it stores the cleaned description in the `filtered_descr` field.

`FiltersFromFile`: This task will serve as a facade to `FiltersFromList` by reading a file with package names and running instances of that task with the obtained chunks.

`ConvertFilteredDescriptionToTXTFromList`: Since I didn't changed the old Mallet component, I still need to store the descriptions as text files in order to be able to use it as input for Mallet. This task will take as input the result of `FiltersFromList` and store each description in a text file.

`ConvertFilteredDescriptionToTXTFromFile`: This task will read a file with package names and split them in chunks. Each chunk will be handled by an instance of `ConvertFilteredDescriptionToTXTFromList`. This task will serve as an entry point of the cleanup phase and the mallet tasks will use it to launch said phase.

`RunMallet`: This task takes the files created by the `ConvertFilteredDescriptionToTXTFromList` tasks and feed it to a mallet instance that is executed as an external program. This program will produce an output in the specified folder.

`MalletChunk`: This task declares `RunMallet` as a dependency and will read the output of this task and write it in MongoDB. It is designed to be parallel. Each instance will read the file, take the applications that belongs to it's chunk and store it.

`Mallet`: This task will take a file with package names and will create `MalletChunk` dependencies. This task is the entry point for the topic modeling phase.

`KmeansTraining`: This task will read the input of the `Mallet` task and will perform clustering using the Kmeans algorithm. It can also optionally do the silhouette of the cluster to check if the cluster parameters are fit to the type of data used.

`DownloadListOfAPKS`: This task will download from the Androzoo repository each application specified. The information needed for downloading is taken from the `FindLatestVersionFromList` task.

`DownloadListOfAPKSFromFile`: As all the other `*FromFile` tasks, this one takes a file with package names and sends chunks of package names to instances of `DownloadListOfAPKS`.

`ExtractAPICallsFromList`: This task will take the APK files downloaded in `DownloadListOfAPKS` and apply a simple static analysis that extracts the API calls related to the Android framework. This list is then filtered using a mapping between API calls and permissions.

`ExtractAPICallsFromFile`: This task reads a file of package names and sends chunks of package names to instances of `ExtractAPICallsFromList` for analysis of their related APKs.

`BuildClusterFile`: This task will augment the information of each application in the clusters with the information about what API calls each application is using.

`KNN`: Using the results of `BuildClusterFile` this task will apply the KNN algorithm to the clusters for searching outliers in them.

4.4.7 Deployment

Using Docker as the deployment solution I deployed the pipeline in a virtual server managed by the IMDEA Software Institute with 72 cores and 62GB of RAM. Since it was a shared server with other researchers of the institute I used only about 20 cores of the ones available.

There are 3 main components to be deployed; a Luigi scheduler, a MongoDB instance and a worker with the pipeline implementation. The two first components where implemented each in a container and launched in the server. The worker component has one container instance for each worker in the system. When the pipeline has to be launched, one container with the needed parameters is launched and connected to the scheduler and database containers. This worker containers can be instantiated several times given that the connections to the outside of the container are not overlapping (i.e: they are not writing to the same directory and files).

4.5 Known limitations of the new architecture

There are two known important limitations of the new architecture. Both of them are related to the size of the data being used. The first limitation is that PyMongo, the python library used to connect to MongoDB has a limit in the amount of bytes that can be sent to the database. This limit is enforced by the library and is caped at 16MB. This

16MB limit is famous in MongoDB but is only enforced by the database in the document size. MongoDB does not have a limit in the size of the data that is being transmitted. PyMongo however enforces a limit of 16MB when sending queries to the database. This force me to split the storage of the Mallet results in several tasks that only would take chunks of the result file. PyMongo didn't allow to send the whole insert query at once because the query was larger than this limit.

The evaluation of the clusters using the silhouette of them has some limitations due to the large size of the dataset. The Silhouette works by performing a Cartesian products of all the elements in the space and stores it in a matrix, this means that with a dataset of more than 700k applications, the matrix will be square that. Sklearn has an implementation of the silhouette that is can handle in memory such large datasets, but as a drawback, the implementation is painfully slow. This can be a very limiting problem. However, the silhouette implementation allows to sample the dataset and calculate the silhoutte on that sample. This means that the algorithm works with an amount of data that can feasibly handle with the drawback that is losing precision. For CHABADA I decided to do a sampling of a 25% of the dataset, this size fits in the computer's memory and can be finished in minutes.

Since the sampling involves a random process, I don't want to take the risk of losing the big picture of the data with the sampling by taking a non representative sample. To make sure the sampling was not screwing with my results I ran a 10% sampling 30 times each pair and then calculated the standard deviation. As Figure 4.7, the deviation is not sufficiently large to suppose that the samples differ too much from one another.

4.6 Ground truth for the Machine Learning algorithms

Although I already have the silhouette algorithm for evaluating the fitness of the hyperparameters to the data, I wanted to see how fit where the hyperparameters to some real world classification of applications. Since I am trying to evaluate the fitness of how related some applications are, I used some information from the Google Play Store. More precisely, I took the recommend, or similar, applications that the Google Play store offers to the users about an app. For example, for Whatsapp, Google Play suggest as similar applications like Telegram or LINE. It is uncertain how Google decides the similarity, but it can be used as ground truth for what the related applications are expected to be.

I built a simple scrapper that took all the applications in my dataset and got the similar applications that the Google Play store suggests. In the results there are applications that are inside my dataset and others that are outside. From this, I built a graph of applications by their similarity. The total graph has 10682425 edges from which 2102355 are edges that connect two applications in the dataset.

I compared the graph to the clusters generated by Kmeans and used a simple score system that gives an score to a pair of apps based on their position in the graph and the clusters (Table 4.1).

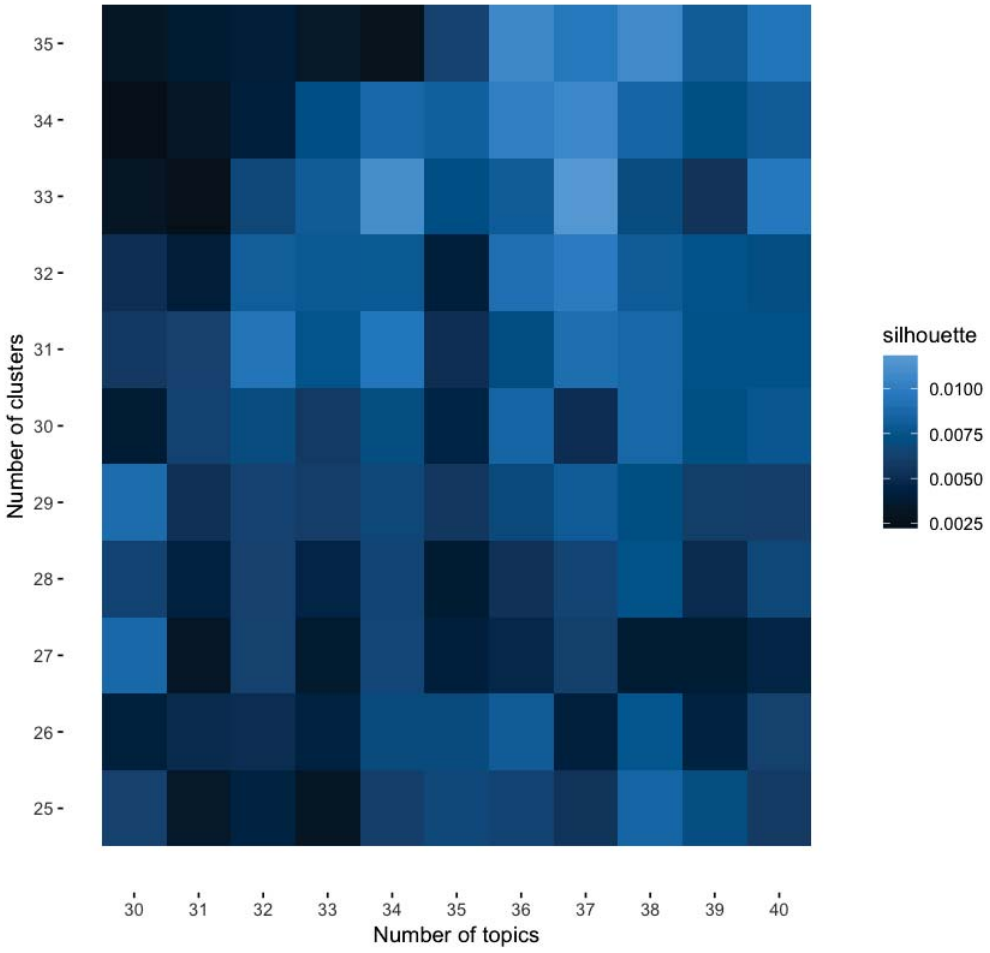

Figure 4.7: The standard deviation of sampling at 10%.

	Connected	**Not connected**
Same cluster	1	-0.5
Different cluster	-1	0.5

Table 4.1: The scoring system for evaluating the clusters using the graph.

The idea behind the scores is to reward clusters that have applications that I know are related and to punish clusters that disconnect applications I know are related. In the case of two applications not being connected in the graph doesn't mean that they are 100% unrelated, but that I don't know if they are or not. Hence the lower score in that case. Once each pair if applications has it's score I take the average of it and that's the silhouette of the clusters according to the graph of applications extracted from the Google Play store.

5 Results

5.1 Dataset creation from Tacyt

In order to be able to run the CHABADA implementation over Tacyt I first needed to build a dataset. This dataset, as explained in Section 4.4.4, starts with the CISPA dataset. This dataset has around 1.6 million of applications. From this dataset I only took the package names. Using this package names I can query Tacyt with the data for each application. During the dataset creation I setup a series of filters or conditions that each application in the final dataset must hold. The raw numbers of the process are described in Table 5.1.

	Number of apps
CISPA dataset	1658808
Apps not found in Tacyt	440184
Apps with missing required fields	143131
Apps with a non English description	334747
Final dataset size	740746

Table 5.1: The results of each filtering step and the final size of the dataset.

From the apps with missing fields, all of them didn't had the `sha256` field. This field was required because this hash is used for referencing the application binary when downloading from Androzoo.

5.2 Evaluation of the dataset in CHABADA

To check the performance of the dataset, I ran the pipeline measuring the time each time took to complete. I configured the chunk size to be 50000, which created 15 chunks. Each chunk was assigned to a worker. Some tasks didn't supported chunking, so the running time of the task is the running time of a single instance. The running time of each task is described in Table 5.2. The times in the table are the rounded average because each task had each own running time but all of them where very similar. Also, given this running times, a difference of seconds or even minutes is irrelevant.

Task	**Approximate running time**	**Number of workers**
Cleanup of description for LDA	2 hours	15
Storing descriptions to text files	4 hours and 30 minutes	15
Execution of Mallet	2 hours	1
Kmeans	20 minutes	1

Table 5.2: The approximate times each task took to complete. Multiply by the number of workers to get the approximate CPU time.

These times are not ideal, but they are to be expected. Once this metrics where taken, I proceeded to optimize the parameters of the machine learning algorithms. The parameters that I wanted to optimize are the number of topics and the number of clusters. The first step was to determine a good range of number of topics. I tried with 20, 30, 40, 50, 70, 90, 100, 150 and 200. After manual inspection of the resulting topics, 30 topics where the number of topics that made more sense. 20 topics produced topics that spoke about too many things and in 50 and above the topics became redundant. 30 to 40 topics seems like the correct number of topics. Table 5.3 shows the topics obtained. Once I had this range the next step is to get the number of clusters.

Topic No.	**Assigned name**	**Most Representative Words**
0	quiz and puzzle games	game, play, puzzle, challenge, quiz, answer, difficulti, correct, memori
1	health and fitness	bodi, health, exercis, workout, weight, fit, train, gym, medicin, doctor, lifestyl
2	military games	game, enemy, shoot, zombi, war, attack, shooter, aim, warrior, aircraft, missile, strategi
3	gambling and casino games	game, football, world, poker, solitari, basketball, casino, prize, bet
4	spirituality and religion	bibl, god, holy, jesus, muslim, spirit, worship, part, religion, roman, gospel, zodiac

5	languages and dictionary	word, learn, languag, translat, dictionary, spanish, vocabulari, italian, japanes, nativ
6	money and business	bank, payment, coupon, financi, budget, card, pay, purchas, market, incom, credit
7	cooking	recip, food, cook, cake, pizza, quick, simpl, diet, beef, vegan, cookbook, fruit, bake
8	platform games	game, jump, collect, adventur, score, coin, obstacl, speed, enemi, funni
9	social media	facebook, whatsapp, off-line, photo, email, social, status, chat, instagram
10	music	sound, rignton, music, play, voic, alarm, guitar, instrument, volum, speaker, melodi
11	photography	photo, frame, pictur, imag, effect, share, camera, wallpaper, blur, stylish, photographi, captur
12	planification apps	news, inform, updat, schedul, program
13	home screen personalization	widget, clock, menu, icon, unlock, bar, hide, lock screen, custom, launch
14	cloud drive apps	video, download, view, edit, player, playback, cloud, pdf, export, librari, upload
15	Candy Crush lookalikes	game, bubbl, free, level, candi, puzzle, jewel, jackpot, destroy, combo
16	kids games	game, kid, child, learn, babi, rhyme, pet, doodl, educ, alphabet, paint
17	self help apps	help, peopl, chanc, posit, learn, chang, success, inspir
18	online television & radio	radio, station, tv, program, televis, youtub, satelli, trailer, wherev, sport
19	calculator apps	call, number, data, measure, unit, rate, convert, currenc, length, mathemat
20	general customisation	theme, keyboard, launcher, background, smiley, decor, screenshot
21	gamer communities	game, top, mod, unoffici, guidelin, friend, skin, tutori, player, clan, fandom, strategy
22	system monitoring apps	devic, use, batteri, system, password, remot, bluetooth, memory, monitor, cpu
23	home decor apps	design, room, home, garden, craft, floor, bathroom, architectur, lamp, colour

24	ads and policies	free, version, ad, rate, feedback, click, websit, licens, polici
25	racing games	car, drive, race, truck, driver, road, citi, bike, speed, traffic, rider, game
26	institutional apps	servic, industri, govern, confer, univers, website, institut,
27	dressing games	design, dress, girl, hair, fashion, style, beauty, necklac, skirt, wear
28	traveling apps	map, local, citi, travel hotel, guid, gps, local, rester, visit, trip
29	wallpaper apps	wallpaper, live, free, image, background, beauty, set, anim, download, picture

Table 5.3: The 30 topics extracted from the dataset of applications.

To find the correct combination of number of topics and clusters I run a matrix of topics and clusters. The parameter of the topics went from 30 to 40 and the parameter of clusters from 25 to 35. For each combination I tried to obtain the silhouette. The silhouette is an score from -1 to 1 that determines how well situated are the elements of a cluster compared to it's neighbors. It is used to evaluate the fitness of the K parameter in the Kmeans algorithm. With the 25% of sampling for the silhouette I obtained the results of the silhouette for that range. A heatmap of the different values of the silhouette is in figure 5.1.

5.3 Results from the graph of applications

5.3.1 Results from using the Graph to evaluate the clusters

Using the scoring system explained in the implementation chapter, I evaluated the hyperparameters in the same range as the silhouette. However, I used a 10% of sampling in this case for efficiency reasons. The results can be seen in Figure 5.2

5.3.2 Missing apps

From the graph of applications I got some interesting results apart from the actual graph. The main one is that most of my dataset was missing from the Google Play store, this can be because many reasons, but the most probable one is that this apps where actually *crapware* and where removed from the store in consequence of a violation of the guidelines.

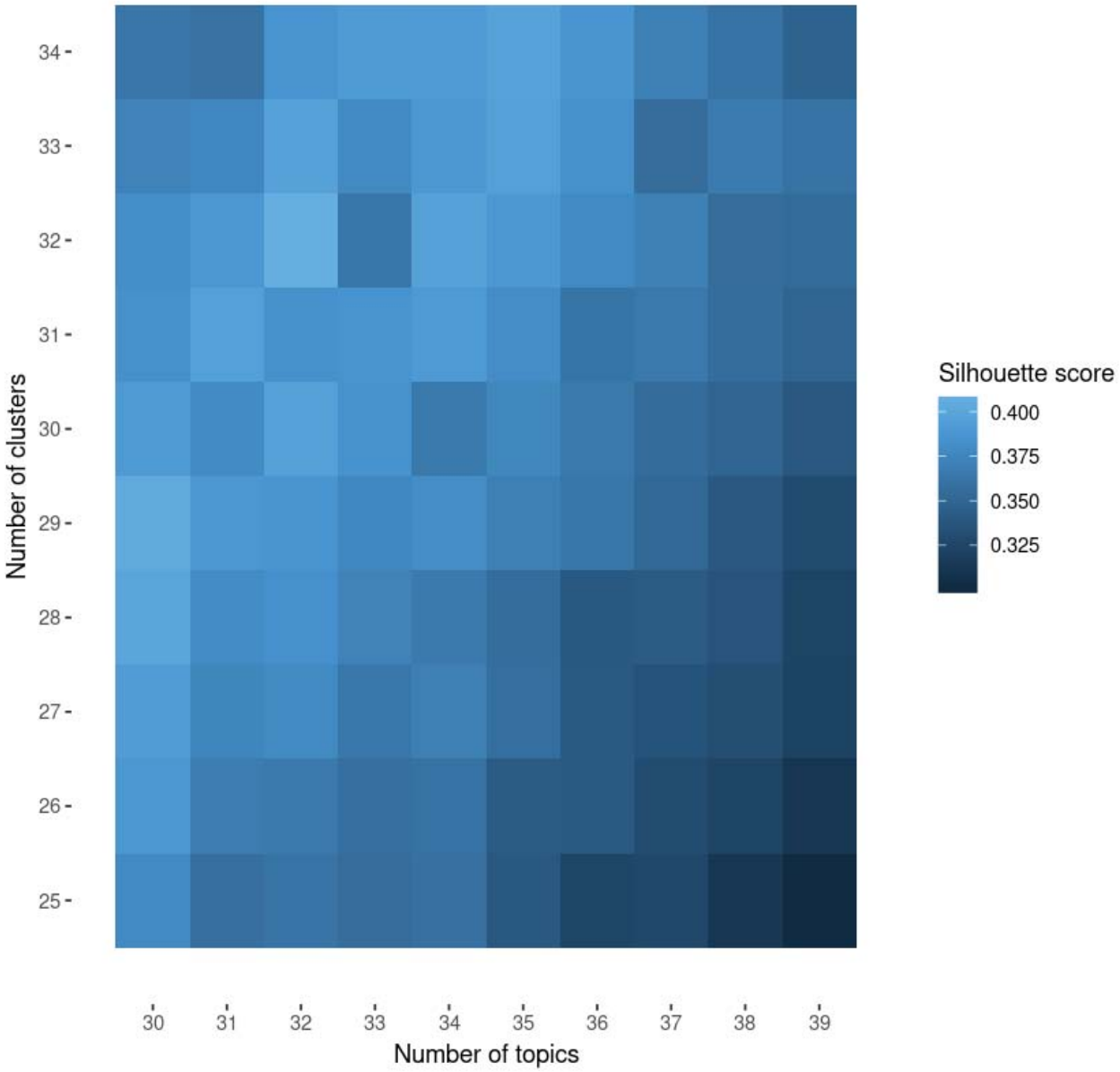

Figure 5.1: The heatmap of the different silhouette values.

One example of this applications is one that predicts whenever a pregnant woman will have a boy or a girl.

One result that is worth noticing is that, as show in Table 5.4, some applications that are missing from the store have applications that are related to them. A possible explanation of this is that Google stores the related apps information separately from the actual app information. And once an app is removed it may take some time to remove that information. It is likely that Google has a very paranoid database administration policy and information will take a long time to be removed.

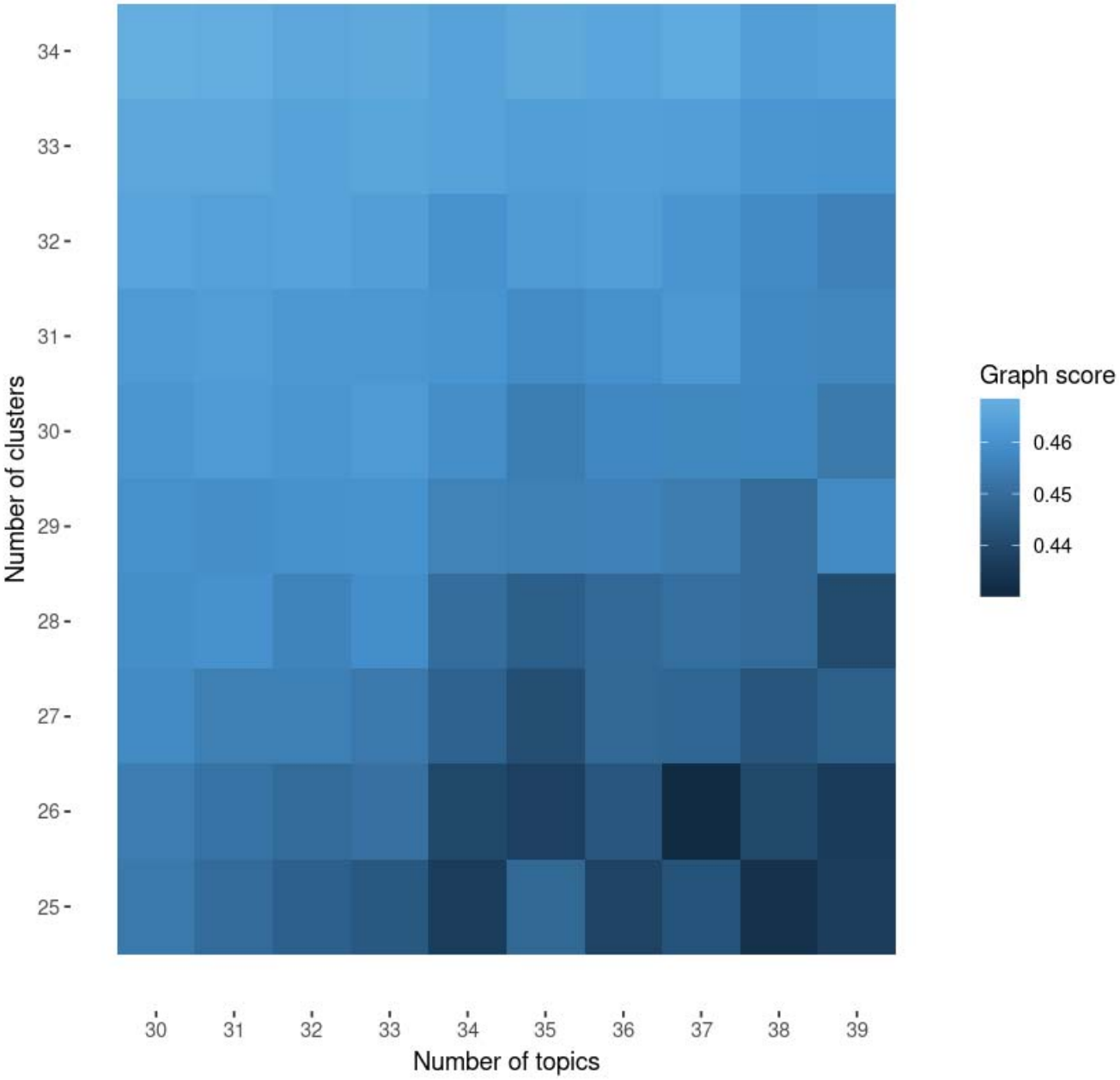

Figure 5.2: The heatmap of the different scores.

Status	**No similar field**	**With similar**	**Without similar**	**Total**
Missing	13257 (1.79%)	33027 (4.46%)	450813 (60.86%)	497097 (67.11%)
Found	6746 (0.91%)	225079 (30.39%)	11809 (1.59%)	243634 (32.89%)
Total	20003 (2.70%)	278123 (37.55%)	462623 (62.45%)	720743 (97.30%)

Table 5.4: The distribution of apps of the dataset in the results of the scrapper.

5.4 Results from classifying the applications with KNN

From the results obtained in the evaluation of the hyperparameters I obtained that, according to the graph scoring, the best hyperparameters are 34 clusters and 30 topics. Using this hyperparameters I calculated the anomalies in the clusters using KNN. For the features used in the anomaly detection I used the permissions of the applications that are declared in their manifests. The permissions are an usable approximation of what an application is expected to do, because it can't do more than what is declared in the permissions. From the results of the classification I got the scores of the applications that I found that are missing in the Google Play Store. Figures 5.3 and 5.4 shows the scores for each application missing and found respectively. As can be seen in the figures, there is no apparent relationship between the fact that an application is missing from the Google Play Store and that an application is considered an outlier by KNN. In the case of the missing applications, around 30% of them have an score greater than 0. In the case of the found applications the percentage is around 35%. This means that being malware is not the only reason Google might have to remove an application from the store. The second most probable reason is an expired developer license. An explanation of a possible way of checking if this is the case is mentioned in the next chapter.

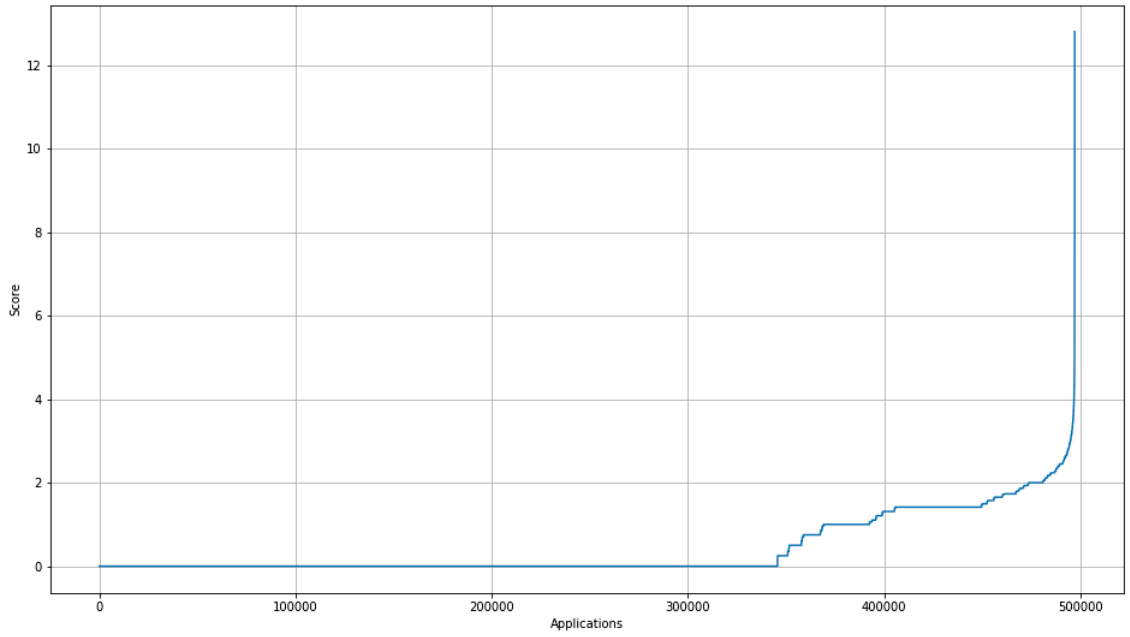

Figure 5.3: Missing apps by their score in KNN

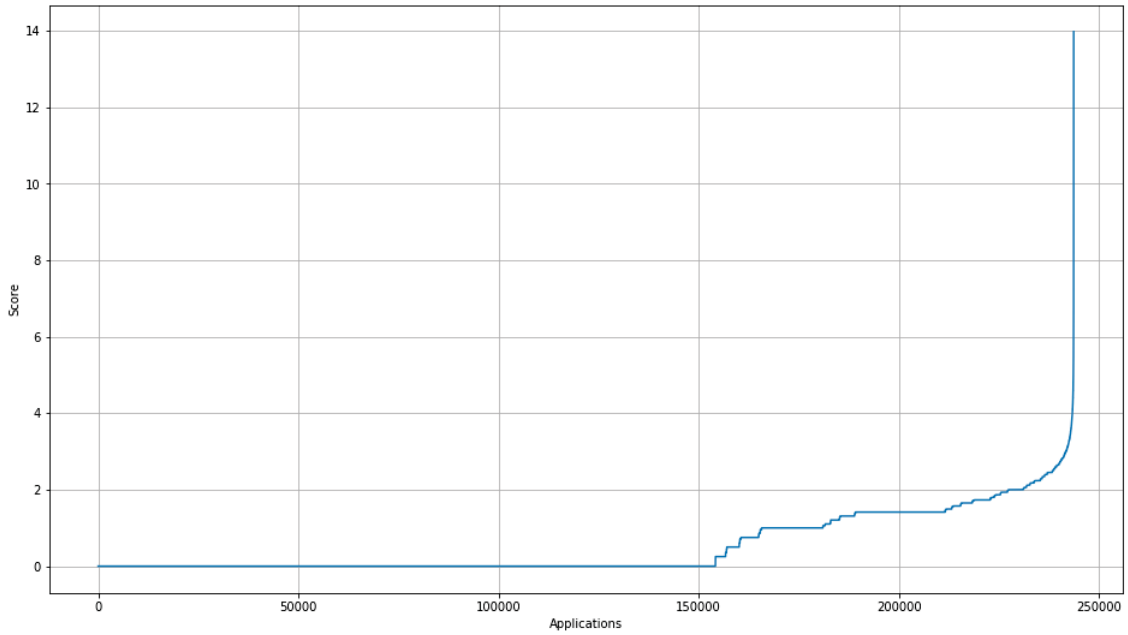

Figure 5.4: Found apps by their score in KNN

6 Conclusions

In this thesis I have presented how the CHABADA technique works and what modifications I have implemented in order to make it scalable enough for running it against the Tacyt dataset. The size of my final dataset was around 700000 applications. This size was large enough to make the pipeline crash in some places and degrade the performance in some others. The changes I have introduced have allowed me to run all of the pipeline with the dataset.

The changes introduced in the implementation have allowed me to run the whole dataset in a reasonable time. This changes involved the usage of a database backend designed for speed, MongoDB, and the reimplementation of the code with scalability in mind. The support for scalability of Luigi is not the most advanced of the world, but is good enough for my purposes.

6.1 Future work

As always, there is room for improvement and this section discuses some possibilities.

The limit of 16MB of transference in PyMongo is more of a annoyance than a bug. It is possible to implement the code so that this problem is avoided. There are two possible solutions to the problem. The first one is to change the Mongo library used. This library needs to be compatible with the MongoDB Luigi modules or those would need to be rewritten too. Of course, this modification is useless if this hardcoded limits or other are present in the alternative libraries. The second option is to change the backend to another database. Not all databases can fit with the schema of the data, but other document oriented databases could be used. One example is Elasticsearch[29].

Elasticsearch can be used as a data lake thanks to its powerful internal search engine, Apache Lucene. As far as I know, Elasticsearch does not have limits in the document size or in the transference of data between clients and servers. It also is very easily scalable thanks to its architecture. The main thing to think about is if the advantages of changing the library or the whole database backend is worth enough the effort. As mentioned, is not a bug, but a limit that can be avoided with proper load balancing.

The second situation is that since the Mallet module has not been changed, there's still need to write the files that the mallet tool uses as input. This problem can be addressed in two ways too. The first one is to write a Java wrapper that connects to mallet as a library and performs all the operations in memory. This wrapper will connect to the database backend and extract the data directly from there. Other option is to change the implementation of LDA to a Python based implementation. There are two options that can be used given the size of the problem. The first one is the library called Gensim[30]. This library implements LDA and has implemented a master slave architecture where dumb workers can parallelize the execution of the algorithms. The other possibility is similar, in PySpark there's an implementation of LDA[31] and has the same parallelization capabilities than Gensim. If more parts of the code is modified to run in Spark workers, then the PySpark option is the most interesting one since the tasks can share worker instances of Spark. Other than that, there are not strong arguments against one or the other.

Other interesting path to explore is the graph that I obtained from scrapping the Google Play Store. I do believe that from this graph more information can be obtained and could be used to reason about applications in the store, and serve as a Ground Truth when processing applications in some kind of manner, like the machine learning algorithms used in this work. The scoring system can surely use some improvements, but is nonetheless promising.

From the results of the anomaly detection I obtained that being classified as an outlier is not correlated to being malware detected by the store and then removed. Other possibility is that the application is abandonware. Possible ways for checking this is looking at the activity by the developer. If the developer hasn't update the application in a very long time (from the scraped results in Tacyt, for example) or the developer is nowhere to be found in the store, that's an strong indicator that the application is abandoned and Google removed it after the license expired.

Bibliography

[1] *Android (GOOG) just hit a record 88% market share of all smartphones — Quartz*. https://qz.com/826672/android-goog-just-hit-a-record-88-market-share-of-all-smartphones/. (Accessed on 14/07/2018).

[2] *Tacyt*. https://www.elevenpaths.com/technology/tacyt/index.html. (Accessed on 01/14/2019).

[3] Alessandra Gorla et al. "Checking App Behavior Against App Descriptions." In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 1025–1035. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568276. URL: http://doi.acm.org/10.1145/2568225.2568276.

[4] Rahul Pandita et al. "WHYPER: Towards Automating Risk Assessment of Mobile Applications." In: *USENIX Security Symposium*. Washington, DC, 2013, pp. 527–542. URL: https://www.usenix.org/conference/usenixsecurity13/whyper-towards-automating-risk-assessment-mobile-applications.

[5] M. Harman, Yue Jia, and Yuanyuan Zhang. "App store mining and analysis: MSR for app stores." In: *IEEE Working Conference on Mining Software Repositories (MSR)*. 2012, pp. 108–111. DOI: 10.1109/MSR.2012.6224306.

[6] Jialiu Lin et al. "Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing." In: *ACM Conference on Ubiquitous Computing (UbiComp)*. Pittsburgh, Pennsylvania: ACM, 2012, pp. 501–510. ISBN: 978-1-4503-1224-0. DOI: 10.1145/2370216.2370290. URL: http://doi.acm.org/10.1145/2370216.2370290.

[7] Lin Tan et al. "/* iComment: Bugs or Bad Comments? */." In: *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 2007, pp. 145–158.

[8] Einar W Høst and Bjarte M Østvold. "Debugging method names." In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2009, pp. 294–317.

[9] Rahul Pandita et al. "Inferring Method Specifications from Natural Language API Descriptions." In: *ACM/IEEE International Conference on Software Engineering (ICSE)*. Zurich, Switzerland, 2012. URL: http://www.cs.illinois.edu/homes/taoxie/publications/icse12-nlp.pdf.

[10] Yajin Zhou and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution." In: *IEEE Symposium on Security and Privacy (SP)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109. ISBN: 978-0-7695-4681-0. DOI: 10.1109/SP.2012.16.

[11] William Enck et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones." In: *USENIX conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–6. URL: http://dl.acm.org/citation.cfm?id=1924943.1924971.

[12] Xuetao Wei et al. "ProfileDroid: multi-layer profiling of Android applications." In: *ACM Annual International Conference on Mobile Computing and networking (MobiCom)*. Istanbul, Turkey: ACM, 2012, pp. 137–148. ISBN: 978-1-4503-1159-5. DOI: 10.1145/2348543.2348563. URL: http://doi.acm.org/10.1145/2348543.2348563.

[13] Cuixiong Hu and Iulian Neamtiu. "Automating GUI testing for Android applications." In: *International Workshop on Automation of Software Test (AST)*. Honolulu HI, USA: ACM, 2011, pp. 77–83. ISBN: 978-1-4503-0592-1. DOI: 10.1145/1982595.1982612. URL: http://doi.acm.org/10.1145/1982595.1982612.

[14] Wei Yang, Mukul R. Prasad, and Tao Xie. "A grey-box approach for automated GUI-model generation of mobile applications." In: *International Conference on Fundamental Approaches to Software Engineering (FASE)*. Rome, Italy: Springer-Verlag, 2013, pp. 250–265. ISBN: 978-3-642-37056-4. DOI: 10.1007/978-3-642-37057-1_19. URL: http://dx.doi.org/10.1007/978-3-642-37057-1_19.

[15] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: an input generation system for Android apps." In: *European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. Saint Petersburg, Russia: ACM, 2013, pp. 224–234. ISBN: 978-1-4503-2237-9. DOI: http://dx.doi.org/10.1145/2491411.2491450. URL: http://doi.acm.org/http://dx.doi.org/10.1145/2491411.2491450.

[16] Domenico Amalfitano et al. "Using GUI ripping for automated testing of Android applications." In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Essen, Germany: ACM, 2012, pp. 258–261. ISBN: 978-1-4503-1204-2. DOI: 10.1145/2351676.2351717. URL: http://doi.acm.org/10.1145/2351676.2351717.

[17] Vitalii Avdiienko et al. "Mining Apps for Abnormal Usage of Sensitive Data." In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE '15. Florence, Italy: IEEE Press, May 2015, pp. 426–436. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818808.

[18] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. "Latent Dirichlet Allocation." In: *Journal of Machine Learning Research* 3 (2002), p. 2003.

[19] *MALLET homepage*. http://mallet.cs.umass.edu/. (Accessed on 01/14/2019).

[20] J. MacQueen. "Some methods for classification and analysis of multivariate observations." In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Berkeley, Calif.: University of California Press, 1967, pp. 281–297. URL: https://projecteuclid.org/euclid.bsmsp/1200512992.

[21] Peter J. Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis." In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53 –65. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: http://www.sciencedirect.com/science/article/pii/0377042787901257.

[22] 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.

[23] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks." In: *Machine Learning*. 1995, pp. 273–297.

[24] Konstantin Kuznetsov et al. "Mining Android Apps for Anomalies." In: *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, Apr. 2015, pp. 257–281. ISBN: 978-0-1241-1519-4.

[25] Hans-Peter Kriegel et al. "Interpreting and Unifying Outlier Scores." In: Apr. 2011, pp. 13–24. DOI: 10.1137/1.9781611972818.2.

[26] Ryan Stevens et al. "Asking for (and About) Permissions Used by Android Apps." In: *IEEE Working Conference on Mining Software Repositories (MSR)*. San Francisco, CA, May 18–19, 2013, pp. 31–40.

[27] *Tasks — Luigi 2.7.6 documentation*. http://luigi.readthedocs.io/en/stable/tasks.html. (Accessed on 14/07/2018).

[28] *0xddom/dex2call: A simple script to get calls to android.jar from the bytecode*. https://github.com/0xddom/dex2call. (Accessed on 01/14/2019).

[29] *Elasticsearch: RESTful, Distributed Search & Analytics | Elastic*. https://www.elastic.co/products/elasticsearch. (Accessed on 01/14/2019).

[30] *gensim: Topic modelling for humans*. https://radimrehurek.com/gensim/. (Accessed on 01/14/2019).

[31] *Clustering - RDD-based API - Spark 2.4.0 Documentation*. https://spark.apache.org/docs/latest/mllib-clustering.html. (Accessed on 01/14/2019).