A Machine Learning Approach to Classifying Android Applications

Matthias Eichhaber

A Machine Learning Approach to Classifying Android Applications

Matthias Eichhaber B.Sc.

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Johannes Feichtner
Institute of Applied Information Processing and Communications (IAIK)

Graz, September 2019

Ein Machine-Learning-Ansatz zur Klassifikation von Android-Applikationen

Matthias Eichhaber B.Sc.

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Softwareentwicklung-Wirtschaft

an der

Technischen Universität Graz

Begutachter

Dipl.-Ing. Johannes Feichtner
Institut für angewandte Informationsverarbeitung und Kommunikationstechnologie (IAIK)

Graz, September 2019

Diese Arbeit ist in englischer Sprache verfasst.

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum	Unterschrift

Abstract

Google Play is the primary application distribution platform for Android. With its rise in popularity, malicious entities have started to use it as a means to spread malware by tricking end-users into downloading harmful applications. In order to mitigate this threat, automated detection mechanisms that apply machine learning techniques are used to flag potentially malicious applications for manual review. With the help of these techniques, it is possible to analyze application behavior, find patterns that are otherwise hard to identify, and study relationships between applications. One approach to studying these relationships is to employ semantic analysis.

The goal of this thesis is to identify semantic similarities between Android applications by evaluating different application-descriptive features. Doc2Vec, a natural language processing algorithm, is used to train models of document and word embeddings for applications using descriptions from Google Play, along with UI strings, layout markup code, and source code extracted from Android application packages. The resulting embeddings are evaluated based on their effectiveness in modeling the application's characteristics, by using the Doc2Vec similarity interface in addition to DBSCAN and k-Means clustering algorithms.

Results show that Doc2Vec models trained with description and source code features are able to create semantically meaningful embeddings. Document and word embeddings of description features can be used to find applications with a semantically similar purpose. Moreover, document embeddings of source code features are suited to spot clone applications and to detect malware applications based on textual source code similarity. The clustering of document embeddings from models trained with UI strings, layout markup code, and source code, however, yielded less useful results, since the fitted clusters did not show any semantically meaningful structures. While not all evaluated features appear to model the characteristics of applications successfully, embeddings derived from application description texts and source code could prove beneficial for Android application analysis, in particular when used alongside other application-specific features in downstream machine learning tasks.

Kurzfassung

Google Play ist die primäre Plattform um Applikationen für Android anzubieten. Mit zunehmender Popularität wird sie immer öfter zur Verbreitung von Malware missbraucht. Um diese Bedrohung zu minimieren, werden automatisierte Systeme, die Machine-Learning-Techniken verwenden, eingesetzt, um potenziell schädliche Applikationen zu erkennen. Mithilfe dieser Techniken ist es möglich, das Verhalten von Applikationen zu analysieren und sonst schwer identifizierbare Muster zu finden. Weiters können Beziehungen zwischen Applikationen untersucht werden. Ein Ansatz zur Untersuchung dieser Zusammenhänge ist die semantische Analyse.

Das Ziel dieser Arbeit ist es, semantische Ähnlichkeiten zwischen Android-Apps zu identifizieren, die auf verschiedenen Applikationsmerkmalen basieren. Untersuchte Merkmale umfassen App-Beschreibungen von Google Play, sowie UI-Strings, Layout Markup Code und Source Code, die aus Android-Programmpaketen extrahiert wurden. Mithilfe dieser Merkmale werden Modelle, die aus Document und Word Embeddings bestehen, generiert, und mit Doc2Vec, einem Algorithmus zur Verarbeitung natürlicher Sprache, trainiert. Die resultierenden Embeddings werden durch Einsatz des Doc2Vec Similarity Interfaces, sowie von DBSCAN- und k-Means-Clustering-Algorithmen hinsichtlich ihrer Effektivität bei der Modellierung der Applikationsmerkmale bewertet.

Die Ergebnisse zeigen, dass Doc2Vec-Modelle, die mit Beschreibungen und Source Code trainiert wurden, in der Lage sind, semantisch sinnvolle Embeddings zu erstellen. Document und Word Embeddings die auf Beschreibungen basieren, können verwendet werden, um Applikationen mit einem semantisch ähnlichen Zweck zu finden. Darüber hinaus eignen sich Document Embeddings von Source Code, um geklonte Applikationen und Malware basierend auf ihrer textuellen Source Code-Ähnlichkeit zu erkennen. Das Clustering von Document Embeddings aus Modellen, die mit UI-Strings, Layout Markup Code und Source Code trainiert wurden, führte jedoch zu weniger nützlichen Ergebnissen, da die trainierten Cluster keine semantisch sinnvollen Strukturen zeigten. Während nicht alle bewerteten Merkmale die Eigenschaften von Applikationen erfolgreich zu modellieren scheinen, könnten sich Embeddings von Beschreibungen und Source Code für die Android-Applikationsanalyse als nützlich erweisen, insbesondere wenn sie zusammen mit anderen anwendungsspezifischen Merkmalen in traditionellen Machine-Learning-Anwendungen verwendet werden.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Over the last decade, mobile applications have become an integral part of everyday life. In 2018, consumers downloaded a total of 205 billion applications, a figure that is projected to grow to over 258 billion downloads by 2022[1]. Applications are distributed via online marketplaces, most notably Google's Play Store and Apple's App Store. With their rise in popularity, those marketplaces have become the target of malicious parties trying to distribute malware disguised as benign applications. For this reason, both Google and Apple have instated guidelines for developers and audit processes to prevent malicious applications from being published. In the case of Google's Play Store, once developers have been reviewed and approved, they can submit applications to the Play Store. However, before applications are published, they are analyzed, scored, and approved or rejected. An automated detection system that applies machine learning techniques is employed to help Google flag potentially harmful applications for manual review [14]. Since introducing machine learning to their audit process in 2017, Google took down 700,000 apps they deemed malicious, which is a 70% increase to the previous year[2]. An overview of Google's review process can be seen in figure 1.1.

Nevertheless, there have been some applications that managed to slip through the cracks of Google's security measures. In early November 2017, news broke that a fake Whatsapp clone has been distributed via Google's official Play Store[3]. It has since been removed, but not before tricking over one million users into downloading it. Moreover, researchers at SophosLabs reported about a phishing attack leveraged by fake banking applications impersonating the State Bank of India, ICICI Bank, Axis Bank, and Citi Bank, that aim to trick the victims into entering banking or credit card information. As of July 2018, those too have been pulled from the Play Store, mainly in part to detection mechanisms that use latest machine learning techniques[4]. Overall it seems as though that machine learning techniques have become an integral part of Android application analysis tools employed by Google as well as independent research teams.

One of the emerging research topics in machine learning over the last few years is semantic source code analysis. Generating continuous representations of formal language texts such as source code or markup code might help in understanding semantic similarities between applications. Two mobile banking applications, for example, might have different codebases, but primarily work the same. After logging in, there is going to be a main form where past transactions and account balances are displayed. Additionally, there will usually be a form for making direct deposits or wire transfers. Could a continuous representation of the codebase or a part thereof be enough to categorize a mobile banking application as such? Could this be done for types of malware applications as well?

[1] statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads

[2] android-developers.googleblog.com/2018/01/how-we-fought-bad-apps-and-malicious.html

[3] thehackernews.com/2017/11/fake-whatsapp-android.html

[4] news.sophos.com/en-us/2018/10/21/fake-android-banking-apps-target-victims-in-india

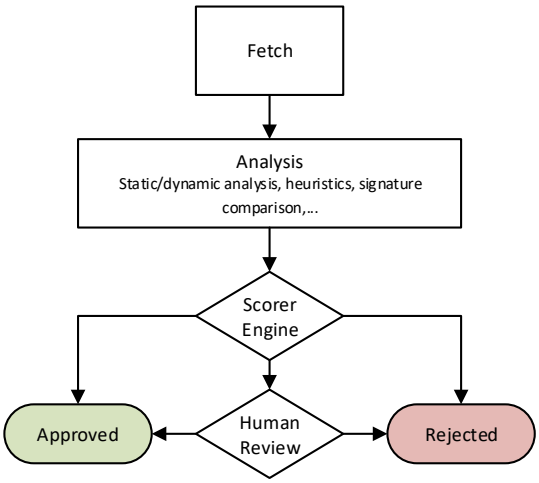


Figure 1.1: Overview of Google's review process for applications submitted to their Play Store. Based on a graphic from Security [14]

1.1 Objectives

The goal of this thesis is to identify patterns and similarities between Android applications with the help of semantic text analysis. By using machine learning algorithms, semantic text analysis tries to model mathematical representations of texts and the relationships between them. In theory, contextually similar texts should result in similar mathematical representations. While most of the research on semantic text analysis focuses on natural language texts, e.g. documents written in the English language, more and more research has been done to understand semantic relationships between formal languages such as source code. This thesis tries to evaluate what features, including both natural and formal language texts, of an Android application can be used to identify semantic relationships to other applications. Specifically, three main problems are tackled:

1. Can Google Play Store descriptions of applications be used for semantic analysis?
2. Can applications be semantically compared based on features found in APK files?
3. Is it possible to identify malware or clone applications based on Smali source code similarity?

To try and answer those questions, a machine learning algorithm called Paragraph Vector, also known as Doc2Vec, is utilized. Published by Le and Mikolov [7], Doc2Vec trains mathematical representations of texts, called document embeddings, with a shallow, two-layer neural network. Document embeddings map documents to n-dimensional vectors of real numbers. In a trained Doc2Vec model, documents with different contents but similar meaning produce document embeddings that are located in close proximity to each other in the vector space. Those embeddings can then be used for downstream learning tasks such as classification or clustering. A Python implementation of Doc2Vec, which is included in the Gensim library [12], is used to train document embeddings containing Android application metadata features as well as features contained in binaries inside an application's APK file. APK is short for Android Package and is a type of archive file, akin to zip files. It contains the compiled binaries as well as metadata information. Online marketplaces such as the Google Play Store attach a set of metadata to each application. In order to improve search functionality and user experience, applications are shelved into different categories and supplied with a description, a list of similar applications, pictures, user reviews and more, while also publishing the application's metadata found in its manifest file. While there is a plethora of information that can be used as features for analysis purposes, this thesis focuses on the following three types of metadata and three types of data found in APK files as shown in table 1.1.

Feature	Data source
Package name	Google Play Store metadata
Description	Google Play Store metadata
Category	Google Play Store metadata
UI strings	APK
Layout markup code (XML)	APK
Source code (Smali)	APK

Table 1.1: List of features

In order to collect a large amount of APKs and metadata, online resources have to be data mined. The most obvious place is the Google Play Store, since metadata can be easily mined from the application's Play Store HTML pages. However, it is not possible to directly download APK files from the Play Store. Third-party downloading sites, browser extensions, and APK repositories try to remedy this by providing alternative ways of collecting APK files. Yet, one major disadvantage of third party services is that most of them limit the amount of APKs that can be downloaded. Another way is to collect the data from online APK data dumps. Data dumps are usually provided as archive files or repositories hosted on websites like archive.org. While limited in size and scope, mining online data dumps should provide a more time effective way of collecting the desired quantity of metadata and APK files. After the needed amount of data is collected, the features have to be extracted from both metadata information and APK files. While the metadata features can be extracted in plaintext, source code, layout markup code, and UI strings are compiled into binaries and have to be decompiled beforehand. Since decompiling Android binaries into working Java code is not always possible, for instance, due to obfuscation, they instead can be baksmali'd to Smali source code using a tool called Apktool[5]. This results in human-readable code that is still true to the original Dalvik bytecode. It can be seen as an intermediate step between bytecode and Java source code. Layout markup code and UI strings, however, can be easily restored to their original form. The following figures 1.2 and 1.3 illustrate all the different sources of data in more detail. After extracting features, they are stored and pre-processed in order to improve the quality of the resulting document embeddings, which are subsequently trained using multiple Doc2Vec models. After the training process is finished, the resulting models are evaluated by using different evaluation methods, including word analogy analysis and downstream machine learning tasks such as clustering and classification. These results will help determine the usefulness of selected metadata and APK features for semantic text analysis purposes, specifically by identifying their effectiveness in solving the three aforementioned problems. This four-step process - data retrieval, data preparation, model training, and model evaluation is illustrated in figure 1.4.

[5] ibotpeaches.github.io/Apktool/

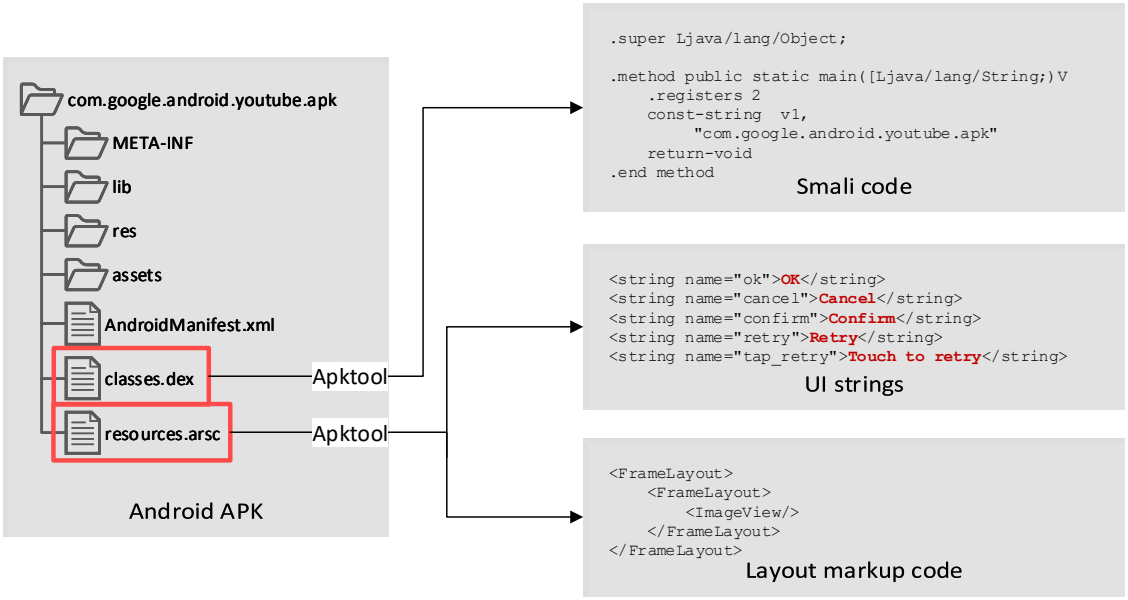


Figure 1.2: Overview of features extracted from an APK

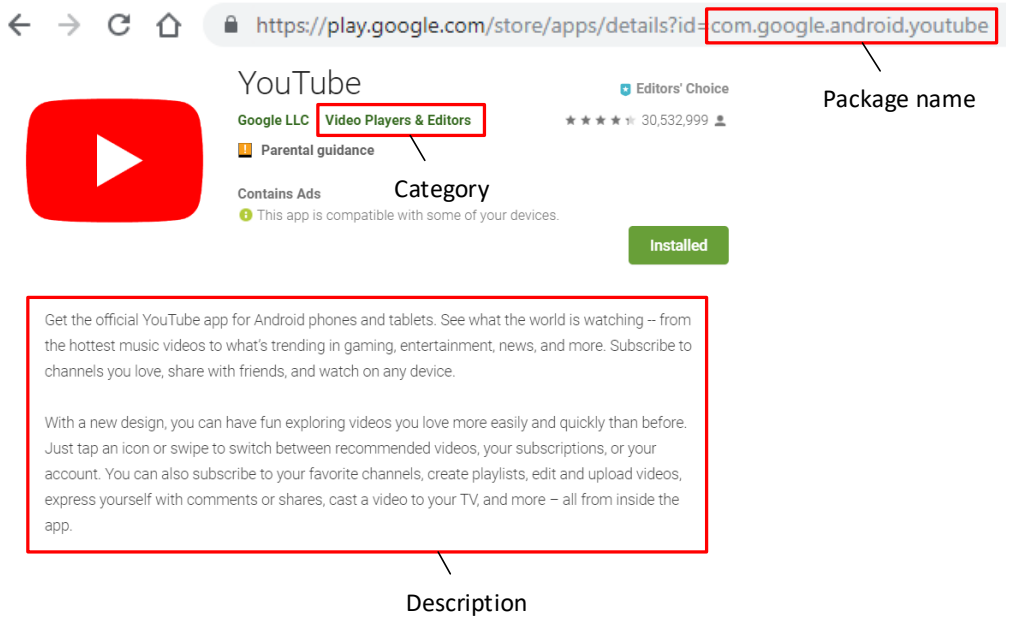


Figure 1.3: Overview of used Play Store metadata features

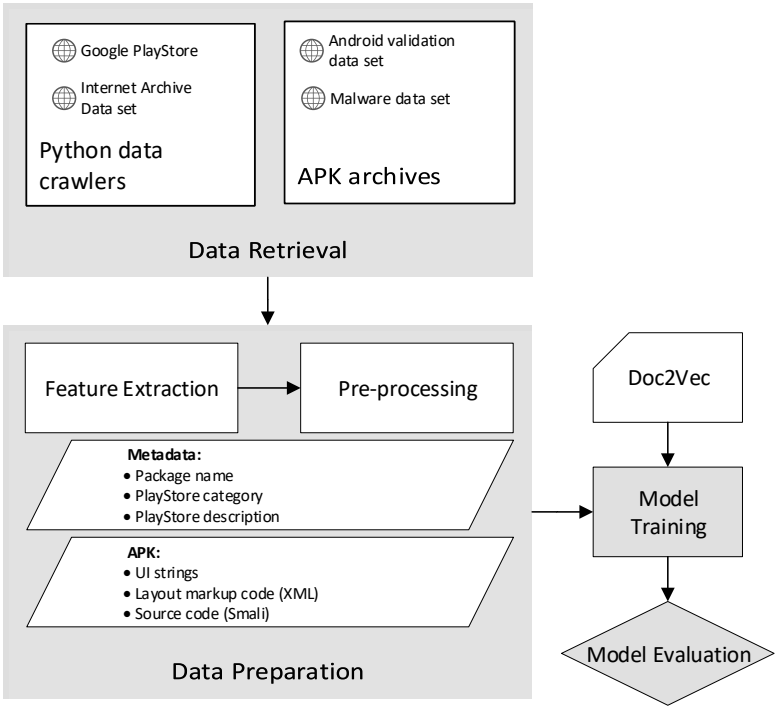


Figure 1.4: Overview of the four step process used to train Doc2Vec models

1.2 Outline

The thesis is structured as follows: after presenting related work in chapter 2, covering works on semantic text analysis, semantic analysis of Android applications, and semantic analysis of source code embeddings, background information relevant to the topic at hand is outlined in chapter 3. Word2Vec and its extension Doc2Vec, the Android application structure, and the Smali syntax are explained during this chapter. Afterwards, the approach, including the first three steps of the four-step process, is laid out in chapter 4. The first step of the process, which is described in 4.1, called data retrieval, encompasses the functionality of the data crawler bots, as well as the structure and origin of the different data sets. The next step, data preparation, is explained in 4.2. This section is comprised of feature extraction and feature pre-processing processes. The model training is outlined in 4.3 and discusses the different hyperparameters and architectures of the Doc2Vec algorithm. Finally, the last step, called model evaluation, is described in chapter 5. This chapter includes detailed explanations of the trained Doc2Vec models in 5.1, as well as methods that were used to evaluate said models, which are explained in 5.2. After finishing describing the four-step process, the results for the three research problems, as well as more general findings are discussed in chapter 6, before a conclusion is drawn in chapter 7.

Chapter 2

Related Work

This chapter concentrates on the established work that has been published on semantic text learning that relates to this thesis. While most of the research on semantic learning focuses on natural language texts, e.g. documents written in the English language, more and more research has been done to understand semantic relationships between formal languages such as source code. First, works using natural language text learning will be discussed. Then, semantic learning with regards to Android application analysis is described, after which works using source code as a basis for semantic analysis will be examined.

2.1 Semantic Text Analysis

Semantic text analysis deals with finding semantic relationships between natural language text. One method to compare texts is to convert them to numerical representations. Those representations are called document or word embeddings, depending on whether they model entire texts/paragraphs/sentences or just one word, and consist of n-dimensional vectors. One method to create document and word embeddings is called Paragraph Vector, which was first introduced by Le and Mikolov [7] and is the basis for the Doc2Vec algorithm that is used in this thesis. In 2015, Dai et al. [2] evaluated document embeddings that were trained using this algorithm in an unsupervised learning task. Experiments were conducted, using two publicly available text corpora, namely a repository of scientific papers from arXiv and articles from the online encyclopedia Wikipedia. The resulting document embeddings were then evaluated by comparing them to other embedding methods, such as LDA and bag of words. Results showed that their document embeddings perform significantly better in terms of accuracy than embeddings created with other methods. They also argued that vector operations can be performed on document embeddings to produce semantically meaningful results.

One of the most prominent areas of semantic text learning is analyzing sentiment in social media postings. Markov et al. [8] identified author demographics based on their writings on social media such as blogs, review sites, and Twitter. They used a Doc2Vec model to learn document embeddings and evaluated their performance on the PAN AP 2014–2016 corpora. They found, that Doc2Vec-learned features consistently outperformed the baseline features when used with a logistic regression classifier. Moreover, they also outperformed state-of-the-art approaches when used with certain settings. Since social media postings are usually limited in size, for example, Twitter's 280 character limit, this shows that Doc2Vec should be suitable for Google Play Store descriptions of Android applications, a feature that is used in this thesis to find semantically similar applications.

Trieu et al. [16] proposed TD2V, a pre-trained document embedding model based on Doc2Vec, which can be used to encode arbitrary text documents written in English. This model was trained on 422,351 news articles from four different data sets and evaluated by applying classification tasks and comparing the results to different baseline methods. They found that their approach increased the accuracy of all evaluated

data sets. Furthermore, they proposed ANTENNA, an online news analysis and visualization tool, which provides an API to create new document embeddings, as well as cluster, and classify documents. Moreover, it can be used to search for related articles based on known locations or persons within a document.

2.2 Semantic Analysis of Android Applications

Assessing relationships between Android applications can be accomplished by analyzing distinct characteristics of a set of applications and modeling them as continuous representations. Those representations are then often used in downstream machine learning tasks like clustering or classification algorithms, to tackle various problems that have emerged due to the rise of popularity of mobile applications.

DroidKin is a tool for assessing relationships between Android Applications and was created by Gonzalez et al. [5] at the Canadian Institute for Cybersecurity at the University of New Brunswick. It is able to predict similarities between applications based on a set of features containing binary data and metadata. Metadata features are divided into descriptive features, such as certificate information and hash values, and numerical features, like the number of internal resource files by resource type, as well as file sizes. Binary features like bytecodes and opcodes found in the DEX file of an APK are used to create n-grams. Relations between applications are assessed with the following categories, with decreasing order of closeness: Twins, Siblings, False siblings, Step siblings, False step siblings, and Cousins. The exact relationship is determined by a filtering process using different similarity values. After testing their approach on a set of manually prepared applications, which are also used in the evaluation process of this thesis, it was used to find similarities in the following three data sets: The Android Malware Genome Project, Drebin, and DroidAnalytics. They found several relationships of all types between applications. Finally, they assessed DroidKin by using it with 8,000 applications from the Google Play Store and Virus Total service. They found 11 relationships with applications from the Google Play Store and 206 from Virus Total. They argued that DroidKin is an efficient and robust way to determine similarities between applications and might even be leveraged to identify malware applications by determining similarities to known malware samples.

With their rise in popularity, application marketplaces have become the target of malicious parties trying to distribute malware disguised as benign applications. Naturally, a lot of research has been done to study automated systems that can detect malicious applications. While this thesis tries to classify Android malware by learning formal language text, a different approach was proposed by Killam et al. [6], which uses unreferenced string literals to detect Android malware. Unreferenced string literals are strings found in the Android executable that are not referenced by the application's source code. These strings literals were learned using a linear SVM with line-bounded word-level 3-grams. They evaluated their approach with The Android Malware Genome dataset and a collection of 4,000 benign apps from the Google Play Store with an accuracy of 99.3% for malware detection and a false-positive rate of 0.5% as well as a 99.2% accuracy and 2.0% false-positive rate for malware family classification.

One of the questions this thesis tries to answer is, if natural language features of an application can be used to assess similarities between Android applications. Similar to that, Yoon et al. [20] proposed a clustering model that automatically determines clusters based on title keywords of applications from the Google Play Store. 2.1 million applications were crawled and used as a training set in a semi-supervised clustering algorithm. Clusters were initialized using title keywords, that were extracted from the actual application's title. Title keywords and application descriptions where then used as a training set to learn document embeddings using Doc2Vec. After training, the resulting document embeddings were used to expand the initial clusters and merge semantically similar ones. Compared to the k-means algorithm, their proposed model increased in purity by 0.19% and decreased 1.18% in entropy. More importantly, however, by using Doc2Vec, they could improve the performance of their model, mostly due to avoiding constructing a synonym dictionary.

In 2018, Narayanan et al. [10] proposed apk2vec, a semi-supervised machine learning algorithm that

creates continuous representations for a given Android application. After disassembling APK files, three distinct dependency graphs are generated and subsequently used, alongside label information to create one embedding per APK. They evaluated their approach by using multiple supervised, semi-supervised, and unsupervised analysis tasks on more than 42,000 Android applications. Their algorithm achieved improvements over other similar approaches in terms of accuracy as well as efficiency. This approach is closely related to Word2Vec, and in extent Doc2Vec, since it also uses skip-gram as the basis of the neural network architecture.

2.3 Semantic Analysis of Source Code

One of the emerging research topics in machine learning over the last few years is using machine learning algorithms to learn source code embeddings. In 2018, Alon et al. [1] created code2vec, a neural network model for representing snippets of code as *code embeddings*. A code embedding maps a code snippet to a single fixed-length vector which can then be used to predict semantic relationships to other code snippets. The model uses a soft-attention mechanism over a snippet's syntactic paths, that are derived from the Abstract Syntax Tree, and aggregates vector representations for all paths into a single vector. They demonstrated their model by predicting a method's name from the vector representation of its body and evaluated their approach by training 14 million methods. Their model outperformed other similar approaches not only for precision and recall but also performance, predicting 1000 examples per second, whereas the next best approach predicted ten examples per second. Furthermore, code2vec was able to predict method names for previously unobserved code snippets.

This thesis uses Doc2Vec models to train document embeddings of Smali source code files, similar to researchers at Lab41, a Silicon Valley-based think tank, who used Doc2Vec to assess similarities in source code written in Python[1]. They used 100,000 and 500,000 scripts as training sets for two Doc2Vec models with different training algorithms, distributed memory (PV-DM) and distributed bag of words (PV-DBOW). The resulting models were then evaluated by using 907 source code submissions from multiple authors that were part of the Meta Kaggle dataset and classified into 25 Kaggle competition classes. Results showed that the learned Doc2Vec models outperformed other state-of-the-art algorithms with the Doc2Vec model trained using the PV-DBOW algorithm slightly outperforming the model trained using the PV-DM algorithm when trained on only 100k scripts. Furthermore, the models were able to predict semantically similar source code words and even perform simple vector arithmetic, for example, 'try' - 'except' + 'if' = 'else'. They concluded that Doc2Vec was indeed capable of generating similar vector representations of semantically similar Python scripts.

Theeten et al. [15] proposed a way to create library embeddings from import statements found in the source code of Java, Javascript, and Python projects, which are able to model semantic relationships between different source code libraries. They adapted the Word2Vec skip-gram model [9] and trained co-occurrence patterns of import statements from different source code languages. To evaluate their model, they collected source code from different online repositories such as GitHub and public package repositories and clustered the resulting library embeddings based on their domain or platform. When projecting the embedding space into two dimensions, they could find similarity clusters of libraries that were frequently used together. Moreover, they successfully applied analogical reasoning to library embeddings, analogous to the famous example 'King' - 'Man' + 'Woman' = 'Queen' used by Le and Mikolov [7] in their work, that served as a basis for Doc2Vec.

Apart from Doc2Vec, different machine learning approaches have been used to create code embeddings. Finding similarities between continuous embeddings of source code using deep learning was studied by Tufano et al. [17]. They used different representations of source code, namely identifiers, abstract syntax

[1] gab41.lab41.org/doc2vec-to-assess-semantic-similarity-in-source-code-667acb3e62d7

trees, control flow graphs, and bytecode to create features for a custom deep learning algorithm. This was done for different levels of granularity, e.g. classes and methods. To detect similarities between source code fragments, they measured the Euclidean distances between their embeddings. They evaluated their approach by using performance and classification metrics for detecting code clones for each source code representation. They concluded that their approach was an efficient way to detect clones, particularly when using multiple representations combined.

Efstathiou and Spinellis [3] created vector space models for six popular programming languages, namely Java, Python, PHP, C, C++, and C#. Those vector space models contain distributed code representations, that were created using a skip-gram training algorithm included in the fastText library. Over 13,000 repositories and over 8 million source code files were processed during the creation of the models. They argued that the created vector space models can be used to address a variety of problems present in software engineering. One of those applications is identifying semantic errors, where a snippet of code is doing something different than its intended purpose. They also argued that the models can be useful in auto-completion or auto-correction processes employed by various programming tools. They described the main challenges as finding the correct input format for optimizing the quality of the code representations and overcoming the limitations of semantic analysis tools that were originally created for natural language texts.

2.4 Summary

Finding semantic relationships between words or texts can be accomplished by comparing their numerical representations, which are called word or document embeddings. Different algorithms exist that are able to create those embeddings. Dai et al. [2] evaluated Paragraph Vector and compared the performance of the resulting document embeddings to other document embedding methods. Their results showed that Paragraph Vector consistently outperformed the other methods. They also showed that vector arithmetic could be used to produce meaningful results based on multiple document embeddings. One popular area of research in semantic text analysis is analyzing sentiment in social media postings. Markov et al. [8] used Doc2Vec to profile author demographics based on social media postings, whereas Trieu et al. [16] proposed TD2V, a document embedding model based on Doc2Vec that was trained using news articles. With the help of this model, they built ANTENNA, an online API to analyze similarities in news articles and search for similar documents based on known locations or persons within the article.

Sentiment analysis has also been used to study relationships between Android applications. Gonzalez et al. [5] created DroidKin, a tool to determine exact relationships between applications. They used binary data and metadata to categorize the exact relationship status into six categories with decreasing order of closeness: Twins, Siblings, False siblings, Step siblings, False stepsiblings, and Cousins. Killam et al. [6] used unreferenced string literals to classify malware applications, and Yoon et al. [20] used the application's title keywords and descriptions to cluster them into more accurate categories. More recently, Narayanan et al. [10] created apk2vec, a semi-supervised neural network, that learns dependency graphs of features extracted from APK files and creates continuous embeddings for them. They show that the resulting embeddings, when used in different analysis tasks, outperform similar approaches in terms of accuracy and efficiency.

Using semantic text learning to analyze relationships between formal language texts is an emerging research area. One of its topics is trying to create meaningful code embeddings, i.e. continuous representations of source code that can be used to semantically compare applications, libraries, classes, or even methods. Alon et al. [1] used a neural network model to create code embeddings of methods to predict fitting method names. Doc2Vec, which is used in this thesis and was originally intended to learn document embeddings of pieces of natural language texts such as sentences and documents, has since been adapted and used with formal language texts. Lab41 used Doc2Vec to create embeddings of Python scripts, whereas Theeten et al. [15] used one of the underlying architectures of Doc2Vec, the Skip-gram model, to create embeddings

for source code libraries. Continuous embeddings using different representations of source code was studied by Tufano et al. [17]. They used identifiers, AST, CFG, and bytecode representations to learn embeddings using deep learning algorithms. Models using one specific representation and models using combined representations were evaluated by measuring their effectiveness and performance in spotting cloned code. Efstathiou and Spinellis [3] created vector space models of distributed code representations for six popular programming languages. They argue that those models can be used in a variety of software engineering-related tasks, like identifying semantic errors, auto-completion, and auto-correction.

Chapter 3

Background

The following chapter focuses on the background knowledge that supplements this thesis. To explain Doc2Vec and how its training process works, Word2Vec, which acts as a basis for Doc2Vec, is described beforehand. The core concepts of Word2Vec - word embeddings, the Skip-gram, and Continuous Bag-of-Words model architectures, as well as optimization techniques, will be described in section 3.1, before explaining Doc2Vec and how it extends the functionality of Word2Vec in section 3.2. The structure and contents of Android packages, which are used as a source for extracting features for the Doc2Vec models will be explained afterwards in section 3.3. Finally, the Smali language will be explained in detail in section 3.4. Smali classes, which can be created by disassembling compiled source code found inside an Android package, act as a feature when training Doc2Vec models.

3.1 Word2Vec

Word2Vec was created by Mikolov et al. [9] at Google. It takes a text corpus as input and trains n-dimensional vectors, also known as word embeddings. Word embeddings are numerical vector representations of words that can be used in a wide variety of machine learning applications. In a trained Word2Vec model, contextually similar words result in word embeddings which are located in close proximity to each other in the vector space. In other words, contextually similar words have a similar vector representation. Word2Vec utilizes two different types of model architectures as a basis to train those word embeddings, Skip-gram, and Continuous Bag-of-Words. In an effort to reduce the size of the resulting neural network and the computational burden of the training process, Word2Vec augments these models with two techniques, subsampling frequent words and negative sampling. A high-level overview of the Word2Vec workflow is illustrated in 3.1.

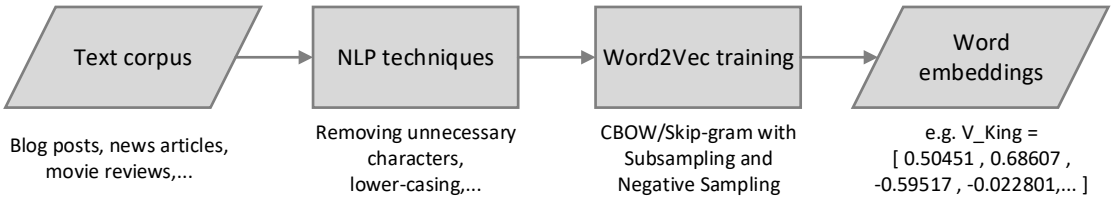


Figure 3.1: Overview of the word2vec process and workflow

3.1.1 Word Embeddings

Word embeddings, also called trained word vectors, map words to n-dimensional vectors of real numbers. Language modeling algorithms such as Word2Vec use pre-trained word vectors to train a model that produces word embeddings that have similar representations for contextually similar words. Those word embeddings are often used as a basis for other machine learning tasks such as clustering or classification. Moreover, relations between two or more word embeddings can be calculated by using simple vector arithmetic. This means that a model of word embeddings is effectively able to calculate word analogies. The most famous example that shows this property is the formula:

```
'King' - 'Man' + 'Woman' = 'Queen'
```

The resulting word embedding of 'King' - 'Man' + 'Woman', provided the model was trained with a sufficiently large corpus of text, will be close in proximity to the actual word embedding for the word 'Queen'. Figure 3.2 shows a basic illustration for this example with word embeddings in 2-dimensional vector space. In practice, however, word embeddings can have around 50 to over 500 dimensions, depending on the task.

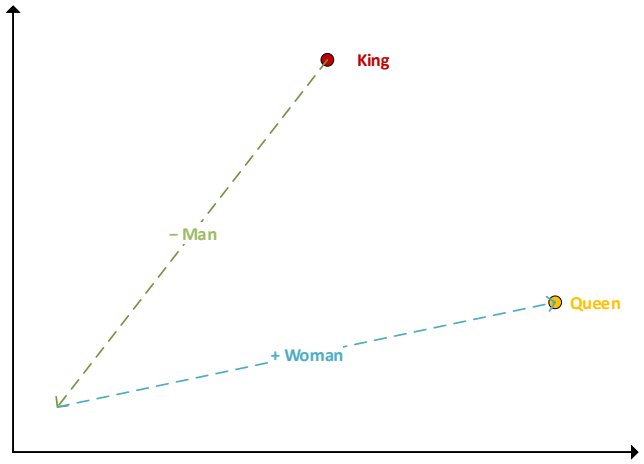


Figure 3.2: Vector arithmetic in 2-dimensional vector space for 'King' - 'Man' + 'Woman' = 'Queen'

To create those n-dimensional word embeddings, Word2Vec trains a neural network with either the Skip-gram or the Continuous Bag-of-Words model in conjunction with optimization techniques, subsampling frequent words and negative sampling, that reduce the computational burden of the training process and reduce the overall size of the neural network. After the process is completed, the weights of the hidden layer of the neural network are used to extract the word embeddings. The detailed processes for each model and the aforementioned optimization techniques are described in the following sub-sections.

3.1.2 Skip-Gram Model

The skip-gram neural network model is a shallow 2-layer neural network with a hidden layer and an output layer. Word pairs found in the training text are used as input and target words for the training process of the neural network. Word pairs are created by scanning the text corpus using a sliding window of size C, where the input is the word x, and the target words are neighboring words $x \pm [1...C]$. Figure 3.3 illustrates

the word pair generation with a sliding window of size 2 on a sample sentence. Since the neural network can not directly use text from word pairings as an input, the words have to be converted to numerical vectors. Input and target words are represented by one-hot vectors with as many dimensions as there are unique words in the training text (i.e. the vocabulary). The vectors are initialized with all 0s apart from 1 at the unique position for the corresponding word.

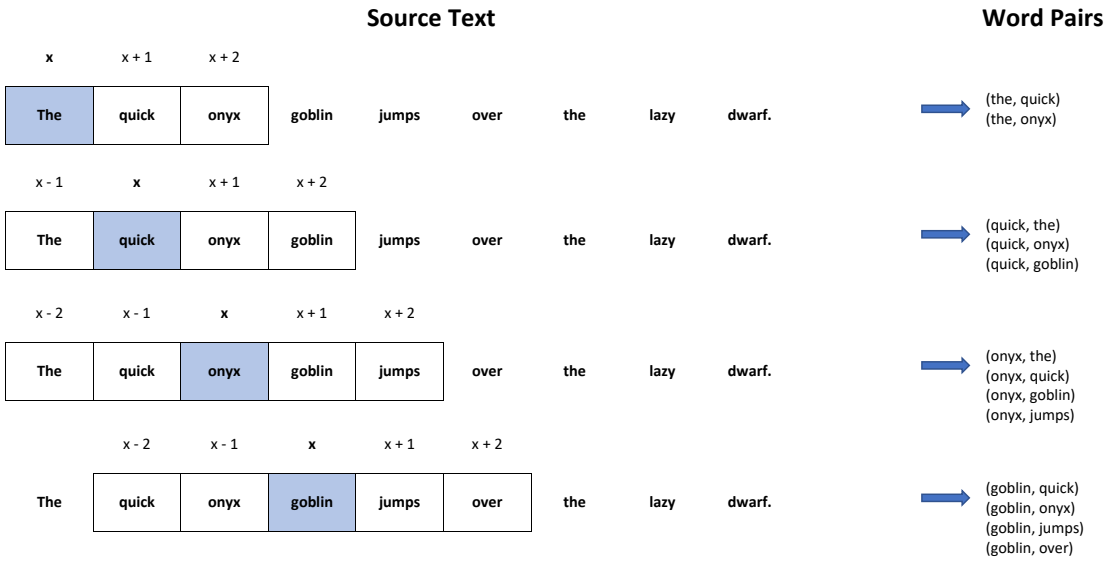


Figure 3.3: Word pair generation with a sliding window of size 2

The following figure 3.4 shows the architecture of the Skip-gram model. The input layer consists of the input word encoded as a one-hot vector. $\{y_1,...y_c\}$ are the target words for the specific input word selected from the training samples, also one-hot encoded. The weight matrix W connects the input layer and the hidden layer. The n^{th} row of this weight matrix corresponds to the n^{th} word in the vocabulary. Each target word also has an associated vector stored in weight matrix W'. The number of nodes of the hidden layer determines the dimensionality of the word embeddings and can be adjusted to fit the task.

At the start of the training phase, the still untrained model initializes the weight matrices W and W' with random values. After that, the word pairs are sequentially fed to the network. Since the input layer is a one-hot encoded vector, the hidden layer will only be affected by the row of the weight matrix W that corresponds to the index of the non-zero value in the input one-hot vector. The output probability distribution is calculated via a softmax function displayed in figure 3.5. The difference between the resulting probability distribution and the expected output, the one-hot vector of the target word, is then used to compute the error gradient with respect to W and W'. Both matrices are then corrected in the direction of this gradient, so that next time the model is less likely to guess the wrong target word. This technique is called stochastic gradient descent. This task of propagating the input values forward to the output layer and propagating the resulting error gradient back to the model is repeated for all word pairs. Figure 3.6 shows an example training step. After one epoch is finished, i.e. all word pairs have been fed through the network once, this process is repeated. The number of epochs can be adjusted to fit the specific task. After the model is finished training, the input-hidden layer weight matrix W contains all the word embeddings as rows.

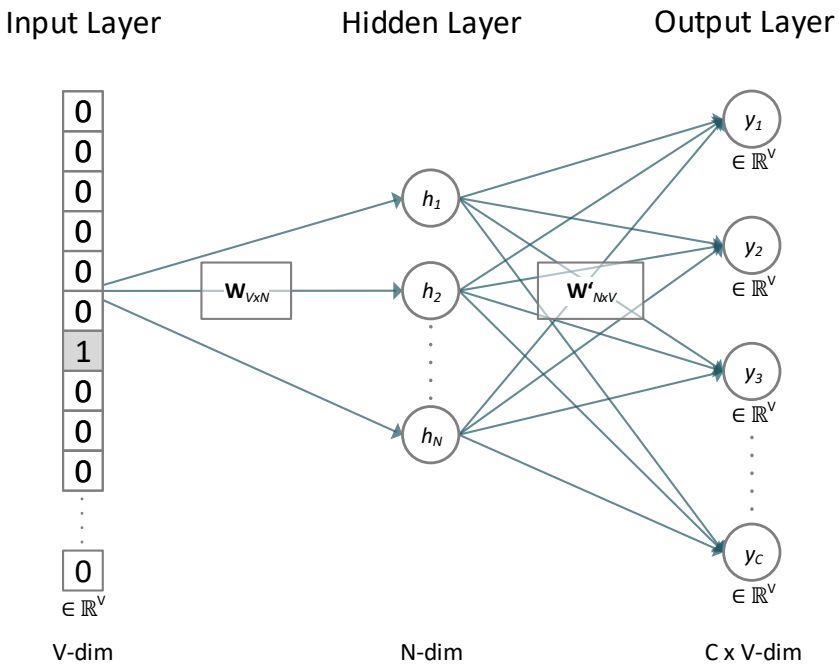


Figure 3.4: 2 layer neural network with Skip-gram architecture. C = number of target words; V = number of words in vocabulary; N = hidden layer neurons

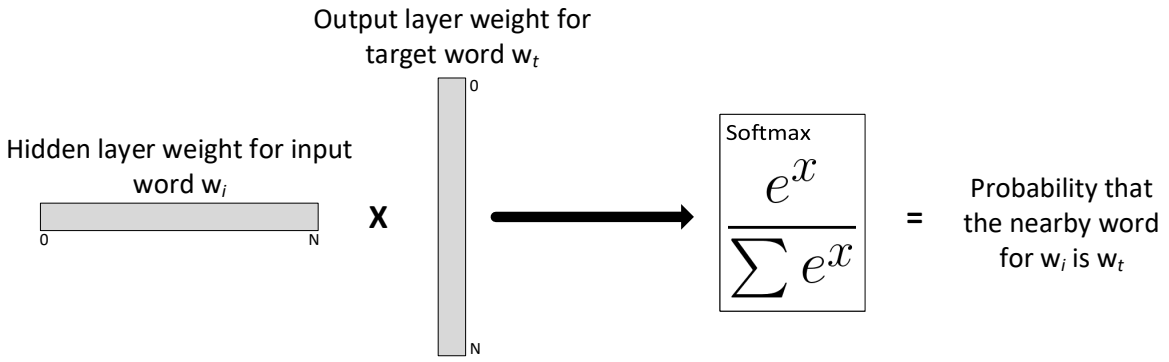


Figure 3.5: Example of calculating an output layer neuron using a softmax function.

3.1.3 Continuous Bag-of-Words Model

The second model architecture is called Continuous Bag-of-Words. In many ways, it is the reverse approach to Skip-gram. Instead of predicting the neighboring words based on the current input word, Continuous Bag-of-Words predicts the current word based on its surrounding words. The architecture of the model shown in figure 3.7 is the reverse of the Skip-gram model. The input layer consists of one-hot vectors $\{x_1,...x_c\}$ of the surrounding words where C is the sliding window size that is used to generate word pairs (see figure 3.3). The input layer and hidden layer are connected via the weight matrix W, whereas weight matrix W' connects the hidden layer and the output layer. The only difference between Continuous Bag-of-Words and Skip-gram, other than being the reverse approach, is the computation of the output h of the hidden layer. With Skip-gram, the output is only affected by one row of the input-hidden

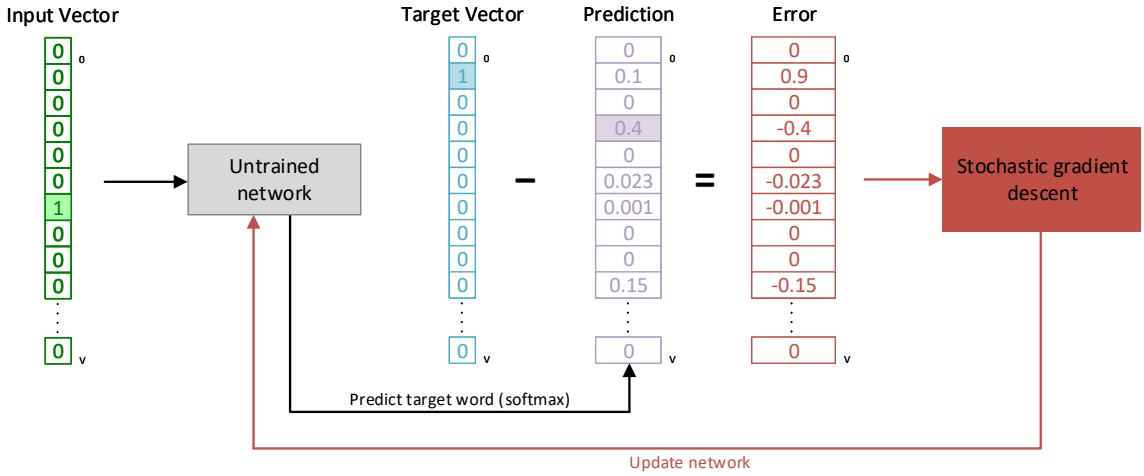


Figure 3.6: A training step for one word pair. Notice, that the network predicts the wrong word, which affects the error gradient that is used to update the network.

layer weight matrix, since the input is a single one-hot vector. However, with Continuous Bag-of-Words, the input layer consists of multiple one-hot vectors, so the output of the hidden layer h must be calculated by averaging the sum of all input vectors weighted by the input-hidden layer weight matrix W. The formula for this calculation is

$$h = \frac{1}{C} W \cdot (\sum_{i=1}^{C} x_i) \tag{3.1}$$

After the calculation of the hidden layer output h, the process continues analogous to Skip-gram. The output prediction is checked against the expected output, and the error gradient is propagated back to the model via stochastic gradient descent.

However, regardless of the aforementioned models, using stochastic gradient descent on a neural network that large (e.g. 10,000 words x 300 neurons = 3,000,000 weights for W and W' respectively) is going to be slow. Moreover, a lot of training data is needed to avoid overfitting the model. Due to this problem, Word2Vec implements subsampling of frequent words and negative sampling to not only improve the performance of the training process but also improve the quality of the resulting word embeddings.

3.1.4 Subsampling

In figure 3.3 the word pair generation with a window size of 2 is visualized. However, there is a problem with common words such as 'the'. They appear very frequently and seldom tell much about the meaning of the context words. Usually, a small subsection of those common words is enough to learn a decent word embedding. Word2Vec addresses this problem by implementing a subsampling algorithm. For each word, that is encountered in the training text, there is a chance that it is deleted from the text. The higher the frequency of the word, the higher the probability that the word is removed. Each word w_i in the training text is removed with a probability that is calculated with the formula

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{3.2}$$

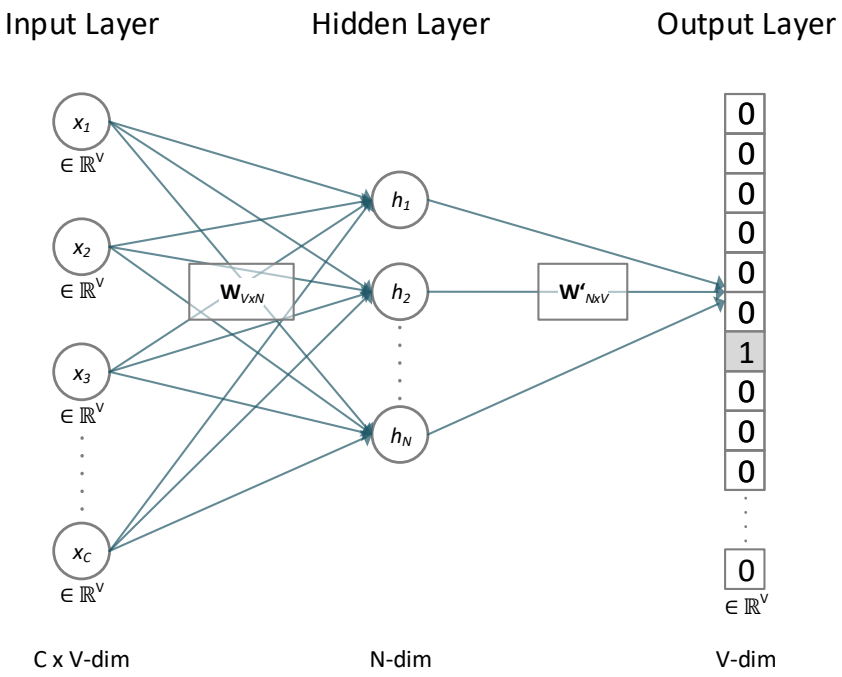


Figure 3.7: 2 layer neural network with Continuous Bag-of-Words architecture. C = number of surrounding words; V = number of words in vocabulary; N = hidden layer neurons

where $f(w_i)$ is the frequency of the word w_i occurring in the training text, and t is a threshold value, typically 0.001. In the C implementation of Word2Vec, this formula is implemented a bit differently, with

$$P'(w_i) = (\sqrt{\frac{f(w_i)}{0.001}} + 1) \times \frac{0.001}{f(w_i)} \tag{3.3}$$

where $P'(w_i)$ is the probability that the word is kept in the training text. The effect of this subsampling approach is that it accelerates the training process and improves the quality of word embeddings for rarer words.

3.1.5 Negative Sampling

Training a big neural network with lots of training data can be quite computationally expensive. With Word2Vec, the training corpus usually exceeds tens of thousands of words, and the hidden layer contains hundreds of neurons, which means that every gradient descent step has to update millions of weights. To address this problem, Word2Vec implements a technique called negative sampling. Rather than updating all weights on each training step, only a small percentage of weights are actually updated. In the case of Skip-gram, only the weights for the neuron that guessed the target word correctly and a small subsection of neurons that guessed wrong words are updated. This subsection of wrong words is randomly selected using a unigram distribution function U(w) raised to the $3/4^{th}$ power, computed by the formula

$$U(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n}(f(w_j)^{3/4})} \tag{3.4}$$

wherein more frequent words are more likely to be used as negative samples than rarer words. Again, the function f(w) denotes the frequency of the word w in the text corpus. Only updating a small percentage of

all weights speeds up the training process. In the original paper, Mikolov et al. [9] suggest that selecting 5-20 negative words for smaller datasets works well, while for large datasets, the value can be as small as 2-5. This means that in a neural network with 10,000 words and 300 neurons, using negative sampling with ten negative words reduces the number of weights that need to be updated from 3 million to only 3,000.

3.2 Doc2Vec

Doc2Vec, also known as Paragraph Vector, is an extension to the Word2Vec model developed by Le and Mikolov [7]. In addition to creating word embeddings, Doc2Vec also creates document embeddings, which are numerical representations of entire documents in vector space. Naturally, the same principles that apply to word embeddings also apply to document embeddings. In a trained Doc2Vec model, documents with similar sentiments create document embeddings whose vectors are located in close proximity to one another. Doc2Vec adapts both Skip-gram and Continuous Bag-of-Words architectures from Word2Vec and adds another input layer feature: a document vector. Both adapted models are explained below.

3.2.1 Distributed Memory Model (PV-DM)

Figure 3.8 shows the PV-DM model which is based on a Word2Vec model with Continuous Bag-of-Words architecture seen in figure 3.7. In addition to using the context words to predict the target word, PV-DM also provides a document vector. After finishing training the model, the new weight matrix D contains the document embeddings while the weight matrix W continues to provide word embeddings.

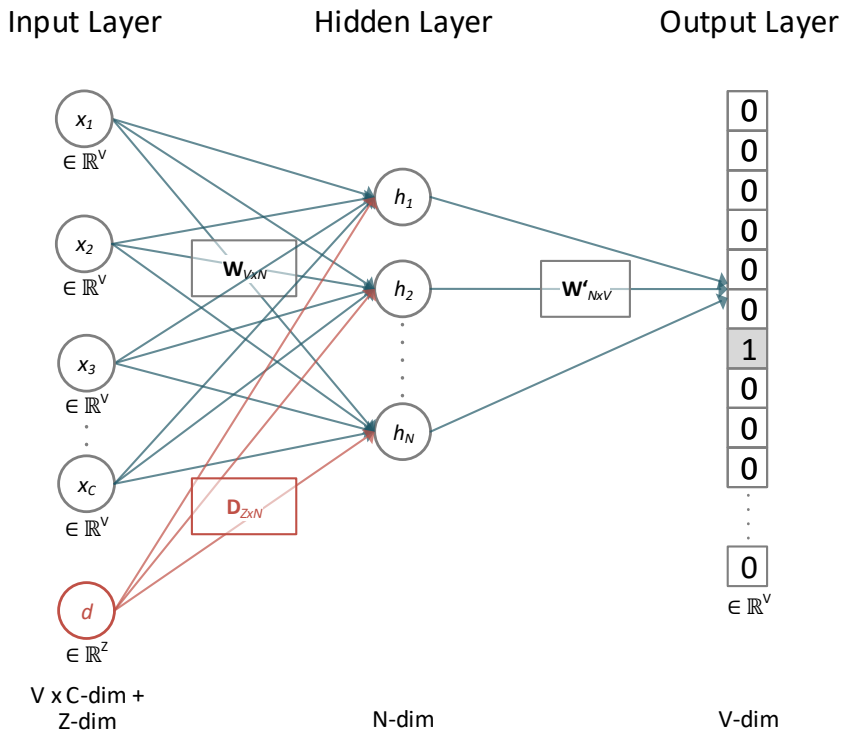


Figure 3.8: Doc2Vec model with PV-DM architecture. Z = number of documents in corpus.

Each document has a document vector embedded in the weight matrix D that is shared for all context

words that are generated from this document. The weight matrix W for word embeddings is shared across all documents. This means that the word embeddings for each unique word of the entire vocabulary are the same for all documents. Document embeddings and word embeddings are trained using stochastic gradient descent that is propagated back from the output layer to the weight matrices D, W, and W'. After training, the document and word embeddings can be used as features for other conventional machine learning algorithms like clustering or classification algorithms. While being computationally more expensive than PV-DBOW, PV-DM trains both document and word embeddings, whereas PV-DBOW only trains document embeddings.

3.2.2 Distributed Bag-of-Words Model (PV-DBOW)

The Distributed Bag-of-Words Model, short PV-DBOW, is based on the Skip-gram architecture from Word2Vec. Figure 3.9 illustrates the adapted model. In comparison to a Word2Vec model with Skip-gram architecture, the entire input layer is replaced by the document vector, and the input-hidden layer weight matrix W is replaced by the document weight matrix D that contains the document embeddings. In contrast to the PV-DM model, PV-DBOW only trains document embeddings, which speeds up the training process significantly.

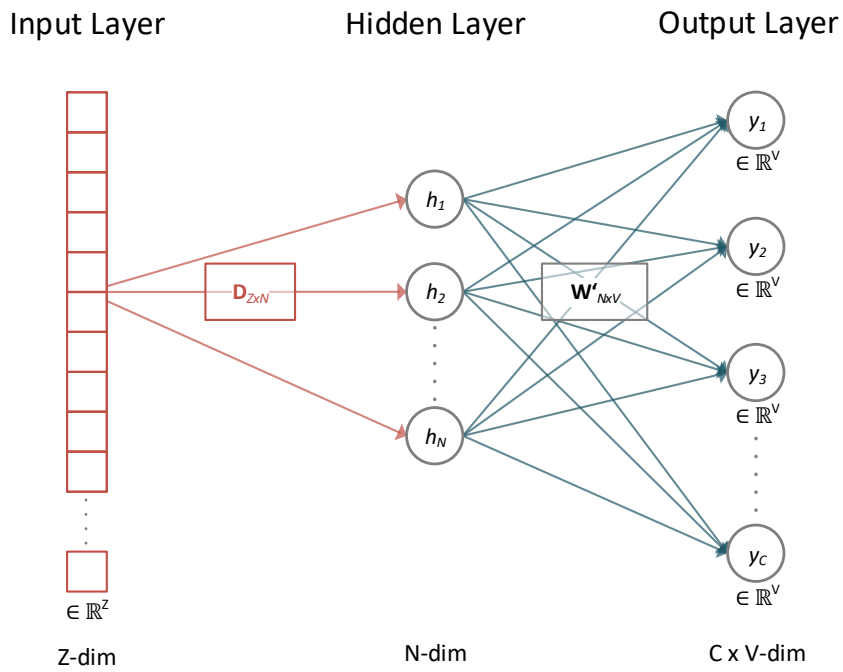


Figure 3.9: Doc2Vec model with PV-DBOW architecture. Z = number of documents in corpus.

For each training step, a subsection of text is selected, and a random word of that subsection is used in a classification task given the document vector. The error is calculated using stochastic gradient descent and propagated back to the weight matrices. After training, only the weight matrix D containing the document embeddings is stored, which means this model requires fewer data and is also faster while training. However, in practice, both models can be used in conjunction to produce word embeddings while still learning document embeddings with the PV-DBOW model.

3.3 Android Application Structure

Android Package, short APK, is the file format used by Android to distribute and install mobile applications. APKs are archive files, similar to .zip files based on the JAR file format and have .apk as the filename extension. By default, APKs can only be installed if they are downloaded from an official source such as the Google Play Store. However, it is possible to install APKs from unofficial app stores or directly from a desktop computer by changing specific settings in the device's security settings menu. An APK usually contains the following files and folders:

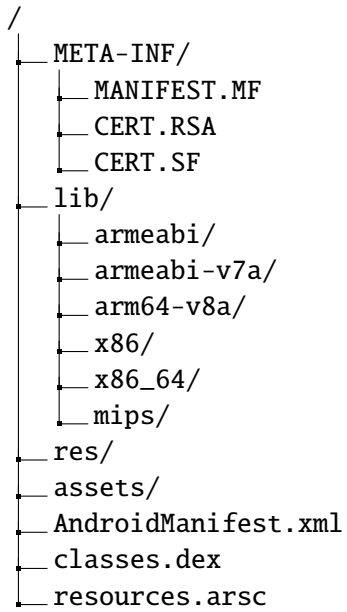


Figure 3.10: Files and folders of an APK file

The META-INF directory contains files that are recognized by the Java Platform to configure the application. It contains the `MANIFEST.MF` file which contains SHA-1 hashes of each file in the application. `CERT.SF` contains a list of SHA-1 hashes of all hashes listed in the `MANIFEST.MF` file and is provided as a fallback mechanism for signature verification. `CERT.RSA` contains the public certificate of the application. Compiled code that is platform dependent is stored in the lib folder. This folder contains multiple subfolders for different hardware architectures. The `res` and `assets` folder contain resources and assets like images, XML files, string dictionaries, icons, etc. used by the application. While resources located in the res folder are directly accessible from code, assets found in the assets folder have to be accessed via the AssetManager class that reads the byte stream. `AndroidManifest.xml` contains important information about the application from the package name, the permissions it requires from the Android operating system, to the software and hardware requirements. Also, metadata can be stored in this XML file. Pre-compiled resources are stored in the resources.arsc file that can be accessed at runtime. Classes.dex contains the Java classes compiled to Dalvik bytecode that can be interpreted by the Android Runtime (ART) and the discontinued Dalvik virtual machine.

3.4 Smali

Decompiling Dalvik bytecode to actual working Java code is often not possible due to obfuscation mechanism employed while compiling the application. However, it is possible to extract the raw Dalvik bytecode and convert it to a more readable format. One of the most common human-readable formats is Smali. The process of assembling and disassembling smali source code is called smali and baksmali,

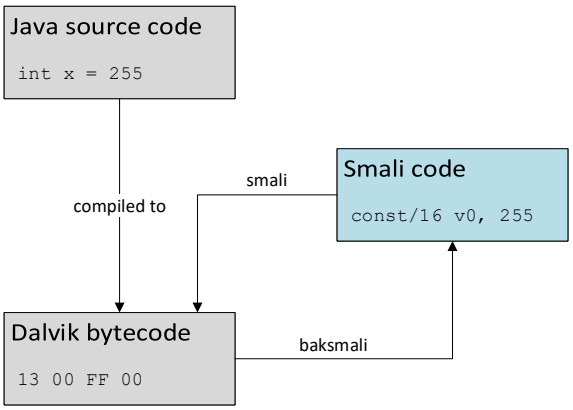


Figure 3.11: Smali/baksmali using a Java source code example.

respectively. Baksmali uses a DEX file to create human-readable Smali code, while smali uses this code and is able to compile a new working DEX assembly. This process is displayed in figure 3.11. The advantage over other decompiling tools that convert DEX files to Java source code is that Smali code is 100% true to the original Dalvik bytecode. After baksmaling a DEX file, each original Java class results in a corresponding Smali class. The layout of an example Smali class can be seen in the code listing below.

```
1  # class information
2  .class public Lcom/example/MyUtils;
3  .super Ljava/lang/Object;
4  .source "MyUtils.java"
5
6  # static fields
7  .field public static final FOO:Ljava/lang/String; = "Foo"
8
9  # direct methods
10 .method public constructor <init>()V
11   .registers 1
12
13   .prologue
14   .line 5
15   invoke-direct {p0}, Ljava/lang/Object;-><init>()V
16
17   return-void
18 .end method
19
20 # virtual methods
21 .method public now()J
22     .locals 1
23
24     .prologue
25     .line 20
26     invoke-static {}, Landroid/os/SystemClock; ->uptimeMillis()J
27
28     move-result-wide v0
29
30     return-wide v0
31 .end method
32
33 .method public IsPositive(I)Z
34     .locals 2
35     [...]
36 .end method
```

Listing 3.1: An example class written in Smali

At the top of the file, the class information is displayed. Classes are always prefixed with the letter `L` and suffixed with the character `;`, as seen in the following two examples:

```
.class public Lcom/example/MyUtils;

invoke-direct {p0}, Ljava/lang/Object;-><init>()V
```

Following the class information, are static field declarations, direct methods, and virtual methods. Parameters and return types of methods are specified by the trailing character and one or multiple characters in the brackets.

```
.method public IsPositive(I)Z
```

The public method above, called `IsPositive`, takes a parameter of type integer (`I`) and returns a variable of type boolean (`Z`). All primitive types are denoted by a character, shown in table 3.1. Since Smali is an assembly based language, it works by registering 32-bit registers before assigning values to variables. There are two ways to define how many registers are needed in a method. The `.register` directive specifies the total number of registers, including parameters, whereas the `.locals` directive specifies the number of non-parameter registers. Registers can be accessed by either the normal v naming scheme or the p naming scheme for parameters.

V	void	**F**	float
Z	boolean	**I**	int
B	byte	**J**	long
S	short	**D**	double
C	char	**[**	array

Table 3.1: Smali syntax for primitive data types

For example, the method `.method public IsPositive(I)Z` reserves two parameter registers. The first register is the object that the method is being invoked on, the second is for the integer parameter. There are two ways to increase the registers for the method. For example, using `.registers 4` or `.locals 2` specifies, that there are two additional registers for the method body (v0-v3/p0-p1). The following table 3.2 shows the v name and p name for each register for this example. One thing to keep in mind, however, is, that each register can only store 32 bits, so long and double primitives (J and D) require two registers each. For basic operation, Smali uses standard Dalvik opcode names, a list of which can be found here[1].

Local	**Parameter**	
v0		local register 1
v1		local register 2
v2	p0	this
v3	p1	parameter I (int)

Table 3.2: Example registers for the method `IsPositive`

[1] `pallergabor.uw.hu/androidblog/dalvik_opcodes.html`

Chapter 4

Approach

The goal of this thesis is to identify patterns and similarities between Android applications with the help of semantic text analysis. This is accomplished by training document embeddings with different features extracted from Android application's APK files as well as their associated metadata using Doc2Vec models. With the help of these models, the following three questions are tried to be answered.

Can Google Play Store descriptions of applications be used for semantic analysis?

A Doc2Vec model, training application's Google Play Store descriptions as document embeddings is used in trying to solve this problem. Specifically, finding meaningful results based on the semantic relationship of two or more application descriptions is of interest. Can vector arithmetic be used to combine or even subtract search terms? For example, will the search query `'City X' + 'Map'` result in similar applications of type `App['Maps of City X']`? This might offer valuable clues about the effectiveness of document embeddings and word analogies in modern search engines.

Can applications be semantically compared based on features found in APK files?

Using document embeddings containing features of an application's APK file in different evaluation tasks will give clues about which feature embeddings can effectively model an application. In particular, features that are looked at encompass UI strings, layout markup code, and Smali source code. Document embeddings containing those features are trained using different Doc2Vec models. After training the models, the resulting document embeddings are used as features in clustering and similarity tasks. Afterwards, it is evaluated which feature can be used to semantically analyze applications.

Is it possible to identify malware or clone applications based on Smali source code similarity?

Smali source code is Dalvik bytecode decompiled to a more readable form. In contrast to decompiled Java source code, which is often unusable due to obfuscation methods employed while compiling the application, Smali source code stays true to the original bytecode. Instead of using Smali source code files in a clustering task as before, Doc2Vec models are used in a classification task to determine if the Smali source code for malware applications are semantically different from benign applications. Furthermore, it is evaluated, if semantic properties of Smali source code can be used to spot cloned applications, which are often used to trick the user into downloading fake applications and are usually filled with malware. Can semantic code analysis be used to create a detection mechanism that might prevent those fake applications from being distributed?

In trying to answer these questions, a four-step process is employed: data retrieval, data preparation, model training, and model evaluation. An overview can be seen in figure 4.1. In order to get an appropriate

amount of training data, different data sets are collected as part of the data retrieval process. APK files and metadata from the Google Play Store and an Android application collection hosted on the Internet Archive[1] are data mined using custom Python crawler bots. Other data sets such as the Android validation data set and the malware data set are provided as archive files containing collections of APKs. Once the metadata and APKs have been stored, the features are extracted, parsed, and cleaned-up. Since APKs have been collected and thus far stored as binaries, their contents are decompiled using a tool called Apktool. Localization strings and UI layout code are decompiled to their original form, while the compiled source code is decompiled to Smali code, in a process known as baksmaling. After extracting the features from the decompiled APK, they are stored in JSON files for ease-of-use. Afterwards, features are pre-processed using different NLP techniques that are explained in section 4.2, and different Doc2Vec models are trained using these pre-processed features as input for document embeddings. Finally, the models are evaluated by how effective they are in solving the aforementioned problems. Different evaluation methods like clustering, classification, and word analogy analysis are employed during this process.

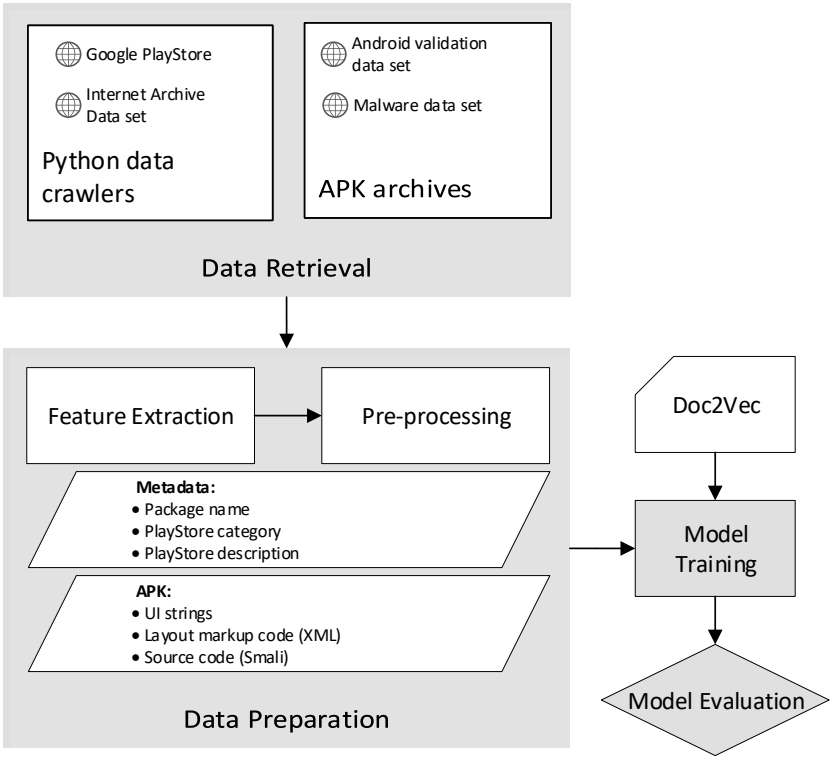


Figure 4.1: Overview of the four step process used to train Doc2Vec models

This chapter goes into detail for the first three of the four aforementioned processes, data retrieval, data preparation, and model training, whereas the last process, called model evaluation, will be covered in the subsequent chapter 5. Section 4.1 explains the data retrieval process and functionality of the crawler bots for Google's Play Store as well as the Internet Archive's application collection. It also explains the contents and origins of the Android validation and malware data set. Section 4.2 explains the used features, their extraction and pre-processing procedures, and the storage of the extracted features. Finally,

[1] archive.org/details/android_apps

the model training is illustrated in section 4.3, describing the design of the used Doc2Vec models and the hyperparameters of the underlying neural network.

4.1 Data Retrieval

Four different data sources for application's APK files and metadata are used in this thesis: the Google Play Store, an Android application collection hosted on the Internet Archive, the Android validation data set and a malware data set. The latter three sources are provided as archive files containing only APKs without further metadata apart from package names, whereas APK files and metadata from the Google Play Store and Internet Archive are collected using crawler bots. The first bot that will be described in this section uses the Google Play Store as its data source. Metadata, like package names and categories, are directly extracted from the Play Store HTML page of the corresponding application. However, since the Play Store only supports installing applications directly to a mobile device but not downloading their APK files to a computer, a third-party downloader site called APK Downloader[2] is used to download the APK files. The second data source that is crawled is a collection of applications and metadata that was gathered using open-source tools created by the PlayDrone project at Columbia University[19] and hosted on the Internet Archive. It contains 1,402,894 applications, 1.1 million of which are free. The data set itself is provided as a JSON file composed of head information objects for each application, which include links to APK files as well as detailed metadata objects. The Android validation data set was created by Gonzalez et al. [5] at the Canadian Institute for Cybersecurity at the University of New Brunswick. It contains a set of 72 original apps with ten transformations for each app, resulting in 792 applications. Lastly, the malware data set was provided by the team behind the now-defunct Android Sandbox project, consisting of 1,889 Malware APK files. First, the Play Store crawler bot, its implementation, and shortcomings are described in 4.1.1. Afterwards, the Internet Archive crawler bot is explained in 4.1.2, followed by the structure and contents of the Android validation and malware data sets in 4.1.3.

4.1.1 Play Store crawler

This crawler bot operates in two stages. In stage one, the bot is crawling the Google Play Store for package names and categories. It parses the category and the package names of the top N similar applications from the Google Play Store HTML page of an initial application. After adding the application's package name with the category to a list, the whole operation is repeated for each of the top N applications. This recursive operation is run, until a certain pre-defined tree depth is reached, after which the bot stops crawling and stage one is finished. Stage two uses the package names from the application set and tries to download their APKs with the help of the APK Downloader website. Since this website does not provide an API and ignores simple POST requests, some of the request-response protocol has to be reverse-engineered. Specifically, the POST parameters for the download link request had to be constructed correctly, by searching for a specific value in the HTML page via regular expressions. After the POST request data has been created and the request has been successfully sent, a download link and the APK version are parsed from the response HTML page, and the APK is downloaded. However, after a few subsequent download link requests, the APK Downloader website usually responds with a timeout error, saying that the rate limit is exceeded. In this case, the bot is programmed to wait for the duration of the timeout, whose value is parsed from the response page. After going through all applications in the application set, the bot stops, and the crawling process is finished. The following flow diagram in figure 4.2 details the crawling process.

[2] apps.evozi.com/apk-downloader/

4.1.1.1 Problems

While crawling the Play Store for package names and categories can be done with reasonable performance, downloading a large amount of APK files via the APK Downloader website turns out to be time-consuming. After 3 to 5 subsequent download link requests, the website denies new requests, until a cooldown period has passed. The duration of the cooldown varies, but it is usually set to around 10 to 12 minutes. The bot is programmed to wait in the meantime and resume crawling after the specified time has passed. However, this constraint limits the amount of APKs files downloaded per hour to about 15 to 20, which is the reason for the development of the Internet Archive crawler, which uses a data set hosted on the Internet Archive that contains both APKs and metadata but, more importantly, does not have a rate limit in place.

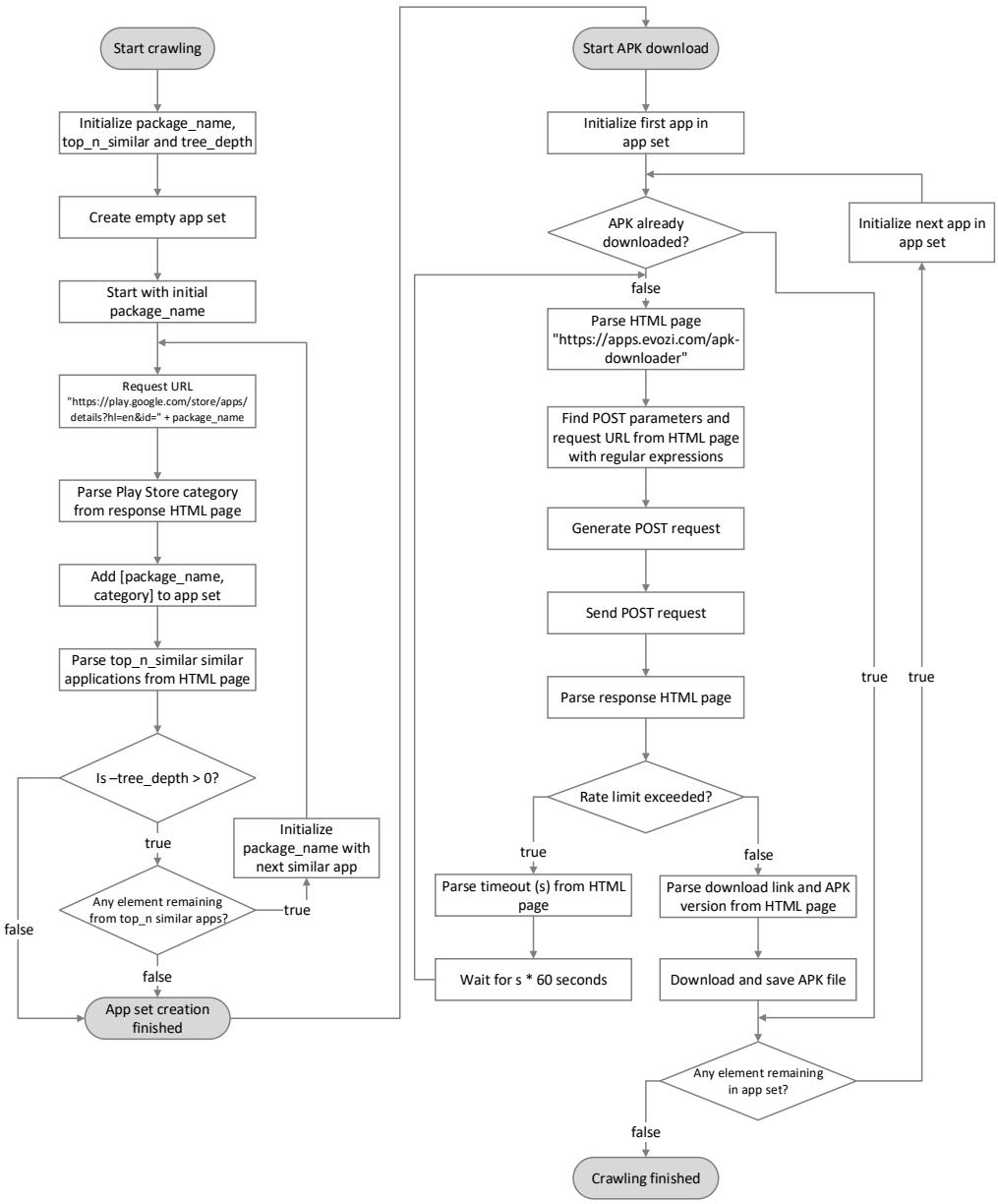


Figure 4.2: Flow diagram of the Play Store crawler

4.1.2 Internet Archive Crawler

The Internet Archive crawler bot uses an application and metadata collection which is hosted on the Internet Archive and was gathered based on research and tools created by the PlayDrone project[19]. It is provided in the form of a JSON file, containing a collection of header objects. A header object consists of basic information pertaining to an application, including URLs to detailed metadata information and, in case of free applications, the archived APK. Overall, the JSON file encompasses 1,402,894 applications, 1.1 million of which are free. The following two listings show the header object information, as well as a condensed version of the detailed metadata information for the YouTube application.

```
1  {
2    "app_id":"com.google.android.youtube",
3    "title":"YouTube",
4    "developer_name":"Google Inc.",
5    "category":"MEDIA_AND_VIDEO",
6    "free":true,
7    "version_code":51405300,
8    "version_string":"5.14.5",
9    "installation_size":10191835,
10   "downloads":1000000000,
11   "star_rating":4.08009,
12   "snapshot_date":"2014-10-31",
13   "metadata_url":"https://archive.org/download/playdrone-metadata-2014-10-31-c9/
         com.google.android.youtube.json",
14   "apk_url":"https://archive.org/download/playdrone-apk-c9/com.google.android.
         youtube-51405300.apk"
15 }
```

Listing 4.1: Header object information for the YouTube application

```
1  {
2    "docid":"com.google.android.youtube",
3    "backend_docid":"com.google.android.youtube",
4    "doc_type":1,
5    "backend_id":3,
6    "title":"YouTube",
7    "creator":"Google Inc.",
8    "description_html":"* Personalized &#39;What to Watch&#39; recommendations<p>*
         Launch a never-ending YouTube Mix from your favorite music videos<p>*
         [...]",
9    "offer":[...],
10   "availability":[...],
11   "image":[...],
12   "details":{
13     "app_details":{
14       "developer_name":"Google Inc.",
15       "app_category":["MEDIA_AND_VIDEO"],
16       "permission":[
17         "android.permission.INTERNET",
18         "android.permission.ACCESS_NETWORK_STATE",
19         "android.permission.CHANGE_NETWORK_STATE",
20         [...]
21       ],
22       [...]
23     }
24   },
25   [...]
26 }
```

Listing 4.2: An excerpt of metadata information for the YouTube application

This bot has two modes of operation. The first mode collects only metadata information, whereas the second mode only collects APKs. Regardless of the mode of operation, one instance of the bot only crawls

one specific category of applications. It traverses a JSON file containing a collection of header information objects of applications. The metadata URL that is linked to each header object is used to download the metadata JSON file, whose contents are subsequently added to a metadata set. After each header object of the current category is processed, the metadata set is saved to disk as a JSON file. This mode of operation is illustrated as a flow diagram in figure 4.4. In the second mode, the crawler inspects the header object for the URL to the APK file and, after checking that the APK has not already been downloaded, uses it to download and save the APK to the hard drive. In two weeks, this crawler bot collected metadata for 1,160,516 applications and 4,480 APKs. The number of applications crawled was dictated by time and storage constraints, especially when looking at the extraction process detailed in 4.2.2 since decompiled APKs can usually double or even triple in size compared to their compiled counterparts. Since an instance of this crawler bot can only process one specific category, multiple instances of the crawler bot can be run concurrently, specifically one per category.

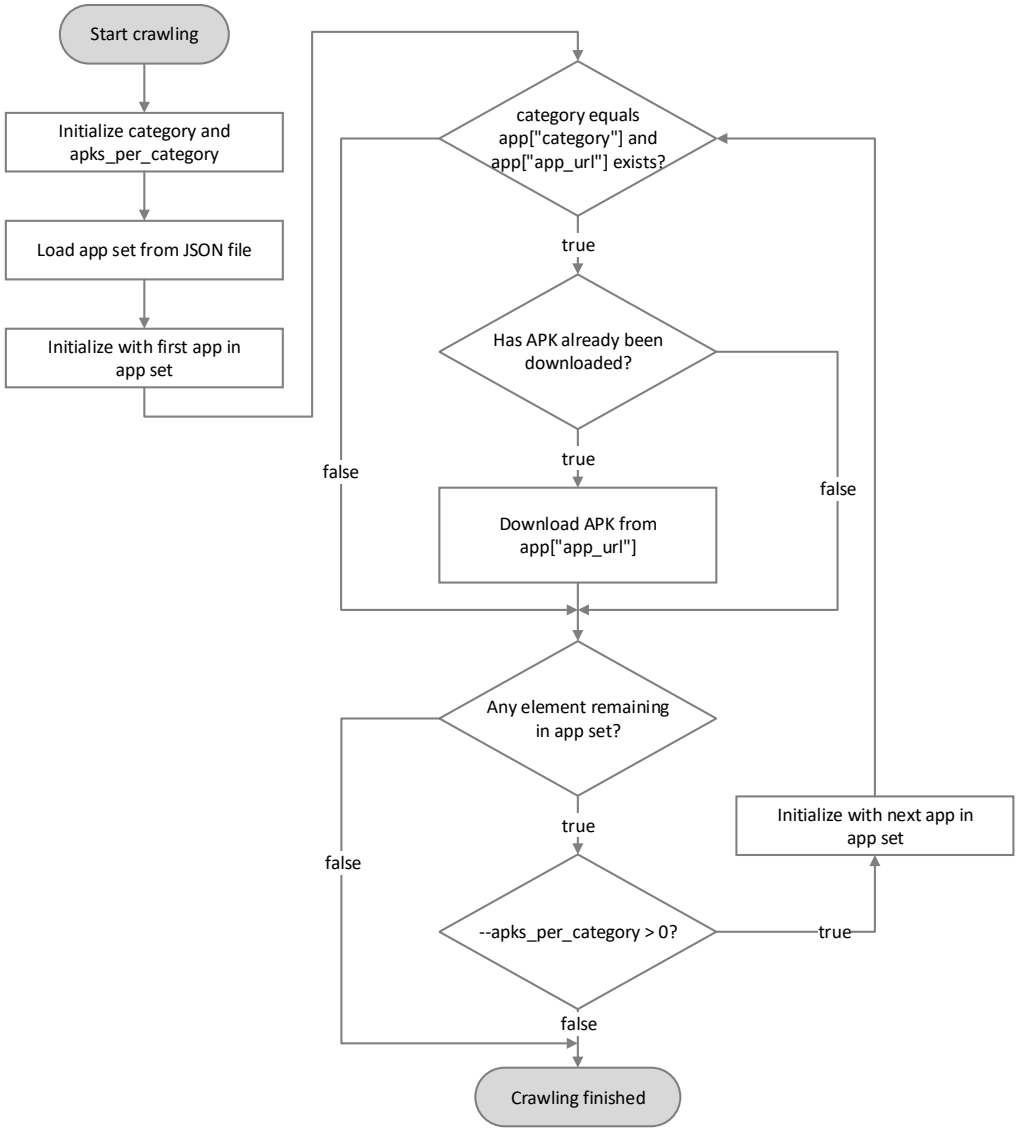


Figure 4.3: Flow diagram of the Internet Archive crawler in APK crawling mode

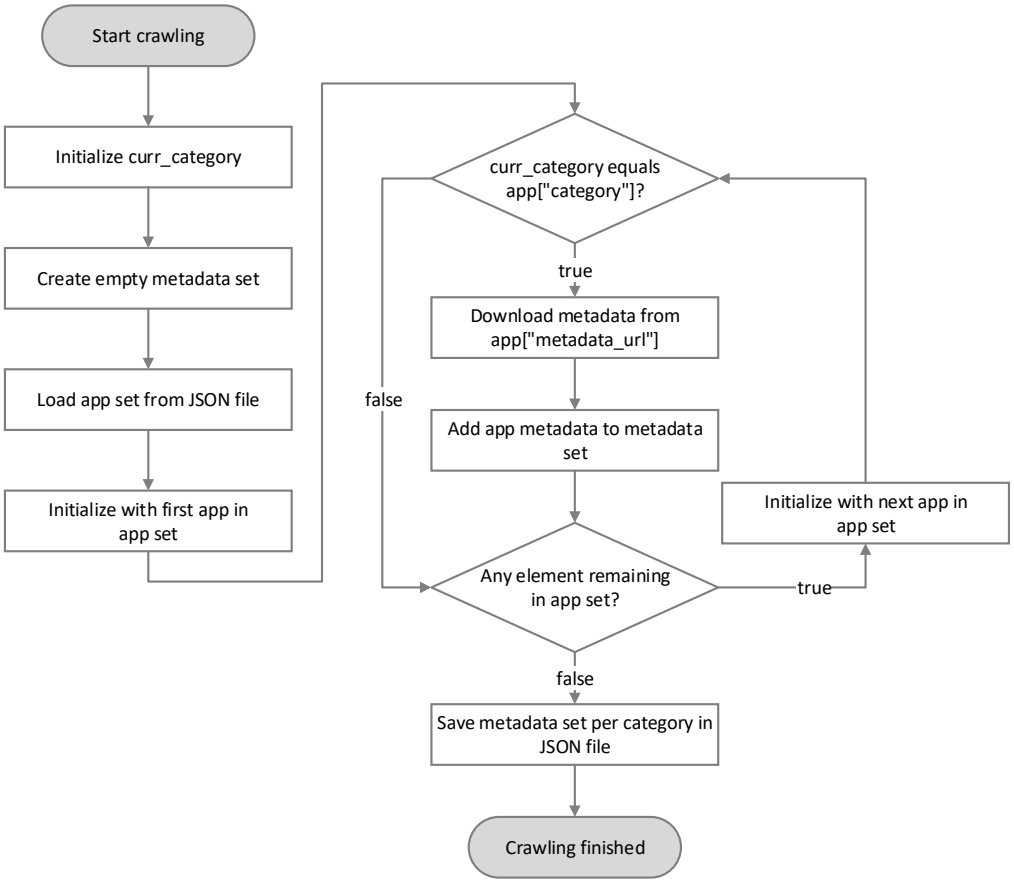


Figure 4.4: Flow diagram of the Internet Archive crawler in metadata crawling mode

4.1.3 Android Validation & Malware Data Sets

The validation data set was created by Gonzalez et al. [5] at the Canadian Institute for Cybersecurity at the University of New Brunswick. It was used in the evaluation process of DroidKin, a system that detects relations between applications based on features such as metadata and source code. It consists of 72 original applications, selected from the Android Malware Genome Project data set[21], the Android Malware Virus Share package[3], the Virus Total repository[4], and the Google Play Store. Furthermore, it contains ten different transformations of the original applications, as seen in table 4.1, resulting in 792 total applications. Some of those transformations modify the original .dex file, and some do not - however, for this thesis only the original application and transformation which had junk code inserted are of interest, since they are the only two versions that differ in source code.

The malware data set was created by Balich IT, the team behind the-now defunct Android Sandbox, an online tool that was used to analyze Android applications. It is provided as an archive file, containing 1,889 malware APKs. Since around May 2015, the website hosting both the Android Sandbox and the data set containing the malware APKs is not online anymore, so a local copy of the data set was provided by the supervisor of this thesis.

[3] virusshare.com
[4] virustotal.com

.dex file modified	.dex file not modified
1. Repackaging .dex file	5. Aligning files to 4 bytes
2. Rerepackaging .dex file	6. Aligning files to 8 bytes
3. Modifying string URLs	7. + 8. Changing icons
4. Inserting junk code	9. Adding junk files
	10. Replacing files

Table 4.1: Transformations of an application in the Android validation data set. Only transformation 4 is of relevance to this thesis.

4.2 Data Preparation

After collecting APKs and metadata in the previous process, features have to be extracted from their data source. So far, metadata features are stored in JSON files, whereas APK features are stored as binary files within the APK files. Before extracting features from APKs, the binaries need to be decompiled. In contrast, extracting metadata features can be done by accessing the JSON files and iterating the collection of JSON objects found within. After extracting features from the metadata JSON files as well as the decompiled APK files, the raw feature data needs to be saved, before being pre-processed and used as input for document embeddings in a Doc2Vec model training process. This section first describes the features selected for the training process in 4.2.1, followed by the feature extraction process in 4.2.2, and pre-processing methods in 4.2.3.

4.2.1 Feature and Label Data

In order to train Doc2Vec models, specific APK and metadata features have to be selected as input for the document embeddings. Moreover, some information that acts as labels for the document embeddings has to be selected as well. For this purpose, package names and categories are used as labels for tagging document embeddings before training them with Doc2Vec, whereas descriptions, UI strings, layout markup code, and source code are used as input features. The following table 4.2 lists all types of data that are used and their purpose in the Doc2Vec model training process.

Data	Type	Used as
Package name	Metadata	Label
Google Play Store category	Metadata	Label
Google Play Store description	Metadata	Feature
UI strings	APK data	Feature
Layout markup code (XML)	APK data	Feature
Source code (Smali)	APK data	Feature

Table 4.2: List of labels and features

4.2.1.1 Metadata Features and Labels

Android applications can store metadata in their AndroidManifest.xml file, which provides users with information about permissions, device compatibility, used SDK version, and the package name among many more. Furthermore, online marketplaces such as the Google Play Store attach additional metadata to each application. In order to improve search functionality and user experience, applications are shelved into different categories and supplied with descriptions, a list of similar applications, pictures, user reviews and

more, while also publishing the application's metadata found in its manifest file. While there is a plethora of information that can be used as features for analysis purposes, this thesis focuses on three specific types of metadata, namely the package name, Google Play Store category, and Google Play Store description. A package name uniquely identifies an application on any individual device and on the Google Play Store, and is declared in the application's manifest file. Naming conventions for package names usually follow the traditional Java package naming conventions, which can be found here[5]. Google Play Store applications are divided into two types, namely apps *Apps* and textitGames. Each type has its own set of categories with which applications are further subdivided. This thesis only looks at *App*-type applications and excludes gaming related applications, since the data retrieval and extraction processes would be prolonged significantly, due to the larger file sizes commonly found in games. Overall, 24 Google Play Store categories are used in this thesis, a list of which can be seen in table 4.3. As a side note, since the data retrieval process ended, Google footnote`https://support.google.com/googleplay/android-developer/answer/113475` has updated its non-gaming related Play Store categories to a total of 32.

Category Name	Category description
Books & Reference	Book readers, reference books, text books, dictionaries, thesaurus, wikis
Business	Document editor/reader, package tracking, remote desktop, email management, job search
Comics	Comic players, comic titles
Communications	Messaging, chat/IM, dialers, address books, browsers, call management
Education	Exam preparations, study-aids, vocabulary, educational games, language learning
Entertainment	Streaming video, Movies, TV, interactive entertainment
Finance	Banking, payment, ATM finders, financial news, insurance, taxes, portfolio/trading, tip calculators
Health & Fitness	Personal fitness, workout tracking, diet and nutritional tips, health & safety etc.
Libraries & Demo	Software Libraries, technical demos
Lifestyle	Recipes, style guides
Media & Video	Subscription movie services, remote controls, media/video players
Medical	Drug & clinical references, calculators, handbooks for health-care providers, medical journals & news
Music & Audio	Music services, radios, music players
News & Magazines	Newspapers, news aggregators, magazines, blogging
Personalization	Wallpapers, live wallpapers, home screen, lock screen, ringtones
Photography	Cameras, photo editing tools, photo management and sharing
Productivity	Notepad, to do list, keyboard, printing, calendar, backup, calculator, conversion
Shopping	Online shopping, auctions, coupons, price comparison, grocery lists, product reviews
Social	Social networking, check-in, blogging
Sports	Sports News & Commentary, score tracking, fantasy team management, game Coverage
Tools	Tools for Android devices
Transportation	Public transportation, navigation tools, driving
Travel & Local	Maps, City guides, local business information, trip management tools
Weather	Weather reports

Table 4.3: List of all 24 Google Play Store categories. As of 2018, Google has expanded the categories to 32. However, this thesis still uses the older category definitions.

4.2.1.2 APK Features

This thesis focuses on three features found in Android applications - UI strings, layout markup code, and source code. These features are compiled into the `classes.dex` and `resources.arsc` binary files which are found within the application's APK file. Figure 4.5 shows an overview of the features and their sources within an APK. UI strings are localized words, phrases, or sentences stored in an XML file. Each UI string can be accessed via a name property, that is shared between all UI string XML files, from the

[5] `docs.oracle.com/javase/tutorial/java/package/namingpkgs.html`

source code which is then localized to the application's current language at runtime. Multiple instances of UI string XML files can exist, typically one for each localized language. Layout files contain the XML definitions of the application's user interface forms. They define the structure of the user interfaces, which are typically built using a hierarchy of `View` and `ViewGroup` elements. The primary purpose of defining a user interface in an XML definition is to separate the presentation of an application from the code that controls the behavior of said presentation, a technique also known as MVC (Model-View-Controller). Lastly, Smali code is a form of human-readable Dalvik bytecode, which can be created by decompiling the binary class files contained in an APK in a process known as baksmaling. Compared to decompiled Java source code, Smali source code is 100% true to the original bytecode, whereas reverse-engineered Java source code is often unusable, due to obfuscation techniques that are applied to the original source code before compiling an application.

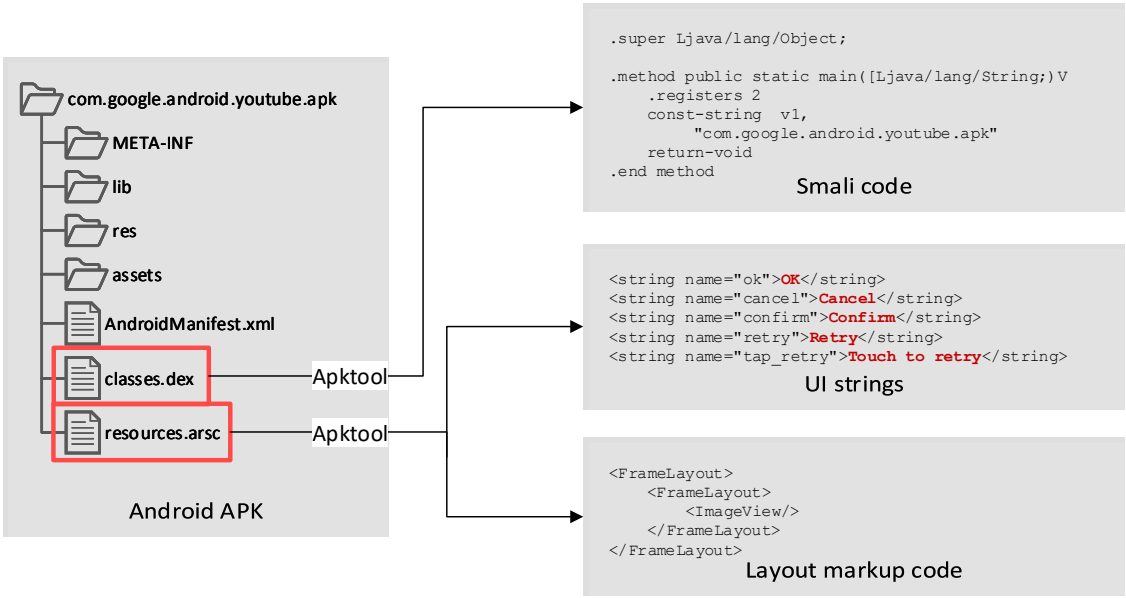


Figure 4.5: Overview of features extracted from an APK

4.2.2 Feature Extraction

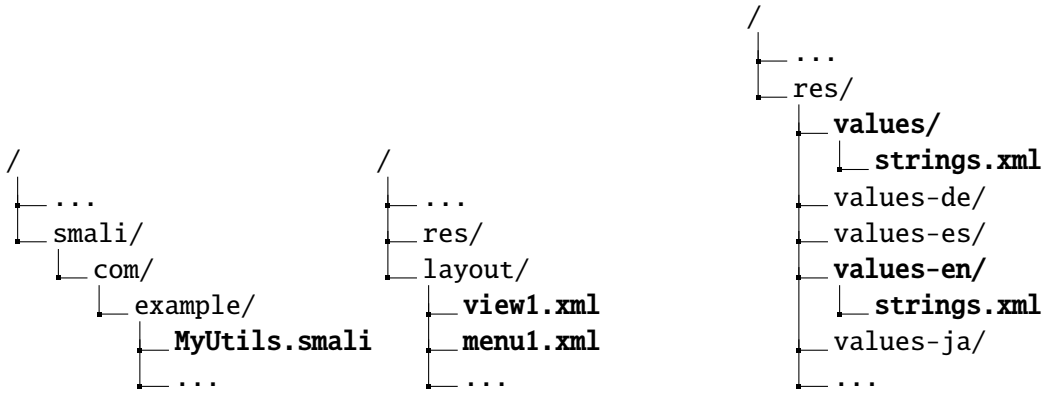
After selecting features for training, they are extracted from the retrieved APK and metadata JSON files. Since Smali files, layout files, and UI strings are stored in binary files within an APK, those binaries first need to be decompiled. Decompiling APK files is done with a tool called Apktool, which is primarily used for reverse engineering Android APK files. For this purpose, it is used specifically to decompile the `resouces.arsc` and `classes.dex` files that are found within an APK file. Smali classes are created in a folder that is usually not part of the Android application structure, since Apktool tries to re-create the original folder structure used while developing the application. UI Strings and layout files are restored to the folders they were originally stored in. After decompiling, the features can be extracted from their corresponding files within the APK folder structure. The following figure 4.6 shows the location of the feature files in decompiled APKs. The Smali and layout features are extracted from all Smali and layout files contained within an APK. However, since multiple language-specific instances of UI strings files can co-exist, the files containing English UI strings have to be determined before extracting the contents. The folder `values/` contains the default language, which in most cases is English. However, sometimes the English UI strings are stored in a different folder named `values-en/`. Because of this, `langdetect`,

a Python port[6] of Google's language-detection library[7], is used to determine the language of the file. After detecting the correct file, only the values are extracted from the file's XML structure. This is in contrast to Smali and layout feature extraction, where the entire contents of files are extracted. Application metadata are so far stored in Play Store category-specific JSON files containing collections of metadata objects. Descriptions are extracted by loading and iterating those metadata JSON files and accessing the corresponding values within the metadata objects.

```
1 {
2   "package_name":"com.google.android.youtube",
3   "category":"MEDIA_AND_VIDEO",
4   "file_name":"utils.smali",
5   "data":"# class information\n.class public Lcom/google/android/youtube;[...]"
6 }
```

Listing 4.3: A Smali feature object, including feature data and identifiers

Since, at the time of extracting the features, the pre-processing techniques discussed in 4.2.3 were still being refined, the features are intermediately saved. Depending on the data source, Smali files, layout files, UI strings, and Google Play Store descriptions are saved in feature-specific JSON files. Every feature is stored as a JSON object, containing the feature data, the package name, and, if available, Google Play Store category and file name as the identifiers, within the feature-specific JSON files. Those files are the basis for all the following processes, since they contain the raw feature data, but stored in condensed and easily-readable JSON objects. An example of a Smali feature object can be seen in listing 4.3. Only after storing the features are they pre-processed and used in training document embeddings using Doc2Vec models. Moreover, since decompiling a large amount of APKs results in a lot of files, decompiled APKs are deleted, after their features have been successfully extracted and stored, so as not to fill up too much hard drive space. The directory listing shown in figure 4.7, illustrates the different files for each data set. Note that, while the Play Store features were still extracted and saved, mainly the features from the Internet Archive data source are used in the model training process, due to the greater number of files and metadata that have been collected from this data source. Following the extraction process, the JSON files, containing the extracted features, are pre-processed before being used as data for the Doc2Vec model training process. Figure 4.8 shows the complete extraction and pre-processing pipeline for one example APK.



(a) Location of Smali classes **(b)** Location of layout files **(c)** Location of the UI strings file

Figure 4.6: Locations of features within a decompiled APK

[6] https://pypi.org/project/langdetect/

[7] https://github.com/shuyo/language-detection/blob/wiki/ProjectHome.md

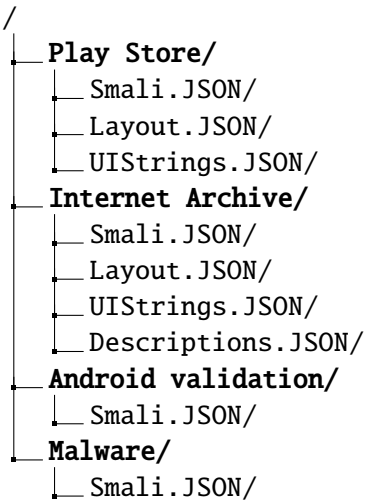


Figure 4.7: Feature data JSON files for each data set after the extraction process

4.2.3 Feature Pre-processing

The feature pre-processing process uses the extracted features and applies natural language processing techniques to them in order to improve the quality of the resulting document embeddings. Natural language processing techniques are text processing tools that try to reduce noise (whitespace, punctuation), standardize (stemming/lemmatizing) and reduce corpus size (stop words removal) in text corpora for use in semantic text analysis. Since different features require different pre-processing techniques, each feature is listed with the natural language processing techniques that are applied to it, including example listings comparing the original and pre-processed features.

Smali Code

The first step in pre-processing Smali classes is to remove all leading whitespace present in the classes. Since comments in Smali are limited to region specifiers, like `# static fields` and `# direct methods`, they are also removed. Afterwards, all debugging lines, like `.line` and `.prologue`, are removed, as well as register allocation code like `.locals` and `.registers`. Moreover, all punctuation and brackets are replaced by whitespace. Following is a comparison between the original source code, and the pre-processed code.

1. Remove any leading whitespace characters
2. Remove any line starting with `#`
3. Remove any line starting with `.line`
4. Remove any line starting with `.prologue`
5. Remove any line starting with `.locals`
6. Remove any line starting with `.registers`
7. Replace any punctuation and brackets with whitespace

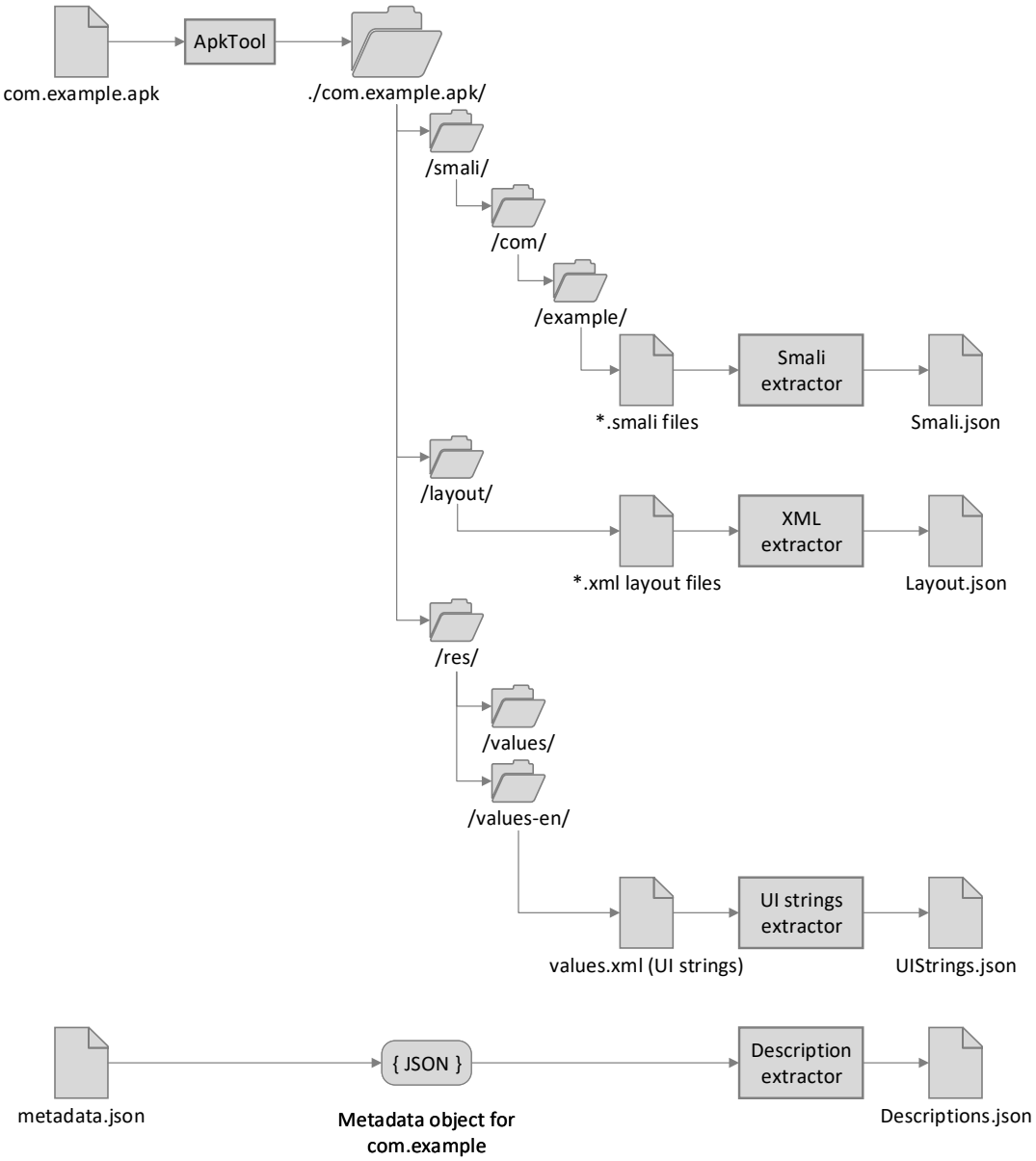


Figure 4.8: The feature extraction pipeline for APK and metadata features

```
.class public Lcom/example/MyUtils;
.super Ljava/lang/Object;
.source "MyUtils.java"

# direct methods
.method constructor <init>()V
    .locals 0

    .prologue
    .line 504
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method
```

Listing 4.4: Smali class

```
class public Lcom example MyUtils
super Ljava lang Object
method constructor init V
invoke direct p0 Ljava lang Object init V
return void
end method
```

Listing 4.5: Pre-processed Smali class

XML Layout Code

Pre-processing the XML layout code is done by parsing the XML tree with the help of lxml[8]. After successfully parsing the XML tree, the XML declaration is removed, before removing all parameters from any XML nodes. Following is an example of a layout code before and after pre-processing.

1. Parse XML tree
2. Remove XML declaration
3. Remove any parameter from nodes

```
<?xml version="1.0" encoding="utf-8"?>
<ViewGroup
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@[+][package:]id/resource_name"
    android:layout_height=["dimension" | "match_parent" | "wrap_content"]
    android:layout_width=["dimension" | "match_parent" | "wrap_content"]
    [ViewGroup-specific attributes] >
    <View
        android:id="@[+][package:]id/resource_name"
        android:layout_height=["dimension" | "match_parent" | "wrap_content"]
        android:layout_width=["dimension" | "match_parent" | "wrap_content"]
        [View-specific attributes] >
        <requestFocus/>
    </View>
    <ViewGroup >
        <View />
    </ViewGroup>
    <include layout="@layout/layout_resource"/>
</ViewGroup>
```

Listing 4.6: Original layout code

```
<ViewGroup>
    <View>
        <requestFocus/>
    </View>
    <ViewGroup>
        <View/>
    </ViewGroup>
    <include/>
</ViewGroup>
```

Listing 4.7: Pre-processed layout code

[8] https://pypi.org/project/lxml/

UI Strings

UI Strings are stored as the extracted values from the strings.xml files. First, all empty entries are removed. Afterwards, any leading, trailing and superfluous whitespace is removed, before replacing punctuation and line breaks with whitespace and lower-casing the values.

1. Remove empty values
2. Remove leading whitespace
3. Remove trailing whitespace
4. Remove superfluous whitespace
5. Replace any punctuation with whitespace
6. Replace line breaks with whitespace
7. Lower-casing

```
1  Done
2  Navigate    up
3  More options
4  Collapse
5       Search
6  Clear query
7  Submit query
8
9  Voice search
10 Messenger
11 Would you like to know \n\r more?
```

Listing 4.8: Original UI strings

```
1  done
2  navigate up
3  more options
4  collapse
5  search
6  clear query
7  submit query
8  voice search
9  messenger
10 would you like to know more
```

Listing 4.9: Pre-processed UI strings

Descriptions

Since descriptions are so far stored as HTML markup, the first step is to convert HTML to plaintext by using html2text[9]. Afterwards, any URLs or E-Mail addresses are removed. Punctuation and line-break characters are replaced by whitespace, and all non-alpha numeric characters are removed before the whole text is lower-cased and encoded in UTF-8. Finally, stoplist words taken from the Natural Language Toolkit[10] corpora are removed.

1. Convert from HTML to plaintext
2. Remove any URLs
3. Remove any E-Mail addresses
4. Replace any punctuation with whitespace
5. Replace line breaks with whitespace
6. Remove all non alpha numeric characters

[9] https://pypi.org/project/html2text/

[10] https://www.nltk.org/

7. Lower-case entire text
8. Encode in UTF-8
9. Remove stoplist words

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

Listing 4.10: Original description

```
1 personalized what watch recommendations launch never ending youtube mix favorite
    music videos find music faster playing albums artists right search cast
    videos straight phone chromecast connected tv devices game consoles search
    using text voice quickly find favorite channels using guide
```

Listing 4.11: Pre-processed description

4.3 Model Training

To train document embeddings using Doc2Vec models, a Python implementation of Doc2Vec included in the gensim library, created by Sojka and Řehůřek[12], is used. It can use labeled or unlabeled data as input for a Doc2Vec model. After pre-processing the raw feature data, which thus far has been stored in data source and feature specific JSON files, each feature document is labeled using the application's package name, file name, and, if available, the Google Play store category. The labeled documents are then used as input for the Doc2Vec model training process. Gensim supports training models using Distributed Memory or Distributed Bag-of-Words as the training algorithm. When using Distributed Memory, both document embeddings and the underlying word embeddings are trained. However, using Distributed Bag-of-Words only trains document embeddings. Word embeddings are initialized with random values and will not get updated during the training process. Therefore, training a model using Distributed Bag-of-Words can be a way to speed up the training process. In addition to that, gensim supports a third mode, which trains document embeddings with Distributed Bag-of-Words, alongside word embeddings with the Skip-gram algorithm employed by Word2Vec. This mode is used as the training algorithm for Doc2Vec models that use formal language text, whereas Distributed Memory is used for models using UI strings and Google Play Store descriptions.

Hyperparameter	Description
Vector size	Dimension of the document embeddings
Window size	Size of the context window when creating word pairs
Epochs	Number of training iterations over entire corpus
Min count	Minimum frequency threshold where words are ignored
Sub-sampling	Frequency threshold for downsampling words
Negative sampling	Number of negative samples used in each training step
Workers	Number of CPU threads used to train the model

Table 4.4: List of important Doc2Vec model hyperparameters

Before training a model with one of the three training algorithm modes, hyperparameters that fine-tune the model's shallow neural network and its training process have to be defined. A list of important hyperparameters and their descriptions can be seen in table 4.4. Vector size determines the dimensionality of the resulting document and word embeddings, which is equal to the number of neurons used in the hidden layer of the neural network. Suggested values for vector sizes range from 100 to 300. Tuning this parameter affects the ability of the model to produce quality document and word embeddings, while maintaining decent performance and memory usage while training. Window size defines the size of the sliding window that is used in the word pair creation process. Epochs define the number of iterations over the text corpus, i.e. the number of times the training process is repeated for all document embeddings. Usually, Doc2Vec models are trained with 10 to 20 epochs. Since this hyperparameter has a significant impact on the performance of the model training process, decreasing the number of epochs for models with larger data sets might be beneficial. Min count describes the minimum frequency threshold of words. All words with a total frequency lower than the specified value are ignored. The threshold for the frequency of words that can be selected to be randomly downsampled is defined by the sub-sampling hyperparameter. The amount of negative samples used for each training step is determined by the Negative sampling parameter. When the value is 0, no negative sampling is used. The default values 1e-3 and 5 are a good starting point for the sub-sampling and negative sampling hyperparameters, respectively. Finally, the workers parameter defines how many threads are used while training the model. This is especially helpful when training on multi-core machines. After specifying the hyperparameters and creating the pre-trained document embeddings using feature data, a Doc2Vec model can be trained. Depending on the size of the input data, selected hyperparameters, and system specifications, the full training process can be time-consuming, with training processes running for multiple days. In order to increase performance, gensim supports multi-threading the training process, a feature that is utilized by training performance-intensive models using a multi-processor setup. After the training process is finished, the resulting model, including document embeddings and, in the case of DM or DBOW+Skip-gram model training, word embeddings, is saved as a binary file.

4.4 Summary

This chapter explained the first three steps of the four-step process used in this thesis. First, the data sets and their retrieval processes were explained, including the Google Play Store crawler bot and the Internet archive crawler bot, which are used to collect metadata and APKs from said data sources. The Android validation and malware data set, however, were provided as archive files and do not need to be crawled using crawler bots. After collecting metadata and APK files, the contained features have to be extracted, stored, and pre-processed. Metadata information is so far stored as collections of metadata JSON objects, split into different JSON files for each Google Play Store category. Google Play Store descriptions can be extracted by simply reading the metadata JSON files and accessing the metadata JSON objects found within. On the other hand, features contained within an APK - localizable UI strings, layout markup code and Smali source code - have to be decompiled from APK files using a tool called Apktool before they can be extracted. Specifically, the `resources.arsc` and `classes.dex` files of an APK are decompiled since they contain the desired features. After extracting the features from the files of the decompiled APK file, their data is saved intermediately to feature-specific JSON files. This is done, so that the pre-processing techniques could be fine-tuned and re-iterated by using condensed and easily-readable JSON objects, instead of repeating the feature extraction process every time. After features are pre-processed, they can finally be used as document embeddings in the Doc2Vec model training process. After training, the resulting models have to be evaluated if they are able to help solve the problems posed at the beginning of this chapter. This evaluation process will be detailed in the following chapter.

Chapter 5

Evaluation

This chapter outlines the evaluation methods and tools used in this thesis. Different Doc2Vec models that use descriptions, UI strings, layout markup code, and Smali source code as input features are trained and evaluated using different methods that are described in this chapter. Figure 5.1 shows an overview of the pre-processing, model training, and model evaluation processes. After Doc2Vec models have been trained and saved, they are tested using evaluation methods like word analogy analysis and downstream machine learning tasks such as classification and clustering algorithms. Those methods help determine the resulting Doc2Vec model's effectiveness in aiding the three main problems this thesis is trying to solve. The following table 5.1 contrasts these research problems with the Doc2Vec models and the evaluation methods used in solving said problems. First, the Doc2Vec models and their input features are explained in section 5.1, including four different Smali models, and one model each for the remaining features. Afterwards, the evaluation methods, including word analogy analysis, Doc2Vec similarity interface, and clustering algorithms, are presented in section 5.2.

Problem	**Models**	**Evaluation method**
Can the Google Play Store description of an application be used to find other, similar applications?	Descriptions.model	Doc2Vec similarity interface
Can applications be semantically compared based on features found in APK files?	UIStrings.model, Layout.model, Smali.model	Doc2Vec similarity interface, k-Means, DBSCAN
Is it possible to identify malware or clone applications based on Smali source code similarity?	Malware.model, Validation.model, Smali.model	Doc2Vec similarity interface

Table 5.1: Models and evaluation methods used in solving problems

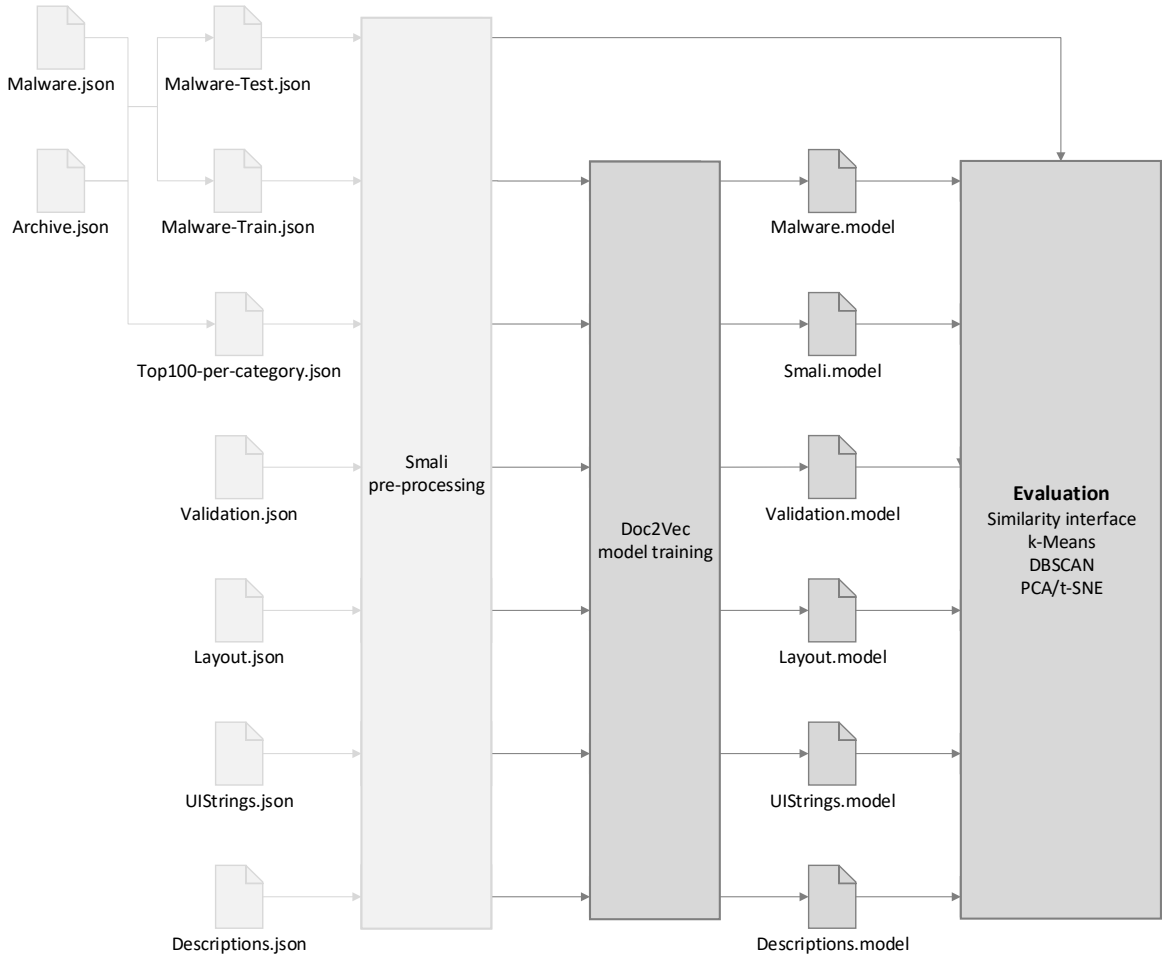


Figure 5.1: Overview of the pre-processing, model training and model evaluation processes

5.1 Doc2Vec Models

Trained Doc2Vec models that have been created with the machine learning framework gensim are the basis of the evaluation. Models are stored in binary files with the file extension `.model`. After the model file is loaded in Python, the document and word embeddings can be accessed and used as input for downstream machine learning tasks. Moreover, gensim's Doc2Vec models provide a similarity interface, which can be used to compare document and word embeddings directly from within a model. In addition to that, models are also able to infer new document embeddings for previously unseen documents, which can then be used in the same way as originally trained document embeddings. This simplifies the process of introducing new data to models, which otherwise would only be possible by re-training a new model using the original and new data combined. Table 5.2 list important model objects which were used in the evaluation process. Afterwards, each model and its purpose is explained.

5.1.1 Play Store Description Model

A Doc2Vec model, using Google Play Store descriptions, that have been collected using the Internet Archive data set, as input features, was trained first. Package names and Play Store categories are used to label the description data for the training process. The model is evaluated by using the Doc2Vec similarity interface to find meaningful relationships between two applications. The relationship results are produced by using document embeddings, word embeddings, and a combination of both. Furthermore, it is evaluated

Objects	Description
`wv`	List of word embeddings
`docvecs`	List of document embeddings. Each object can be accessed by using its associated labels.
`wv.vocab`	The vocabulary of the model, containing all unique words from every document embedding.
`trainables`	Represents the inner shallow neural network. Contains hyperparameters as well as the list of documents used in the training process.

Table 5.2: List of important Doc2Vec model objects

if the model is able to produce meaningful output given word analogies. This might give clues if Doc2Vec models trained with Play Store descriptions can be used as a tool for searching related applications.

5.1.2 UI Strings & Layout Models

Two Doc2Vec models were trained using UI strings, and layout code features, respectively, which are extracted from APK files that were collected from the Internet Archive data set. The trained models are evaluated using clustering algorithms such as k-Means and DBSCAN, as well as the Doc2Vec similarity interface. The results are evaluated based on the effectiveness of both features to model an application as a continuous embedding for semantic analysis purposes.

5.1.3 Smali Models

Model	Data source
Smali model	Internet archive data set
Validation model	Android validation data set
Malware model	Malware data set + Internet archive data set

Table 5.3: List of Doc2Vec models using Smali source code features

Three different Smali models, as shown in table 5.3, which have been trained using Smali source code from applications, that were collected from three different data sources, are evaluated with different evaluation methods. The Smali model contains the top 100 applications from different Google Play Store categories and is evaluated by using the inbuilt Doc2Vec similarity interface, which can be used to determine the similarity scores of related document embeddings. Are the resulting document embeddings effective in predicting semantic similarities between applications? The validation model, as well as the Smali model are used to spot fake clone applications. This might, for example, help in finding malware applications, which impersonate popular applications. Document embeddings for the validation model are created from the Smali source code files of 144 applications contained in the validation data set, including 72 original applications and 72 transformations that have junk code inserted. The Doc2Vec model's document embeddings are evaluated using the inbuilt similarity interface. Lastly, the malware model is trained using Smali source code feature files from 600 malware and 600 benign applications. The source code files were labeled, using the package and file name. After training the model, the resulting document embeddings are evaluated by using 200 previously unseen malware and benign applications as well as the trained document embeddings themselves. The data source for malware applications is the Malware data set, whereas benign applications are drawn from the Internet Archive data set's top 100 applications for the following six categories:

1. Communication
2. Finance
3. Personalization
4. Productivity
5. Social
6. Tools

The categories were chosen specifically to include applications like internet browsers, messengers, banking apps, and antivirus apps, among others since these types are most commonly impersonated by malware. Again, the Doc2Vec similarity interface is used to evaluate this model.

5.2 Evaluation Methods

This section describes the methods used to evaluate the aforementioned Doc2Vec models, including document and word embeddings. While document and word embeddings can be used in downstream machine learning tasks, gensim's implementation of Doc2Vec provides a similarity interface, that can be used to directly compare similarities of embeddings within the model itself. First, the similarity interface is described in more detail, including word analogy analysis, before describing the k-Means and DBSCAN clustering algorithms used in the evaluation process.

5.2.1 Doc2Vec Similarity Interface

Method	Return	Description
`infer_vector( doc_words, alpha=None, min_alpha=None, epochs=None, steps=None)`	Document embedding as an n-dimensional NumPy array	Can be used to create a new document embedding for previously untrained documents. Accepts a list of words called `doc_words` followed by multiple hyperparameters used for the training process of the new document.
`wv.most_similar( positive=None, negative=None, topn=10)`	List of `(word, similarity)`	Finds the top-N similar words. `positive` and `negative` are list of words that contribute positively or negatively to the similarity score. Returns a list of words and their similarity scores.
`docvecs.most_similar( positive=None, negative=None, topn=10)`	List of `(tag, similarity)`	Finds the top-N similar document embeddings. The `positive` and `negative` lists of document embeddings contribute accordingly to the similarity score. Returns a list of document tags/labels and the similarity scores of the corresponding document embeddings.

Table 5.4: List of important Doc2Vec model methods

Gensim's Doc2Vec implementation provides a way to both compare document and word embeddings, without using separate machine learning tasks like classification or clustering. Similarities can be computed

for document embeddings and word embeddings that have been learned during the Doc2Vec model training process. Moreover, the model can be used to infer document embeddings for previously unseen documents, which subsequently can be compared to the model's document embeddings. This provides a simple way to introduce new data to an already trained model without the need for re-training the model. Table 5.4 lists the important methods. `infer_vector` creates new document embeddings for previously unseen documents. The method `most_similar` can be used to find the top-N similar word or document embeddings. For example, asking for the top 5 similar words to 'facebook' might result in the similarity scores seen in listing 5.1, normalized to values between -1 and 1. Additionally, it is possible to find document embeddings based on the similarity of one or multiple word embeddings.

```
1 [('twitter', 0.9196221828460693),
2  ('instagram', 0.8319854140281677),
3  ('social', 0.7319374084472656),
4  ('pinterest', 0.7220771312713623),
5  ('tumblr', 0.6985977292060852)]
```

Listing 5.1: Top 5 similar words for 'facebook' including the similarity scores for each word

The similarity score is calculated based on the cosine distance between document/word embeddings and is known as *cosine similarity*. More precisely, the cosine similarity is the cosine value of the angle between two n-dimensional vectors. For example, the cosine distance for document/word embeddings in 2-dimensional vector space can be illustrated as seen in figure 5.2. The similarity can be calculated with the formula

$$similarity = cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{5.1}$$

where θ is the angle between the associated vectors A and B, and the similarity is derived by using the Euclidean dot product formula

$$A \cdot B = \|A\| \|B\| cos(\theta) \tag{5.2}$$

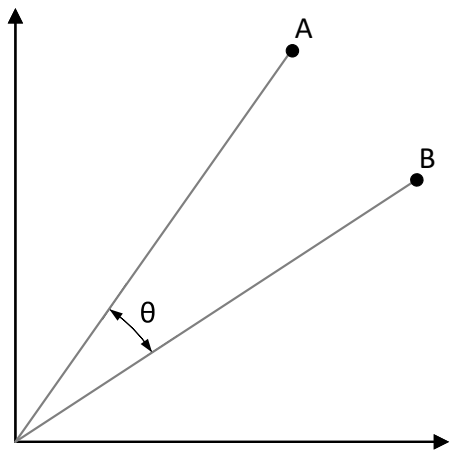


Figure 5.2: Cosine distance between two document/word embeddings A and B in 2-dimensional vector space

5.2.1.1 Word Analogy Analysis

Word2Vec is well-known for successfully predicting word analogies. The most famous example, 'King' - 'Man' + 'Woman' = 'Queen' was already discussed in 3.1.1. Gensim's Doc2Vec similarity interface is also able to provide similarity scores for document and word embeddings based on vector arithmetic. This is used to evaluate if the Doc2Vec model trained with Google Play Store descriptions can successfully predict different word analogies.

5.2.2 k-Means

A Python implementation of k-Means, included in the skicit-learn framework[1], is used to evaluate Doc2Vec models trained with UI strings, layout code, and Smali source code. k-Means is an unsupervised clustering algorithm that tries to divide n data points into a specified number of clusters, denoted by k. More specifically, k defines the number of centroids, i.e. the center of the clusters. The algorithm tries to allocate every data point to the nearest cluster while keeping the centroids small. After initializing k centroids with either randomly generated values or randomly selected values from the data set, each data point is assigned to its nearest centroid, based on the squared Euclidean distance. Afterwards, all data points of each centroid are averaged, and the centroid is being adjusted based on this 'mean' value. The algorithm iterates between these two steps, until pre-defined stopping criteria are met.

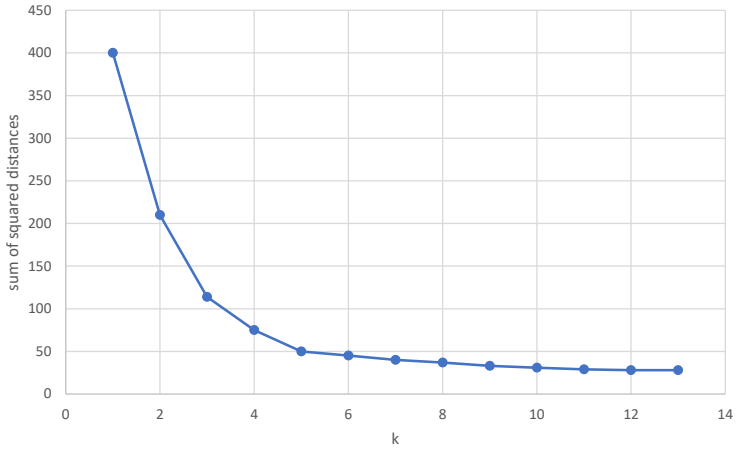


Figure 5.3: Elbow method for finding the optimal k for k-Means clustering

One of the main challenges when using k-Means is determining a good value for k, since the optimal number of clusters for the document embeddings is not known beforehand. Because of this, the mean distance between data points and centroid clusters is used to chose the value for k, in a method known as *elbow method*. Since increasing the number of clusters always decreases the mean distance, to the point where the mean distance equals zero when k equals the number of data points in the data set, the distances for increasing k are plotted, and the value on the *elbow* of the plot is chosen as k. It usually is the point where increasing k starts to see diminishing returns. In the example plot in figure 5.3, the elbow is at k=5, indicating that 5 is the optimal cluster size for this particular data set. Document embeddings that were trained in models containing features from UI strings, layout code, and Smali source code are clustered using k-Means, and an optimal number of clusters is determined using the elbow method mentioned above.

[1] scikit-learn.org

5.2.3 DBSCAN

DBSCAN, short for Density-Based Spatial Clustering of Applications with Noise, is an unsupervised data clustering algorithm, introduced by Ester et al. [4] in 1996. DBSCAN is based on the concept of identifying dense regions of n-dimensional data, which can be measured by calculating the number of close objects to any given data point. Unlike k-Means, which assumes, that every point in the data set belongs to some cluster, density-based clustering algorithms like DBSCAN introduce the concept of noise, where some points are presumed to be outliers that do not belong to any cluster. Two important parameters that are needed for the clustering operation are *epsilon*, the neighborhood radius around a point and *minimum points*, a value that defines the minimum number of points in the epsilon neighborhood for a so-called *core point*. *Border points*, on the other hand, are data points that have less than the minimum number of neighborhood points, but in turn, belong to the epsilon neighborhood of another core point. Any other point that lies outside of those boundaries, and is neither a core nor a border point, is regarded as an outlier (i.e. noise) and will be ignored. Figure 5.4 illustrates all three point types used in DBSCAN, where X is a core, Y a border, and Z an outlier point.

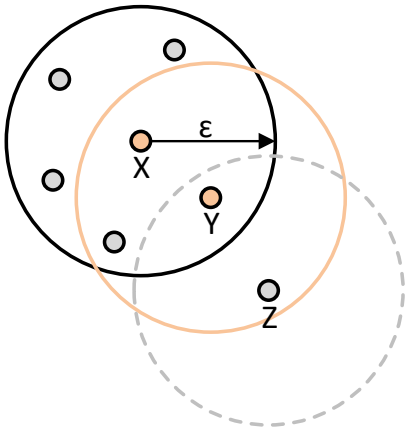


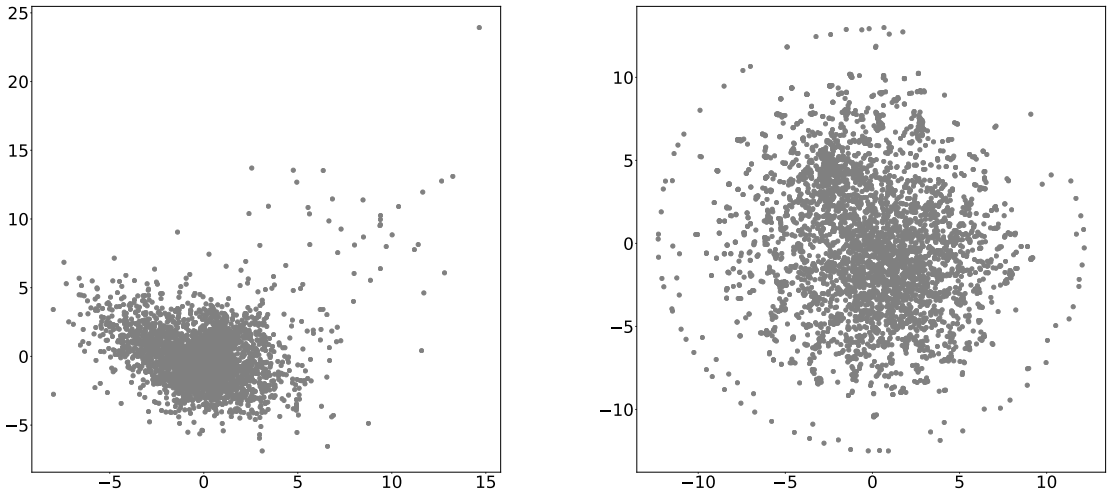
Figure 5.4: Core, border and outlier points used in DBSCAN.

The algorithm starts out with an arbitrary initial point. If the point has more than the minimum amount of neighboring points in a radius *epsilon*, it is labeled as a core point. Nearby border points are iterated and in turn, checked if they fulfill the criteria of a core point. If one of those points is indeed a core point, the cluster is grown from it. All neighboring points of the new core point are then in turn checked. After finishing this breadth-first search procedure, the cluster is finished, and a new arbitrary initial point that is not already part of an existing cluster is used as the starting point for the next cluster. If however, an initial point does not have the minimum amount of points in its neighborhood, it is labeled as noise and assumed not to be part of any cluster. Unlike k-Means, DBSCAN does not require a pre-defined number of clusters before running the algorithm. However, the two values *epsilon* and *minimum points* are crucial to achieving sensible clusters. Results can vary significantly based on those values, so they have to be chosen carefully. Rahmah and Sitanggang [11] proposed an algorithm that can automatically determine the optimal value for *epsilon*. The distance from every point to the nearest n points are calculated and sorted. In the same vein as the k-Means elbow method, which was explained in 5.2.2, the results are plotted, and the y-axis value on the *elbow* of the plot is chosen as the optimal *epsilon*. Again, document embeddings containing UI strings, layout code, and Smali source code are clustered using this algorithm. Afterwards, results of the DBSCAN clustering are contrasted with the results of the k-Means clustering.

5.2.4 Dimensionality Reduction Methods

In order to visualize document and word embeddings, the underlying high-dimensional vectors have to be reduced to two dimensions while trying to preserve the structure and inherent relationships present in the high-dimensional data. Moreover, reducing dimensions can also be useful when clustering document embeddings with k-Means and DBSCAN. Two dimensionality reduction algorithms are used in this thesis, namely Principal Component Analysis and t-Distributed Stochastic Neighbor Embedding.

Principal Component Analysis (PCA) is a linear dimensionality reduction method that transforms high-dimensional data to low-dimensional *principal components* that have maximum variance. First, the high-dimensional data is standardized, before computing a covariance matrix, that models the relationship between features. After computing the covariance matrix, the eigenvectors of the matrix are used as principal components, with the eigenvalues describing how much variance exists in the direction of their corresponding eigenvectors. Subsequently, the top n eigenvectors are used to create a new matrix that represents the data in lower n-dimensional space.



(a) PCA-reduced data in 2d vector space.

(b) t-SNE-reduced data in 2d vector space.

Figure 5.5: Dimensionality reduction of the same data with PCA and t-SNE.

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a non-linear dimensionality reducing machine learning algorithm, proposed by van der Maaten and Hinton [18]. It is well suited for high-dimensional data like word and document embeddings and is frequently used to visualize Doc2Vec model data. Whereas PCA tries to maximize variance for large distances between pairs of data points, t-SNE only preserves small pair-wise distances and local similarities by using the Gaussian distribution to create a probability distribution that defines the relationships of high-dimensional data points. In order to map the high-dimensional data to low-dimensional space, a mapping is created using the Student t-distribution, which recreates the probability distribution in low-dimensional-space. Afterwards, gradient descent is used to optimize the low-dimensional embeddings. Visualizations of the 2d representations of word and document embeddings use either PCA or t-SNE. Since both methods use vastly different approaches, the resulting visualization of word and document embeddings differ as well, as can be seen in figure 5.5, where the same data set is projected differently for each algorithm. Because of this, the dimensionality reduction algorithm is chosen based on the quality of the visualization on a case by case basis. However, clustering with k-Means and DBSCAN was done exclusively with PCA-reduced data, since t-SNE does not preserve distances and densities for low-dimensional data, both of which are needed for successful clustering tasks.

5.3 Summary

This chapter discussed the Doc2Vec models and the evaluation methods that were used for solving the three main research problems. Overall, seven Doc2Vec models are trained - four containing Smali source code, and one each for Google Play Store descriptions, UI strings, and layout markup code. The description model is used to determine similarities between applications based on their Google Play Store descriptions. This is evaluated by using the Doc2Vec similarity interface. This method is also used to determine if malware can be detected based on their relationships with other malware and benign applications, using the malware model. Moreover, the validation model uses the same method to determine if Doc2Vec models are capable of spotting fake clone applications based on Smali source code similarity. The remaining models - the layout, UI strings, and Smali model use two clustering algorithms called k-Means and DBSCAN, as well as the Doc2Vec similarity interface to analyze the semantic properties of the trained features. The layout model uses the layout code found in APK files, whereas the UI strings model uses the localizable UI strings found in strings.xml files. The Smali model uses Smali source code features found in the top 100 applications per category. In order to visualize document and word embeddings, their underlying high-dimensional vectors have to be reduced to two dimensions. This is done with PCA or t-SNE, which are chosen case by case for each model, based on the quality of the visualization. Additionally, clustering was done with PCA-reduced document embedding data. Results for both clustering algorithms and results for the other research problems are described in the following chapter.

Chapter 6

Results

This chapter discusses the results that have been acquired during the course of this thesis. Three main problems have been evaluated:

1. Can Google Play Store descriptions of applications be used for semantic analysis?
2. Can applications be semantically compared based on features found in APK files?
3. Is it possible to identify malware applications based on Smali source code similarity?

First, the choices and challenges that were faced during the data retrieval, data preparation, and model training processes are outlined in 6.1, including summaries for how many metadata and APKs were collected, the composition of trained Doc2Vec models, as well as training systems. Afterwards, the results that were found while evaluating the three main problems are discussed. The results and findings for the model trained with Google Play Store descriptions are described in 6.2, after which APK features and their semantic properties are discussed in 6.3. Lastly, the results for malware and clone detection based on Smali source code are explained in 6.4.

6.1 Data Retrieval, Data Preparation and Model Training

During the course of this thesis, Doc2Vec models have been trained, using different APK and metadata features and evaluated using similarity analysis, word analogy analysis, and clustering algorithms. However, before features could be trained, sufficient data had to be collected. Four different online data sources were used as a basis for the data retrieval process. First, the Google Play Store was crawled using a custom crawler bot. The bot collected metadata directly from the Play Store and used the collected package names to download APK files from a third-party APK downloader site. The crawling process continued in a recursive manner, by parsing the top-N similar package names from the Play Store HTML page and repeating the process for each of those package names. The process was initially started with the *com.facebook.orca* package name, representing the Facebook Messenger application, which is assigned to the *Communication* category. After processing the application, crawling continued using the top eight similar applications. Figure 6.2 illustrates the relationships and categories of applications that were subsequently crawled. Most notably, the categories *Communication* and *Social* appear to have substantial overlapping in terms of Play Store category similarity. Since the task of categorizing applications is left to the individual developer, applications that share the same context or purpose might end up in different categories. Moreover, Sanz et al. [13] acknowledge that applications are sometimes categorized into different categories solely to increase their visibility on the Play Store. As an example, figure 6.1 shows two versions of the Telegram messenger application, categorized into two different categories, *Communication*, and *Social*, respectively. Overall, this crawler bot collected metadata for 1,123 applications. However, since the third-party APK downloader site limited the amount of APKs that can be downloaded in quick

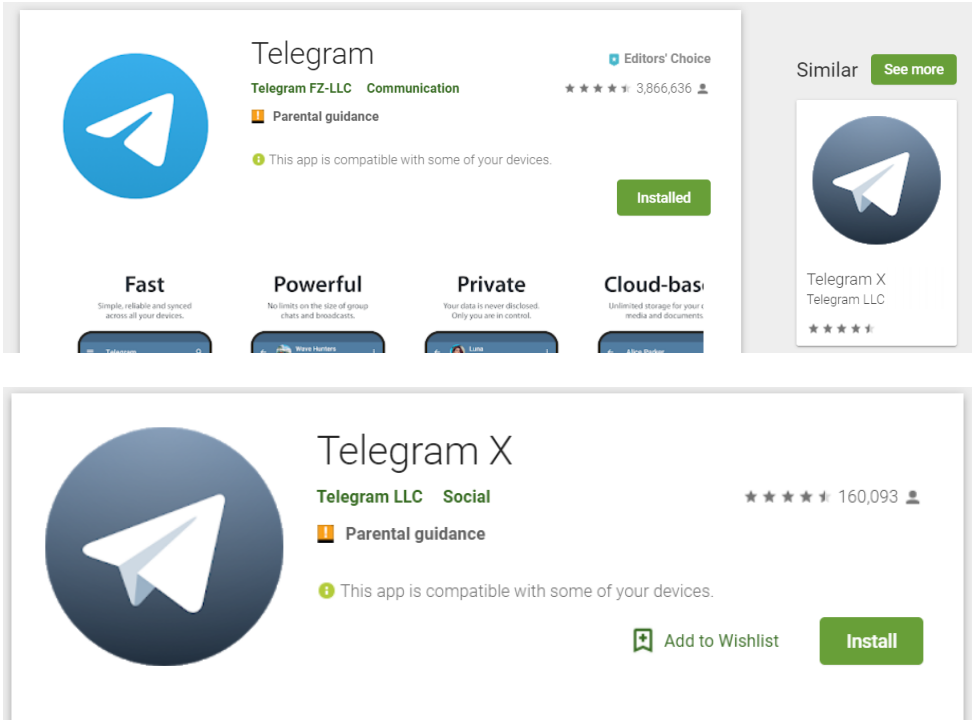


Figure 6.1: Two versions of the Telegram messenger application with different Google Play Store categories

succession, the crawling process was slowed down significantly and was aborted after only crawling 350 APKs. Because of this problem, a second crawler bot, which used an online APK and metadata dump hosted on the Internet Archive website, was subsequently developed. With this bot, it was possible to collect metadata for 1,160,516 applications and 4,480 APK files throughout a three-week crawling period. Furthermore, two additional data sets were collected as archive files containing APK files, which eliminated the need for additional crawler bots. Table 6.1 shows how many APK files and metadata were collected from the different data sources in the data retrieval process overall. Crawler bots, as well as the feature extraction process was first run on a single CPU home PC but was quickly taken over by the computer cluster at the Institute of Applied Information Processing and Communications, due to better performance when running concurrent processes on multiple CPU cores. Since the raw APK data surpassed 250GB disk space, with the amount of extracted Smali code features amounting for over 20GB, models using Smali source code features were also built using the cluster, reducing the calculation time significantly, while models with metadata, UI strings, and layout code features were built on the home PC with reasonable performance. Table 6.2 lists the systems and their specifications, which were used to data mine application metadata and APKs, extract features, and train Doc2Vec models. Tables 6.3 and 6.4 list the Doc2Vec models, their features, data source, the training platform, and the hyperparameters that were used in the training process. For natural language text features like descriptions and UI strings, the Distributed Memory training algorithm was used. All other models, using either Smali code or XML markup code, were trained using Distributed Bag of Words for document embeddings and Skip-gram for word embeddings. This decision was made based on the results of researchers at Lab41, who noted that this algorithm slightly outperformed Distributed Memory when training document embeddings of Python scripts. While using this mode resulted in a slower training process compared to Distributed Memory, this was mitigated by training the models on computer cluster nodes. For vector size, window size, epochs, and min count, values were used that lie in the range of recommended values published by gensim. The worker parameter was adjusted to fit the system the model was trained on.

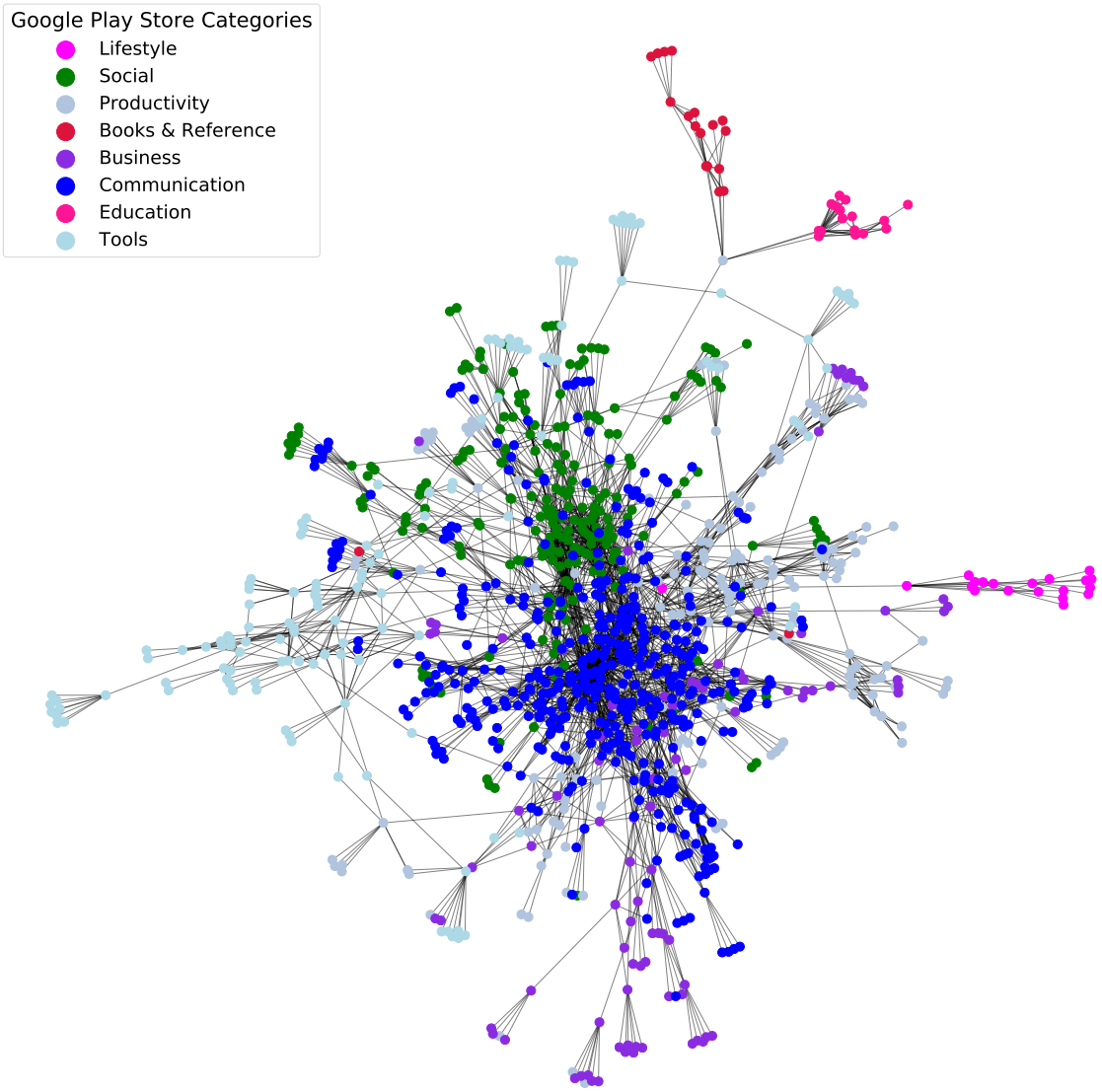


Figure 6.2: Similarity graph of Google Play Store application categories

Data source	Metadata	APKs
Google Play Store	1,123	350
Internet Archive data set	1,160,516	4,480
Android Validation data set	-	792
Malware data set	-	1,889
Total	1,161,639	7,772

Table 6.1: Total summary of how many metadata and APKs were gathered from different data sources

Description	CPU	RAM
Cluster node *xeon512g0*	2 x Xeon E5-2699v4, 44 Cores@max 3.6GHz (88 Threads)	512GB DDR4 RAM
Cluster node *nehalem192g0*	2x Xeon X5690, 12 Cores@3.46GHz (24 Threads)	192GB DDR3 RAM
Home PC	1x Intel i7-920, 4 Cores@2.66GHz (8 Threads)	16GB DDR3 RAM

Table 6.2: List of systems used to mine data, extract features and build models

Model	Feature	Data source	Training system
Descriptions	Play Store descriptions	Internet archive data set	Home PC
UI strings	UI strings	Internet archive data set	Home PC
Layout	Layout markup code	Internet archive data set	Home PC
Smali	Smali source code	Internet archive data set	xeon512g0
Validation	Smali source code	Validation data set	nehalem192g0
Malware	Smali source code	Malware data set + Internet archive data set	nehalem192g0

Table 6.3: List of all Doc2Vec models, their features, data sources and systems with which they were trained

Model	Model hyperparameters					
	Training algorithm	Vector size	Window size	Epochs	Min count	Workers
Descriptions	PV-DM	200	10	20	5	8
UI strings	PV-DM	200	10	20	5	8
Layout	PV-DBOW + Skip-gram	200	10	20	5	8
Smali	PV-DBOW + Skip-gram	200	10	15	5	88
Validation	PV-DBOW + Skip-gram	200	10	20	5	24
Malware	PV-DBOW + Skip-gram	200	10	15	5	24

Table 6.4: List of all Doc2Vec models and the hyperparameters that were used in the training process.

6.2 Application Similarity based on Google Play Store Descriptions

The first problem that was evaluated is, if document and word embeddings of application's Google Play Store descriptions can be used to find semantically similar applications. Moreover, it was tested if the model could produce accurate results for word analogies. The model consists of document and word embeddings of Google Play Store descriptions from applications collected from the Internet Archive data set. Since not all descriptions that were initially gathered from this data source are written in English, only a subset of descriptions was used to train the model. Figure 6.3 shows a histogram of the word count of descriptions after the pre-processing phase. Most descriptions hover around 20 to 40 words, while some go as high as 450 words. However, a substantial amount of descriptions have fewer than ten words, which is mostly due to the applied pre-processing techniques. Those too were discarded and not used in the model training process. Table 6.5 contrasts the number of descriptions that were used for training the model, with the number of descriptions that were discarded. Overall, the model was trained using 769,582 descriptions, containing 60,074,045 words and a vocabulary of 146,196 unique words. This section is split into three parts. First, the word embeddings are discussed. Afterwards, category embeddings, which are trained document embeddings of all descriptions from applications of a certain category are evaluated, before finally assessing document embeddings of descriptions.

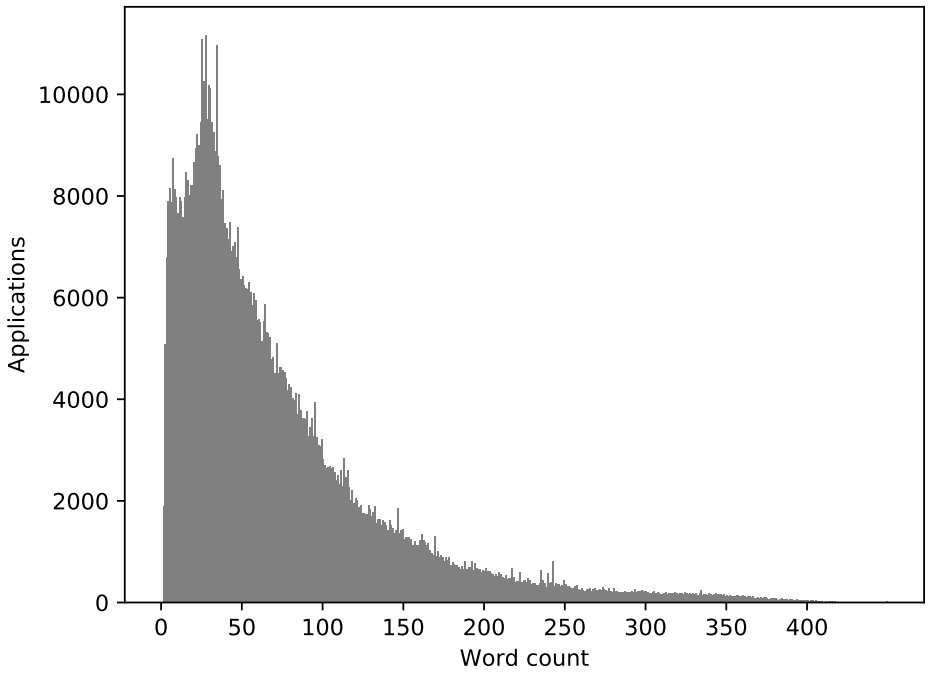


Figure 6.3: Frequency of words in Google Play Store descriptions

#	
1,160,516	Overall # of descriptions that were crawled
328,341	Non-English descriptions
62,593	English descriptions with a word count lower than 10
769,582	**Training set**

Table 6.5: Number of descriptions that were collected, discarded and used in the training process.

6.2.1 Word Embeddings

Doc2Vec trains word embeddings alongside document embeddings. Evaluating word embeddings is often easier than document embeddings, since their context (i.e. word similarities) is self-explanatory, especially when training a natural language text corpus, whereas document embeddings need some external label data that has to be interpreted in order to assess the quality of the results. Because of this, the Doc2Vec similarity interface was used to test the quality of the model's word embeddings, before evaluating document embeddings and combinations of word and document embeddings. The similarity scores and their associated words for different word queries are shown in the following four listings 6.1, 6.2, 6.3, and 6.4. All queries produced meaningful results with good similarity scores. The similarity query for *malware* produced different synonyms for the same word, whereas the query for *germany* resulted in different (European) countries. Interestingly, the query for *facebook* predicted a near 100% similarity to the word *twitter*. This might be due to the commonly used phrase *[find us, share, follow us,...] on facebook, twitter [...]*. Indeed, this phrase and variations thereof are found over 22,600 times within the data set, which explains the high similarity score between these two words. Lastly, the query for *messenger* returned words describing different messenger applications, as well as the plural form for the search word. This also happened for the *malware* query, which returned *virus* and *viruses*. Getting multiple results for the same word could have been mitigated by stemming and lemmatizing the word corpus before training. However, Le and Mikolov [7] and Dai et al. [2] did not use any stemming or lemmatizing in their experimental setups, and the pre-trained GoogleNews vectors[1] were also not pre-processed using these methods. Figure 6.4 illustrates the word embeddings for all four search queries in 2d vector space. As expected, the word embeddings of the *facebook* group and the *messenger* group are closest to each other, since the context for both is similar. The other two groups for *germany* and *malware* are concentrated quite nicely, which is further confirmed by figure 6.5, which shows the word embeddings of the same origin words, but this time with their top 200 similar words. With an increasing number of word embeddings, the groups for *facebook* and *messenger* start to overlap, since they share semantic similarities. Overall, the same segmentation is still visible with higher top-N values, which suggests that a Doc2Vec model using 769,582 Google Play Store descriptions is able to produce meaningful word embeddings.

```
model.most_similar('malware')
[(u'viruses', 0.8334431052207947),
 (u'antivirus', 0.826315701007843),
 (u'malicious', 0.8114053010940552),
 (u'spyware', 0.8081693053245544),
 (u'virus', 0.7919743061065674),
 (u'threats', 0.7588896751403809),
 (u'adware', 0.7371416687965393)]
```

Listing 6.1: Most similar words to *malware*

```
model.most_similar('germany')
[(u'italy', 0.7349196672439575),
 (u'france', 0.7297219634056091),
 (u'austria', 0.7014596462249756),
 (u'belgium', 0.7005186080932617),
 (u'finland', 0.6829525828361511),
 (u'spain', 0.6815527677536011),
 (u'sweden', 0.6807953119277954)]
```

Listing 6.2: Most similar words to *germany*

```
model.most_similar('facebook')
[(u'twitter', 0.9470336437225342),
 (u'email', 0.7984024286270142),
 (u'share', 0.7904098033905029),
 (u'friends', 0.7327014803886414),
 (u'mail', 0.7249593734741211),
 (u'sharing', 0.722528338432312),
 (u'social', 0.7129271030426025)]
```

Listing 6.3: Most similar words to *facebook*

```
model.most_similar('messenger')
[(u'kik', 0.6761946678161621),
 (u'whatsapp', 0.6522836685180664),
 (u'messengers', 0.647438645362854),
 (u'viber', 0.6429873108863831),
 (u'icq', 0.6173261404037476),
 (u'telegram', 0.6091070175170898),
 (u'skype', 0.5982436537742615)]
```

Listing 6.4: Most similar words to *messenger*

One feature of word embeddings is that they can be used to predict words based on analogies. Table 6.6 shows the results for some word analogy examples that were calculated using this model. While the

[1] https://code.google.com/archive/p/word2vec/

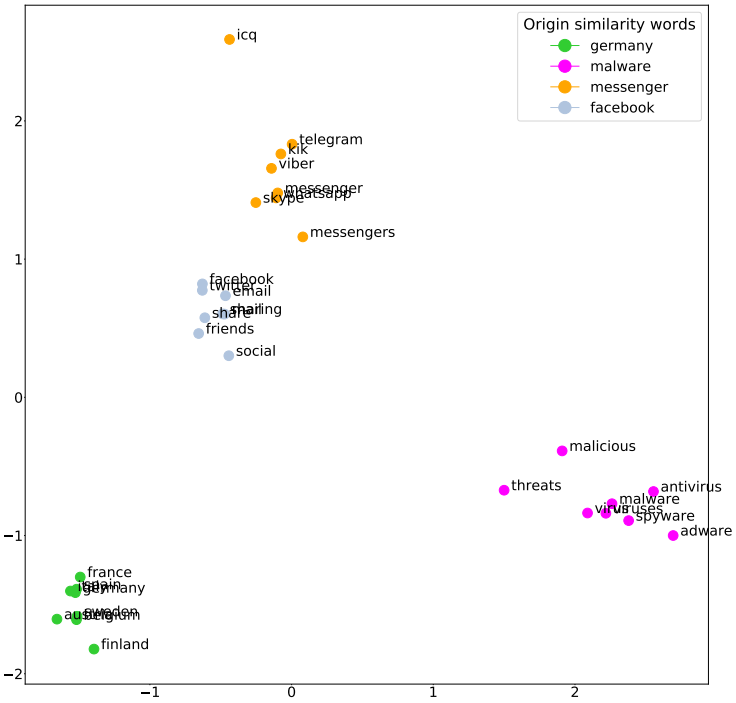


Figure 6.4: PCA-reduced word embeddings of *germany*, *malware*, *messenger*, *facebook*, and their top 7 similar words

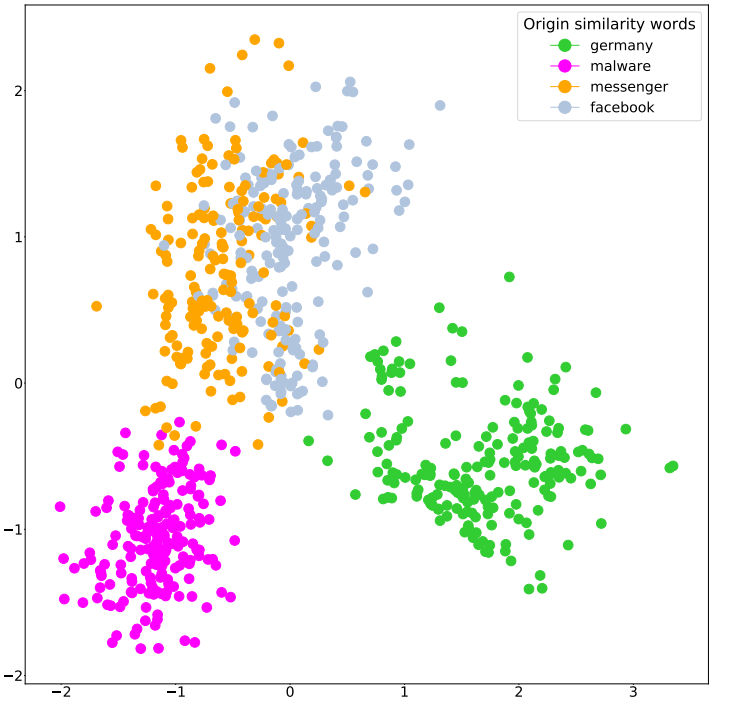


Figure 6.5: PCA-reduced word embeddings of *germany*, *malware*, *messenger*, *facebook*, and their top 200 similar words.

correct answer was not always the most similar result, with the exception of the last analogy, all correct answers are at least in the top five results. A larger and more varied text corpus will definitely improve the accuracy of the word analogy predictions.

Analogy	**Results (similarity scores)**
Man is to *Woman* what *King* is to...?	james (0.5724004507064819) **queen** (0.55512934923172) luther (0.545451283454895) daughters (0.5195327997207642) surnamed (0.517264723777771)
Germany is to *England* what *Berlin* is to...?	boston (0.6519495248794556) orleans (0.6105227470397949) **london** (0.6102681159973145) philadelphia (0.5963624715805054) birmingham (0.5921129584312439)
Boy is to *Girl* what *Man* is to...?	**woman** (0.7017908692359924) men (0.5938155651092529) valliere (0.5770934820175171) murder (0.570452868938446) mysterious (0.5656972527503967)
Google is to *Apple* what *Android* is to...?	*ipad* (0.5310356616973877) blackberry (0.4950388967990875) pux (0.48740434646606445) powerbook (0.46722352504730225) *iphone* (0.4537336230278015)

Table 6.6: Word analogy examples.

6.2.2 Category Embeddings

Before the training process, description embeddings were labeled with their associated application's package name and Google Play Store category. By doing that, the model trained embeddings for not only each description but also embeddings for each Play Store category. This *category embedding* can be seen as the mean embedding of descriptions for all applications, which belong to a specific Play Store category. Figure 6.6 shows the location of those category embeddings in 2d vector space. The plot shows some similarities between certain learned categories. Especially categories *Social/Personalization* and *Photography/Sports* appear to have a lot in common. However, when assessing the most similar applications for a category embedding, it quickly shows, that the original Play Store category often differs from the most similar category embedding. For example, table 6.7 shows the top five similar descriptions for the category embedding *Communication*. While their descriptions share the same context, the associated categories differ, which might indicate that the trained category embedding more accurately describes the semantic properties of applications than the developer-assigned Play Store category. This is further exemplified by illustrating the most common words for each category embedding, as seen in table 6.8. Every category embedding seems to produce sensible common words that share a semantic similarity with the category. With further refinement, for example, by building a classification system, trained category embeddings might be able to reliably predict the Play Store category for new applications based solely on the description. This system could then be used to automatically recommend suitable categories to developers while uploading applications to the Play Store.

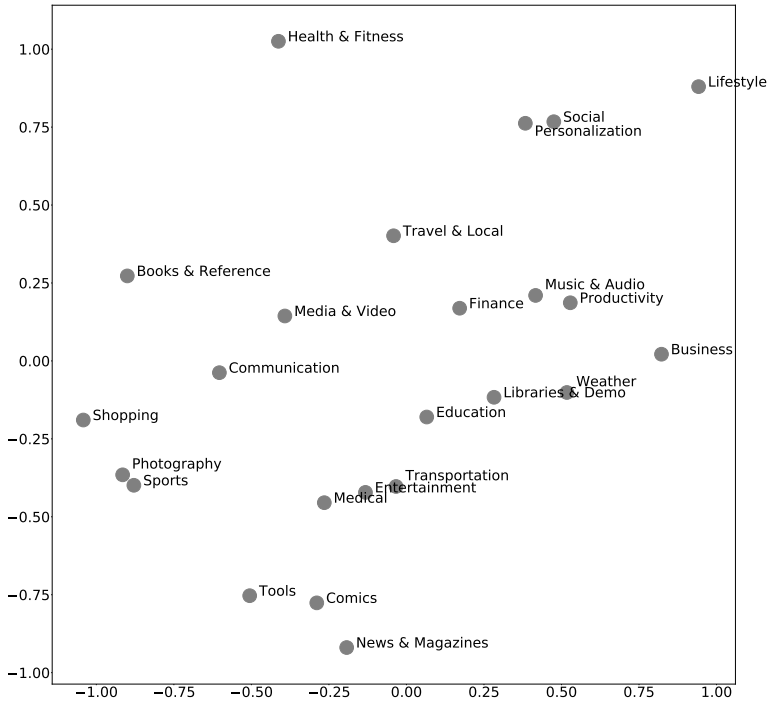


Figure 6.6: PCA-reduced document embeddings for Play Store categories (category embeddings).

Package name	Original category	Description
appinventor. ai_nazrinasirrmn. AutomaticSMS Response3	Tools	This app will reply your message or SMS automatically while you cannot reply it at that time. You also can set your message to response.
com.asr.smslater	Productivity	SMS Later is an application that enables you to send SMS messages whenever you want. You only have to specify the recipient phone number, the text and when to send the SMS.
com.rohdeschwarz. sit.topsecapp	Productivity	The TopSec Phone app is a VoIP client for encrypted and unencrypted voice communications. The app can be used for plain calls. For encrypted calls it is necessary to connect the TopSec Mobile encryption device via Bluetooth.
br.com.integriz. keepsms	Tools	BackupSms is an Android application that allows you to back up your SMS text messages. The messages are stored in a text file and optionally sent to your email. Can also delete all messages sent and or received. A very useful application!
com.koimiki.spyme	Social	This application is designed to monitor and automatically save sms, incoming and outgoing calls in the log file.

Table 6.7: Top five similar appliations for the category embedding for 'Communication'

Category	Words
Books & Reference	quote, need, book, time, work, help, application
Business	client, supplier, marketing, koenigrubloff, sale, business, corporate
Comics	comic, manga, story, horror, vampire, witch, alien, cartoon
Communication	message, call, sms, incoming, communication, contact, sender, outgoing
Education	teacher, learning, elementary, student, grammar, vocabulary, mathematics
Entertainment	horror, jacksepticeye, songpop, legendary, effect, soundboard, himym
Finance	financial, investment, transaction, mortgage, investor, loan, equity
Health & Fitness	diet, healthy, workout, exercise, anxiety, dieting, nutrition
Libraries & Demo	arduino, sdk, roadex, raspberry, accon, deltalogic, makerdroid
Lifestyle	spiritual, cured, bhaktas, thought, believe, remedy, giribapu
Media & Video	video, music, movie, playback, media, film, television
Medical	patients, medical, clinical, physicians, treatment, medication, dental, clinic
Music & Audio	music, song, listen, playing, saxophone, musician, story
News & Magazines	newspaper, information, world, headlines, politics, magazines, publication
Personalization	wallpaper, cat, twinkle, blured, smoke, pattern, theme
Photography	photo, camera, photography, picture, image, frame, photographer
Productivity	tasks, input, indiviual, versa, taskmaster, tasklist, utility
Shopping	shop, product, retailer, brands, deal, stores, coupon
Social	egroups, chatting, someone, invite, join, sending, chat, meet
Sports	football, soccer, team, basketball, tennis, hockey, club, championship
Tools	utility, device, admin, configuration, operation, specified, administrator
Transportation	vehicle, passenger, car, taxi, driver, ride, cab
Travel & Local	hotel, travel, tourist, restaurant, attractions, route, sightseeing
Weather	weather, forecast, radar, noaa, map, meteorological, estofex

Table 6.8: Most relevant words per category embedding.

6.2.3 Word & Description Embeddings

Word embeddings were used to query gensim's Doc2Vec similarity interface, to predict relevant applications based on their description embeddings. The similarity interface can accept a single word embedding, the output of vector arithmetic operations or two lists of word embeddings that contribute either positively or negatively to the resulting similarity score. Table 6.9 shows a selection of search queries and their results. Since the output is subjective, all resulting applications were manually categorized into relevant, semi-relevant, and non-relevant results based on their descriptions. While most queries returned semantically suitable applications, they are most of the time, not the most desirable results. For example, the search query *messenger* did not return any of the popular messaging applications as the most relevant results. Additionally, using a combination of two or more search terms only sometimes returned desirable results. While the search queries *berlin* + *map* and *london* + *taxi* produced relevant and semi-relevant results, the search query *food* + *paris* produced more non-relevant than relevant results. This occurred because additional informative metrics such as total downloads and ratings were not taken into account while training the model. Nevertheless, document and word embeddings trained with Google Play Store descriptions could be used as additional features in combination with other metrics in custom search engines, since they do predict semantically relevant applications, albeit not the most popular ones.

Search query	Relevant results	Semi relevant results	Non relevant results
mozilla	org.mozilla.firefox org.mozilla.firefox_beta	com.dataii.openbadges at.paul.firefoxwidget net.the4thdimension.mozillanews net.the4thdimension.firefoxapps tw.idv.gasolin.ffosud org.tint.firefoximporter	net.kokonotsu.addressbook com.wTBGWebsite
antivirus	com.antivirus.autoremove81 com.hotdog.antivirusautoremovevirus1 com.smilevillage.app.phonecleanvirusautoguide com.flame.antivirus com.apps.freeandroidvirusremoval com.hotdog.phonecleaningvirus2 com.remove.malware wb.virusdetectorforandroid com.homelbtscanmobilevirus com.smvscan.scanmobilevirus		
messenger	jp.my.findtalkfriends com.MSYApp.CepChat.WeLove com.agilemobile.im.android2 by.istin.android.vktalk com.facebook.katana	com.gofa.emoticons com.JorLais.emoticon_birthday com.JorLais.emoticon_natal com.plusgetteams.whatsappguide	std.iapp.tom_cute com.chudapa.hairpiecemyapp com.surinapp.photodecorkrondon
berlin + map	de.dmaicher.berlinmap com.festville.achtungberlin13 org.gc.metroberlino com.kuifangxu.BerlinMetro	com.axisapps.subwaymapseurope istanbul.maps com.appli.MetroBrussels org.gc.metromosca tapgap.london	
london + taxi	com.jothisofttech.willesden com.jothisofttech.chelsea com.autocab.taxibooker.tripleaexpress.hayes com.didmo.magandxxairportcars appinventor.ai_sazfarz.Minicab com.airportfare com.jothisofttech.birminghamtaxibooking com.jothisofttech.bristoltaxibooking com.jothisofttech.brightontaxibooking com.jothisofttech.stanstedtaxibooking		
food + paris	com.paristravel com.carin.eiffeltowerparisfrancelwp com.parism com.syscraft.frenchbakery		com.tts.android.fd.zhtw com.tts.android.hr.zhtw com.goomeoevents.sialmiddleest com.hanintel com.nakaborigawa.katabiragawa com.quickmobile.Cartier_Mtg

Table 6.9: Doc2Vec similarity interface search query results.

6.3 Semantic Analysis of Applications based on APK Features

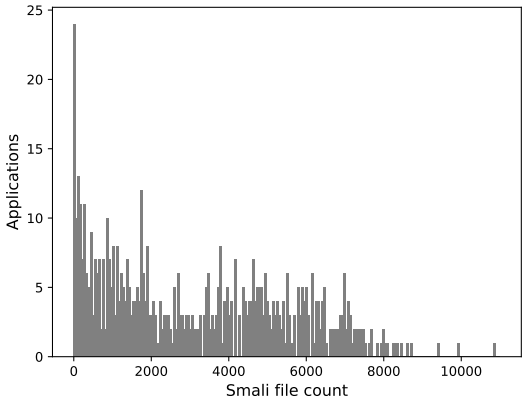
To evaluate the semantic properties of UI strings, layout code, and Smali source code, three models were trained using these features. Since APKs consist of multiple layout and Smali source code files, all files that were used in training, were only labeled using the associated package name, resulting in one embedding per application, instead of one embedding per feature file. This is in contrast to the UI string model, where only one UI string document per APK exists. However, one problem with this approach is that this additional abstraction layer makes it difficult to interpret and evaluate results. So far, the quality of word embeddings has been an indication of whether the model is reasonably trained or not. However, with formal language texts like layout or Smali code, this becomes increasingly difficult. External label data has to be used alongside different metrics like package names, Play Store categories, or an application's purpose of use, to produce results that can be interpreted. Figure 6.7 shows the frequency of layout and Smali files as well as the UI string word count of APK files that were used in the training process. While the layout file count for applications tapers out at around 400 to 450 files, Smali file count can reach well over a few thousand files for most applications. This is due to the baksmali process, which creates a Smali file for each class that exists in the original source code. Most UI strings have a word frequency from around 50 to 500 words, which is quite close to the word distribution for Google Play Store descriptions, as seen in figure 6.5. Table 6.10 lists how many APKs and feature files were used in training those models. First, the UI string model's word embeddings are discussed, using evaluation methods akin to the previous model. Afterwards, the results of the clustering of UI strings, layout, and Smali embeddings using k-Means and DBSCAN are described.

Model	APK count	Feature files count
UI strings	4,480	4,480
Layout	1,959	219,324
Smali	600	1,914,462

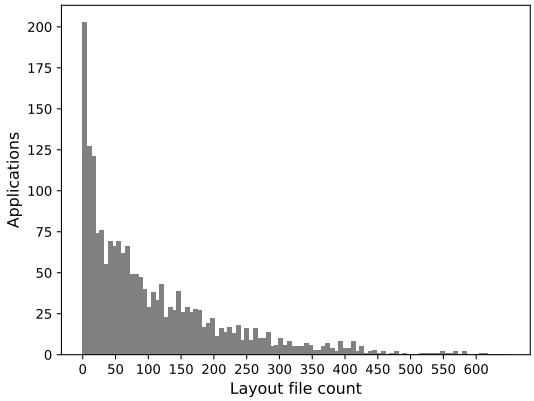
Table 6.10: APK and feature file count for layout, UI strings and Smali models.

6.3.1 UI String Word Embeddings

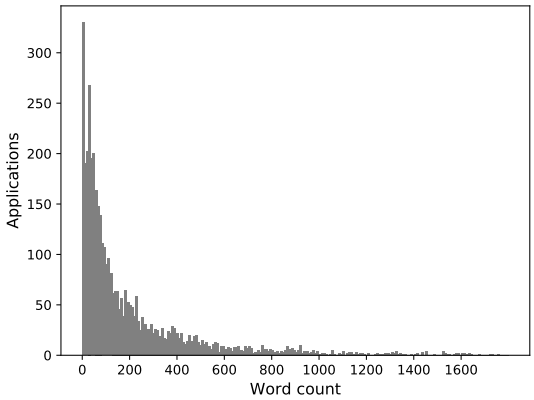
Since UI strings consist of natural language texts, the same analysis methods that had been used in 6.2 could be used to evaluate this model. UI string words embeddings were evaluated, by using the similarity interface to query the top similar words for *malware* (6.5), *germany* (6.6), *facebook* (6.7), and *messenger* (6.8). While the results are passable, a drop in the quality of the word embeddings compared to the word embeddings of the description model can be observed. Notably, similarity scores have decreased to a mean of 0.5077 in contrast to the mean of 0.7269 for the description model examples. Furthermore, 2d plots of those word embeddings, illustrated in figures 6.8 and 6.9, show a much less clean separation between the word embeddings for both 7 and 200 similar words respectively. This drop in quality can be explained by the reduced training size of the UI string corpus compared to the description corpus. While the description model consists of 60,074,045 words and a vocabulary of 146,196 unique words, the UI strings model only features 4,029,542 words and a vocabulary of 19,391 unique words. The drop in quality might indicate that the size of the data set is too small to produce quality embeddings. Furthermore, UI string files contain lots of noise, like placeholder values, unused strings, mixed localizations, and are generally not as pure as descriptions.



(a) Frequency of Smali files in applications.



(b) Frequency of layout files in applications.



(c) Frequency of words in UI strings.

Figure 6.7: Histograms of the frequency of files and words of APK features.

```
model.most_similar('malware')
[(u'spyware', 0.6604266166687012),
 (u'trojans', 0.5825715065002441),
 (u'virus', 0.5814374685287476),
 (u'threat', 0.5707137584686279),
 (u'viruses', 0.5645267367362976),
 (u'threats', 0.5536139011383057),
 (u'malicious', 0.5313403606414795)]
```

Listing 6.5: Most similar words to *malware*

```
model.most_similar('germany')
[(u'spain', 0.7110351920127869),
 (u'ireland', 0.702124834060669),
 (u'poland', 0.6477463245391846),
 (u'berlin', 0.6470950841903687),
 (u'netherlands', 0.6433355808258057),
 (u'malay', 0.6379404664039612),
 (u'hamburg', 0.6324115991592407)]
```

Listing 6.6: Most similar words to *germany*

```
model.most_similar('facebook')
[(u'twitter', 0.4447627067565918),
 (u'google', 0.42380857467651367),
 (u'etmek', 0.3995349705219269),
 (u'socialize', 0.3990570306777954),
 (u'openly', 0.3971394896507263),
 (u'vkontakte', 0.38915443420410156),
 (u'mulai', 0.3831460773944855)]
```

Listing 6.7: Most similar words to *facebook*

```
model.most_similar('messenger')
[(u'ted', 0.40609607100486755),
 (u'hey', 0.39229047298431396),
 (u'ascension', 0.3913536071777344),
 (u'kik', 0.3842655420303345),
 (u'unfortunatelly', 0.3825146555900574),
 (u'fring', 0.37944777796554565),
 (u'wanna', 0.3770503103733063)]
```

Listing 6.8: Most similar words to *messenger*

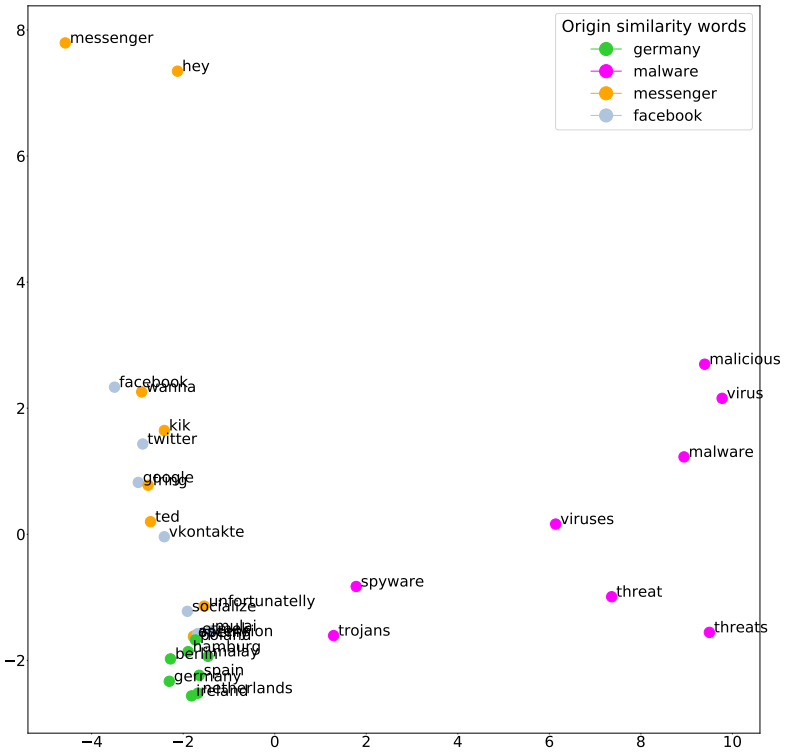


Figure 6.8: PCA-reduced word embeddings of origin words and top 7 similar words.

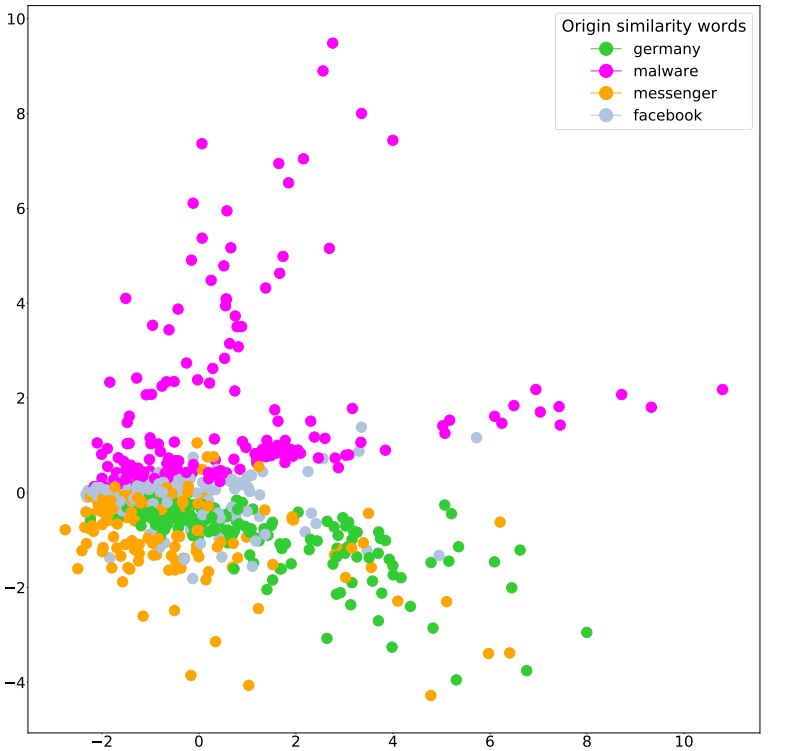


Figure 6.9: PCA-reduced word embeddings of origin words and top 200 similar words.

Table 6.11 shows word analogies that have been predicted using UI string word embeddings. The quality is noticeably worse than the description word embedding analogies. Because UI strings inherently have a more restricted, technology-focused context than descriptions, it makes sense that only the Google/Apple analogy was successfully predicted. However, the similarity scores are relatively low, even for the successfully predicted analogy.

Analogy	**Results (similarity scores)**
Man is to *Woman* what *King* is to...?	pushups, (0.5283721685409546) promociones, (0.5011328458786011) lng, (0.4875529408454895) compra, (0.4867666959762573) pron, (0.48676323890686035)
Germany is to *England* what *Berlin* is to...?	disagreement, (0.4282784163951874) rpg, (0.4111740291118622) amorous, (0.4092555046081543) hostile, (0.3944903612136841) phenomenon, (0.38355737924575806)
Boy is to *Girl* what *Man* is to...?	isle, (0.6773624420166016) darth, (0.6761913299560547) creed, (0.6503317356109619) blonde, (0.6373347043991089) cake, (0.6372594833374023)
Google is to *Apple* what *Android* is to...?	jailbroken (0.40168413519859314) ipad (0.370983362197876) iphone (0.3450864553451538) **ios** (0.3376222252845764) honeycomb (0.3229621648788452)

Table 6.11: Word analogy examples for the UI string model word embeddings.

Results for the word embeddings for UI strings indicate that a data set of 4,480 APK files contains not enough UI string samples to train quality word embeddings. A noticeable drop in quality, compared to word embeddings from Google Play Store descriptions could be observed. While data size for UI strings is relatively small, extracting the features with Apktool creates a lot of overhead, since additional unused files are decompiled as well, which results in slow extraction processes. Additionally, collecting large amounts of APK files is time-consuming. However, with more data and more refined pre-processing techniques, it would definitely be possible to create embeddings that match the quality of the word embeddings of Google Play Store descriptions.

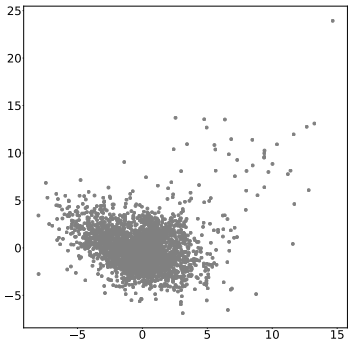
6.3.2 Clustering Applications based on APK Features

DBSCAN and k-Means were used to train clusters of three different types of Doc2Vec models containing application embeddings of UI strings, layout code, and Smali code. In order to reduce the dimensions of the underlying vectors, PCA is applied to all high-dimensional application embeddings. Effectively, the embeddings were reduced from 200 to 2 dimensions. Figure 6.10 visualizes the reduced data in 2d vector space. Often, these plots can be an indicator of the success of potential clustering applications. Because of this, t-SNE-reduced data was also visualized in figure 6.11, to further gain information about the structure of the data. UI string embeddings seem to concentrate around one giant structure both in PCA and t-SNE reduced space, making it hard for potential clustering algorithms to produce meaningful results. On the contrary, layout code embeddings seem to have 3-4 distinct structures, whereas Smali code embeddings are spread out more evenly and only seem to cluster in smaller structures, both in t-SNE and PCA reduced space, making both models far more likely to produce sensible clusters. Since k-Means needs a pre-defined number of clusters, the elbow method is used to find the optimal value for k. For each model, 50 k-Means passes were done, and the sums of the squared Euclidean distances were plotted. Afterwards, the values that lie on the *elbow* of the plots were chosen as the optimal number of clusters for the k-Means algorithm. DBSCAN, on the other hand, calculates the optimal amount of clusters automatically. However, clustering results can vary significantly based on the values of *epsilon* and the number of *minimum points*. Therefore, the method proposed by Rahmah and Sitanggang [11] was used to determine the value for *epsilon*. Different values for the number of *minimum points*, ranging from 2 to 5 were tested manually, and the value with the best results was chosen. Figures 6.12 and 6.13 show the resulting plots for both methods. The values on the *elbow* of the plots are chosen as the optimal k, as well as optimal *epsilon*. Ultimately, table 6.12 lists the selected values for both k-Means as well as DBSCAN for each model. Interestingly, the optimal value of clusters for k-Means for the Smali models is the same as the number of Google Play Store categories from which applications were used in training the document embeddings. However, this could be a coincidence, and due to the small sample size of 600 applications.

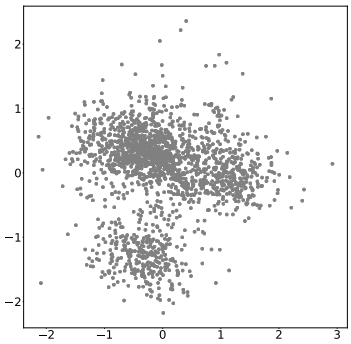
Model	**k-Means**	**DBSCAN**
UI strings	k = 6	epsilon = 1, minimum points = 3
Layout	k = 4	epsilon = 0.1, minimum points = 3
Smali	k = 6	epsilon = 0.15, minimum points = 3

Table 6.12: Values for k-Means and DBSCAN hyperparameters per Doc2Vec model.

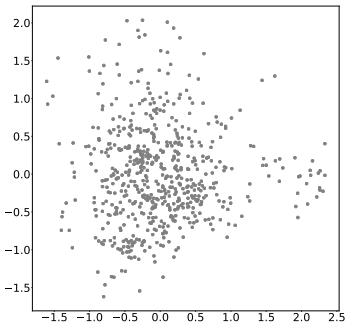
A big challenge in clustering high-dimensional data, apart from projecting the data set to lower dimensions without losing the structure and inherent relationships, is interpreting the results. Since document embeddings introduce an abstraction layer similar to deep learning, external label data has to be used to evaluate the quality of the finished clusters. In order to accomplish this task, a selection of applications per cluster were chosen and manually reviewed. For k-Means, applications that are nearest to the cluster's centroids were selected, whereas DBSCAN clusters were randomly sampled for applications. Afterwards, the selected applications were manually reviewed and evaluated based on the semantic similarities of the underlying feature type. A more thorough evaluation could be done by manually tagging every application based on the semantic properties of the feature type. However, this process would be very time-consuming. Moreover, the similarity interface was used to calculate the average similarities score values for all clusters. Those values were then, in turn, averaged to calculate a mean similarity score per clustering algorithm.



(a) UI string embeddings

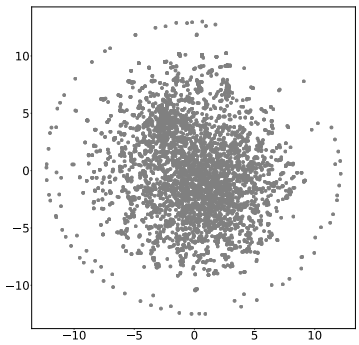


(b) Layout code embeddings

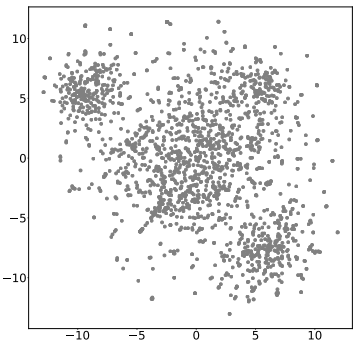


(c) Smali code embeddings

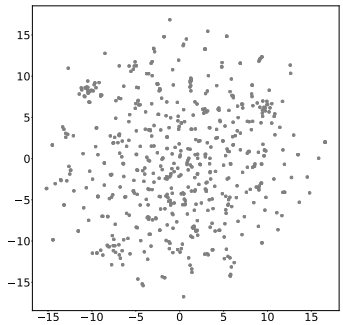
Figure 6.10: PCA-reduced document embeddings of APK features.



(a) UI string embeddings



(b) Layout code embeddings



(c) Smali code embeddings

Figure 6.11: t-SNE-reduced document embeddings of APK features.

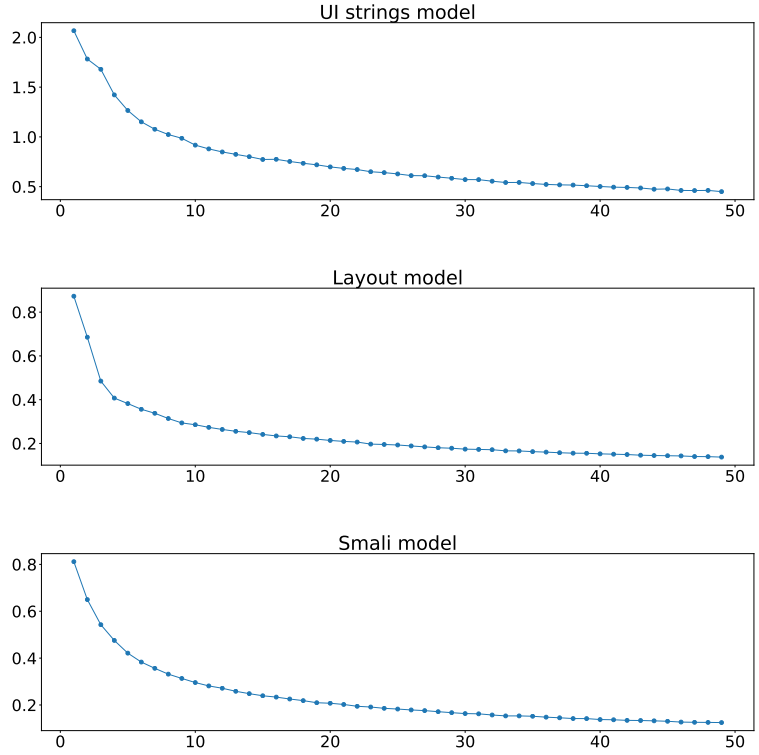


Figure 6.12: Elbow method plots of UI strings, layout and Smali embeddings used for finding optimal k for k-Means.

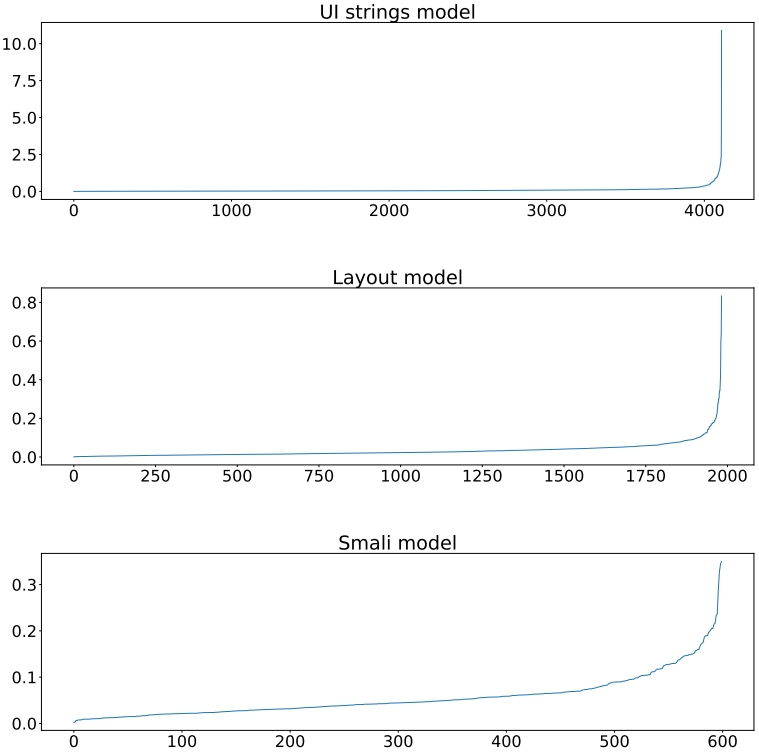
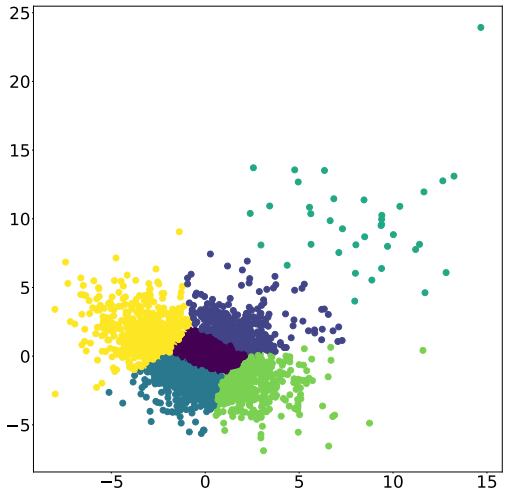
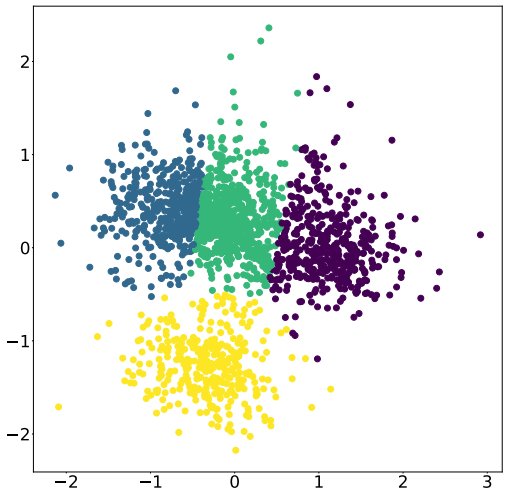


Figure 6.13: Nearest neighbor method plots for UI strings, layout and Smali embeddings used for finding optimal epsilon for DBSCAN.

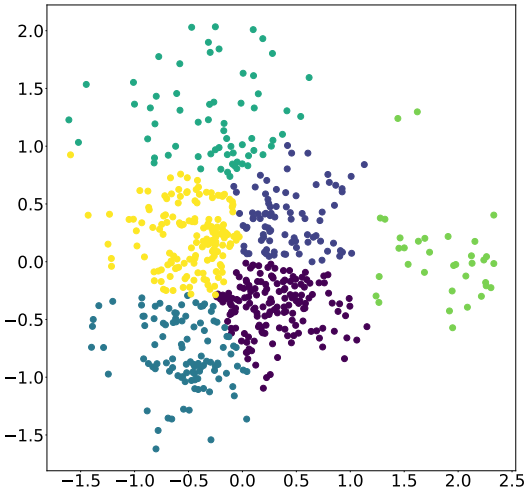
Figures 6.14 and 6.15 show the finished clusters for k-Means and DBSCAN, respectively. As expected, both clustering algorithms had a hard time fitting clusters to the data set of UI string embeddings. Where k-Means allocated the sparely populated region in the upper right part to the nearest cluster, DBSCAN discarded this region almost completely as noise. Since most applications center around a large structure in the lower-left half of the vector space, DBSCAN regarded it as one big cluster, whereas k-Means tried to fit the pre-defined amount of clusters to this structure. Interestingly, DBSCAN automatically determined the same amount of clusters that were selected as k from the elbow method plot but fitted those clusters in a completely different way. Both clustering algorithms separated the data set for layout code embeddings in two big structures. Where DBSCAN fit smaller clusters around those two big structures, k-Means segmented the larger structure into three smaller clusters. This might once again be due to the fact that k-Means tries to fit the specified number of clusters that were pre-defined before training.



(a) UI string embeddings (k=6)

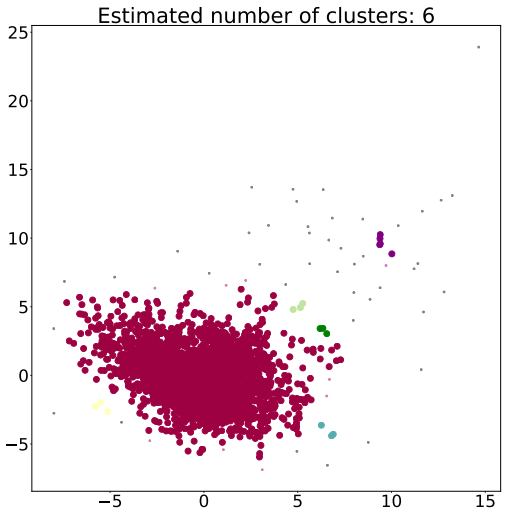


(b) Layout code embeddings (k=4)

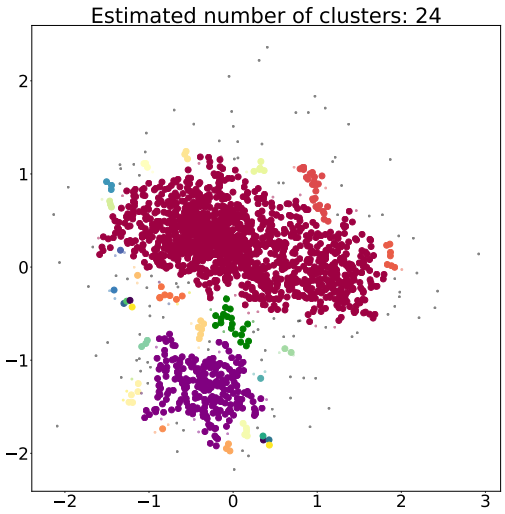


(c) Smali code embeddings (k=6)

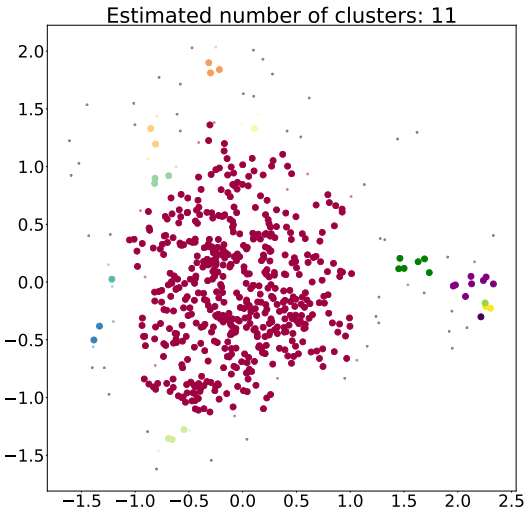
Figure 6.14: k-Means clustering of APK features.



(a) UI string embeddings



(b) Layout code embeddings



(c) Smali code embeddings

Figure 6.15: DBSCAN clustering of APK features.

In order to evaluate the quality of the resulting clusters, the mean similarity of applications within clusters was calculated with the help of the Doc2Vec similarity interface, by averaging the similarity scores of all applications in a cluster and, in turn, averaging all cluster similarity scores. Table 6.13 lists the similarity scores of all models for both clustering algorithms. Unsurprisingly, the mean similarity scores for the UI strings model are very low compared to the other two models. For layout code and Smali code embeddings, the k-Means algorithm averaged a significantly better similarity score than DBSCAN. It seems as though that some information about the structure and relationships is lost while reducing UI string, layout, and Smali embeddings to lower-dimensional space with PCA. In order to remedy this problem, different dimensionality reduction methods could be tried, as well as other approaches to clustering, for example, by using higher-dimensional data that retain more of the semantic information. Unfortunately, manual review of the selected applications per cluster did not give clear insights into their semantic similarity. The distribution of applications within clusters seemed random, especially for the UI strings model. This

might be due to the lack of training size, wrong pre-processing techniques, or an inherent inability of the trained feature to model the semantic characteristics of an application. However, quality UI string word embeddings could be generated by increasing the training size, as indicated by the promising results when using a small sample size.

	Mean similarity score	
Model	**k-Means**	**DBSCAN**
UI strings	0.0033	0.0696
Layout	0.6189	0.4952
Smali	0.5059	0.3810

Table 6.13: Mean similarity of all clusters.

6.4 Malware and Clone Detection based on Smali Source Code

The malware model was trained using application embeddings of 600 benign and 600 malware applications, as well as consolidated malware and benign class embeddings. Those class embeddings were then used in a classification task, using 200 unseen samples of malware and benign applications. It was evaluated if the model could successfully predict the class (malware/benign) for the 200 test set applications. Furthermore, the validation model, which was trained using the Smali source code files from applications of the validation data set, was used, to predict the original application based on transformation samples. Both unseen and seen samples (i.e. samples that were or were not known to the model while training) were used in this classification task. Both classification tasks were performed using the Doc2Vec similarity interface by querying the model for the top 1 similar prediction. As an example, listing 6.9 shows a successful class prediction for the malware *mw_Color_Grab_2_2_0*. Table 6.14 shows the accuracy of both classification tasks. The validation model was very successful in finding the original sibling applications based on the transformation applications, with both accuracy scores well over 90%. This is corroborated by figure 6.16, which shows the application embeddings of sibling applications in close proximity. However, classifying applications as malware or benign proved to be not as accurate, as the model only classified 65.52% of samples correctly. Additionally, the benign test set was used to find potential similarities between benign applications and malware applications. Only two applications have scored a similarity higher than 0.6, as seen in listings 6.10 and 6.11. The similarities between *com.netqin.aotkiller*, a performance-boosting application and the malware *taxi.apk* are hard to find, especially because there was no context provided for the applications of the malware data set. However, the context for the second similarity query between the benign application *embware.phoneblocker* and the malware *Blocker_4.95.apk*, could be deduced as call blocking functionality, that is shared between both applications. Lastly, 600 application embeddings of Smali source code were used to find high-similarity application pairs. Out of 600 applications, 247 application pairs had a similarity score greater than 0.8. Most of those application pairs seem to have similar semantic context. Listing 6.12 illustrates a few examples of high-similarity application pairs that were found.

```
model.docvecs.most_similar('mw_Color_Grab_2_2_0.apk')
[('malware', 0.5107153654098511),
 ('benign',  0.17020379006862641)]
```

Listing 6.9: Malware classification using Doc2Vec similarity interface

```
model.docvecs.most_similar('com.netqin.aotkiller')
[(u'taxi.apk', 0.6075764894485474)]
```

Listing 6.10: Most similar malware application to *com.netqin.aotkiller*

Task	Model	Accuracy
Finding sibling applications (unseen samples)	Validation	95.31%
Finding sibling applications (seen samples)	Validation	98.50%
Classifying malware and benign applications (unseen samples)	Malware	65.52%

Table 6.14: Accuracy scores of classification tasks.

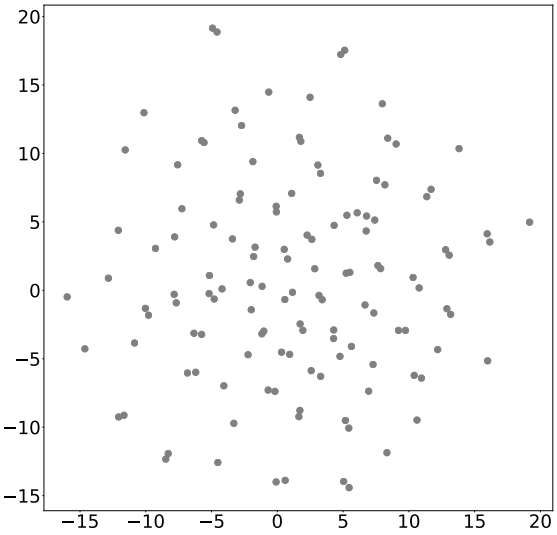


Figure 6.16: t-SNE-reduced document embeddings of Smali source code of applications from the Validation data set.

```
model.docvecs.most_similar('embware.phoneblocker')
[(u'Blocker_4.95.apk', 0.6302837133407593)]
```

Listing 6.11: Most similar malware application to *embware.phoneblocker*

```
[(u'com.antivirus', u'com.antivirus.tablet'): 0.9349891543388367,
 (u'com.antivirus', u'com.avg.cleaner'): 0.9013761281967163,
 (u'com.avg.cleaner', u'com.antivirus.tablet'): 0.8721846342086792,
 (u'org.mozilla.firefox', u'org.mozilla.firefox_beta'): 0.9359119534492493,
 (u'nh.smart', u'nh.smart.card'): 0.9664639234542847]
```

Listing 6.12: Examples of high-similarity application pairs

Overall, the results for the malware classification task were far from impressive. Apart from the classifier accuracy, one big concern was training and testing performance. Since applications can contain thousands of Smali classes, all of which have to be converted to document embeddings, the Doc2Vec training process, as well as the classifying process, was time-consuming. Pre-processed Smali features included in the malware model data set amounted to over 50GB of data, which might be the central issue in terms of performance. In order to remedy this problem, models could be built in PV-DBOW mode without training word embeddings, though the quality of the model would have to be evaluated anew. A potentially better approach would be only to use a subset of source code that is more significant than, for example, some commonly used code library. Nevertheless, with more samples combined with additional metrics and feature selection processes, a robust classifier might be possible. Finding clone applications, however, delivered promising results, even with a model with relatively small sample size. The validation model successfully predicted over 90% of test samples from previously seen or unseen data sets. Moreover, 600 application embeddings trained with Smali source code were used to find 247 high-similarity application pairs, most of which share a semantic context.

6.5 Summary

Results showed that a Doc2Vec model that was trained using Google Play Store descriptions contains semantic information of relationships between applications. It was possible to produce meaningful word embeddings and even calculate simple word analogies. Moreover, category embeddings, which contain information about every description from applications of a certain Google Play Store category, produced common words for each category that were semantically sensible. With further refinement, for example, by implementing a downstream classification task, those category embeddings could be used to automatically predict a fitting Play Store category for applications based solely on their descriptions. Furthermore, word and description embeddings could be used to search for relevant applications. Results showed that while the predicted applications were semantically relevant, they were often not the most desirable or popular choices. This could be mitigated by using those embeddings alongside additional metrics like download count or rating in a downstream recommender task. With decreasing training set size, the quality of embeddings decreased as well. This could be directly observed with the word embeddings generated from UI strings, that were extracted from 4480 APKs, which resulted in only a tenth of the word information available to the description model. While the word embeddings still produced sensible predictions, a noticeable drop in quality compared to the description model could be observed. Nevertheless, a bigger training size will help in increasing the quality of the word embeddings. Document embeddings of those UI strings, alongside layout code and Smali code embeddings from the same set of APKs, were subsequently used in downstream clustering tasks using k-Means and DBSCAN. However, results were inconclusive, since the resulting clusters appeared not to contain applications that share a semantic similarity. This again might be due to the small sample size, but could very well be an inherent inability of the selected features to model semantic characteristics of their applications. However, it could be calculated, that with the selected hyperparameters for k-Means and DBSCAN, the mean similarity scores for clusters was better with k-Means than with DBSCAN. Lastly, malware and clone applications were classified by using Smali source code embeddings as well as consolidated class embeddings trained from 600 benign and 600 malware applications. While the malware classification task only resulted in a 65.52% accuracy, spotting clone applications with the validation model was done with an accuracy of over 90%. Furthermore, it was possible to identify 247 high-similarity application pairs from a set of 600 application embeddings trained with Smali source code.

Chapter 7

Conclusion

The goal of this thesis was to use semantic text analysis to analyze semantic relationships between Android applications. Specifically, three research problems were considered, elaborated, and the found results evaluated:

1. Can Google Play Store descriptions of applications be used for semantic analysis?
2. Can applications be semantically compared based on features found in APK files?
3. Is it possible to identify malware or clone applications based on Smali source code similarity?

In order to answer these problems, four types of features, including Google Play Store descriptions, as well as UI strings, layout, and Smali code found in APK files, were used to create document and word embeddings by training Doc2Vec models. Document embeddings map documents to n-dimensional vectors of real numbers. In a trained Doc2Vec model, documents with different contents but similar meaning produce document embeddings that are located in close proximity to each other in the vector space. These embeddings can then be used for downstream learning tasks such as classification or clustering. The same principles apply to word embeddings. However, instead of modeling entire documents, word embeddings only model single words and their semantic meaning. Training quality document and word embeddings necessitates a large text corpus. In order to collect the required amount of data, four different sources of data were mined. First, the Google Play Store was used to collect application package names and their corresponding Play Store categories, which in turn were used to download APK files from a third-party APK downloader website. Using this method, metadata for 1,123 applications could be collected. However, since the third-party APK downloader site denied additional download link requests after 3-5 successful downloads until a cooldown period had passed, the crawling process was slowed down significantly and was aborted after only crawling 350 APKs. This problem sparked the development of a second crawler bot, which used an online APK and metadata dump hosted on the Internet Archive website. With this bot, it was possible to collect metadata for 1,160,516 applications and 4,480 APK files throughout a three-week crawling period. Furthermore, two additional data sets, the Android Validation data set created by Gonzalez et al. [5] at the Canadian Institute for Cybersecurity at the University of New Brunswick, as well as a data set containing malware samples were collected as archive files containing APK files, which eliminated the need for additional crawler bots. Both data sets contained 792 and 1,889 APKs, respectively. Overall metadata for 1,161,639 applications and 7,772 APK files were collected. After collecting metadata and APK files, the features were extracted and stored in JSON files, before being pre-processed using different NLP techniques, and used as input for six different Doc2Vec models.

The three methods that were used to evaluate the trained Doc2Vec models included the Doc2Vec similarity interface and two clustering algorithms, namely k-Means and DBSCAN. The Doc2Vec similarity interface provides functionality to semantically compare two or more document or word embeddings from within a model, without the need for separate machine learning tasks like classification or clustering. It uses

the cosine distance between the embeddings to calculate a similarity score that is normalized to values between -1 and 1. Moreover, it is possible to compare unseen samples to the trained set of embeddings by using the model to infer new document/word embeddings and using them as input for the similarity interface. k-Means and DBSCAN are unsupervised data clustering algorithms, that were used to cluster document embeddings of UI strings, layout, and Smali code. Whereas k-Means needs a pre-defined number of clusters, DBSCAN computes the optimal number of clusters based on two hyperparameters *epsilon* and *minimum points*. The optimal k for k-Means was selected by plotting the sum of squared distances for 50 passes of k-Means and selecting the value on the elbow of the plot. Similarly, the optimal value for *epsilon* was chosen by using a method proposed by Rahmah and Sitanggang [11], which also uses the y-axis value on the elbow of the resulting plot to determine the optimal value. These three methods were used to evaluate the research problems mentioned above.

Can Google Play Store descriptions of applications be used for semantic analysis?

Results showed that a Doc2Vec model trained with descriptions from the Google Play Store was able to create meaningful word embeddings that contain semantic information. The model was even able to calculate simple word analogies. Category embeddings that contain semantic information about all descriptions from applications of a specific Play Store category were able to predict common words for each category which were semantically sensible. For example, using the category embedding for the category *Media & Video* to predict the most similar word embeddings resulted in the following words: *video, music, movie, playback, media, film, television*. With the help of these category embeddings, it was possible to assign new category labels for applications solely based on their Play Store descriptions. Those category embeddings could potentially be used to automatically recommend suitable categories to developers while uploading applications to the Play Store. Furthermore, word and description embeddings could successfully be used to predict relevant applications based on single keywords or combinations of keywords. While results showed that predicted applications were semantically similar for most test cases, the resulting applications were not the most relevant or popular suggestions. This could be mitigated by using additional metrics like download count or rating alongside the existing description embeddings in a custom search engine system.

Can applications be semantically compared based on features found in APK files?

Three models were trained using UI strings, layout, and Smali code features found in APK files and used in two downstream clustering tasks using k-Means and DBSCAN. Additionally, since UI strings consist of natural language text, the word embeddings of the UI string model were evaluated with methods similar to the word embeddings of the description model used to answer the previous research problem. Since the UI string model only contained 4,029,542 words and a vocabulary of 19,391 unique words, in contrast to 60,074,045 words and a vocabulary of 146,196 unique words in the description model, the quality of the word embeddings was noticeably lower. Clustering embeddings from the UI string, layout, and Smali models produced inconclusive results. After manual review, the resulting clusters appeared not to contain applications that share semantic similarities. This again might be due to the small training size, but could very well be an inherent inability of the selected features to model semantic characteristics of their applications. However, the average similarity score of clusters from k-Means was higher than clusters fitted with DBSCAN, which might indicate that k-Means is more suitable to cluster document embeddings created with Doc2Vec.

Is it possible to identify malware or clone applications based on Smali source code similarity?

Malware detection was done by training a model containing class embeddings using Smali source code from 600 benign and 600 malware applications. Using this model and its similarity interface, it was possible to predict 65.52% correct guesses from 200 unseen samples. While far from impressive, spotting

clone applications with the validation model proved more successful, with a classification accuracy of over 90% for both seen and unseen samples. Furthermore, 247 high-similarity application pairs could be identified from a set of 600 application embeddings trained with Smali source code. Manual review showed that most pairs share semantic similarities.

Unsurprisingly, Doc2Vec worked best when used with a large natural text corpus like Google Play Store descriptions. With decreasing corpus size, the quality of document and word embeddings decreased as well, as seen with the model trained with UI strings, which only contained a tenth of the word information of the description model. While the model trained with layout code was unusable, results for models trained with Smali code were promising. With more refined pre-processing techniques and feature selection processes, the quality of the resulting embeddings could definitely be improved. Overall, Smali source code embeddings and description embeddings created with Doc2Vec could prove beneficial for Android application analysis, in particular when used alongside other application-specific features in downstream machine learning tasks.

Bibliography

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. *code2vec: Learning Distributed Representations of Code*. CoRR abs/1803.09473 (2018). http://arxiv.org/abs/1803.09473 (cited on pages 9–10).

[2] Andrew M. Dai, Christopher Olah, and Quoc V. Le. *Document Embedding with Paragraph Vectors*. CoRR abs/1507.07998 (2015). https://arxiv.org/abs/1507.07998 (cited on pages 7, 10, 58).

[3] Vasiliki Efstathiou and Diomidis Spinellis. *Semantic Source Code Models Using Identifier Embeddings*. CoRR abs/1904.06929 (2019). https://arxiv.org/abs/1904.06929 (cited on pages 10–11).

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. *A density-based algorithm for discovering clusters in large spatial databases with noise*. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. KDD'96. Portland, Oregon: AAAI Press, 1996, pages 226–231. https://dl.acm.org/citation.cfm?id=3001460.3001507 (cited on page 49).

[5] Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. *DroidKin: Lightweight Detection of Android Apps Similarity*. International Conference on Security and Privacy in Communication Networks. Edited by Jing Tian, Jiwu Jing, and Mudhakar Srivatsa. Cham: Springer International Publishing, 2015, pages 436–453. ISBN 9783319238296 (cited on pages 8, 10, 27, 31, 77).

[6] Richard Killam, Paul Cook, and Natalia Stakhanova. *Android malware classification through analysis of string literals*. Text Analytics for Cybersecurity and Online Safety (TA-COS) (2016) (cited on pages 8, 10).

[7] Quoc V. Le and Tomas Mikolov. *Distributed Representations of Sentences and Documents*. CoRR abs/1405.4053 (2014). https://arxiv.org/abs/1405.4053 (cited on pages 2, 7, 9, 19, 58).

[8] Ilia Markov, Helena Gómez-Adorno, Juan-Pablo Posadas-Durán, Grigori Sidorov, and Alexander Gelbukh. *Author Profiling with Doc2vec Neural Network-Based Document Embeddings*. Advances in Soft Computing. Edited by Obdulia Pichardo-Lagunas and Sabino Miranda-Jiménez. Cham: Springer International Publishing, 2017, pages 117–131. ISBN 9783319624280 (cited on pages 7, 10).

[9] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. *Distributed Representations of Words and Phrases and their Compositionality*. CoRR abs/1310.4546 (2013). https://arxiv.org/abs/1310.4546 (cited on pages 9, 13, 19).

[10] Annamalai Narayanan, Charlie Soh, Lihui Chen, Yang Liu, and Lipo Wang. *apk2vec: Semi-supervised multi-view representation learning for profiling Android applications*. CoRR abs/1809.05693 (2018). https://arxiv.org/abs/1809.05693 (cited on pages 8, 10).

[11] Nadia Rahmah and Imas Sukaesih Sitanggang. *Determination of Optimal Epsilon (Eps) Value on DBSCAN Algorithm to Clustering Data on Peatland Hotspots in Sumatra*. IOP Conference Series: Earth and Environmental Science 31 (Jun 2016), page 012012. doi:10.1088/1755-1315/31/1/012012 (cited on pages 49, 68, 78).

[12] Radim Řehůřek and Petr Sojka. *Software Framework for Topic Modelling with Large Corpora*. Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks. Valletta, Malta: ELRA, 22 May 2010, pages 45–50. https://is.muni.cz/publication/884893/en (cited on pages 2, 40).

[13] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, and Pablo Garcia Bringas. *On the automatic categorisation of android applications*. 2012 IEEE Consumer Communications and Networking Conference (CCNC). Jan 2012, pages 149–153. doi:10.1109/CCNC.2012.6181075 (cited on page 53).

[14] Google Security. *How We Keep Harmful Apps Out of Google Play and Keep Your Android Device Safe*. Android Open Source Project (2016). eprint: https://source.android.com/security/reports/Android_WhitePaper_Final_02092016.pdf (cited on pages 1–2).

[15] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. *Import2vec - Learning Embeddings for Software Libraries*. CoRR abs/1904.03990 (2019). https://arxiv.org/abs/1904.03990 (cited on pages 9–10).

[16] Lap Q. Trieu, Huy Q. Tran, and Minh-Triet Tran. *News Classification from Social Media Using Twitter-based Doc2Vec Model and Automatic Query Expansion*. Dec 2017, pages 460–467. doi:10.1145/3155133.3155206 (cited on pages 7, 10).

[17] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. *Deep Learning Similarities from Different Representations of Source Code*. Proceedings of the 15th International Conference on Mining Software Repositories. MSR '18. Gothenburg, Sweden: ACM, 2018, pages 542–553. ISBN 9781450357166. doi:10.1145/3196398.3196431 (cited on pages 9, 11).

[18] Laurens van der Maaten and Geoffrey Hinton. *Visualizing data using t-SNE*. Journal of machine learning research 9 (2008), pages 2579–2605. eprint: http://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf (cited on page 50).

[19] Nicolas Viennot, Edward Garcia, and Jason Nieh. *A Measurement Study of Google Play*. SIGMETRICS Perform. Eval. Rev. 42.1 (Jun 2014), pages 221–233. ISSN 0163-5999. doi:10.1145/2637364.2592003 (cited on pages 27, 29).

[20] Yeo-Chan Yoon, Junwoo Lee, So-Young Park, and Changki Lee. *Fine-Grained Mobile Application Clustering Model Using Retrofitted Document Embedding*. ETRI Journal 39.4 (2017), pages 443–454. doi:10.4218/etrij.17.0116.0936. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.4218/etrij.17.0116.0936 (cited on pages 8, 10).

[21] Yaijn Zhou and Xuxian Jiang. *Dissecting Android Malware: Characterization and Evolution*. 2012 IEEE Symposium on Security and Privacy. May 2012, pages 95–109. doi:10.1109/SP.2012.16 (cited on page 31).