Stefan Gruber, BSc

Code Between the Lines: A Machine Learning–Based App Publishing Assistance System

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisors

O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch
Dipl.-Ing. Johannes Feichtner
Dipl.-Ing. Dr.techn. Peter Teufl

Institute of Applied Information Processing and Communications

Graz, September 2019

Statutory Declaration

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 document uploaded to TUGRAZonline is identical to the present thesis.

Signiert von:	Stefan Gruber
Datum:	18.09.2019 16:57

www.handy-signatur.at

Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr. 910/2014 vom 23. Juli 2014 ("eIDas-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Dieses Dokument ist digital signiert!

Prüfinformation:
Informationen zur Prüfung der elektronischen Signatur finden Sie unter:
www.handy-signatur.at

TRUST
einfach sicher

Date

Signature

Abstract

As the central app market, Google Play is the primary source for Android users to obtain applications. Each app's presence in the store and in particular its description text, must answer crucial questions for the user: Does the app meet my needs? What features are included? What privacy-critical data does it share? A thoroughly composed description allows users to make a sound decision about whether to install an app.

However, it has been shown that app publishers do not always provide comprehensive descriptions. In many cases, the lack of enforced quality standards leads to inaccurate descriptions, both wittingly and unwittingly. Even with the best intentions, drafting a description text that answers the questions above is cumbersome.

Therefore, we propose an app publishing assistance system. It consists of three machine learning models that examine a given app for the sake of improving its description. The first model finds related apps regarding the description, and the permission set it requests. For this purpose, we employ autoencoder neural networks and t-SNE dimensionality reduction. The evaluation demonstrates that we can accurately identify similarities and dissimilarities, information which altogether serves as a starting point for composing app descriptions and initial app analysis. The second model infers the app's main features based on its application package file. By extracting resource identifiers, string constants, and methods from the application package, we use a dense neural network to predict words and phrases that are likely to occur in its description. The model predictions are precise and generalized phrases that can easily be used to verify and enhance the actual description. Moreover, the third model assesses how well permissions are reflected in the description. The text is analyzed with a convolutional neural network that was trained to predict nine of the dangerous permission groups. Our evaluation shows that the model very well captures permission-related words and that their absence can encourage publishers to add them. Ultimately, our proposed app publishing assistance system could help to improve description quality in the thriving Android app ecosystem significantly.

Kurzfassung

Als zentraler Marktplatz ist Google Play die Hauptquelle für Android-Nutzer, um Apps zu erhalten. Die Präsenz jeder App dort, insbesondere der Beschreibungstext, muss dem Benutzer essentielle Fragen beantworten: Erfüllt die App seinen Zweck? Welche Funktionen enthält sie? Welche Daten, die meine Privatsphäre betreffen, werden geteilt? Eine sorgfältig zusammengestellte Beschreibung ermöglicht es Benutzern, eine fundierte Entscheidung zu treffen, ob sie eine App installiert sollen.

Es hat sich jedoch gezeigt, dass die Entwickler und Veröffentlicher von Apps nicht nur vollständige Beschreibungen einreichen. In vielen Fällen führt der Mangel an erzwungenen Qualitätsstandards zu unzureichenden Beschreibungen, sowohl bewusst als auch unbewusst. Selbst mit den besten Absichten ist es mühselig, einen Beschreibungstext zu verfassen, der die obigen Fragen beantwortet.

Daher schlagen wir ein App-Publishing-Assistenzsystem vor. Es besteht aus drei Machine Learning Modellen, die eine bestimmte App untersuchen, um schlussendlich ihre Beschreibung zu verbessern. Das erste Modell findet verwandte Apps in Bezug auf die Beschreibung und die benötigten Berechtigungen. Dazu verwenden wir Autoencoder und die t-SNE zur Dimensionsreduktion. Die Auswertung zeigt, dass wir Ähnlichkeiten und Unterschiede identifizieren können. Gefundene Informationen können insgesamt als Ausgangspunkt für die Erstellung von App-Beschreibungen und für erste App-Analysen dienen. Das zweite Modell schließt auf die Hauptfunktionen der App basierend auf der Installations-Datei. Durch Extrahieren von Resource-Identifiern, String-Konstanten und Methoden versucht ein neuronales Netzwerk, Wörter und Ausdrücke vorherzusagen, die wahrscheinlich in der Beschreibung vorkommen. Die Modellvorhersagen sind präzise und verallgemeinerte Phrasen, mit denen die eigentliche Beschreibung leicht überprüft und erweitert werden kann. Darüber hinaus bewertet das dritte Modell, wie gut Berechtigungen in einer Beschreibung widergespiegelt werden. Der Text wird mit einem Convolutional Neural Network analysiert, das trainiert wurde, um neun der kritischen Berechtigungsgruppen vorherzusagen. Unsere Auswertung zeigt, dass das Modell berechtigungsbezogene Wörter sehr gut erfasst und dass ihr Fehlen die Verfasser der Texte ermutigen kann, sie hinzuzufügen. Letztendlich könnte unser vorgeschlagenes Assistenzsystem für App-Veröffentlichungen dazu beitragen, die Beschreibungsqualität im florierenden App-Ökosystem signifikant zu verbessern.

Contents

Acknowledgments

Throughout the writing of this thesis, I have received a great deal of support from many sides.

First and foremost, I would like to acknowledge my supervisors for their collaboration. Continuous support that is both challenging and motivating is not to be taken for granted. Their ideas and perspectives shaped the outcome of this work. I want to especially thank Johannes Feichtner and Peter Teufl for their excitement on the overall topic of deep learning and for sparking my interest to eventually confront this challenge, which has endured until its completion. I am appreciative of the countless working hours that they have invested for discussing my results, contributing new approaches, and revising this written document.

Furthermore, the support of my family and friends made finishing this thesis significantly easier. Thank you for understanding the impacts a master thesis has on someone's schedule, priorities, and mood.

Stefan Gruber
Graz, Austria, September 2019

Chapter 1

Introduction

Their third-party app ecosystem has been a catalyst for the success of smartphones in the last decade. Benefiting from its platform-openness, Android has become the most popular smartphone operating system among developers in terms of available apps. The primary way of installing third-party applications on mobile devices are app markets like Google Play. As of August 2019, the store contains over 2.4 Million published apps. [1] There are additional possibilities to obtain apps, like manually downloading installable packages from the web. Most Android apps, however, is acquired through this software distribution hub.

Google Play provides convenience features that allow users to decide whether an app is geared to their needs. Almost every store entry contains meta information like a category, a one-sentence summary, multiple screenshots, a descriptive text, and a list of user permissions the app requests. This information helps the user to decide whether (a) the app's features are of service to them, and (b) the app is trustworthy. Although there may be users that skip reading the description, the information available should always cover these two aspects.

To some extent, store providers like Apple and Google attempt to assess app trustworthiness and the contents of the description. They have implemented mechanisms to scan for abuse and malware. Different approaches exist: While human code reviews are the norm for all apps submitted to Apple, Google mostly uses automated analysis. In recent years, Google's pre-publish security tests have been found to be circumventable without much effort.[2] Adversaries also use techniques like cloning existing apps, adding malicious behavior, and re-submitting them to the store. These apps are then presented just like the original, legitimate version, but with a deceiving store entry. Detecting repackaged apps in mobile app stores, but also malware on mobile platforms in general has become a growing research area in recent years [7, 13, 19].

The identification of malware before its distribution is, however, not the only concern for apps in Google Play. For this thesis and its three main contributions, we formulate the underlying problem much more general. As we pointed out, the information about the app stated in the store entry, in particular, the description, is crucial for deciding whether someone should download the app or not. In other words, the app developer is obligated to formulate the description so that it briefly, but thoroughly, covers the core functionality without omitting important features and permissions. In case there is a gap between the description and the actual functionality, the user should be alarmed. Reasons may reach from inattentive, careless app releases to intentional malware distribution. Therefore, we deem the examination of the alignment between description, permissions, and functionality momentous.

[1]https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[2]https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/

A thorough assessment of app store descriptions would have great value but is expensive. From the store provider's point of view, comprehending what an app does is a complicated endeavor and time-consuming. Every app would have to be dissected and thoroughly analyzed to understand its behavior. From the developer's point of view, even with best intentions, selecting the right meta information that the users are about to see is also challenging. It requires the ability to abstract one's own detailed view of the implementation and to succinctly depict the developed piece of software from the end user's perspective. On the one side, manually drafted app descriptions are an essential tool for app marketing, making the app appealing for users and allowing a distinction. On the other side, accurate depiction for the sake of comparability can be measured and eventually improve description quality. Practically, a good app description thus contains objectively measurable parts as well as creative wording to describe its behavior.

Hence, we propose an app publishing assistance system (APAS) that can guide both app developers and store providers. The overarching objective of such a system is to enhance the app's store presence. Based on this objective, we have identified three questions that should be asked before publishing an app:

1. How does the app relate to other existing apps?
2. What core phrases should the description include, based on the app's functionality?
3. How well does the description reflect permission-related functionality?

By developing a system that helps to answer these three questions for an app that is about to be published, we aim to improve its description quality before publication.

We intend to build a system that gains knowledge from real-world apps. Statically defining rules for classifying source code is an option. The vast number of API calls, third-party library methods, and their combinations, however, makes it cumbersome to find and set up these rules by hand. Another factor is the evolving nature of smartphone apps, with continually changing APIs, design patterns, and use cases. With the advances in machine learning in recent years and especially in the field of neural networks, we attempt to make use of established neural network designs that capture and classify relationships based on a large number of apps. Analyzing already published applications and learning behavior from them, however, leads to large amounts of data to be processed and any problems associated with that.

Neural networks, in general, learn patterns from a large set of examples that they are given during training. Conventionally, before training machine learning models, humans evaluate each training sample regarding its quality and properties. For classification tasks, i.e., predicting the class assignments for samples, someone goes through the training data beforehand to validate the assigned target classes. The manual examination aims to reduce the influence of incorrectly labeled samples and improves the prediction quality. In domains where a machine learning model is used to make consequential, non-monitored decisions, meticulously reviewing each sample is essential.

In this thesis, however, we do not work with manually labeled and hand-picked samples for practical reasons. For our tasks on smartphone apps, manually identifying the complex relations between training input and training output is a demanding and time-consuming task for humans. In contrast, for an image classification task, revision by humans would take comparably less time per sample. Reviewing an Android application and deriving a "normalized" description requires considerably more time. Practically, to train on a broad spectrum of smartphone app types, we would have to correctly label thousands of apps regarding their descriptions, their source code, and static resources, as well as their permission sets.

However, since our goal is an assistance system that merely makes suggestions, we make use of a real-world app set that has not been manually reviewed. As this makes the learning process more prone to errors, i.e., learning irrelevant or missing correlations, we try to adjust our system to that as good as possible. On the one hand, we train our machine learning models with coarsely filtered samples. We check them for properties like minimum description lengths, description language, or whether the app implementation is non-native, like HTML5 cross–platform apps. On the other hand, we design the system so that it outputs not only predictions but also their reasoning. Through the use of model explanation algorithms, each prediction, e.g., description fragments based on the application package, is always presented with a list of the input features that have led to it. Hence, although we train our model on noisy, potential low-quality samples, we coarsely filter them during preprocessing and provide a transparent explanation for each prediction. As our evaluations show, despite this trade-off, we still achieve good results.

As the first step for pre-publishing assistance, we deem description-related and permission-related comparison with other apps essential. Finding apps that are related to each other can serve multiple purposes. From a security perspective, apps can be clustered by their malignancy, i.e., if an app is malicious or which malware group it belongs to [32, 33, 54]. Clustering algorithms have also already been successfully applied to smartphone app descriptions [2, 37, 58]. The main goal is usually to categorize apps according to their usage, like finding a suitable category for Google Play or Apple's App Store. For our system, we intend to use clustering and dimensionality-reduction algorithms without always requiring strict boundaries. Our goal is to visualize apps closer to each other or further apart. By measuring distances in a low-dimensional space, we aim to find related apps to the one being examined. As pointed out before, even though store descriptions contain app-specific phrasing, particular parts should make the description relatable as well. A clustering system for descriptions should thus be able to position a particular app in the proximity of other apps that serve a similar purpose in case the description is sufficient. Analogical to the description similarity, the permission sets of related apps should theoretically also resemble each other. In case the permission sets are very dissimilar for related apps, developers and users may question them.

The first one of our three contributions is, therefore, an app clustering and distance measurement approach using undercomplete autoencoder neural networks and t-SNE. Two separate autoencoders are trained on the descriptions and the fine-grained permission sets. Their encoders produce latent space representations that are used for further reducing the dimensionality with t-SNE. After these non-linear transformations, we obtain 2-d representation models that we can feed the app that should be examined. For descriptions, the approach shows that with t-SNE we can find similar apps better than with PCA. The permission dissimilarity analysis shows that based on permissions only, we can find outliers that are not only or not at all doing what the description claims. Reducing app complexity and representing them in a 2-d space for descriptions and permissions can facilitate initial assessments as we are showing in our evaluation.

Upon finding similar and dissimilar apps, the APAS should suggest potential elements the description text might include. These suggestions could be used to enhance the first draft of a description. Such a system thus has to analyze the app for clues that allow for inferring descriptive phrases. Deriving explanations from source code is a challenging field that has made progress in recent year with the use of deep learning. Neural architectures have been used to summarize code, suggest method names, or label software projects [3, 4, 16, 40]. Specific to Android, user interface descriptors have been analyzed to obtain words for the description [28]. These methodologies, however, only cover either source code or a subset of the static resources, are very domain-specific (non-mobile), or computationally expensive preprocessing and training. The description inference part of the APAS should thus make use of both the app's source code and its resources, while still maintaining low model complexity.

The description inference model we propose thus receives static parts of the app as well as its source code. In particular, given an individual application package, including source code and resource files, the system should come up with an overall picture of what the app does. Therefore, we train three different neural networks. The networks are trained to output description fragments. The first network receives all resource identifiers an app holds in its XML files. The second one gets string constants found in the app's source code and its XML files. The third model gets all method calls an app makes to the Android API. All inputs and outputs, i.e., descriptions, identifiers, strings, and methods, are represented via term frequency–inverse document frequency (TF-IDF), a commonly used text representation algorithm. After training, we use the SHAP framework to provide explanations for the predictions, e.g., in case a predicted description phrase is *photo frame* it might be inferred from static resources in the app that contain tokens like *photo*, *crop*, *frame*, or similar. Model explanations make the predictions more reliable since we use non-manually labeled samples. As we show later, the three models of our description inference system often output very concise, generalizing illustrations of what an app does. To summarize, the description inference part of the APAS uses an occurrence-based representation of the app's resource identifiers, string constants, and API method calls to predict phrases of the description. Additionally, it employs a machine learning model explanation algorithm to show how these predictions come about.

After automatically computing app similarity and description inference, the third contribution of this thesis focuses on textual permission clarity. User permissions are one of Android's cornerstones to protect the user's privacy. Whenever a third-party app requires access to privacy-critical features, e.g., the camera, the microphone, or the address book, the user has to white-list the corresponding permission group. Consequently, a well-drafted description text in Google Play should reflect why an app needs these permissions and what it does with the data it collects. Based on a description text, the required permissions should be identifiable. In recent years, this has become an active research area [15, 45, 49]. These systems, however, have mainly used comparably small, manually labeled sample sets, match the permission-related parts with specific strings in the source code, or used conventional machine learning algorithms like support vector machines. These approaches are infeasible for our noisy, non-handcrafted datasets.

We thus propose a convolutional neural network (CNN) for text classification in combination with a model explanation algorithm. During preprocessing, we tokenize the raw text and use a pre-trained embedding model before feeding the texts into the neural network. The CNN's different filters capture words and word groups. Their combination is used to predict the likelihood that an app requires a certain permission. Then, we also apply the LIME model explanation algorithm in the prediction phase to find the words in the text that cause the prediction score for each permission. Our evaluation shows that the textual permission clarity model can be applied to identify shortcomings of permission usage reflection in the description. The model explanation is used to visualize the impacts of particular words. We see that although we use a noisy dataset, the trained model can also give valuable about the absence of permission-usage related wording. Overall, the textual permission clarity system as the third part of the APAS can help with identifying and improving the expression of an app's permissions in the description text.

The following chapters describe our contributions in detail and are structured as follows. Chapter 2 establishes the necessary background knowledge. The background includes details about Android's app structure, permissions, static code slicing, natural language processing methodologies, and selected machine learning fundamentals. In Chapter 3, we show existing, related work similar to our approaches. After that, we continue with the three main contributions of the thesis. First, we cluster apps with the use of autoencoders and t-SNE in Chapter 4 to show their similarity and dissimilarity in regard to descriptions and permissions. Second, Chapter 5 deals with the description inference system where we use a dense neural network and TF-IDF to predict description fragments for an app based on its resource identifiers, string constants, and method calls. Third, in Chapter 6, we describe our textual permission clarity model in detail, which uses a CNN to predict the use of permission groups based on the description text. Each main chapter includes an evaluation where we show the application of each approach. Finally, we complete this thesis with a conclusion in Chapter 7.

Chapter 2

Background

In this chapter, we provide background knowledge about Android applications, static code analysis, and machine learning. For Android apps, an overview of the application package structure, build process, runtime, and code obfuscation is given, as well as details about the runtime permission system. Then, we turn to the fundamentals of static slicing which we need later to analyze applications. The section about natural language processing aims to give insights on how human language can be represented and processed computationally. Ultimately, the machine learning part focuses on relevant types of neural networks and methods to evaluate their performance.

2.1 Android

Android has become the world's most popular operating system for smartphones. Bought by Google in 2005, the platform has evolved over time. Running on smartphones only at first, the ecosystem now covers tablets, wearables, cars, smart home equipment, and more. The Linux-kernel based OS can, by design, be adopted by hardware manufacturers while still providing a public market platform for third-party applications. As of June 2019, 2.5 billion devices running the operating system are active.[1] Over the years, Android has become the most widely spread smartphone OS.

The third-party application ecosystem has played a significant role in that development. Allowing developers to cover all sorts of custom use cases has drastically changed the way we use hand-held internet devices. Android was designed to be an open platform for developers with a considerable amount of documentation. The following pages cover a broad range of topics necessary to comprehend packaged and distributed Android apps.

2.1.1 Application Packages

Installable third-party Android apps are distributed using application packages (APK). This platform-specific archive format encapsulates runnable code plus necessary static resources, e.g., images, language translations, or user interface descriptors. The APK file's format is a Java archive (JAR) with ZIP file compression. The archive itself needs to contain particular descriptors like an app manifest and an appropriate sub-folder structure.

The Manifest file defines relevant parts of the app and serves as an entry point. [2] Listing 2.1 gives an idea of the basic structure. The main `manifest` tag defines the app package name which must be

[1]https://9to5google.com/2019/05/07/android-pie-distribution-numbers/

[2]https://developer.android.com/guide/topics/manifest/manifest-intro

unique on the system, which we refer to as app package name. Line 6 with `uses-sdk` puts a limit on which minimum OS versions the app can be installed on, which is essential, i.e., for compatibility reasons and availability of newer SDK functionality. User permissions requested by the app must be listed, as exemplified in line 8. A more detailed look on the permission system is given later in Subsection 2.1.7.

The functional core is defined within the `application` tag. In general, four different component types are available and can be implemented multiple times:

- **Activities** are user-interface related parts of the app.
- **Services** contain logic running independently from the user-interface and potentially after closing or without an activity.
- **Broadcast receivers** are listeners for messages from other apps or the operating system.
- **Content providers** are abstracted, well-defined interfaces that can be used by other applications to access the app's data.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3              android:versionCode="5" android:versionName="1.1"
4              package="at.tugraz.example.myapp">
5
6      <uses-sdk android:minSdkVersion="19" android:targetSdkVersion="26" />
7
8      <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
9
10     <application android:allowBackup="true"
11                     android:icon="@mipmap/ic_launcher"
12                     android:theme="@style/AppTheme">
13
14       <activity android:name=".MainActivity">
15         <intent-filter>
16           <action android:name="android.intent.action.MAIN" />
17           <category android:name="android.intent.category.LAUNCHER" />
18         </intent-filter>
19       </activity>
20
21       <service android:name=".PushMessageService"
22            android:label="@string/push_service_name" />
23     </application>
24 </manifest>
```

Listing 2.1: Android app manifest file: example structure.

Table 2.1 lists the mandatory files and folders inside the APK archive. Upon development, the Java/Kotlin source code is compiled to Dalvik executable bytecode and stored inside the `classes.dex` file. The DEX files are then loaded into Android's virtual machine and executed on the device.

`META-INF/`	list of archive contents and data integrity hashes
`assets/`	file-system like folder for resources
`lib/`	non-Java, native library code
`res/`	pre-loaded resources like strings, icons
`AndroidManifest.xml`	Application manifest file
`classes.dex`	Dalvik VM source code of the whole application
`resources.arsc`	pre-compiled version of the app's resources

Table 2.1: APK file structure.

The `assets/` and `res/` folders both contain non-executable resources. Assets are files that are accessible by their filename. In contrast, `res/` can store multiple variants of data which are referenced in the code via a resource identifier. The information located inside the `res/` folder is what we subsequently refer to as resources.

2.1.2 Resources

Android development offers a refined system for storing and accessing resources program logic uses.[3] Items such as activity layout definitions, icons, strings, animations, and others are stored under `res/`. The developer is encouraged to externalize elements like the language-specific text used in the UI. For graphical elements, device-specific content can be provided (resolution, orientation, etc.). This view separation via resource elements facilitates code maintenance and allows for covering a broad range of device types and users with one installable package.

In other words, this means that one resource can bundle several specific items serving the same purpose. Listing 2.2 shows a simple example of string localization. The developer creates language-specific folders under `/res`, e.g., for English, German, and a fallback case. Within these folders, XML files with the same file name are provided. Specific items, like strings in the example case, can be referenced via the `name` attribute. The operating system handles the selection of the exact, underlying item.

(i) XML file for English.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="btn_send">Send</string>
  <string name="txt_enter_name">Enter last
    name:</string>
  <string name="msg_sent">Request has been
    sent!</string>
</resources>
```

(ii) XML file for German.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="btn_send">Absenden</string>
  <string name="txt_enter_name">Nachnamen
    eingeben:</string>
  <string name="msg_sent">Anfrage
    abgeschickt!</string>
</resources>
```

(iii) Accessing multilingual resource strings programmatically.

```java
private void setButtonText() {
  Button okBtn = (Button) findViewById(R.id.btn1);
  okBtn.setText(getResources().getString(R.string.btn_send));
}
```

Listing 2.2: Example usage of multi-lingual strings in the user interface.

Internally, name attributes in XML files are translated to numeric symbols. Each name attribute corresponds to an integer ID (resource ID) which is randomly assigned by the compiler. Within the program logic, the `R.*` constants are integers holding these values. The build system creates a table stored in the `resources.arsc` file, linking each integer value with multiple versions of the resource.

2.1.3 Entry Points

In regard to activities, services, broadcasts, and providers, the app's manifest provides a list of entry points for the app. These Java classes contain methods the operating system looks for when it launches an app or sub-parts of it. Examples are starting and stopping app activities and services. Resolving these gateways is particularly useful for app analysis.

[3] `https://developer.android.com/guide/topics/resources/providing-resources`

The associated XML tags in the manifest contain `android:name` attributes which point to Java classes. These classes can either be referenced using the fully-qualified class name, i.e., full package name plus class name or in an abbreviated way: The leading dot in line 14 in Listing 2.1 stands for the whole app's package name as defined in the top-most `manifest` tag. This list of Java classes can then be found in the DEX file.

All entry classes are derived from SDK components, meaning they have to implement or inherit a specific class or interface.[4] For activities and services, important methods are listed in Table 2.2. After instantiating the object, i.e., calling the constructor, the system calls these lifecycle methods, similar to the `main()` function in conventional JVM Java programs. As an example, when an Activity is launched, the OS calls `onCreate`, `onStart`, and then `onResume`. Within these methods, the app is free to create other objects, threads, callbacks, and to access their methods. By resolving these lifecycle entry methods, the relevant program code can be analyzed systematically.

Activity	**Service**
`onCreate()`	`onStartCommand()`
`onStart()`	`onBind()`
`onResume()`	`onCreate()`
`onPause()`	`onUnbind()`
`onStop()`	`onRebind()`
`onDestroy()`	`onDestroy()`

Table 2.2: Important Android lifecycle methods.

2.1.4 Build Process

The Android build process defines several steps on how to transform source code to installable packages.[5] A set of compilation operations, file alignments, and data integrity measures assembles the APK file. Understanding this build process is necessary to reverse engineer applications and analyze their behavior.

Figure 2.1 depicts the steps necessary for building an APK. The compile step merges app-specific Java/Kotlin source code with third-party libraries. These libraries can contain, e.g., UI handlers, custom cryptography, HTTP request handling, etc.Libraries are always included as a whole, with potentially large amounts of code that is not used by the app. Apart from all the source code, also device and language-dependent images and XML resources are compiled into the aforementioned `resources.arsc` table. The compilers output one or more .dex files, containing program code, and an .arsc file, containing the organized resources.

Upon compilation, the APK packager adds the installable archive file. The packager includes asset files and native libraries to the APK. For data integrity measures, the user has to sign the data using an asymmetric key-pair that is shared with the app market, i.e., Google Play Store. The APK file is then ready to be distributed or to be installed directly on the device.

2.1.5 Dalvik Bytecode

Dalvik Executables (DEX) are the files containing the program logic loaded and run by the Android OS. Code execution is similar to Java. A virtual machine (VM) translates bytecode statements to CPU operations. Before Android 5.0, Dalvik VM was used and then replaced by the Android Runtime (ART).[6] An Android-specific compiler to translate Java code to bytecode executable on hand-held devices is necessary.

The d8 compiler is part of the Android Build Tools. It takes a set of Java `.class` files as input, optimizes them, converts them to bytecode, and compresses them into the DEX file. For apps exceeding the number of 64k methods, two or more DEX files are generated. These DEX files are stored in the APK file.

[4] `https://developer.android.com/guide/components/activities/activity-lifecycle`

[5] `https://developer.android.com/studio/build`

[6] `https://source.android.com/devices/tech/dalvik/index.html`

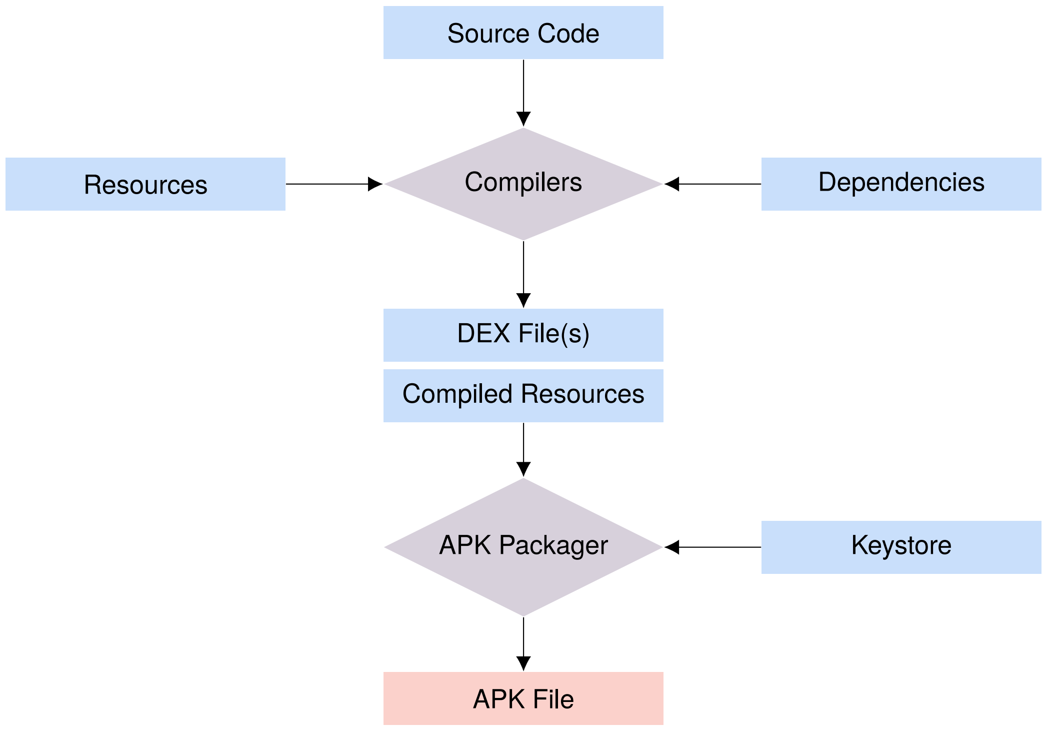


Figure 2.1: Android build process.

The resulting bytecode for ART is different from the Java VM's bytecode.[7] For Java, local variables are put on the stack, but the Dalvik VM assigns these variables to hardware registers. Hence, DEX opcodes perform their operations directly on registers. The set of opcodes also differs from the Java VM, as well as handling the pool of constants. For object references and reference comparison, Dalvik uses a numeric zero as a null pointer and does not have separate OP codes for that. Amongst others, these adaptations allow for faster code execution on embedded devices but hamper recovering the original Java code.

Reverse engineering DEX files to Java code is thus an effortful endeavor. Tools like dex2jar[8] attempt to undo one step in the build process, turning DEX files back into its Java .class files. Peculiarities of the VM and the DEX format often cause issues since build transformations are not unambiguously invertible.

Considering these challenges, a decision has to be made whether obtaining Java code is necessary for the app analysis. As it has been shown, work can be done on DEX code level [17, 30, 31]. Working directly with DEX files also bypasses tools like HoseDex2Jar, which try to prevent the decompilation of Android apps. Depending on the task, reversing the APK's logic back to Java code might thus not be needed and working with DEX code can be feasible.

2.1.6 Code Obfuscation

Source code obfuscation describes a transformation of program code without changing its behavior. During the transformation process, semantic information that was needed for development is stripped away that does not affect the execution. Obfuscation is an optimization process that also aims to protect intellectual property.

[7]`https://forensics.spreitzenbarth.de/2012/08/27/comparison-of-dalvik-and-java-bytecode/`

[8]`https://github.com/pxb1988/dex2jar`

Tools provide obfuscation on different levels [36]. Layout transformations alter identifiers such as variable names, class names, and methods names. Since Java is an interpreted language that offers reflection features[9], identifiers are included inside the compilation. Data obfuscation adapts the access, storage, and aggregation of values, e.g., inside arrays. Control obfuscation rearranges statements, duplicates them, or rewrites them in other ways. On all the different levels, all obfuscation measures must not change the program's behavior.

Code obfuscation serves two purposes. Replacing identifiers with shorter ones leads to lower memory consumption. Mainly, however, obfuscation tools are used to make it harder to reverse-engineer the original source code. Semantic information like variable names facilitate comprehending code fragments. The removal of such semantics is supposed to impede theft of intellectual property, as well as program analysis.

On Android, the most popular code obfuscation tool is ProGuard.[10] Google recommends the use of ProGuard or its own, newer compiler R8 that also provides similar functionality, to all developers. While ProGuard only affects Java code, the R8 compiler also checks for unused references to XML resources. It does not, however, obfuscate the resource identifiers, e.g., for language strings or activities. Both ProGuard and R8 only alter Java code on Android.

Obfuscation hampers app analysis on a source code level. The absence of development semantics adds additional challenges to understanding the app's behavior. Several code properties are not modified. Apart from the aforementioned resource identifiers, direct API calls to the OS have to be hard-coded at some point so they can be executed. Since the transformation process can never alter the desired app behavior, the control and data flow have to remain unchanged in the end. Despite obfuscation, structural code analysis methods can still extract valuable information.

2.1.7 User Permission System

The Android platform security model in regard to user applications is based on a three-party consent [41]: App developer, device user, and the OS have to mutually agree for an action to be executed. Three-party consent especially affects privacy-related data. In contrast to traditional operating systems, apps are security principals. An Android app has significantly fewer privileges over the whole OS, like file-system encrypting malware or cross-app data leakage on desktop systems. These design aspects improve the protection of private data when a user installs third-party apps.

In practice, apps work in isolated spaces and with limited authority. A sandboxing system prohibits rigorous access between third-party apps and isolates their execution and their private data folders. Additionally, a long list of mandatory access control features is implemented via the Security-Enhanced Linux (SELinux) kernel extension, e.g., direct hardware access or file operation restrictions. Sandboxing and SELinux are measures of the operating system to restrict critical app-related operations on an operating-system-level fully. They are enforced regardless of user settings.

In addition to these measures, Android comes with a user permission system.[11] These user permissions affect all kinds of information, from communication (internet access, NFC, SMS, ...) over phone data (call logs, wallpaper management, file access, ...) to direct hardware access (microphone, camera, location, ...). Each permission allows, if granted, the usage of one or several API calls. The list of permissions has to be declared in the `AndroidManifest.xml` file. In case a developer does not state the necessary permission upfront, the app is not able to access certain features.

[9] https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html

[10] https://www.guardsquare.com/en/products/proguard

[11] https://developer.android.com/guide/topics/permissions/overview

The permission system, however, has seen changes across OS releases. Up to Android 5.1.1 (Lollipop), the list of permissions was presented to the user at install time. The user could either accept all permissions and proceed with the installation process or decide not to install the app at all. This potentially long and overwhelming list of critical and trivial permissions caused many users to skip assessing the requested permissions and install apps regardless of the dialogue's contents.

A major change was made in Android 6.0 (Marshmallow) when Google switched to runtime permissions. For third-party apps, Android then started differentiating between normal permissions and dangerous permissions. Normal permissions cover features that are considered non-privacy sensitive, e.g., internet access, time-zone management, or alarms. Dangerous permissions are related to camera, location, or shared file storage. While normal permissions are granted automatically at install time, for dangerous ones, the user is presented a confirmation dialog when the app first requests the feature. Smartphone users, in general, see this shift positively, adapted quickly, and prefer the new system [6]. This change to the permission system aimed to make the user more aware of the permissions an app uses, but also give more access control.

Granted Permission Group	Entailed Permissions
`CALENDAR`	read and write calendar
`CALL_LOG`	read and write call log process outgoing calls
`CAMERA`	camera access
`CONTACTS`	read and write contacts get system accounts
`LOCATION`	access both coarse and fine location
`MICROPHONE`	audio recording
`PHONE`	read phone state and numbers make and answer calls; add voice mail SIP (voice-over-IP)
`SENSORS`	access to body sensors
`SMS`	send, receive and read SMS receive WAP push notifications and MMS messages
`STORAGE`	read and write external storage

Table 2.3: Permission groups in Android 6+.

In practice, the Marshmallow permission system change has significant security implications. Developers are forced to state what the requested permission group is used for. In a large-scale empirical study, however, it was found that less than a quarter of apps contained such rationales [35]. Especially the PHONE and CONTACTS group included a plethora of incorrect rationales. The exact usage description is either often omitted, the quality tends to be low containing only trivial details and is given a positive spin in terms of privacy.

Another downside is that previously single permissions are now clustered into permission groups for usability reasons. Table 2.3 shows the ten permission groups and the entailing fine-grained permissions. It is opaque to the user which permissions within the group are requested and used. For instance, if an app only needs to create a calendar entry, the user would not notice that it can also read all existing calendar entries. Another example would be an app that merely needs to connect to a nearby device, e.g., a smartwatch; it can also easily pinpoint the user down to several meters using GPS. Since the internet permission is considered non-dangerous and granted to any app, data could be sent to an adversary in the background. Grouping multiple permissions and creates new attack vectors for Android apps.

While the newer permission system gives more control to the user at runtime, grouping fine-grained permissions can mislead the user to share more private data than obvious. High-quality app descriptions in the app market can give guidance on how the app uses permissions and can potentially compensate for lack of runtime permission rationales.

2.2 Code Slicing

Slicing is a technique to analyze how programs behave during execution [63]. While natural language can be processed linearly, programming languages fundamentally rely on conditions and jumps. A program usually consists of a large set of instructions that are not executed linearly. Depending on the input, e.g., from the user, different paths are taken. Program slicing is a method that models the specific behavior of a program execution path.

In general, slicing differentiates between control flow and data flow. The control flow describes the relationship between statements. It is useful to obtain a sequence of operations to describe an algorithm. The data flow maps the change and transition of variables (data) throughout the program.

Slicing decompositions of programs can be created using mathematical graph structures. A flow graph is a directed graph. The nodes represent program statements. For control flow graphs, edges are modeled for each transition between such statements, i.e., conditions (multiple outgoing edges), loops (multiple incoming edges), or simple statements like assignments (one in, one out). Treating programs as graph structures facilitate finding the relationship between program statements.

Call graphs are a sub-category of control flow graphs specific to the execution of procedural functions [53]. Conditions and jumps are omitted when creating the graph, only called sub-routines are followed through, with each call being added as a graph node. In its most straightforward way, the nodes can be collected statically when analyzing source code. In the end, call graphs yield an abstract, compact list of relationships between sub-procedures of a program.

In practice, call graphs are used as a compressed visualization for humans to comprehend what the program does. Nowadays, programs, and especially Android applications, contain a large code base, including frameworks and libraries, embedded into operating systems with specific API calls. Developers add third-party libraries to their projects. Classes and methods might only be relevant for debugging or might be artifacts from older releases. Hence, call graphs can tell which parts of an Android application are really used.

2.2.1 Implicit Control Flows

The sequence of triggered sub-procedures cannot always be found via the project's source code. In high-level programming languages and frameworks, a variety of calls is made implicitly [8]. Java offers constructs like Reflection, which allows modifying the program code during runtime. Android provides so-called Intents, used for communication between components. Another example in Android are event listeners, where the registration statement and the code to be executed are triggered internally by the operating system. For someone reading the code, the source and the target of the flow may be apparent. However, it cannot be captured by the previously outlined flow graph creation algorithm.

EdgeMiner [12] aims to find such implicit control flows. By analyzing the Android framework's codebase, it looks for potential callbacks. Performing a backward-analysis of the framework's flow graph, it generates pairs of method calls, consisting of a registration method and one or several callback methods. The list that EdgeMiner generates enhances a flow graph with additional edges.

As with any object-oriented language, object types in Java can be decided at runtime. In case a variable is defined using an abstract class or interface, the particular method implementation to be called is versatile. Inheritance complicates the creation of call graphs since type-related conditions are not incorporated. It is thus necessary to decide how to treat inheritance in the call graph by potentially resolving all possible implementations of a parent class.

As for the overall goal of this thesis, call graphs can be used to detect particular Java and Android API calls. Adding information about implicit flows to the graph enables resolving graph edges that are not defined by the app but hidden inside the Android/Java framework. Call graphs can thus be used to separate library code in the app from code needed for its execution and trace the methods that are actually called.

2.3 Natural Language Processing

In recent years, there has been a shift in how users interact with their devices. It was common that computing devices accepted only a narrowly defined set of inputs, e.g., commands following a strict syntax or user interfaces with dedicated action elements. Advances in the field of natural language processing (NLP) allow programs to interpret human language better. Potential applications are information retrieval (IR), text classification, or sequence-to-sequence translation. Despite speech recognition playing an important role in NLP as well, this section solely focuses on processing text, which is needed later in this thesis.

NLP is a challenging task, spanning over the fields of computer science, mathematics, and linguistics. Languages are inherently different from each other, both semantically and syntactically. Words come in grammatical variations and can have similar or multiple meanings. Depending on the context, the contribution of a single word for the overall meaning varies. The dictionary limits processing capability. For handling human text, a program must take all these properties into account [61].

2.3.1 Text Preprocessing

To be able to digest the text further, certain procedures have become established. In general, the text is stored as a sequence of characters. As a first step, this character array has to be split into meaningful tokens. Some of these tokens are eliminated, e.g., because of their occurrence count, while others are kept and morphologically adjusted. Text preprocessing represents written human language so that machine learning models can handle it efficiently.

Tokenization refers to splitting the whole text into smaller, processable entities. Semantically correct tokenization is a complicated task. The notion of a "token" needs to be defined beforehand [62]. In practice, as a first step, splitting is done through whitespaces (spaces, newlines, tabs, etc.). Special characters like dots, colons, semi-colons, apostrophes are often also removed. Depending on the use case, it makes sense to retain special characters, e.g., for third-person singular or hyphenated word combinations.

Stopwords are words that carry comparably little meaning in a text. Stopwords are usually defined in a list that contain low-content words, high-frequency words, or function words, i.e. articles ("a", "the"), forms of "to be", or conjunctions ("and", "or", "but"). In theory, stopword influence should be low if the subsequent machine learning model is able to balance out tokens occurring significantly more than others. In practice, the purpose of stopword removal is to keep the data to be processed low, e.g., memory and time complexity for training a neural network.

Stemming is a transformation where different morphological word forms are reduced to a single word, the root or stem [22]. Most stemming algorithms remove affixes (prefixes and suffixes) of words according to a list of pre-defined rules. Words shorter than a certain length are not truncated. The stem does not necessarily exist in a dictionary.

Before applying stemming, the quality-speed trade-off has to be considered. Table 2.4 gives some examples of stemmed words. "Photography" is an example for under-stemming, where not enough letters are removed. Stemming algorithms do not know irregular nouns and verbs. Unlike lemmatizers, which use dictionaries, stemming transformations are partially inaccurate. The impact of these mistakes may be neglected. For many tasks, efficient and straightforward actions like stemming (and also stopword removal) significantly improve the performance of NLP tasks, while more complicated techniques only add very little improvement [11]. Despite their shortcomings, stemming algorithms in practice yield good, fast results and are widely used.

Words	**Stem**
take, taker, takes, taking	take
computer, computing, computation	comput
goes	goe
photography	photographi

Table 2.4: Stemming examples, Porter algorithm.

2.3.2 Term Frequency–Inverse Document Frequency

Term Frequency–Inverse Document Frequency (TF-IDF) is a popular algorithm to encode text samples for NLP [23]. A text or document first has to be tokenized. The text is then treated as a bag-of-words, meaning the order is not relevant anymore. Each word is assigned a weight calculated by its relevance. The result for each document is a vector, with each cell containing a decimal weight value, corresponding to a word in the dictionary.

The weights are generated using two components: the term frequency per document, and the inverse document frequency of a word across all documents. Equation 2.1 shows the calculation of the term frequency. The number of occurrences of a term in a single document is divided by the total number of terms in that document (TF). This census implies that if a term occurs multiple times, it has a higher weight. For the inverse document frequency (IDF, Equation 2.2), the total number of documents, and the documents a term is used in are divided, and log-scaled.[12] Multiplying these two values gives the full TF-IDF value (Equation 2.3). This weighing algorithm gives a vector for each document, where the terms most significant within the document have the highest values.

$$TF(d,i) = \frac{|w_i|}{|d|}. \tag{2.1}$$

$$IDF(i) = log\left(\frac{N_{docs}}{1 + n_i}\right). \tag{2.2}$$

$$TFIDF(d,i) = TF(d,i) \cdot IDF(i). \tag{2.3}$$

Despite its frugalness, TF-IDF and its adaptations are still widely used in information retrieval systems. Used by many search engines, TF-IDF serves for querying documents [50]. A model is fitted using all items in the database, e.g., websites, books, or scientific publications. In particular, the model contains the word dictionary, the corresponding vector positions, and the inverse document frequencies. The document vectors of all documents are calculated. The input query is then processed using the previously generated

[12]The 1 in the denominator is merely a smoothing value to avoid division-by-zero errors

model. In combination with k-NN, a text categorization system can be built [57]. After fitting TF-IDF on a corpus of data (the training set), new documents can easily be transformed. Then, for instance, the Euclidean distance measure can be used to calculated document similarity.

TF-IDF has two notable disadvantages. First, the transformation removes any order between the words. Treating documents as a bag-of-words makes leveraging local semantics impossible. Depending on the task, this compromise might be acceptable or not. Second, the TF-IDF document vectors mathematically have a size as large as the dictionary. For text in natural language, depending on the preprocessing (stemming), these dictionaries can have millions of entries. Since most documents only use a minimal subset of the complete dictionary, sparse vectors and sparse vector operations are used.[13] Some implementations only store the non-zero elements of the vector and their indices. Another measure to limit the dictionary size is to set upper and lower threshold on a term's document frequency, i.e., a term has to occur in at least ten documents and not more often than in 50% of the documents. Sparse vectors and document frequency bounding significantly decrease memory and computational complexity.

2.3.3 Word Embeddings

Data sparsity is a problem when working with natural language. A transformed TF-IDF document only depends on the dictionary size and not on the length of the text. Systems that process the document as a sequence of word tokens require a per-word representation. Vector space models, in particular word embedding models in NLP, offer memory-efficient and arithmetically meaningful word representations. Subsequently, we outline the approaches of two popular ones, Word2vec and GloVe.

The most straightforward word representation is one-hot encoding: a vector of zeros, where one cell is set to 1, with the index corresponding to a word. This sparse encoding has two disadvantages. The memory allocated for a full document sample scales both with the dictionary size and the input length. Systems that subsequently process these one-hot vectors necessitate operations and internal parameters for the whole dictionary, despite most input vector entries being zero. Additionally, one-hot encoding does not utilize the relationship between words: "ship" to "ships" is as dissimilar as "cat" to "car". Statistical neural models that learn the probabilistic distribution of words have been proposed already 20 years ago to tackle this curse of dimensionality issue [9]. By transforming one-hot vectors to data points in a continuous vector space, the dictionary can be processed much more memory efficient and allows for arithmetic operations.

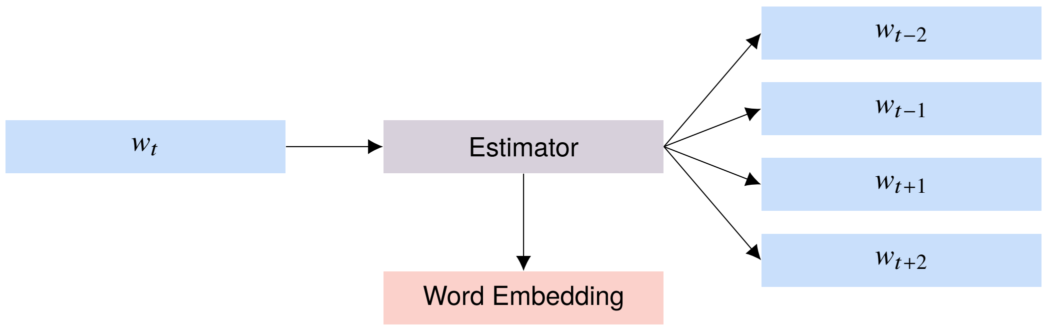


Figure 2.2: Skip-gram architecture and the generation of word embedding as a by-product. Based on a word w_t, e.g., the two words before and after it are guessed.

The most common word embedding model, Word2vec, is based on the work of Mikolov et al. [44]. The probabilistic distribution of words is modeled via skip-grams. In other terms, based on a single word, the

[13]`https://docs.scipy.org/doc/scipy-0.14.0/reference/sparse.html`

estimator predicts the words to most likely occur around it. Figure 2.2 depicts this probability estimation model. The model is designed so that at one point in the chain of calculations, a low-dimensional vector contains all the information to make a prediction. This vector is then stored as the embedding for the corresponding input word. In short, Word2vec tries to statistically find the local context of a word and encode the probabilistic information inside a dense vector, called word embedding.

The estimator model itself is a neural network and requires large amounts of training data. Depending on the used training set, properties of the embeddings may vary. Numerous pre-trained word embeddings can be used out-of-the-box as dictionaries.[14] As a text corpus, e.g., a dump of Wikipedia articles was used for training. For generating skip-grams, the text is windowed. The word embeddings commonly have a dimensionality between 50 and 1000. The model is trained until the average prediction quality of the surrounding words plateaus.

In contrast to Word2vec, GloVe [46] is a count-based embedding model. While Word2vec uses a predictive approach, GloVe creates the word embeddings based on co-occurrence: a large matrix X is used where each cell $X_{i,j}$ contains information how often word i occurs in the local context of word j (n-gram). In order to model the occurrence probabilities as dense vectors, they use a simple weighted least squares regression model instead of a neural network. The authors claim that their embedding model performs similar to Word2vec in general, but they also show that it can outperform predictive models for specific tasks. As there is no general rule to decide whether to use count-based or probability-based embedding models, the performance of both should be tested, especially if the word embeddings are part of a larger system.

$$cos\,(\boldsymbol{e_1} \angle \boldsymbol{e_2}) = \frac{\boldsymbol{e_1} \cdot \boldsymbol{e_2}}{||\boldsymbol{e_1}||_2 \cdot ||\boldsymbol{e_2}||_2}. \tag{2.4}$$

By using addition and subtraction, as well as a similarity measure, interesting embedding properties can be shown. Table 2.5 lists a few examples. The mathematical vector operations are performed element-wise. The resulting vector is compared to all existing vectors using the cosine similarity (Equation 2.4, with L2 norms in the denominator). The word or words corresponding to the closest vectors are listed. These calculations show that words are assigned interpretable values in the vector space that take semantic properties into account.

Operation	**Result**
"Merkel" − "Germany" + "Cameron"	"Britain"
"man" − "male" + "woman"	"female"
"knife" − "knives" + "tool"	"tools"
Top 4 words around "sea"	"ocean", "seas", "oceans", "waters"

Table 2.5: Arithmetic examples for Word2vec, Google News Negative 300 dataset.

Using pre-trained word vectors has become common practice in NLP systems. They can significantly improve the generalization of machine learning models when there is comparably little training data [14]. Small training sets in specific use cases often do not contain enough information to derive strong relationships between words. Subsequent systems can partially also handle words that they were not trained on due to this generalization. It makes thus sense to employ such word-to-vector dictionaries.

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

The selection of a word vector dataset is not trivial. For the generation of the embeddings, essential parameters have to be set, like window size and vector dimensionality. For common domains, the default settings yield good results. However, for extrinsic data, more attention has to be paid to tuning these parameters [42]. The training corpus used should also be as similar as possible to the target domain, i.e., regarding dialect, slang, or word order. Many publicly available data sets rely on encyclopedias, news articles, or web sites. Taking these pitfalls into account, word embedding models like Word2vec and GloVe are an important building block for modern NLP systems.

2.4 Neural Networks

In recent years, artificial neural networks have shown great success in solving highly complex tasks such as image or speech recognition. Despite the mathematical fundamentals dating back as much as 60 years [52] [24], the steep increase in computational power has aided the rise of machine learning. The principle of machine learning is simple: leave it up to the model to find patterns in the data by providing numerous examples. Neural networks allow computers to learn from experience were manually setting up rules is practically infeasible, e.g., for NLP. Since machine learning is a vast field, we will only briefly outline some concepts here that are necessary for our contributions in the main chapters.

2.4.1 Training

A machine learning model differentiates between the training phase and the prediction (or inference) phase. Training a machine learning model like a neural network involves setting the model parameters, presenting data to the model, and evaluating the performance. Modern machine learning frameworks and guiding scientific publications facilitate neural network engineering.

Feed-forward neural networks consist of several connected layers that attempt to approximate a function [18, p. 164]. Each layer holds several neurons, with links between the layers, called weights. A neuron thus takes weights, sums them up, and applies an activation function. The weights are the parameters that the network learns. During the learning phase, the model is repeatedly presented the training samples, calculates the output, and measures the error. By back-propagating the error through the network, the weights are adjusted until the performance converges. In the prediction phase, the model then can make predictions on unseen examples through a learned generalization of the training samples' properties.

The most challenging part with neural networks is the choice of hyperparameters. Hyperparameters steer the training process and model capacity. Major influences are:

- **General network architecture**: dense network, convolutional network, recurrent network, ...
- **Activation functions**: ReLU, sigmoid, softmax, tanh, ...
- **Learning optimizer**: stochastic gradient descent, ADAM, AdaGrad, ...
- **Regularization**: L1/L2, dropout, batch normalization, ...
- **Weight initialization**
- **Number of training epochs**
- **Batch size**

Neural network engineering is an empirical field. The search for correct hyperparameters can be tedious and time-intense due to the high-dimensional parameter space. While there is no universal solution to every problem, following standard practices and adapting existing scientific publications have shown to yield good results.

For a long time, so-called grid search was used to find a good set of hyperparameters. For every adjustable parameter, an exhaustive set of values is defined. After evaluating every possible combination, the parameter setup giving the best performance is used for the final model. Practically, however, this has shown to be a comparably ineffective approach in contrast to random parameter search [10]. Neural networks are not equally sensitive to different adjustments. For example, evaluating five batch sizes obtains less knowledge compared to five vastly different hidden neuron configurations. Random parameter search is thus to be preferred over grid search.

For the implementation of neural networks, machine learning frameworks like TensorFlow [1] are widely used. The core of TensorFlow is a two-step approach: first, the separate creation of a data-flow graph, and second, the execution phase. The framework allows executing the code on different devices, i.e., CPUs, GPUs, and TPUs (Tensor Processing Units). Depending on implementation requirements, different abstraction levels are provided, ranging from fine-grained mathematical calculations to pre-implemented, stackable network layers, loading architectures, and fully-trained models. Especially for the use of established architectures, high-level TensorFlow abstractions like Keras[15] can be used. Essential features like back-propagation by differentiating the computational graph, different cost functions, or standard optimizers are implemented and can be used out-of-the-box. Although there are alternative machine learning frameworks, TensorFlow is the most widely used one in research and production.

2.4.2 Convolutional Neural Networks

Fully-connected, dense neural networks impose a constraint on the processing capability. For dense layers, each neuron within a layer is connected to every neuron of the adjacent layers. This dense connectivity assumes that the input data follows a constant structure, where input features always have a fixed global position. For certain domains, like image, text, or speech, utilizing local properties makes more sense. Convolutional neural networks (CNN) [29] offer a solution to these limitations.

The architectural changes between fully-connected networks and CNNs are motivated to achieve three properties [18, p. 329–339]:

1. Sparse interactions: A filter with a kernel smaller than the whole input is windowing the input data. That leads to fewer parameters in memory and thus also fewer computing operations.
2. Parameter sharing: A learned pattern can be used for multiple inputs; it is not tied to a single one.
3. Equivariant representations: Although the detection of patterns is not strictly limited to input regions, e.g., image position, a translation is still detectable in the output.

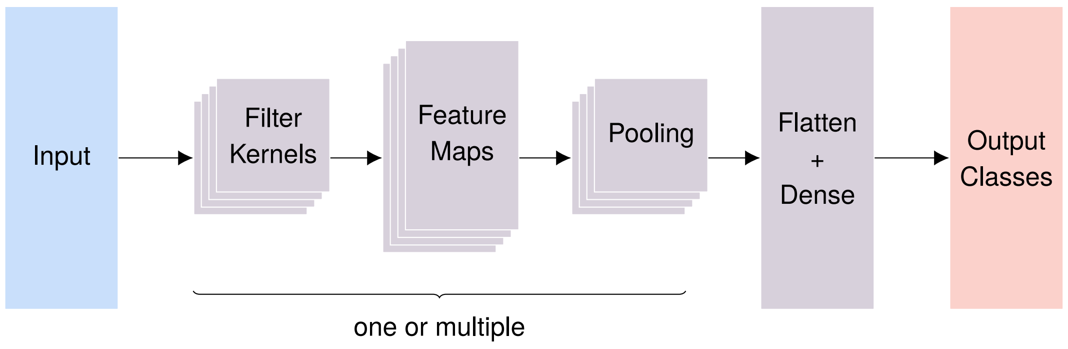


Figure 2.3: Conventional CNN architecture.

Figure 2.3 shows the basic elements of a CNN. Several filters are applied to the input and learn spatial properties, e.g., specific small arches for an image object. The convolution operations of all filters are intermediately stored in feature maps. The pooling layer acts as a subsampling layer of the feature maps. It effectively reduces the number of parameters by taking the max or average value of several feature map

[15]`https://www.tensorflow.org/guide/keras`

entries. In many state-of-the-art CNNs, multiple combinations of filter kernels/pooling layers are stacked on top of each other, leading to a deep convolutional architecture. One or more dense layers commonly flatten the pooled values and calculates the ultimate classification or regression output.

Originally, CNNs have been developed for image processing. For images, the input has three dimensions: the resolution plus the color channels (e.g., RGB). The CNN then learns many 2-D filters to find local patterns in the image, combining the information to obtain a global interpretation. This hierarchical pattern recognition has been inspired by the way that the human brain processes visual signals [27].

Recently, however, CNNs have shown to be useful for NLP as well. Text processing has intuitively been tackled by recurrent neural networks (RNN), which have been designed to process sequences of data. Due to their recursive structure, RNN training cannot be parallelized as efficiently as for CNNs, leading to longer training times. Instead of a 3-channel image pixel, the CNN is fed word embeddings of, e.g., dimensionality 300. CNNs do well in tasks where positional features have to be extracted, which is a useful characteristic of human language as well.

Yin et al. [64] did a comparison study for RNNs and CNNs for NLP . They have evaluated various typical NLP tasks, such as sentiment classification or handling question/answer schemes. In general, their findings show that CNNs perform only slightly worse and overall very similar to RNNs in their experiments. They also conclude, however, that the selection of right hyperparameters for the CNN's batch size and hidden neurons is crucial to obtain similar results for RNNs and CNNs. These findings show that convolutional architectures are a reasonable and time-efficient approach for NLP tasks.

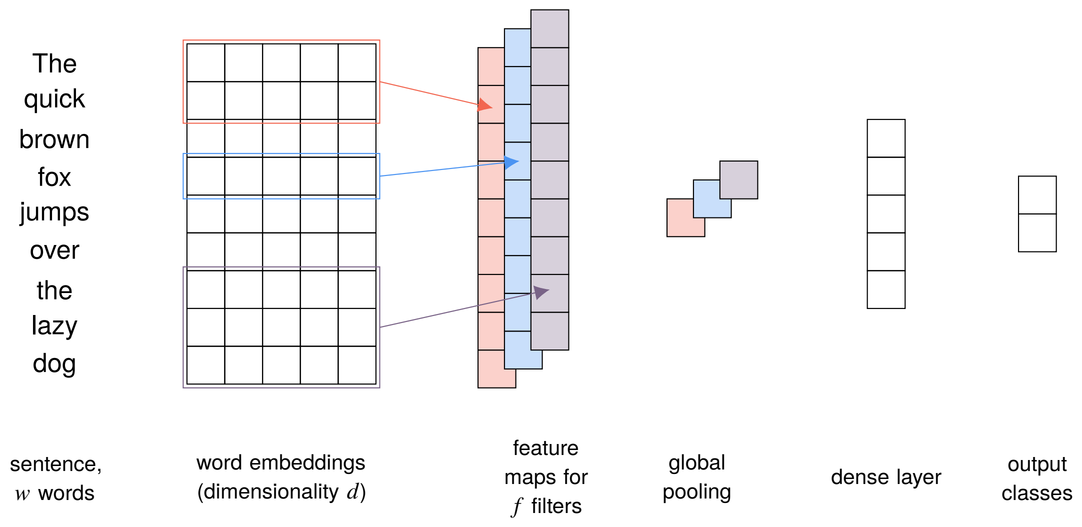


Figure 2.4: Text classification CNN: A sentence is represented via pre-trained word embeddings. Different filter sizes are applied on a single convolution layer (1-d convolutions). After pooling the maximum values, a dense layer does the classification. (Note: Weight connections are not drawn for the sake of readability. Stride is 1, padding is set to same).

A CNN for sentence classification has been proposed by Kim et al [25]. Figure 2.4 depicts the architecture. The model is capable of processing a fixed number of words, so the sentence has to be padded or capped. As a first step, the input words are translated to pre-trained word embeddings and concatenated to form a matrix. The 1-dimensional convolutions are calculated on the input matrix. The per-filter output for the whole input is stored in a feature map. The maximum value per feature map (global max pooling) is passed on to a dense layer. As network output, a classification is learned, e.g., with two target classes.

The model uses different filter sizes. This has the effect that filters can learn the impact of a single word as well as short word combinations. Unlike in Figure 2.4, there are multiple filters of one kernel size in practice, so that numerous words or (word combinations) can be learned. Utilizing word embeddings (Word2vec, Google News, 100B words in their case) allows for better generalization of the filter kernels. The max-pooling layer finds high-impact input features. Global max-pooling allows the dense layer to make a classification based on the whole sentence by combining single words from local contexts.

2.4.3 Undercomplete Autoencoders

Traditional compression algorithms follow a manually defined rule set to encode and decode data. From end to end, the input is first converted to require less space and then reconstructed again, either with loss or ideally lossless. Undercomplete autoencoders (UAE) follow this concept but learn the encoding and decoding rules through training a neural network. These encodings have properties so they can be used for e.g. classification and clustering.

Autoencoders aim to find a mathematical transformation between the input and the output [18, p. 499–501], as schematized in Figure 2.5. UAEs receive the same input and output during training. The special layer architecture causes the undercompleteness: Between the input and the output layer, there is at least one layer of lower dimensionality than the input (and output). This forces the network to learn important characteristics of the training data, expressed by the low-dimensional layer only, the so-called latent space. For one forward-pass of a sample, the data there is an encoded representation of the input data. UAEs learn a non-linear principal subspace of the data as part of neural network training.

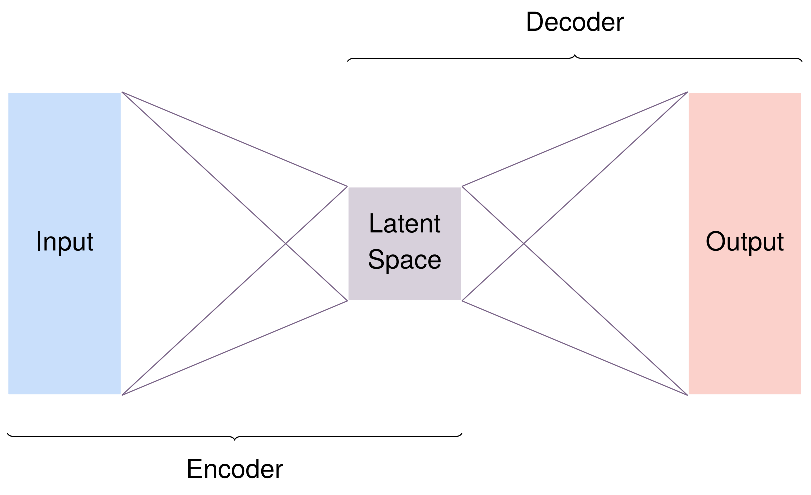


Figure 2.5: Undercomplete autoencoder: The input training data equals the output training data, which lets a low-dimensional hidden layer learn a principal subspace transformation. After training, the model can be split into an encoder and decoder part.

The latent-space vector representation of samples can be used for dimensionality reduction and clustering tasks [55]. Traditional clustering algorithms like K-Means try to find similarities directly in the input. For data with a vast feature space, the curse of dimensionality problem often impedes finding meaningful clusters. UAEs address this problem by transforming the original input to a low-dimensional subspace before clustering. Subsequently, conventional clustering algorithms are applied.

2.4.4 t-SNE Dimensionality Reduction

t-SNE is a parameterized, non-linear dimensionality reduction [39]. Data points are projected to a 2- or 3-dimensional subspace based on probabilistic estimations. The mapping both takes global and local context of data points into account. The t-SNE reduction is commonly used to visualize high-dimensional data efficiently.

Stochastic Neighborhood Embedding (SNE) calculates occurrence probabilities based on point-wise distances [20]. The Euclidean metric is used to compute distances. The probability is estimated by a Gaussian model for each data point. The concept is that the whole neighborhood around a certain point, meaning all other points, can be expressed by a probability function. Close points have a high co-occurrence likelihood, while for the ones farther away, the neighboring likelihood is close to zero. This assumption has to hold for both the original space, as well as the transformed, low-dimensional space. The problem can be solved by approximating one distribution to another. The parameters of the low-dimensional transformation are approximated by optimizing the Kullback-Leibler divergence between the distributions. While SNE was found to yield reasonable visualizations, the algorithm is costly and tends to produce crowded point regions.

For this reason, two modifications were made by Van der Maaten and Hinton to obtain t-SNE. The SNE function was tweaked to be symmetric, i.e., the probability for point p_A being next to point p_B is the same as for p_B being next to p_A. High-dimensional data points in practice have often the same distance, leading to crowded regions–another effect of the curse of dimensionality. Thus, a t-distribution is used for the low-dimensional probability function. Due to this distribution's shape, it pulls crowded, distant points further apart. The t-distribution creates more visual space in the 2-d or 3-d space. Using gradient descent t-SNE can be trained much more effectively compared to SNE and gives compelling visualizations.

When used in practice, several properties of t-SNE have to be considered. Since t-SNE uses gradient descent, commonly there is more than one local minimum and therefore no single solution. The iterative process takes several epochs and might not converge depending on the data. t-SNE also has an adjustable hyperparameter, the perplexity, which balances the importance of neighbors against more distant points. For perplexity, usually, several configurations are visualized (values between 2 and 50 according to the original paper). Different perplexity settings should be tested since they can give very different results, as shown in Figure 2.6. With knowing t-SNE's properties and setting its hyperparameters correctly, it has shown to be a very effective and useful tool for data visualization and clustering.

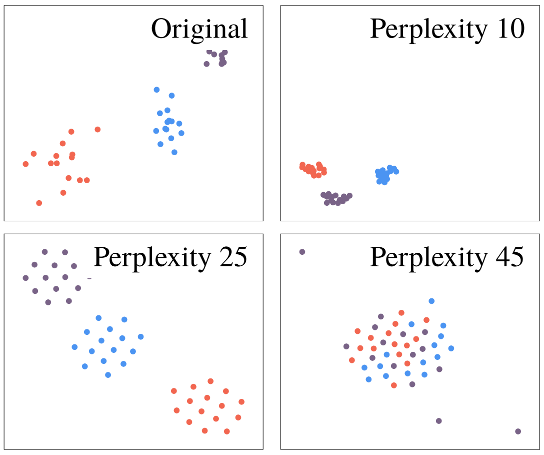


Figure 2.6: t-SNE: Sample data transformed with three different perplexity settings. A good perplexity value must be chosen empirically and depending on the domain and task.

2.4.5 Performance Evaluation

For evaluating a machine learning model, a quantitative performance measure must be chosen. There are various mathematical metrics commonly used, each having a different explanatory power. Additionally, special attention must be paid to data set partitioning. Selecting a reasonable performance metric and sample splits is crucial for training reliable machine learning models that generalize well in production. Subsequently, we describe a few metrics and pitfalls for classification tasks.

The accuracy rate is the most common metric (Equation 2.5, according to a two-class confusion matrix[16]), but can be deceiving depending on the data. As a first reference value, the accuracy should exceed the random guessing rate i.e., for two classes: 1 : 2 = 50%. However, this baseline does not factor in the class distribution of the training data. If the majority of data belongs to one class, e.g., 98 out of 100 samples, a classifier could always guess the majority class–hard-coded, independently of the actual sample. It would still achieve a deceivingly high accuracy of 98%. Especially for imbalanced data, the accuracy rate is insufficient to portray model performance in a good way based on this accuracy paradox.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.5}$$

For said reason, other metrics which show the cost of specific mistakes are advisable. Precision, recall, and F_β score (Equations 2.6) take into account all possible result configurations [18, p. 417–419]. While accuracy only differentiates between correct and incorrect results, these metrics into account how many true-labeled samples result in true predictions, in false predictions, and vice versa.

- **Precision** describes how the model makes true predictions, compared to samples that are labeled true. It thus shows the reliability of a model, the preciseness of a true prediction.
- **Recall**, on the other hand, shows how many correct true predictions a model captures. A low recall means the model is very cautious about making a positive prediction.
- The F_β**-score** is the harmonic mean of precision and recall, merging the two metrics into one. β can be used as a parameter to either weigh precision or recall higher. With F_1 ($\beta = 1$) both have the same weight.

In practice, precision, recall, and F_β-score are always provided for a transparent presentation of the model's performance.

$$\begin{aligned} \text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}} \\ \text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}} \\ F_\beta\text{-Score} &= (1+\beta) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\left(\beta^2 \cdot \text{Precision}\right) + \text{Recall}} \end{aligned} \tag{2.6}$$

Scenarios in which a classification task outputs multiple categories per sample lead to additional challenges. In multi-label classification, a single sample can be part of multiple output classes, e.g., an Android app has multiple permissions. The class imbalance in this case example stems from the fact that more applications require the camera permission than the calendar permission. In other terms, the sample support for one class is higher than for the other class, which should be reflected in a performance metric.

The class support must be dealt with when a summarizing metric for a model, independent of the individual classes, is required. Appropriate metrics, like precision, recall, or F_β-score, are averaged. The averaging is done either per-sample or per-class. The micro average computes the metric over all samples. In contrast, the macro-average computes the metric per class first and then averages all the classes results. If a class has low support, its influence vanishes in the micro-average. Thus, for imbalanced multi-label classification tasks, macro-averaging is preferred.

Training a neural network also requires evaluating its generalization quality. According to the universal approximation theorem [21], a neural network with a sufficient number of neurons can approximate any function. This theorem implies that neural networks are very good at memorizing exact data, not

[16]TP...True Positives, TN...True Negatives, FP...False Positives, FN...False Negatives

necessarily learning only relevant properties. For unseen samples, later, the performance might be significantly worse.

To prevent over-fitting on the training data, the training data is split [18, p. 108–120]. Three partitions are created: a training set, a validation set, and a test set. The training set contains the majority of samples. After training the network for a few epochs, the performance metrics, e.g., the ones mentioned before, are evaluated on both the training and the validation set. Training is stopped when the training performance keeps increasing, but the validation performance stagnates or decreases again. Finally, the performance is also evaluated on the test set, which has never been used for training. This regularization technique, ending training when the validation performance decreases, is called early stopping.

For comparably few training samples, statistical differences in the sets can influence the performance calculation. Dividing one's training data into the sets mentioned above without negative side-effects, in theory, requires the samples to be independent and identically distributed. For a theoretically infinite amount of data, the set a particular sample is assigned to should not influence the performance. In this hypothetical case, the distribution of, e.g., target labels, is identical in all sets. If in practice, this assumption does not hold, it makes sense to create multiple set configurations.

A common way to achieve this end is k-fold cross-validation. The k parameter stands for the number of splits. The whole training data set is divided into k partitions. One partition is used as a validation set, the other ones as a combined training set. All possible combinations are used to train the model. In the end, the validation performances are averaged over all splits. When the data set does not have hundreds of thousands of examples, this method is an appropriate way to deal with statistical uncertainty in the set configurations.

2.4.6 Model Explanation

The essence of machine learning is finding patterns via function approximations in a given set of data. The inherent underlying problem is that based on the model output only, it is not always obvious why models make the predictions they do. Learned relations are still correlation-based. Before putting a model into production, often more trust than that is required. Model explanation algorithms such as LIME and SHAP are methods that interpret how ML models make their decisions.

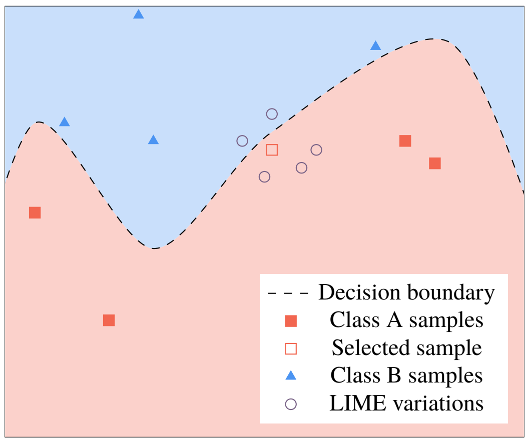


Figure 2.7: LIME finds parameter influences by varying a single sample and handing the different samples to the trained model. By varying different properties, e.g., shifting positional features, the property importance for the actual sample's prediction is estimated.

LIME is a method proposed to give local, model-agnostic explanations [51]. Model agnosticism means that the ML model is treated as a black box, i.e., LIME does not know about the inner workings of the model. Locality means that it works based on a particular sample which is slightly varied to obtain new samples, one by one. These samples form a local context around the original sample. The impact of each modification causes different model outputs. In the end, all differences are aggregated to show which properties of the input are most influential for the original sample's prediction. This process is shown in a 2-d example in Figure 2.7: A classifier has been trained for two classes and has learned a decision boundary. The decision boundary itself is not evident and should be found, i.e., the influence of the x and y coordinate. Thus, Sample □ is varied in several directions to obtain the new samples ∘. Classifying these new samples and taking the variation, i.e., how much the x/y value is changed, into account reveals where the decision boundary is located. For high dimensional, complex input data knowledge about the decision boundary can show which words of a sentence or which pixel areas of an image affect the output most. LIME has an implementation in Python that modifies text or image samples systematically, feeds them to a trained ML model, and visualizes the impacts.[17]

The SHAP framework is another approach to find estimate the importance of sample features [38]. The algorithm behind it is based on Shapley values, a concept in cooperative game theory: of n potential players, several combinations of $k \leq n$ players are possible, i.e., can play together against the bank. Each combination of players achieves a different (monetary) result. The Shapley value shows the contribution of each player by incorporating different combinations. Sundberg et al. used this concept in combination with additive feature attribution. By masking out several parts of the input features, different model results per sample are obtained. The results can be united according to Shapley, but this is computationally expensive. SHAP provides several approximations, e.g., one for neural networks called Deep SHAP.[18] By leveraging knowledge about the network's parameters and structure, and not treating it as a black box, Deep SHAP creates a simpler, approximated model. The Shapley values are linearly calculated on this model. Unlike LIME, Deep SHAP is not model-agnostic. It uses the simplified model to estimate feature importance and applies it to a feature. Deep SHAP is computationally efficient, consistent with human intuition, and useful for explaining class differences.

Model interpretability, however, is only a loosely defined concept [34]. There is no formal verification approach to prove that a neural network always works as it is designed. The claim that linear models are more comfortable to explain than deep neural models has not yet been proven mathematically–it is merely easier for humans to see causality. Although deep ML models have yielded astonishing results recently, their complex inner structure requires additional effort to build trust in them. Despite that the field is in its early stages, current explanation methods like LIME and SHAP can give useful information about a model's operating principles when we analyze app descriptions, permissions, and app internals.

2.5 Summary

In this chapter, we explored some concepts our subsequent contributions are based on. We started by showing how Android apps are built and packaged. Thereupon we named starting points and associated challenges for app reverse engineering and program behavior investigation. For processing natural language, we looked at how text samples can be represented, both with keeping their sequential structure (word embeddings) and without it (TF-IDF). Regarding neural networks, we first covered convolutional nets and undercomplete autoencoders. We also summarized concepts on how to interpret results (t-SNE), evaluate model performance, and approaches to explain complex ML models. Building on these blocks, we propose our app publishing assistance system in the main chapters.

[17] `https://github.com/marcotcr/lime`

[18] `https://github.com/slundberg/`

Chapter 3

Related Work

Analyzing mobile apps with machine learning has become a successful field in recent years. Numerous authors proposed solutions regarding app descriptions and app features. While some are more aimed towards permission-based malware classification, others focus on description text quality. Out of the vast number of partially related publications, we give an overview of notable work similar to ours and show some of their particular shortcomings that motivate this thesis.

We looked at three different fields in the broader context of app analysis in recent literature. (i) App similarity is calculated for different reasons based on both description and program behavior. (ii) It is common to extract a chosen set of attributes from the app, i.e., the permissions, API calls, strings, or similar, to interpret the app to determine its purpose. (iii) Since app permissions play a crucial role in protecting the user's privacy, depicting them correctly within the description has become a growing field of research. Subsequently, we give examples of interesting work in these partially overlapping fields. While our own contributions do not directly focus on malware analysis, it is still a form of app behavior interpretation and thus worth mentioning.

Finding similarity between apps, i.e., categorization or clustering, is desirable for several reasons. The most apparent clusters are the categories that developers assign to their apps when they publish them in markets, e.g., Lifestyle, Travel and Local, or Social. Google provides guidance on selecting the appropriate category based on signal words.[1] Aminordin et al. [5] conducted a preliminary study of around one thousand apps in three categories that showed the Google guide's shortcomings. Using word frequency in the description, they verify category assignments. Their findings show that around a third of these apps were miscategorized. Al-Subaihin et al. [2] proposed a system to categorize apps based on their description automatically. They extract textual features (n-grams), apply TF-IDF, and group apps with hierarchical clustering. They conclude that overall, their approach yields significantly better class assignments and that for now, Google Play does not exhibit a sound categorization system.

CLANdroid [59] aims to find similar apps by defining semantic anchors. These anchors are internal app properties such as sensor information, permissions, intents, and identifiers. A matrix per app is generated with Latent Semantic Indexing, a method improving keyword-based information retrieval on several "phrases"–in their case, "phrase" refers to several of their anchors. The similarity between apps is calculated using the cosine distance. The proposed system can be used for search engines and recommendations.

Other research is done evaluating whether a description states clearly which permissions an app uses. AutoCog [49] divides the text into sentences and analyzes them with the Stanford NLP Parser[2] to obtain

[1]`https://support.google.com/googleplay/android-developer/answer/113475?hl=en`

[2]`https://nlp.stanford.edu/software/lex-parser.html`

their grammatical structure. Explicit Semantic Analysis, a specialization of TF-IDF that takes semantic relatedness into account, is used to find related pieces of text. Subsequently, their system gives a list of phrases correlated to specific permissions, which is then used to annotate the relevant sentences in the description. WHYPER [45] annotates sentences as well, but with a critical difference. This system, in contrast to AutoCog, creates a semantic graph from the Android API docs. Based on class names and method names, relevant words are identified, e.g., the `Contracts` class containing a member named `email`. Sentences are then compared with these words in the semantic graph. One of the most recent publications is AC-Net [15]. A recurrent neural network is used to model the semantics of description sentences. The authors claim their work is the first to apply deep learning to the permission-to-description fidelity. To represent their words, they use self-trained embeddings. Instead of outputting a simple yes/no answer, their system gives a score corresponding to the level of consistency. Compared to keyword-based searches, these approaches perform better at identifying permission-related sentences as well as their absence.

For modeling an app's behavior, in general, more aspects than the set of permissions can be taken into account. Takahashi and Ban [56] use several thousand APK-based features. They combine permissions, API features, category, and their calculated cluster assignments and attempt malware classification on them. They use a modified Support Vector Machine derivative called SVM-RFE, which recursively eliminates features that have little impact on the prediction. Although their approach is used on a binary classification problem, the comparably high number of app properties should intuitively be useful for a more general behavior modeling as well. AndroVault [43] addresses a similar problem by creating a knowledge graph. Also based on multiple app attributes, AndroVault creates a graph with the nodes being the apps. The edges are probabilistic properties of the app (e.g., code similarity) and deterministic ones (e.g., author). It uses more properties than Takahashi and Ban's work, such as malware classes or more APK metadata information like the description. According to them, machine learning algorithms can then be applied on top of this graph to find malicious apps or repackaged apps. These two systems show that the interpretation of particular application attributes to analyze the behavior is feasible.

While most work is done on the logic parts of the code, some focus also on user interfaces. Kuznetsov et al. [28] extract identifiers and strings in human language from activity XML definitions and verify them against the description. The descriptions are represented as bag-of-words in their work. They then use Latent Dirichlet Analysis to find clusterable topics for both the descriptions and for the user interfaces. The comparison of these two affiliations is used for the assessment of whether an app is sufficiently well described. They propose that their approach can be used for enhancing insufficient description texts.

The work listed above shows the scientific interest in the intersecting fields of app behavior and description quality analysis. Our review shows, however, that most of the work has been done with conventional machine learning such as Support Vector Machines and segmentation algorithms like hierarchical clustering or k-means. Apart from AndroVault, most projects are done on comparably low sample sizes. Using as many app attributes as possible, like from the user interface, depicts the app more accurately and improves the performance of all kinds of analysis tasks [26].

Hence, we motivate our work by filling several scientific gaps regarding the publications as mentioned earlier. First, our work uses real-world datasets that are significantly larger than the datasets used in the research above. Although our samples are not manually labeled, we still achieve good results. Second, we cluster apps with non-linear algorithms and show superiority over linear algorithms. Also, third, we use neural networks for app clustering, permission prediction, and description generation to learn more complex relations between model inputs and outputs, compared to simpler machine learning algorithms. Finally, we not only rely on source code parts but also make use of the knowledge stored in an Android app's resource system. Based on these five significant differences between our work and the current research state, we propose our three-part app publishing assistance system for Android apps.

Chapter 4

Clustering based on Descriptions and Permissions

After having summarized necessary background knowledge and related work, the first of our contributions is the clustering-based analysis of apps. We focus on two attributes to cluster apps, permissions, and descriptions. We start by briefly motivating our approach and continue with describing the preprocessing steps required for our machine learning algorithms. Then, we lay out an autoencoder neural network architecture to convert descriptions and permissions to low-dimensional embeddings. The clustering on a 2-d space is done by t-SNE. We evaluate our approach with a real-world app dataset. Based on the obtained subspaces, we first visualize a significant subset of our apps. Then, we analyze single apps to show particularly interesting findings.

4.1 Motivation

Clustering information is an essential subdivision of data science, which is also interesting for app analysis. Due to the high number of Android apps being published every day, it stands to reasons that many apps have a similar purpose. By visualizing apps as data points and positioning them closer to each other or farther away from others, quick assessment of an app is possible. With the use of cluster and dimensionality reduction algorithms, we aim to capture apps that are alike as well as dissimilar.

As listed in Chapter 3, many previous publications do their analysis with non-neural machine learning techniques and linear subspace transformations. With our approach, we attempt to identify similar apps through exploiting the non-linear, adaptive characteristics of neural networks and t-SNE. Our goal is to visualize related apps and position them close together. Additionally, we aim to be able to identify outliers that can express potential misuse, e.g., requesting more permissions than similar apps or dissimilar description texts despite the functional similarity.

To the best of our knowledge, no extensive evaluation of non-linear clustering approaches for apps has been published. We argue that to efficiently capture the meaning of apps based on their permission sets and their description texts, exploiting non-linear properties of this data is superior to linear methods. As part of an app publishing assistance system, clustering similar apps based on their descriptions and permissions is a useful starting point for app analysis.

4.2 Approach

To attain permission- and description-based clusters, we stack undercomplete autoencoders and t-SNE. The autoencoders have different architectures and preprocessing steps for permissions and descriptions. We use the data representation in the middle, the latent space, as input for t-SNE. Based on the final, low-dimensional output of the t-SNE models, we conduct our analysis in the evaluation section.

4.2.1 Permission Autoencoder

The permission autoencoder (PAE) receives single permissions as binary values as input and output. The preprocessing is outlined in Figure 4.1. In a first step, we parse the manifest files of all apps for their `<uses-permission>` tags. Since developers can define app-related permissions as well[1], we impose a lower limit on permission occurrence. This reduction leaves us with a list of permissions of size N_p that occur in a minimum of 10 apps. We assign each permission an index $0 \leq p_i < N_p$ to be able to represent them with vectors. In a second step, we represent each app's permissions with one-hot encoding. For each app, we initialize a row vector of size $1 \times N_p$ with all zeros. If the app has a particular permission declared in its manifest, we set the field p_i corresponding to the permission to 1. Each app sample is then represented with its permissions only as a one-hot encoded binary vector.

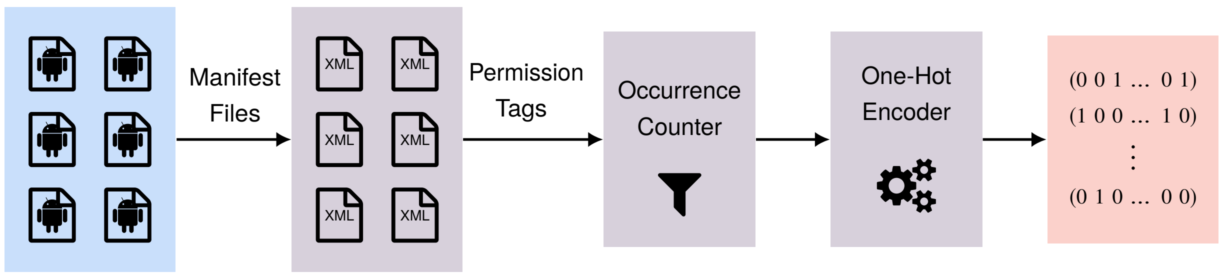


Figure 4.1: Preprocessing for the permission autoencoder: APK Manifest permissions occurring in at least 10 apps are one-hot encoded.

4.2.2 Description Autoencoder

The description autoencoder (DAE) is supposed to receive TF-IDF vectors of the app description texts. To obtain the tokens necessary for the TF-IDF vector calculation, we perform the following steps:

1. **HTML tags**: The formatting of our description texts is done with HTML. We remove all structural information by stripping the strings of all HTML tags and keep the inner texts.
2. **Special characters**: Many publishers format texts not only with HTML, but also with Unicode characters, like for listings, paragraph separation, etc. We remove all non-alphanumeric characters, including dashes, apostrophes, and dots.
3. **Tokenization**: We split the texts non-alphanumeric character, including whitespace characters. This gives us a list of tokens, where we remove all tokens that are shorter than two characters.
4. **Stopword removal**: In a next step, we remove the tokens that are considered to have little meaning for a sentence, commonly referred to as stopwords. We use the list from the Python Natural Language Toolkit.[2] Although the TF-IDF algorithm would penalize such stopwords with a high document

[1]`https://developer.android.com/guide/topics/permissions/defining`

[2]`https://www.nltk.org/`

frequency, removing these tokens based on a list before the machine learning model sees the data is computationally more efficient.

5. **Stemming**: Finally, we attempt to strip words of their grammatical variations by applying the Porter Stemmer algorithm [47] on each token.
6. **Lowercase**: All tokens are transformed into lowercase to compensate for sentence starts, parts of brand names, misspelling, etc.

Based on these six steps, we transform each app description in a list of stemmed tokens.

These tokenized descriptions are subsequently converted into TF-IDF vectors. The TF-IDF algorithm requires several configuration parameters to be set, which we choose empirically: A token must occur in at least 0.5% and not more than 15% of all app descriptions. We discard any app description that does not contain at least five of these tokens. Then, we calculate the inverse document frequency using the smoothed log variant as described in the background chapter. Based on the obtained TF-IDF vectors per app description, we finally do a per-sample normalization (L2 norm), so that the sum of squares of each TF-IDF vector's elements is 1. Upon normalization, these vectors can be used as inputs for our neural network.

4.2.3 Training

For training our networks, we need to find suitable performance metrics for the autoencoder's reconstruction ability. The PAE is a common, binary multi-label task, which we evaluate using the F_1-Score. For the TF-IDF vectors, which are floating-point numbers between 0 and 1, we need a different approach. Unlike the PAE, which is a classification problem, the DAE is a regression task. As a first metric, the mean squared error can be used, but it has a severe interpretability downside: It does not give any details on the actual reconstruction of words. Thus, we need a more refined performance metric for the DAE evaluation.

$$u_j^{(i)} = \begin{cases} 1 & \text{if } v_j^{(i)} \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Only for the performance metric, we propose a binary transformation of our regression problem as in Equation 4.1. We set a threshold θ which transforms each TF-IDF vector $v^{(i)}$ for app description i. If a TF-IDF value at position j in vector $v^{(i)}$ is greater than θ, it is set to 1, otherwise to 0. We chose the value for θ so that it ignores output noise of the network, i.e., values that are close to zero but are not exactly zero due to approximation errors. By observing the network output, we decided for θ as 10% of the average non-zero TF-IDF vectors' values. We emphasize that this transformation is only used for the performance metric, not for the loss function. It allows us using binary classification metrics for early stopping.

Random search is used to decide for good network architecture. We randomly try different setups of layer depth and hidden neurons per layer within a defined range for the autoencoders. We impose the following constraints on our random search:

- The autoencoder has to be undercomplete, i.e., at least one layer has to be of lower dimension than the input/output.
- The encoder part is symmetrical to the decoder part. We only generate the encoder's architecture and mirror it except for the latent space layer.
- The encoder has to consist of one or more hidden layers, not including the input and the latent space layer. The number of hidden layers L can range between 1 and 8.
- The latent space layer consists of 5 hidden neurons for the PAE and 10 for the DAE. We do not require a 2-dimensional representation since t-SNE performs the final dimensionality reduction. DAE has a higher dimensionality because the input vectors are larger.
- Each other hidden layer j can consist of a random number of neurons n_j that is derived from the input size: between 30% and 100% of the full input size.

- Since deeper autoencoders are generally preferred [18, p. 505–506], we limit the hidden neuron count further. The more layers our encoder has, the fewer hidden neurons per layer it should have. We thus divide the randomly chosen neurons per layer by the square-root of the total number of layers: $n_j/\sqrt{L}$.

The other hyperparameters of the network are fixed and listed in Table 4.1.

Optimizer	Adam, $\eta = 0.001$
Batch Size	32
Weight Initialization	Glorot (Uniform)
Loss Function	Binary Cross-Entropy (PAE)
	Mean Squared Error (DAE)
Latent Activations	Linear
Hidden Activations	ReLU
Final Activation	Sigmoid (PAE)
	Linear (DAE)
Early Stopping	Patience: 10 epochs
	Delta: 0.5% F_1-Score

Table 4.1: Neural network hyperparameters for the permissions clustering autoencoder (PAE) and the descriptions clustering autoencoder (DAE).

The random search is done on half of the training data, so the search takes less time. Based on the performance metric described above, the F_1-Score, we choose the best architecture for permissions and descriptions, respectively. During the random search phase, we adopt early stopping. After each epoch, the performance is evaluated on the validation set (25% validation, 75% training). If the performance value does not improve within the subsequent ten epochs again, this network state is used as the final state. Otherwise, training continues. After these ten epochs where the validation performance has not improved anymore, the previously best model state is restored and used to calculate the embedding vectors.

Figure 4.2 outlines the whole process. We first extract the permissions from the app and get their textual descriptions. The permissions are one-hot encoded, while the descriptions are tokenized, the tokens are stemmed and TF-IDF transformed. After finishing the parameter search, we use the final architectures to obtain the 5-dimensional embeddings for each app's permission set and the 10-dimensional embeddings for each description. These embeddings are further scaled down in regard to dimensionality and clustered with t-SNE. The number of embeddings used for t-SNE is reduced. Due to t-SNE's gradient descent optimization, the computational complexity heavily depends on the sample size. The reduced sample set is used as input for t-SNE with various perplexities. After finding a suitable 2-dimensional representation, we can analyze app properties visually and mathematically. In the next section, we demonstrate the applicability of this process with a large app dataset and some exemplary findings.

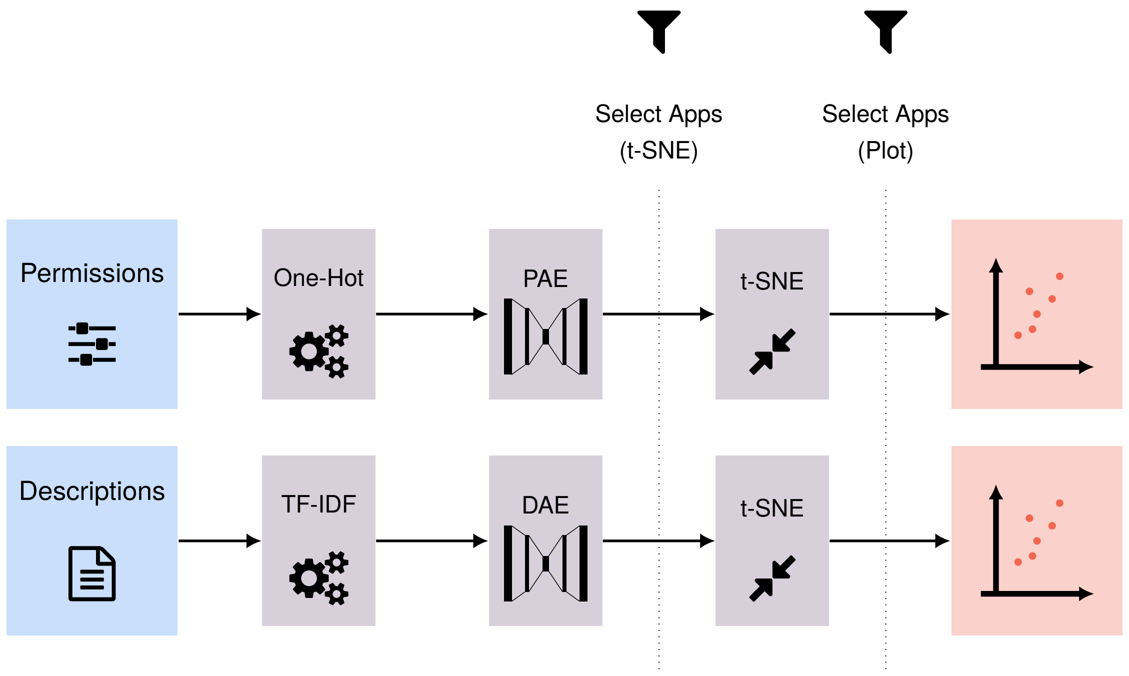


Figure 4.2: System overview: Both permissions and descriptions of the apps are preprocessed and then used to train separate autoencoders. The embeddings calculated via their latent spaces. For the t-SNE transformation, only a subset of all embeddings is used for performance reasons, which is reduced further to give meaningful plots. Apps of interest are then highlighted in the plot or distance-wise evaluated.

4.3 Dataset

In order to evaluate our approach, we use the PlayDrone dataset [60]. We explain how we select a subset of these apps, depending on certain app properties and metadata. Based on this list, we further reduce the apps that are used for our analysis. Finally, we list some statistical features of the dataset to provide a clearer understanding of the data we base our evaluation on.

PlayDrone is a Google Play Store crawler that has created a dataset of over 1 million apps. The project uses several attack vectors to circumvent the store's protection so that it can mass-download applications. Importantly, the project crawls not only APK files, but also the matching store's metadata, e.g., description, category, and download count. We use the PlayDrone snapshot provided by the Web Archive.[3]

We limit the number of apps we initially process to around 110,000. We download apps from all categories except for the *Games* category. Since games are inherently differently developed, we entirely exclude this category. Based on the download count, we sort the available apps and obtain them separately for each category.

In the next step, we filter out cross–platform apps. Nowadays, many Android apps are built with hybrid technologies that have their logic implemented in a language other than Dalvik bytecode. The most common way is to use web technologies, i.e., HTML, CSS, and JavaScript. For these website-like cross–platform apps, many frameworks and libraries have become established. Famous examples are Cordova, Xamarin, and Adobe AIR. Many developers, especially of large companies, create their custom

[3]`https://archive.org/details/android_apps&tab=about`

cross–platform architectures that suits their needs. PlayDrone's latest available snapshot is from 2014, so the frameworks used do not reflect the current distribution in 2019. While for the clustering task-irrelevant, we need a way to detect these apps since the work in the next two main chapters focuses on native Android applications.

Covering all these frameworks and custom implementations is out of scope for our work, which requires us to find a simple and efficient detection mechanism. We focus on the 20 most widely used third-party frameworks we found in the PlayDrone dataset. Most of them can be identified by looking at the app's source code folder layout. We use Apktool[4] to unpack the APK and decompile its code to *Smali*. Smali is a representation of DEX code that contains all class files in a hierarchical folder layout. For each app, we create a list of directory paths and check them against our list of known paths. If one app contains such path that relates to a known third-party framework, we dismiss this app and do not use it for any of our subsequent work. A full list of the directories that we use to identify specific frameworks is given in the Appendix in Table A.1.

For our two clustering tasks for descriptions and permissions, apart from filtering out games and cross–platform apps, we set the following constraints on the samples to be selected:

- **Descriptions**: The raw description has to be in English, and the raw text (non-HTML) has to consist of more than 30 characters.
- **Permissions**: An app has to request at least one permission. As described in the previous section, a particular permission has to occur in more than 10 apps to be taken into account.

Table 4.2 shows some statistical details about the apps our dataset has.

Total number of apps	**115,294**
Categories (except gaming)	24
Detected cross–platform	7,543 (6.5%)
English descriptions	81,785 (70.9%)
English descriptions (>30 characters)	81,393 (70.5%)
No-permission apps	41 (0.04%)
5+ million downloads	1,169 (1.0%)

Table 4.2: App dataset properties.

4.4 Evaluation

Subsequently, we evaluate our clustering approach with the previously described PlayDrone app dataset. We begin by listing the performances of our randomly chosen architectures and the final two ones we opted for. For the descriptions, we show visualizations that compare t-SNE and PCA in regard to app categories. Then, we select several apps manually and analyze the subspace neighborhood of certain app descriptions. Finally, we use permission embeddings to describe interesting properties. Our visual and textual descriptions show the usefulness of our approach for app relatedness.

Figure 4.3 shows a total of 100 evaluated architectures for the permission autoencoder and 100 architectures for the descriptions autoencoder. For permissions, all setups yield F_1-scores between 85% and 94%. The plot shows that for more than 4 layers, performance starts to decrease. The best performances are achieved with 2 to 4 layers. The description autoencoder performed best with 2 or 3 layers. There was

[4] `https://ibotpeaches.github.io/Apktool/`

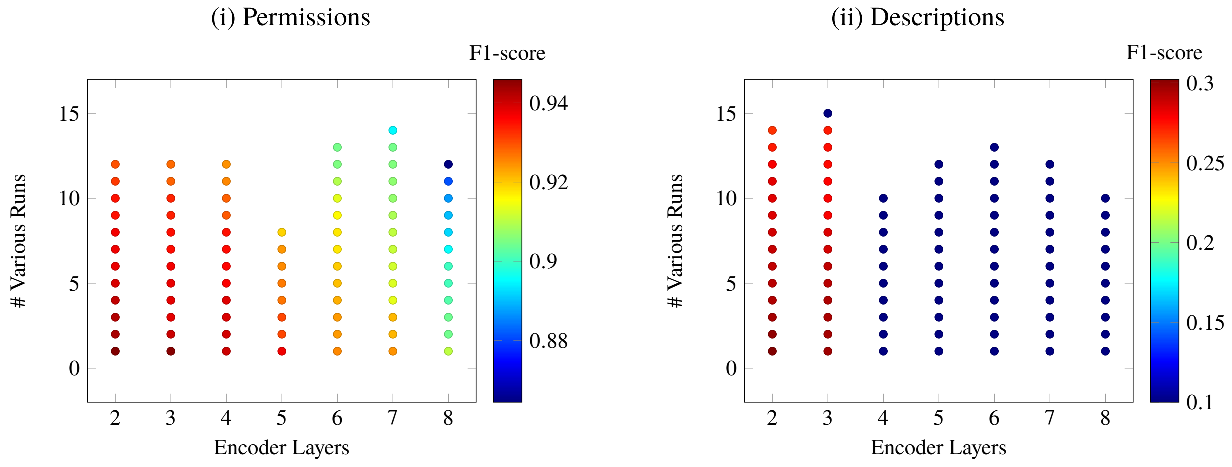


Figure 4.3: Random architecture search for 100 setups for permissions and descriptions. Gradient bar signifies the F_1-score.

a significant performance difference over all architectures, where the ones with 4 layers or more resulted in an F_1-score between 8% and 10% only, while the other ones achieved 26% to 30%. The number of hidden neurons per layer had a comparably low impact. The final architectures and their performances are listed in Table 4.3. These are the autoencoder networks we use for the following examples.

Despite the assumption by Goodfellow that deep autoencoders perform better, we found that fewer layers yielded a better performance for our data. We attribute these effects to the sparsity of the TF-IDF vectors and one-hot vectors. In comparison with permissions, description recovery through the autoencoder is significantly worse. We blame this on two aspects. The regression task, i.e., the linear output activation function necessary for the TF-IDF vectors, makes function approximation harder. The input variance is also higher for the descriptions. The variance stems from the very different subsets of words per description compared to all descriptions. Permissions are more likely to occur together compared to words, especially compared to rarer ones.

	Permissions Autoencoder	**Descriptions Autoencoder**
Layers (Encoder)	3	2
Hidden Neurons	210–196–5–196–210	1368–970–10–970–1368
Precision	97%	23%
Recall	93%	45%
F_1-score	95%	30%

Table 4.3: Best architectures and performances for the PAE and the DAE. Layer count does not include latent layer and input layer.

The autoencoders are trained with the complete set of apps. However, we only use a subset of all our apps for the clustering part for the sake of readability and computability. For the subsequent PCA and t-SNE algorithms, we use the 1169 apps in the dataset with over 5 million downloads. We take their embeddings and reduce their dimensionality to 2-d with both algorithms. We use the 2-d representations for descriptions and permissions of these top-downloaded apps to describe our observations.

4.4.1 Description Similarity

We start by comparing t-SNE with PCA for the descriptions. We assume that textually similar apps are put into the same category. Thus, we cluster descriptions based on their text and evaluate how the result corresponds to the category distribution. Since we have a total of 24 categories in our dataset, it makes sense highlight three of them for the sake of readability: *Weather*, *Communication*, and *Media and Video*. We process and compare our description embeddings with PCA and t-SNE and single out specific apps or app groups.

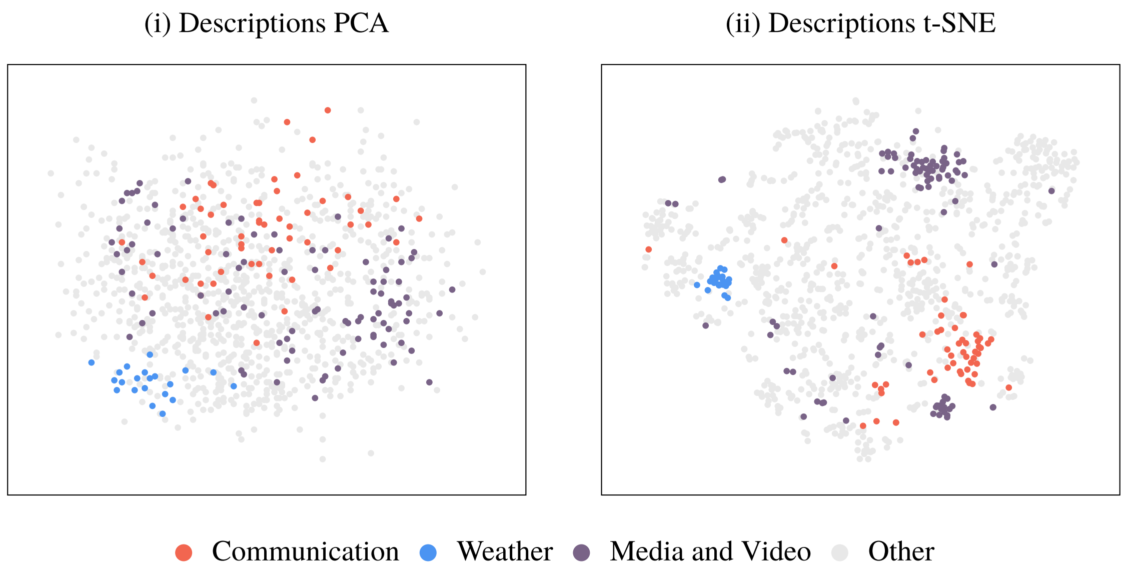


Figure 4.4: Embeddings of descriptions visualized with PCA and t-SNE. The highlighted data points show that t-SNE puts apps that are in the same category closer to each other.

First, we visualize the three categories, which is shown in Figure 4.4. Compared to PCA, t-SNE (Perplexity 45) does better at finding similar description texts and putting them in the vicinity of apps that belong to the same category. While *Weather* apps are already closely together with PCA, t-SNE clusters them even better, i.e., emphasizes the textual similarities. The other two categories have several hotspots all over the 2-d subspace of t-SNE, but are separated better than with PCA. In PCA, the *Communication* category and the *Media and Video* category are significantly more overlapping. In our observations, t-SNE is superior to PCA in clustering similar apps according to their store categories.

In addition to the visual evaluation for descriptions, we select two apps and look at their description neighbors. By calculating the Euclidean distance to all other apps of our transformed 2-d data and sorting them, we list the top-5 data points which are closest to the selected ones. We assume that similar apps should be placed in each other's vicinity.

Table 4.4 shows the description embedding neighborhood for two apps. We selected a social messenger and a web browser. The surrounding apps in their 2-d spaces are different for PCA and t-SNE. PCA finds apps that relate to multimedia for WhatsApp and productivity apps for Opera. t-SNE performs better at capturing the messenger- and browser-related properties in the descriptions, i.e., the combination of words and their different entropy within the text. t-SNE's top five neighbors of our two apps are all messengers and browser apps, respectively, while for PCA they are not. We emphasize that we do not use the app's title for our calculations, only the description text. Clustering description embeddings with t-SNE yields significantly better description similarity results than PCA.

WhatsApp Messenger		
#	**PCA**	**t-SNE**
1	Little Photo	Mercury Messenger (Free)
2	Mobo Video Player Pro	Yahoo Messenger Plug-in
3	Textgram - Instagram Text	Telegram
4	Photo Warp	Yahoo Messenger
5	Meme Generator Free	Azar-Video Chat&Call, Messenger

Opera Mobile Browser		
#	**PCA**	**t-SNE**
1	JuiceDefender - battery saver	Next Browser for Android
2	SmartWho Task Manager	Opera Mini – Fast web browser
3	Battery Widget	Boat Browser for Android
4	Desktop VisualizeR	Opera Mobile Classic
5	History Eraser - Privacy Clean	Maxthon Browser - Fast

Table 4.4: Top-5 neighbors for the PCA/t-SNE reduced **description** embeddings of two apps.

4.4.2 Permission Dissimilarity

Unlike finding similarities between descriptions embeddings, we want to find dissimilarities between apps based on their permissions instead. Thereby, we want to capture apps with divergent behavior. Apps that claim to be similar, but have significantly varying permission sets, are of interest for security reasons. Many permissions allow accessing privacy-critical information. A closer look at why a particular app requests the permissions when related apps do not is a useful starting point for further analyses. Hence, we use our t-SNE embeddings to analyze the permission distribution of two common app types.

As an example, we visualize all anti-malware apps and web browsers in our reduced dataset based on their permission embeddings. Figure 4.5 depicts the t-SNE reduction of our reduced dataset with perplexity 35. We also use the Euclidean distance as a similarity measure here.

Anti-Malware Apps

Although the 17 highlighted anti-malware apps appear mostly in one region in Figure 4.5 (i), there are three outliers. According to the visualization, *Antivirus Free*, *Antivirus for Android*, and *Dr. Web Lite* have fairly different permission sets. A closer look reveals that these three do not ask the user to give them

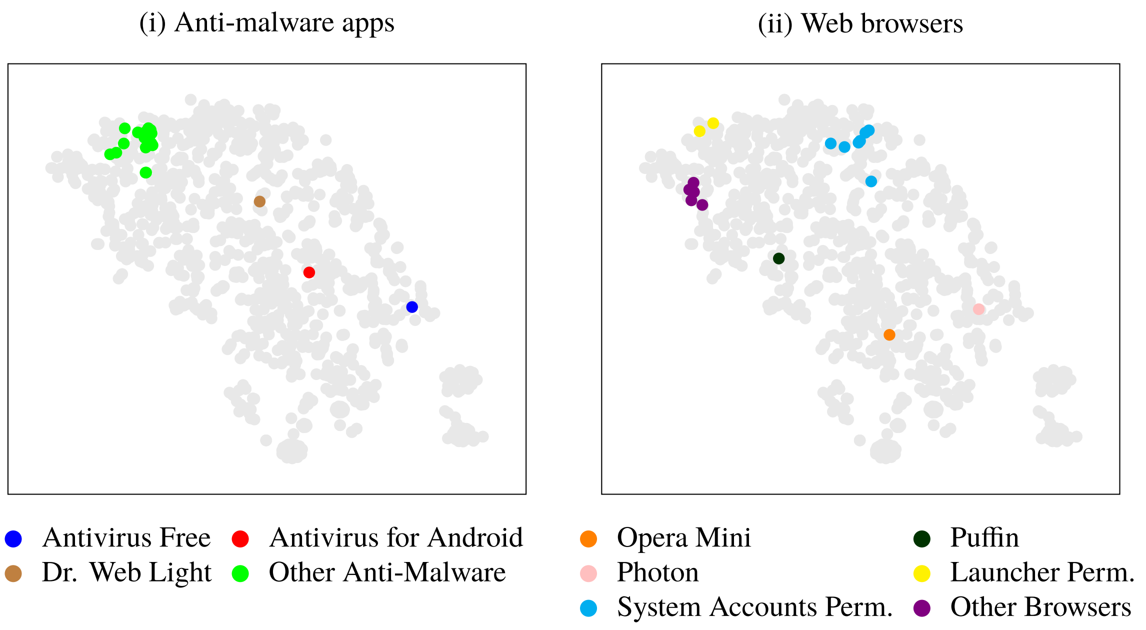


Figure 4.5: Permission embeddings dissimilarity: Highlighting of anti-malware apps and web browsers in the dataset (5M+ downloads). Outliers and clusters are useful starting points for in-depth analysis.

access to, e.g., location, call logs, SMS, or disk storage. The other anti-virus apps request between 33 and 81 single permissions, while the two outliers only request 5 to 14. Apps that aim to find malware on smartphones need access to critical, privacy-sensitive information to analyze suspicious behavior. For outliers that do not request such permissions, it stands to reason that they do not serve the purpose of identifying malware.

Web Browsers

Figure 4.5 (ii) focuses on 21 web browsers. Three isolated browsers appear in the plot: Opera Mini, Puffin, and Photon. Opera Mini is presented as a fast browser without unnecessary features. This claim seems to be correct in regard to permissions: the app only requests eight of them. Puffin and Photon are two browsers that serve mainly to run Adobe Flash scripts, but barely access other permission-critical resources. Thus, they are also positioned farther away from the other browsers. Two browsers stand out because of their launcher permissions, i.e., permissions for creating icons in the HTC app launcher or the Google app launcher. Apart from that, there are two larger groups. The blue cluster contains apps that are more intertwined with the operating system. These apps read and manage accounts stored directly on the device[5], e.g., a Google or Facebook account managed by the OS. Popular examples for this group are Firefox and Chrome. Apps in the remaining group (purple) are also full-fledged browser apps (location access, audio recording, system settings, etc.) that do not request the system accounts. Overall, our approach performs well in identifying related permissions that serve a similar purpose, like for launchers or system accounts, and positions them logically correct within the subspace.

Based on these examples, we conclude that autoencoder embeddings, in combination with t-SNE, are well-equipped to find similar and dissimilar applications for both descriptions and permissions. By selecting three categories in our reduced dataset, we showed that t-SNE is superior to PCA in clustering textually related description texts. In addition to the visual analysis, we listed the neighbors of two apps

[5]Android Permissions: `MANAGE_ACCOUNTS`, `GET_ACCOUNTS`, `USE_CREDENTIALS`

in our 2-d description subspace of both PCA and t-SNE that confirmed the better performance of the latter due to the unequal importance of words. In regard to permission embeddings, our goal was to identify characteristic behavior of anti-malware apps and web browsers. We succeeded in using visual outliers as a starting point for inspecting the permission sets further. Our evaluation indicates that the non-linear transformations by both an undercomplete autoencoder and t-SNE dimensionality reduction yield meaningful and improved results compared to linear transformations for app description and app permission analysis.

4.5 Summary

Over the previous pages, we have demonstrated our approach for clustering Android apps in regard to their descriptions and their permissions. The system started by preprocessing the textual data with TF-IDF and the permission sets with one-hot encoding. Then, undercomplete autoencoders were trained on the preprocessed data. The layer architecture was decided by random search. The latent-space representation of the descriptions and permissions was used to perform further dimensionality reduction via PCA and t-SNE. The resulting 2-d subspace was then ready to be used for visualizations and distance calculations.

We evaluated our approach with a real-world dataset to train the autoencoders, generate embeddings, and analyze the data. We calculated the 2-d subspaces for descriptions and permissions for around 1,000 apps. The description t-SNE transformation performs better than PCA in clustering apps with similar categories. We evaluated the neighbors of apps and found that the t-SNE description subspace also positions descriptions more closely together if looked at the actual use case, e.g., messaging apps and browsers. For permission embeddings, we selected anti-malware apps and web browsers and analyzed the resulting clusters and outliers. We found a meaningful correlation between cluster positions and permission-related functionality.

Based on our findings, we conclude that the non-linear subspace created by autoencoders and t-SNE clusters apps both via permissions and descriptions better than linear algorithms. Furthermore, we consider our clustering approach a useful tool for quickly identifying similarities and dissimilarities between Android apps. In regard to app publishing, the approach can be used, especially as a starting point for subsequent in-depth analysis of permission usage and description quality.

Chapter 5

Description Inference

The second chapter of this thesis deals with describing an app's purpose in an automated fashion. After giving motivating aspects for a description inference system, we explain the preprocessing of our three-model approach, using three different kinds of app information to infer description parts: resource identifiers, string constants, and API calls. Subsequently, we show the neural network model we use to infer description words. We also use a model explanation framework to find out which input features cause particular description words. After briefly listing details about the app samples we use, we give an extensive evaluation of our approach and conclude with a short chapter summary.

5.1 Motivation

Obtaining descriptions of an app's behavior in human language can have numerous applications. With the increasing complexity and number of smartphone apps, the difficulty of comprehending what an app does is also on the rise. The wide range of use cases, in combination with the easy installation procedure, calls for a solution that helps understanding app behavior. In this work, we focus on the human-readable, descriptive explanation of the behavior. We claim that both end-users and experts can benefit from such an app description inference system.

From a user's perspective, a simple app feature description can be useful. Straightforward installation procedures via app markets like Google Play promote users to install more third-party apps on their devices than ever. Gaining a glimpse into what functionality an app provides can be a valuable decision-making point on whether to install an app or not. App descriptions are among other meta information like screenshots, title, and reviews, crucial to help the user determine if they should download an app. This implies that a short, concise overview of the app's features can be especially beneficial for users that do not bother with reading longer description texts but still want to be reasonably informed about what an app does. Automatically obtaining short app explanations helps smartphone users to figure out whether the app does what they desire fast.

Conversely, app description inference can also help the other side, namely app publishers. As pointed out before, the description is a significant part of the app's store presence, which demands creators to draft texts that describe the behavior broadly. While there are different approaches to writing a descriptive text, we claim that a description inference system could be a helpful starting point. Many new, semi-professional apps are put on the store with comparably short texts or no texts at all. An automatic system can propose keywords for the description text that could either be accepted or adopted by the publishers, as they most likely do not want their app to be misrepresented. The suggestions could function as an incentive to write a better description. For these apps, the application of our system would significantly improve the description texts and therefore, also user satisfaction. Additionally, even for already drafted texts, automatic description inference could help to put one's app in a new perspective that makes the app better relatable. We found numerous store entries that missed keywords explicitly describing an app's core functionality, e.g., for wallpapers or comics. Non-mandatory assistance can improve Google Play descriptions in general, while still leaving its philosophy of openness untouched, in contrast to the iOS App Store and its rigorous vetting. It could help app marketing departments to write descriptions that are appealing, distinct, but still do not omit any crucial feature explanations. By providing an app summary that functions as a comparison, publishers could be motivated to improve their descriptions and to align them to similar apps better.

Automatically generating brief app explanations is thus a useful instrument in nowadays' fast-paced smartphone app ecosystems. On the one hand, users can benefit from them to make sounder decisions on whether to download and install an app or whether to avoid it. On the other hand, app publishers can use auto-generated phrases as a reference to improve description quality. Hence, we claim that a solution to infer app descriptions as a relevant contribution and extension to an app publishing assistance system.

5.2 Approach

In what follows, we outline our machine learning approach for creating a description inference system for Android apps. Initially, we describe measures undertaken to extract significant information that models app behavior. We focus on different information sources, coming from static elements and methods in the source code. First, we use the resource identifiers as model inputs. Second, we make use of the resource values (string constants). Third, we gather method calls to the Android API and use them as model inputs. Based on these three input entities, we use a dense neural network to infer tokens that should occur in the app's description. After model training, we add functionality to explain which resource identifiers, string constants, and API methods stimulate the prediction of a particular description token.

5.2.1 App Behavior Modeling

Before we can design a description inference system, we need to answer a fundamental question: which attributes of an app describe its behavior? A user's answer to this question would probably be installing the app and trying it out. A developer's take would likely be a thorough analysis of the source code. Both perspectives influence the strategy we undertake to infer app behavior. Our goals are extracting significant properties of an app that relate to its description while keeping the approach's computational complexity manageable.

As mentioned in the chapter for related work, valuable information for describing an app resides in its user interface. The average user would describe what an app does by what they see on the screen. Elements presented to the user may include media resources like images, videos, and sounds. This also implies that functionality running in the background goes unnoticed. Ultimately, however, displayed strings yield a comparably high amount of information. Consequently, we assume that text in the user interface, i.e., static string resources shipped with the app, can give reliable indicators about what must appear in an app's description text.

Understanding source code is a challenging task for computer algorithms. Programming languages do not have a linear structure like natural languages. Despite the progress in recent years in the complex field of source code analysis, we want to focus on straightforward ways to infer behavior based on the app's program logic. Our assumption for understanding source code is that we can describe a program by looking at how it interacts with the operating system and the user. In particular, we want to describe a program by finding relevant Android API calls, like accessing sensitive information, UI effects, or event listeners. The number of these calls, as well as their co-occurrence, should then explain an application's modus operandi to some extent.

Hence, we attempt to model Android app behavior relevant to its description from two different angles. On the one hand, string resources show what an app does from a user's perspective. On the other hand, we want to track a particular method calls on a source-code level in a simple way. Subsequently, we outline in detail how we extract, handle, and interpret this information.

5.2.2 System Overview

With the use of these app properties, we want to create three independent machine learning models that can infer the basic purpose of an app. First, we must preprocess these properties and represent them in a way the neural network can understand. Our goal is to predict words and word groups, which is why we also need to preprocess app descriptions. After that, we can start the training phase. The trained network is then used to infer descriptions of new, unseen apps. For them, we also want to explain which input features lead to a particular word prediction. In what follows, we give an overview of the whole description inference system.

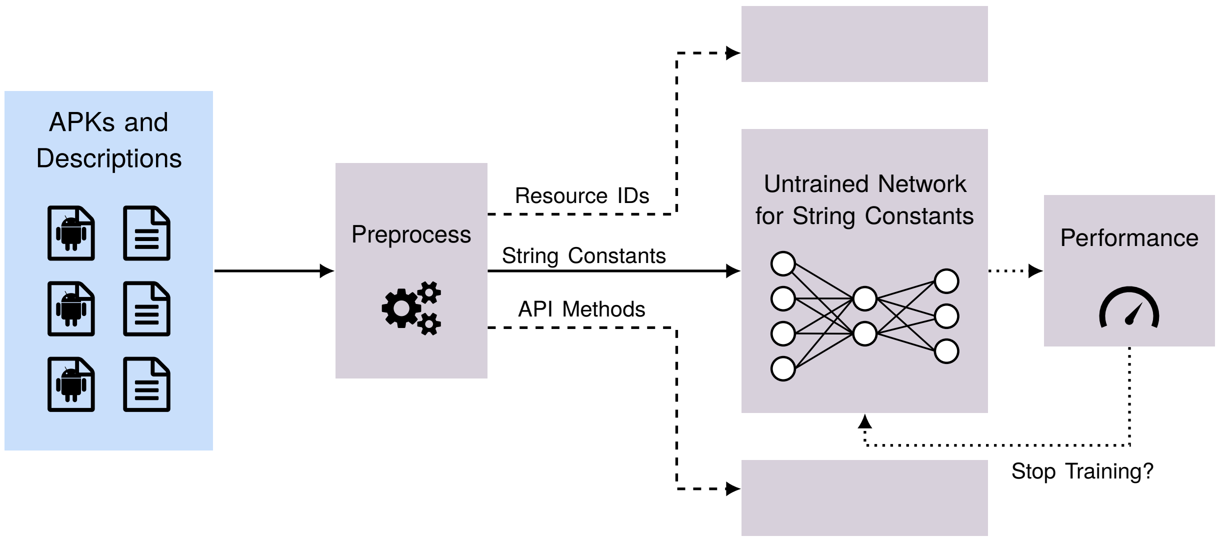


Figure 5.1: Description inference system: training phase. Three separate models are trained for the three app property types.

Figure 5.1 summarizes the parts of the system's training phase. For training, the APK files are preprocessed, yielding three different TF-IDF models: one for resource identifiers, one for string constants, and one for API methods. The descriptions are also represented as TF-IDF vectors. For each of the three input types, a separate dense neural network is trained. Training continues over several epochs up to until the performance plateaus. We stop then to reduce overfitting on the training set. Upon finishing the training, the sample prediction phase can start.

In the prediction phase, the three individually trained neural network models receive samples from the test set, as shown in Figure 5.2. These apps receive the same preprocessing as the other apps in the training phase, meaning that the previously created TF-IDF models are used to transform the resource identifiers, string constants, and API methods, and to decode the model output to readable words. So, each model outputs tokens with a particular decimal value, with a descending order accounts for their importance. For each output token, the model explanation framework SHAP is used to find the input features that contribute most to that token's prediction. Thus, during the prediction phase, the final models yield separate lists of tokens that describe the app and gives a list of influencing inputs for each of these tokens.

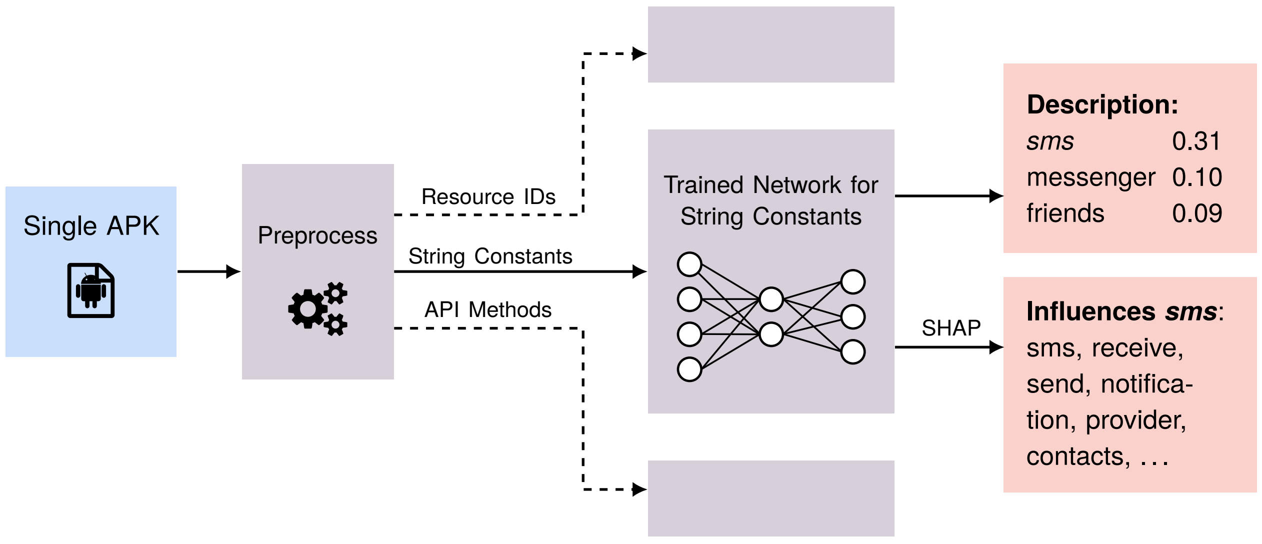


Figure 5.2: Description inference system: prediction phase. Each of the tree models can predict description tokens, with SHAP (model explanation) giving the input influences for these tokens.

5.2.3 Resource Identifiers and String Constants

The first preprocessing step deals with static resources in Android apps. Developers are encouraged to place user-interface related strings in separate XML files that facilitate, e.g., creating multi-lingual apps. Based on our assumption that these resources can explain an app's behavior, we extract them from each APK file. In particular, we focus on the identifiers, which are used as a reference, as well as on their assigned values. In what follows, we describe how we select and preprocess these resource identifiers and string constants.

There are numerous different kinds of resources saved in an APK file. For our approach, we leave out XML files that describe activities/fragments, animations, menus, and drawable. Instead, we solely focus on language strings, which include the following three file types: single strings, string arrays, and plurals. In these XML files, each node consists of at least a name tag, which we call resource identifiers, and one or more values, which we refer to as resource values. As the first step of our analysis, we parse these XML files and obtain a list of key/value pairs per app.

Resource Identifier Names

Resource identifiers are strings used in the program's logic to refer to separately stored static values. Unlike variable or function names in the source code, these are in general not obfuscated. The name assigned to a resource by the developer is maintained throughout the build and obfuscation process. The name, or identifier, usually reflects its purpose to some extent and can also give hints about the overall app.

Identifier
`pay_btn`
`select_image_dialog`
`confirmRemove`
`welcome_message_1`
`start_quiz_headline`

Table 5.1: Examples names for resource identifiers.

Table 5.1 shows examples of resource identifiers that could be used in practice. Like variable names, they can only consist of alphanumeric characters and underscores. Developers can assign arbitrary names and use random or uncommon, hardly interpretable resource identifiers. By randomly checking a plethora of apps, we found, however, that most apps contain resource identifiers made up of words or partial words that are either separated by underscores or formatted via camel-case. Hence, we need a tokenization strategy to split these into smaller, comparable entities so that a neural network can eventually work with them.

String Constants

By design, Android apps can show user interfaces in different languages. APKs then contain various strings for the same use. Resource names then potentially identify more than one corresponding value or set of values, whereby the exact resolution depends on the language setting on an OS-level. So in practice, the selection process of a particular string is handled by the OS.

We have to come up with a way to extract the desired strings by hand. For our system, we prefer English values. However, even for the English language setup, string constants can be defined in various sub-language files. For example, there might be phrases that are specific to Australian English (`en-au`), while others are the same for all English language regions (`en`), and particular words, like brands, can be language-independent (`default`). In the Android OS, this fallback-lookup creates an inheritance-like structure for language resources in Android apps. As a result, we need to come up with a strategy that is similar to the OS-approach to single out the most appropriate language variant from the APK.

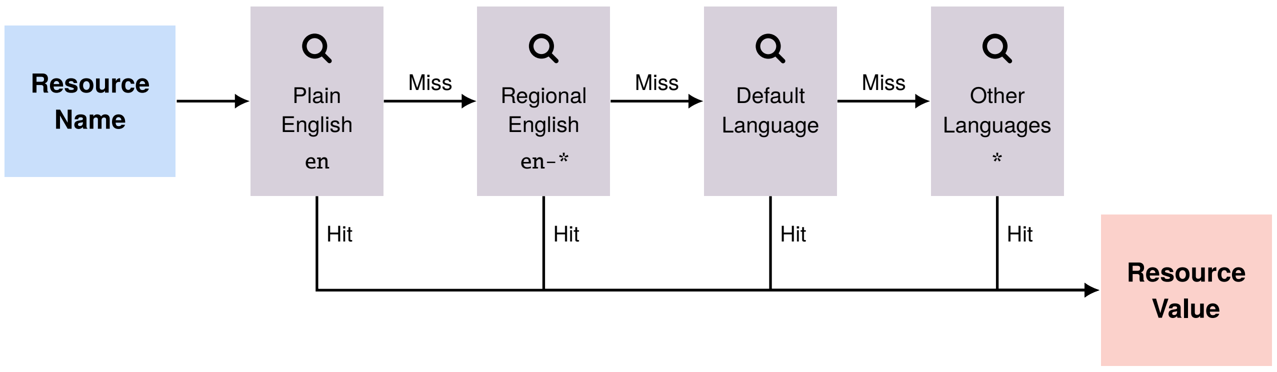


Figure 5.3: Strategy for resolving a localized resource value that occurs in multiple language XML files in order to obtain an English value. In case the default value or the fallback value in other language is futile, it is filtered out later.

The process for resolving resources is shown in Figure 5.3. The desired variant of a resource value is plain English, as we want to build our system based on frequently occurring words. Thus, we first search these files for matching identifiers, e.g., `values-en.xml`. If we cannot find the identifier in there, we check all other more specific English language files, e.g., `values-en-us.xml` or `values-en-gb.xml`. If we still have a miss, the default language file is scanned (e.g., `values.xml`). In the unlikely case that up to now there has been no hit, we check all other language files for the identifier and take the first value we can find. Our resolving strategy ensures that we obtain the most widely used words while ruling out that we miss a resource value, e.g., in case a developer misplaced it.

Apart from string constants stored in resources, we also want to include string constants in the source code. Parts of the app may not be language-dependent, but still contain strings useful for us in its code, like log output, URLs, or API parameters. Additionally, individual app developers may not outsource their string constants at all. If we would only cover resource files, we could not infer information about the behavior of such apps.

To obtain strings residing in the program code of an app, we extract them from the DEX file(s). The DEX format consists of several memory areas, whereby one of them itemizes all constants. The constant pool is thus an easy possibility to gather all app string constants. We add these code strings to the list of strings we already have from the resources. Then, we can start converting these string constants into tokens and count their frequency.

Tokenization

For the neural network architecture we intend to use, tokenization of string constants and resource identifiers is necessary. Unlike with text in natural language, these strings follow different patterns we need to address. We process string constants and resource identifiers so we can train our machine learning algorithm with them.

As mentioned above, resource identifiers are mostly understandable words separated by underscores or formatted in camel case. Their length and character set are limited. In comparison, string constants in resource files and source code files can have arbitrary length and contents. Examples for their purpose can be short phrases, URLs, hashes, or long HTML code. Apart from splitting all constants into small parts that still convey meaning, we also want to maintain URLs and Java packages and Java class names (including package names). We assume that standard app features can be derived from their frequent usage. To tokenize these strings, but also for the steps afterward, we have to take these properties into account.

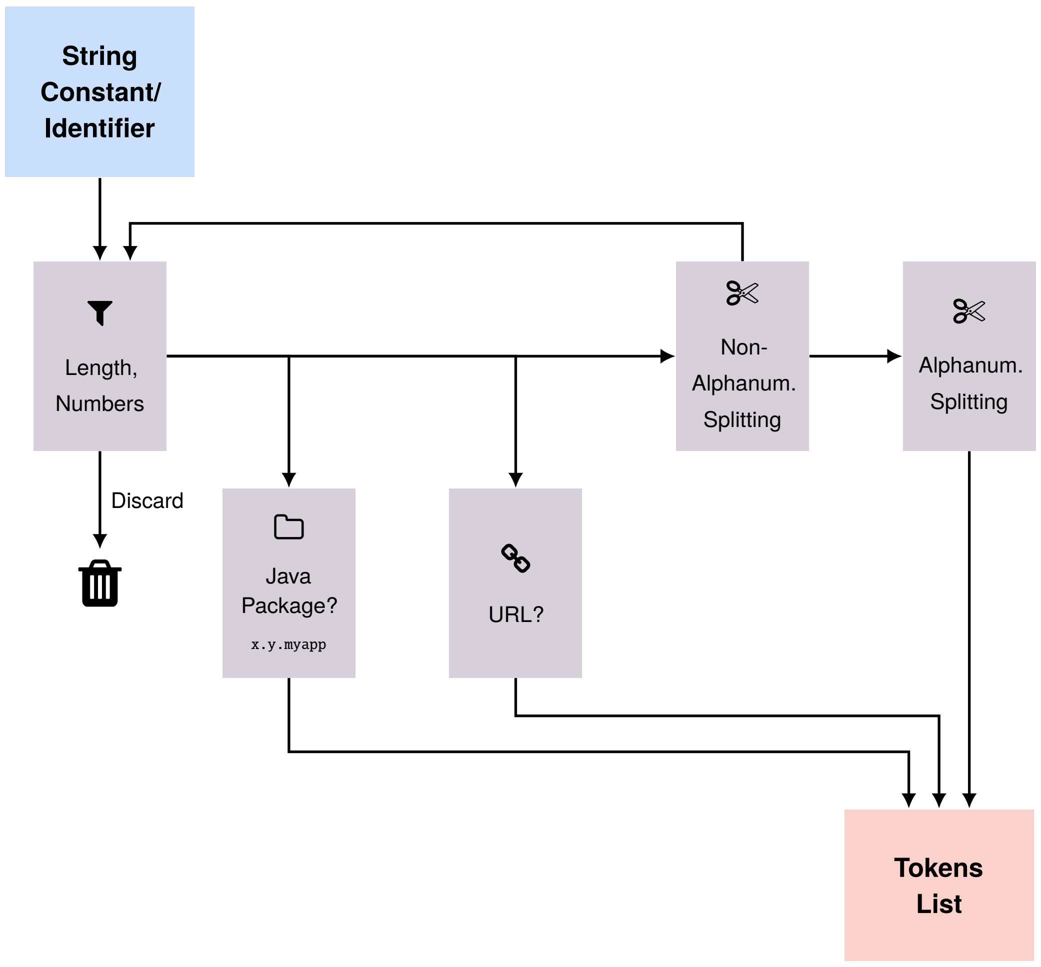


Figure 5.4: Tokenization approach for resource identifiers and string constants, which is the same for both.

Figure 5.4 illustrates the tokenization procedure for resource identifiers and string constants. We process them both the same way because the resulting tokens follow a similar pattern. Hence, the input for this tokenization procedure can either be a string constant (from the source code or an XML file) or a resource identifier. First, the input is checked in regard to length. Any sequence longer than 1,000 characters or shorter than one character is discarded.[1] Purely numeric inputs are also considered irrelevant. Next, in case the whole input is either a URL, a Java package name, or a class name including package name, we add them to the output tokens list. Either way, meaning for inputs that follow the previous pattern and for those that do not, we attempt to split the input by non-alphanumeric entities, e.g., whitespace characters, special characters like dots, HTML tag brackets, etc. These parts are treated like new inputs and processed again, the same way as the original input (e.g., for resulting intermediate tokens that are too short). In case the non-alphanumeric splitting yields only one element, we split the token further. Alphanumeric splitting breaks up camel-case notation and transforms all tokens to lower case. This procedure is thus capable of splitting the names of resource identifiers, their string values (constants), and the string constants in the source code into small, processable entities while also maintaining the URLs and Java packages/classes.

With this tokenization procedure, we obtain a list of tokens from both string constants and resource identifiers that are alphanumeric and can be processed further. While particular tokens are likely to be app-dependent, others are supposedly frequent across multiple apps. Therefore, we use the tokens and their occurrence count to build a TF-IDF model later and obtain a TF-IDF vector for each app. One property of TF-IDF is that rarely occurring and very frequently occurring tokens are ignored to maintain a reasonable dictionary size. We list the thresholds for all TF-IDF models later on.

5.2.4 Source Code Preprocessing

As for processing source code, our goal is to scan the app for API calls. In detail, we build a directed graph of the app's internal methods to find these calls. We also add implicit edges to this graph to better capture relevant parts. Based on the graph, we extract relevant exit nodes and count their occurrence. A list of method names combined with how often they are found in each app is the result of this preprocessing step.

Our search for Android API calls relies on each app's call graph. This directed graph expresses the method in the Android app as nodes and connects them via edges in the way that they are called. For methods that have a concrete Java implementation, i.e., lie in the codebase of the app, method calls are further resolved. The method calls that do not, like location retrieval or user-interface callbacks, the graph nodes occur as terminal nodes. Our goal is to find these terminal nodes.

As we explain subsequently, we need to add more edges to the call graph to access relevant code parts. Figure 5.5 shows the process of tracing API calls and how often they occur in the app. We take the code from the APK file and build a call graph based on the static, explicit code statements. We enrich the graph with additional edges: First, we resolve inheritance connections, and second, we resolve implicit flows given on a list provided by EdgeMiner. Then, we search for entry nodes of this graph defined in the app manifest file. After building this extended call graph, we can use find paths between nodes of API method calls and nodes of entry nodes.

Unlike conventional Java programs, Android apps do not have single entry points like the main function. Hence, we must find entry points that the operating system triggers. We focus on classes defined in the manifest that are Activities, Services, and Providers. These classes usually contain overridden methods like `onStart`, `onCreate`, `onBind`, etc. that appear as starting nodes in our directed graph. We start any search through the graph from these overridden methods.

Furthermore, we have to take class inheritance into account. Java is an object-oriented language that allows for type assignments during runtime. A static call graph cannot model this property. In the source

[1]Values are chosen empirically.

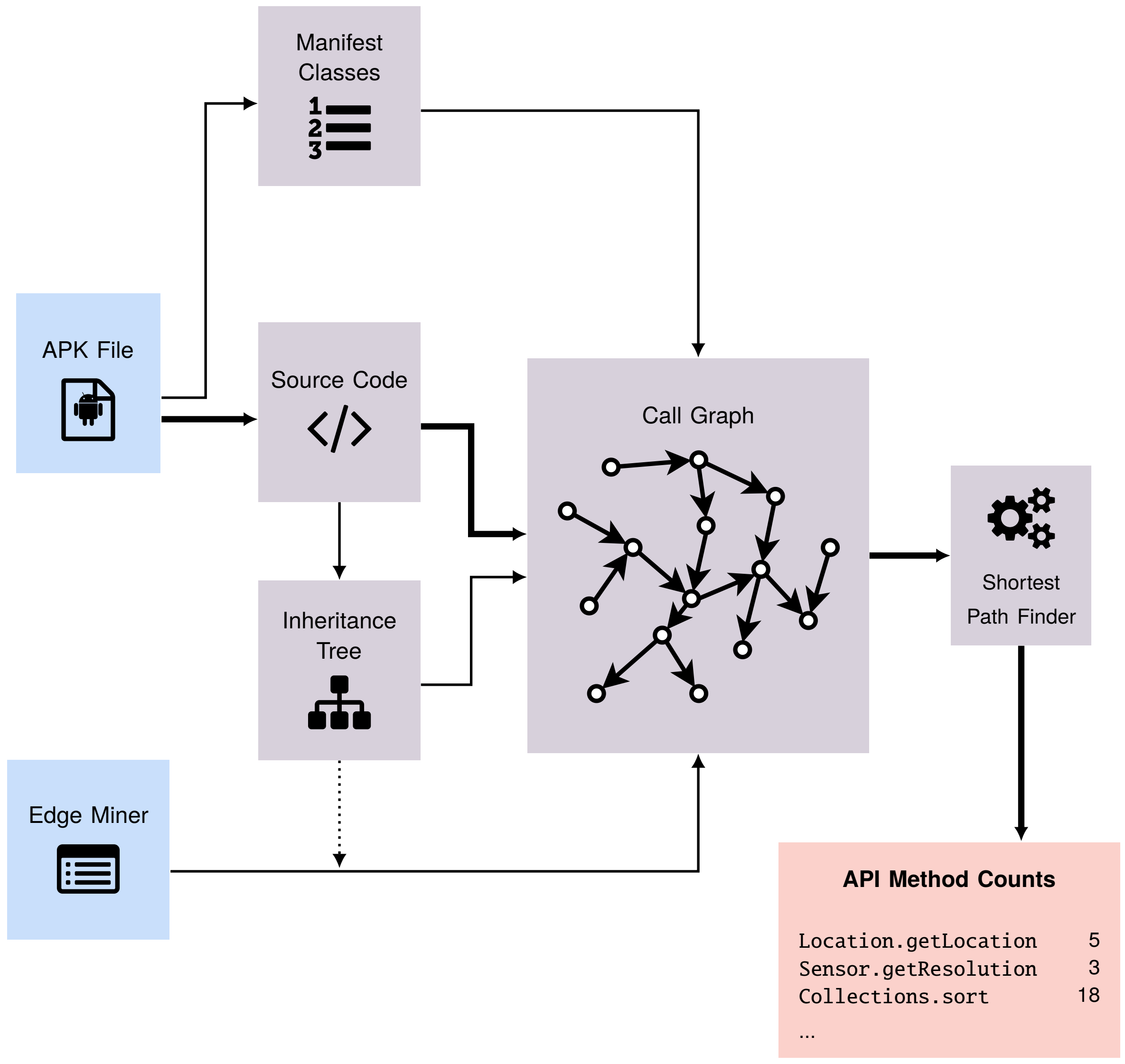


Figure 5.5: Preprocessing of Android app source code to receive API call occurrence. We create a call graph and add additional edges before we count all API methods, starting from the app's entry classes.

code, such calls are stated via the parent class' method, and the decision which implementation to use is made during runtime. There would be simply no edges between methods that are, in fact, called successively. An in-depth solution to this problem is out of scope for this work, as we would have to fully incorporate every app's control and data flow, including possible user inputs and actions. However, neglecting these linkages altogether would sharply reduce the app parts we can cover through graph traversal.

Thus, we strongly simplify the code so we can handle runtime type-assignments. First, we create a tree that keeps track of all the inheritances within one app. In case we have an object of class C, and a method is invoked on the object, we also invoke it for any class that derives from class C, given that the derived class has an implementation of that method.[2] While this, in theory, causes many false positives for single execution paths, we argue that in practice, these additional code parts have a strong tendency to be

[2]This approach is applied to both interfaces and abstract classes.

relevant for the whole app's behavior.

Furthermore, we have to approach the implicit edge problems. Certain execution paths are only possible in case the user or the operating system triggers it. A prominent example is the `OnClickListener`, where a custom class deriving from it implements an `onClick` method that is only called when a user clicks the associated UI element. While the nodes for the registration of this listener and the implemented `onClick` method exist in our call graph, there is no edge between them.

To add many implicit edges to our graph, we use EdgeMiner [12], which aims to close this gap for Android source code analysis. Their project finds implicit edges in the control flow of programs, both Android-specific and in Java. We use the list of method calls that are linked provided by the authors. For these nodes, we also resolve the inheritance of app-specific method calls, as EdgeMiner only lists Android API and Java functions. Then, we add edges between these additional nodes to receive the final version of our call graph that we can start processing.

After adding all these additional edges between the nodes, we can start the path search. Our goal is to count execution paths, i.e., connections, between entry methods $E_{in,j}$ and API call methods $E_{out,k}$. Thus, we use the Dijkstra algorithm to check whether a link between two nodes in a graph exists. For each node $E_{in,j}$ we check whether there is a connection to $E_{out,k}$. If so, we increment a counter for API call k. We count all methods as a combination of their class (fully qualified) and their method name, but without their arguments and return type. After iterating over all possible combinations, we receive a list of API call counts.

Like with the resource identifiers and string constants, we also create a TF-IDF model for the API methods. For each app, we now have a list of pairs. Each pair consists of a method call and how often it was found in the app's source code. By using the TF-IDF algorithm, we decide which method names end up in the dictionary, based on their frequency. TF-IDF then transforms this information and eventually obtains a 1-dimensional, decimal vector for each app sample that we can use as neural network input.

5.2.5 Description Preprocessing

As the output of our machine learning model, we want to infer feature-related parts of the app description. We opt for TF-IDF representation of the description, similar to the description clustering approach in Chapter 4. However, we expand it to cover not only single words but also short phrases. Besides that, we also improve the human readability of stemmed tokens. The outcome serves as the network output for three models, i.e., the training target per app.

In the clustering chapter, we calculated the TF-IDF model solely on single words, i.e., tokens. Because of treating description texts like bag-of-words, lots of relevant information is lost. For example, we cannot capture concrete phrases and their word order, like *take photo* or *SD card*, as they would be split into separate tokens. This circumstance was not that significant for app similarity and dissimilarity, but with the description inference task, the predicted outputs are supposed to be meaningful entities.

As a consequence, we generate n-grams from the description texts. Intuitively, we use $n = (1, 2, 3)$ to cover phrases that include one, two, and three words. Due to stopword removal and Porter stemming, we reduce the number of frequent word combinations that carry comparably few meaning beforehand, e.g., *take some photos* and *take a photo* both become *take photo*. An important aspect here is setting minimum and maximum document frequency thresholds in the TF-IDF algorithm. That ensures that the number of resulting dictionary entries, i.e., single tokens, 2-token combinations, and 3-token combinations, does not explode and the subsequent computations remain feasible. By stemming tokens, removing stopwords, and windowing with three different window sizes, we aim to capture more meaning.

The tokens, regardless of whether the model finds frequent single occurrences or combinations, are stored in their stemmed form. Stemming removes parts of the word to subsume multiple word variations and facilitate computation. It often does not, however, reduce the words to a stem that can be easily read by humans. Since the stemming transformation is not a bidirectional transformation due to the loss of information, an accurate *un-stemming* method cannot exist. Stemmed tokens contrast our goal to provide human-readable description fragments.

As a solution, we use a greedy algorithm to recover original words from their stem. Therefore, we keep track of all the stemming transformations, i.e., whenever a token T is altered and results in its stemmed version $\widehat{T} = f_{\text{stem}}(T)$. The suffix removal of stemming leads to $\widehat{T}$ having multiple corresponding original tokens T, so we collect the number of times of: $T \to \widehat{T}$. After processing all descriptions, we have obtained an association count table that lists how often $\widehat{T}$ was caused by each original T. Two examples are given in Table 5.2. For example, if the stemming result of a token is *locat*, it is then replaced with *location*, regardless whether the original one was *location*, *located*, *locating*, or *locate*. By counting how often an original token results in a particular stemmed token, we can replace the stemmed token by its most common origin.

$\widehat{T}$	T	#
locat	**location**	**890**
	located	418
	locating	356
	locate	208
effect	**effects**	**286**
	effect	109
	effecting	75

Table 5.2: Example for a stemming association count table. *locat* would be re-transformed to *location*.

Ultimately, each description is represented by a list of tokens that we want to transform via a TF-IDF model. The TF-IDF model has features that consist of single tokens, 2-grams, and 3-grams. The stemmed tokens are re-transformed to their most likely original, non-stemmed word to be more easily readable afterward. Hence, for each app, we here also obtain a 1-d vector with normalized, decimal numbers between 0 and 1 for each description, which we can subsequently use as machine learning targets.

5.2.6 Network Model

As we have now processed both input and output data, we can continue to the neural network architecture and the training process. We create three similar models that have the same output, descriptions but receive either resource identifiers, string constants, or method names as inputs. Following preprocessing, these three inputs and the output are represented non-positional, decimal values for which we can build a dense neural network regression model. Subsequently, we outline our design decisions for network and training.

Figure 5.6 shows the dense network architecture we use for all three input types. The bag-of-words representation via TF-IDF vectors enforces positional constraints, i.e., the value for a particular word is always put into the same vector cell. This input property allows us to use a standard dense network structure in comparison to other convolutional or recurrent architectures that factor in positional and sequential information. Between the dense layers, we apply a dropout for regularization. Since the output of each neuron consists of floating-point numbers, we have a regression task and use a linear activation function for the output layer and mean-squared error as a loss function. All hyperparameters are listed in Table 5.3.

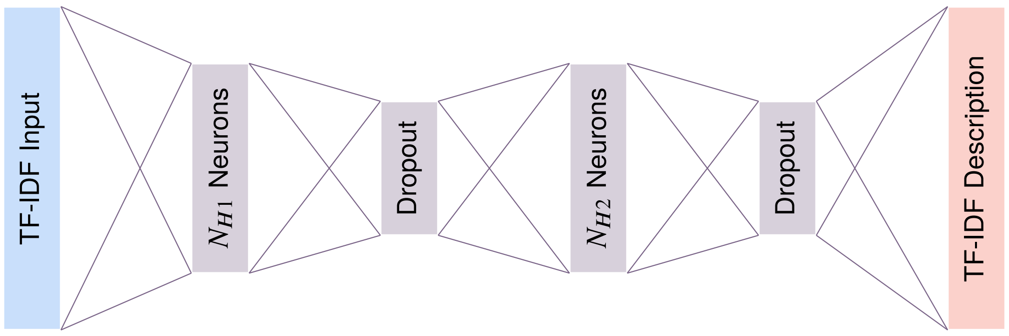


Figure 5.6: Feed-forward dense neural architecture that either receives TF-IDF vectors of resource identifiers, string constants, or method calls. The number of hidden layers and their neuron counts varies for the three different input types, whereby a dropout layer follows a hidden layer.

Optimizer	Adam, $\eta = 0.0001$
Batch Size	32
Weight Initialization	Glorot (Uniform)
Loss Function	Mean-Squared Error
Hidden Activations	ReLU
Final Activations	Linear
Early Stopping	Patience: 6 epochs
	Delta: 0.5% $F_{0.5}$-score

Table 5.3: Neural network hyperparameters for the description inference model.

The size of the input and the output depends on the dictionary size of the TF-IDF models. In particular, the dictionary sizes are limited by the minimum document frequency and the maximum document frequency, e.g., in how many apps a certain method call or resource identifier token occurs at all. Especially the lower boundary is crucial. If the minimum document frequency is too high, we miss information the neural network could use to infer the output more precisely. In case the minimum is too low, the model remembers too many tokens that rarely occur, and the dictionary becomes very large. The larger the input space, the more inputs and weight parameters are stored in memory, and the longer the training process takes. The selection of TF-IDF model parameters thus binds the training process. Additionally, we used random search similar to the previous chapter to find suitable network architectures. For this non-exhaustive search, we trained networks with one to three hidden layers, having 1,000 to 15,000 hidden neurons. Dropout was randomly set between 0% and 40%.

We choose the parameters empirically by trying out different setups and observing the resulting dictionary sizes and model performances. Thereupon we set the minimum document frequency for each of the three input types to 2% of the total number of documents (apps), and the corresponding maximum frequency to 20%. This range means, e.g., if we have an app dataset of size N and a token occurs n times, it only ends up in the dictionary if $0.02N \leq n \leq 0.2N$ apps to be used in the dictionary. Table 5.4 lists the networks and TF-IDF dictionary sizes we found to give good results. We describe the results in detail later in the evaluation part in Section 5.4.

	Input TF-IDF # Features	**Network Hidden Layers**	**Description TF-IDF # Features**
Resource Identifiers	3315	2968, 3265, 1393 (3 Layers)	6140
String Constants	6391	2898, 3105 (2 Layers)	6140
API Methods	11735	5891 (1 Layer)	6140

Table 5.4: Neural network configurations of the three models, including the size of the TF-IDF dictionaries' models.

5.2.7 Training

The training process resembles the one in the previous chapter, but with minor adaptations. Like before, we again split the dataset into different sets, i.e., training, validation, and test, and use a parameterized F-score for monitoring. We also use early stopping here. For the discretization of TF-IDF values, however, we use a different threshold.

The three models are trained using the mean-squared error measure as a loss function. As a performance metric, however, it is not fit for the purpose. It could be used to compare different models and runs but does not give any intuitive expression of how well the model performs. Thus, similar to Section 4.2.3, we discretize the description TF-IDF vectors by choosing a threshold θ, above which we set the vector element to 1, and 0 otherwise. We can then use standard performance metrics like F-score, precision, and recall on these binary vectors for actual description's vector and description prediction's vector.

For this task, we also want to factor in the actual model's certainty for predictions. As we show later in the evaluation, for uncertain predictions, outputs a particular state. We call this state prediction *baseline*, which could also be viewed as noise floor. It stems from model input that is insufficient to infer any clear output. We use the highest value of such a baseline prediction as a basis for θ. This way, the discrete classification metrics do not include too much noise.

One specific property of this task is that we likely cannot recover large parts of the description. The correlation-based learning approach tries to find similarities between apps and thus neglects app-specific terms in the description. From a performance point of view, this means that we can expect a lower recall than precision. For early stopping, we are required to choose a pivotal performance metric that measures whether training should stop or continue. We thus use a weighted F-score ($\beta = 0.5$) that rates precision higher than recall. Consequently, we use the $F_{0.5}$ performance for early stopping to find a good final training state.

5.2.8 Prediction Explanation

Once we trained our final three machine learning models, each one of them predicts a list of description words based on their whole input. Apart from seeing this result, we also want to know which description word predictions are caused by which input items. For this reason, we apply SHAP to our network to explain model predictions.

For example, if our resource identifier model outputs the word *dictionary*, we want to find the influences of these predictions. A reasonable, for humans understandable relation would be input tokens like *search*, *word*, or *translate*. If input tokens show up that do not make sense, the model has learned that correlation from similar apps but not regarding a particular app feature. For example, a developer could reuse parts of an app in multiple others while mentioning the same word in all or most of their description texts. Especially in our case, where we deal with real-world, noisy datasets, explanations where the model prediction stems from, are essential.

To find sample-based input-output causes, we use the Deep SHAP explainer, as part of the SHAP framework.[3] The explanation works on a sample-basis, meaning it has to be repeated for each new sample again. For an app sample, the explanation algorithm returns a Shapley value for every input feature, for every output. The Shapley values give relative information about the significance of one input feature for a particular model output neuron. Therefore, it receives similar inputs and uses differences between them to calculate overall influences. This method aims to find the influence of a certain input feature on one sample's output. Deep SHAP by default does this by incorporating the internal structure of a given neural network, calculating partial Shapley values and adding them linearly. Eventually, we obtain a cascaded list of Shapley values. The neural network itself gives a list of description tokens. For each predicted description token, there's a list of Shapley values that correspond to the input features. Each list can then be sorted to see the most significant input influences per predicted description token.

The neural network has an ample output space, with many predictions being close to but not exactly zero. The output space comes from each TF-IDF description model's dictionary, which has several thousand entries. Deep SHAP tries to find the influence of all values that are not zero, even the ones that are insignificant. Practically, we do not need to evaluate predicted output words which have rather low value, as they are most likely noise. Thus, we instruct Deep SHAP to only look at the input and output values that are relevant, which we determine via the selected θ threshold (Section 5.2.7, Training). By that, we significantly reduce the execution time.

To sum up the model explanation, we calculate the Shapley values for relevant, predicted description tokens via Deep SHAP. This approach is the same for resource identifiers, string values, and method calls. We sort in descending order and receive a list of the top influencing input values that explain the model's prediction for that sample.

5.3 Dataset

Upon setting up the preprocessing pipeline and the three neural networks, we can train and evaluate them on a dataset. Like in the other two main chapters, we use the PlayDrone project to obtain app samples from Google Play. Subsequently, we briefly describe which samples we use and their splitting into subsets.

For the description inference part, we rely on Android–specific development properties. Our system cannot handle cross–platform implementation. A large part of these apps' logic is not stored in bytecode or APK-specific resource files. The description inference system could thus not learn from these apps or make predictions for them. We ignore such apps during training and prediction based on the cross–platform app detection approach in Section 4.3.

The remaining apps are split into training, validation, and test set. We design the test set so that it only includes descriptions with a reasonably good description. We make the simplifying assumption that apps with a higher download count tend to have higher description quality. For the test set, we thus take samples from these comparably popular apps. We sort all apps in the whole dataset by download count and take every third app until we obtain 1,000 test samples. The remaining samples are split into a training set and a validation set, where 20% is used for validation and 80% for training. Additionally, we check each sample if the resulting TF-IDF vectors for input and output contains more than non-zero values. This way, we avoid training on empty samples. Table 5.5 shows the exact number of apps we use in the training and prediction phase that follows from these criteria.

[3]`https://github.com/slundberg/shap`

	Resource Identifiers	String Constants	API Calls
Training set	33,490	54,369	53,560
Validation set (20%)	8,381	13,608	13,405
Test set	804	937	914

Table 5.5: App dataset properties.

5.4 Evaluation

This section deals with the sample-based evaluation of our description inference system. On the one hand, we do this by looking at performance metrics, and on the other hand, we evaluate single samples. First, we look at the performance metrics of the discretized TF-IDF description vectors and compare the three input types. Second, we demonstrate description predictions as word clouds. Ultimately, we show SHAP explanations for some samples. In what follows, we show our findings and draw conclusions based on a large spectrum of apps.

Table 5.6 lists the numeric performance results of our three models. The direct comparison of the F-scores shows that the resource identifiers yield the best results, while API calls perform significantly worse, with string constants in between. Especially the recall value is comparably low. We attribute this mainly to two reasons. First, descriptions contain lots of words specific to the app that are hard to generalize. This makes reconstructing many of these rarely occurring words difficult. Second, the TF-IDF model for the description output does not take synonyms into account. For example, if the description contains the word *image*, but the word *photo* is predicted, it counts as a mismatch and lowers the recall despite the semantic correctness. Although the table lists these low-performance values, the models yield good results as we show subsequently.

	Resource Identifiers	String Constants	API Calls
Precision	56%	64%	51%
Recall	12%	6%	6%
$F_{0.5}$-Score	33%	22%	20%

Table 5.6: Performance on the test set of the three input types via discretized TF-IDF vectors. Discretization threshold: $\theta = 0.04$.

Figure 5.7 demonstrates the illustration of our results via word clouds, sometimes also referred to as tag clouds. It shows the top-ranked predictions of the three models for the app **Vevo**, a popular video streaming platform similar to YouTube. A word cloud illustrates a limited number of words by setting the font sizes of the tokens to weigh them. Given a list of tokens and numeric values, each token's font size is chosen relative to the other values. The position of a token is chosen randomly and has no meaning. In the figure, for each model, the table lists the top-8 tokens and their predicted values. The tokens with lower values are excluded. We employ word clouds to obtain a representation of our result that is easy to understand for humans.

As we work with n-grams as well as with single tokens, we combine some of them for a clearer illustration. In case the model predicts not only the two tokens T_i and T_j, but also their combination as a 2-gram (T_i, T_j) (any order), we prefer the n-gram. Hence, we do not display the single tokens T_i and

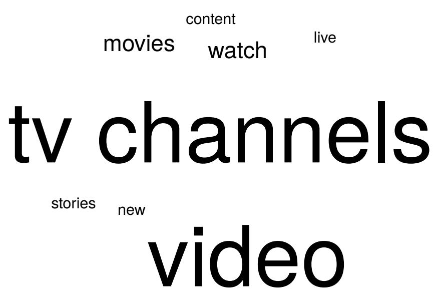
(i) Resource Identifiers

Value	Description Token
0.295	video
0.261	tv show
0.227	kids
0.148	music
0.123	watch
0.081	songs
0.066	subscription
0.064	content

content
music
songs
subscription
kids
video
watch
tv show

(ii) String Constants

Value	Description Token
0.136	tv channels
0.132	video
0.041	watch
0.033	movies
0.025	new
0.024	content
0.024	live
0.024	stories



(iii) API Calls

Value	Description Token
0.131	video
0.098	watch
0.094	tv
0.062	news
0.056	live
0.049	movies
0.046	sports
0.045	us

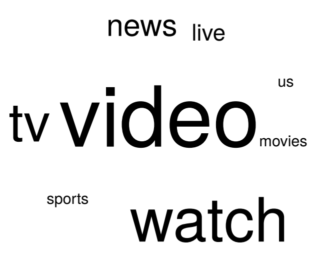
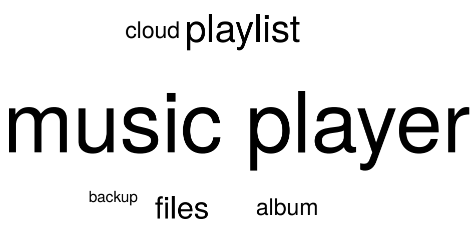


Figure 5.7: Word cloud representation for each model's prediction for the app *Vevo*.

T_j. Regarding their weights, we take the maximum of the accumulated singular weights and n-gram: $\widehat{w}_{i,j} = \max\left[w_i + w_j;\ w_{i,j}\right]$. For 3-grams, we take the maximum of the three single weights added up and the 3-gram's weight. The combination of tokens and n-grams reduces the number of top tokens without lowering the word cloud's expressiveness.

Concerning the Vevo app, the three models predict expressive words about what this app is about. Apart from *video* being top-ranked, the nature of a video streaming and sharing platform is expressed through the phrases *tv show* / *tv channels*, *movies*, *subscription*, *music*, *live* and *content*. The tv-related phrases show that the models cannot differentiate between traditional television and online video streaming. Because these predictions stem from many other apps, we reason that the neural networks understand the domain of the input and learn to cluster video-related applications internally. We also see that the inferred tokens based on API calls are much more general. The overall domain of the app becomes clear, but, e.g., no n-grams like *tv channels* or *tv show* are learned. All in all, despite their independence, the three models yield descriptive information about a potential description.

(i) Suggested Words



(ii) Actual Description

4shared Music was created for those, who can't live without music and don't want their attention to be attracted with anything else, but music while listening to it. When you install this app on your Android device, you may easily access your favorite tracks from 4shared. Using your 'Search' menu item you can look for music files you like and add them to your playlist at 4shared Music. You don't have to look for them each time you use the app, just create a playlist, add your favorite tracks and listen to them exactly from your 4shared Music anytime you want. Moreover, you can upload tracks from your Android device to your 4shared Music. With 4shared Music you can enjoy 15GB of space for your music and nothing out of place. Upload and add all music files you like and make your life even more enjoyable with 4shared!

Figure 5.8: A comparison between the actual description of *4shared Music* and our system's prediction shows that despite the accurate representation of the app's purpose, the performance is hard to measure. Many words do not exist in the text literally.

The following example demonstrates the performance metric shortcomings we mentioned above. Figure 5.8 deals with the app **4shared Music**. 4shared[4] is a cloud storage provider, similar to Dropbox. This app is a music player that accesses audio files on their platform. The description inference predicts, based on resource identifiers, that the app is a *music player*, dealing with *playlists* and *albums*. The neural network also captures the second domain of this app, the online storage platform, via *cloud*, *backup*, and *files*. While all these predictions make sense, the description does not mention all of them, e.g., *player*, *cloud*, and *backup* are absent. In other words, since the description text does not cover the tokens literally, the measurable performance decreases despite the good generalization. An accurate but abstracted word cloud that is intelligible to humans is thus difficult to measure.

Apart from giving a list of words, it is also important to estimate the confidence of a prediction. The manual examination of several apps shows that uncertain model prediction could be easily identified. Table 5.7 shows the top values for such a case. As the model cannot find any inputs it can make sense of, it outputs very generic words. The numeric values at the output also have low variance. We described this

[4]`https://m.4shared.com/`

state in Section 5.2.7 as a baseline model, which can be used to differentiate between certain and uncertain predictions. Since the highest value here is 0.024, we used a slightly higher value of $\theta = 0.040$ for the discretization we need for performance measuring.

Value	Description Token	Value	Description Token
0.024	application	0.022	photo
0.019	com	0.017	android
0.017	phone	0.016	features
0.016	device	0.016	share
0.015	images	0.015	video
0.015	new	0.015	free
0.014	mobile	0.014	contact
0.014	time	0.014	set
0.013	information	0.013	map

Table 5.7: Uncertain model prediction: When the neural network cannot interpret a certain app, it outputs generic words with a relatively low numeric value.

The predictions based on resource identifiers and string constants can have different quality, as also shown for the app **365Scores** app in Figure 5.9. This app provides features for fans of different sports, teams, and leagues, like checking past results and statistics, live standings, or video sequences. The resulting word clouds show that although the inference via string constants roughly explains what the app does, the one via resource identifiers does a better job. While the former shows functionality related to a social messenger, the latter one more accurately paints a picture of the actual purpose and the app features. Apart from the quantitative performance comparison in Table 5.6, we could also confirm this tendency through our qualitative analysis. We attribute this tendency to the more precisely worded tokens in resource identifiers that overall have lower variance than the ones in string constants, which are in general longer and use more creative language. It depends on the app, which of the three neural models performs better, but in general, we observed a higher accuracy for the inference based on resource identifiers.

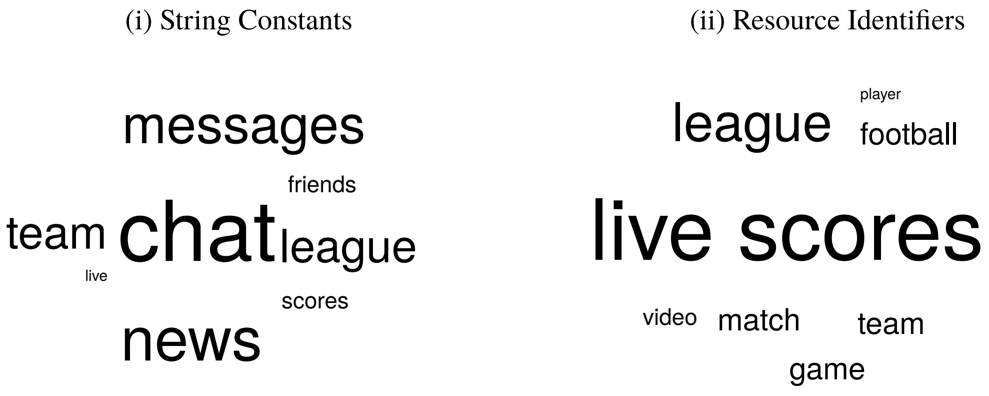


Figure 5.9: Word cloud representation based on the prediction for string constants and resource identifiers for the app *365Scores* for sports fans, whereby the latter one is more accurate.

Figure 5.10 shows the system's output for **360 Security–Antivirus FREE**. Resource identifiers and string constants show meaningful results that make it easily identifiable for humans what the app does. In contrast, the inference via API calls does not give much information. Merely the *location* token could remotely describe the app's feature to find the device. The comparably worse performance for API calls is a behavior we commonly observed across many apps. For complex behavior, the mere occurrence count of API calls via TF-IDF seems to be insufficient, especially with our noisy training data. For certain app types, however, we found that the methods-based explanation generation performed significantly better than for others. Examples are wallpapers, which often have a small code base, and themes or add-ons for other apps that resemble each other. We still found, however, that the information provided by the API calls neural network in addition to the other two models can be used for many apps.

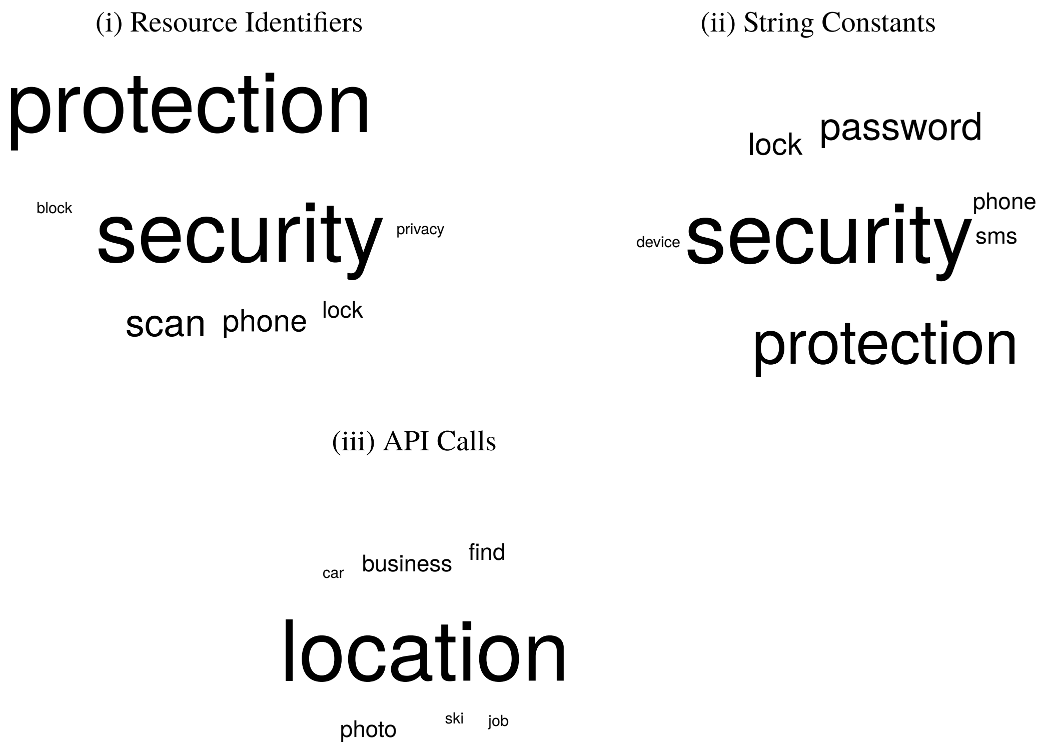


Figure 5.10: For the *360 Security–Antivirus FREE* here, and for many other samples, inference via API calls is comparably difficult for the system.

Next, we also want to evaluate input-output relations. We thus take one sample and get the top prediction for it, i.e., we focus on the network's output with the highest numeric value. Then, we calculate the SHAP values for all inputs and list the corresponding input features and their SHAP values. Table 5.8 shows this result for the app **Slacker Radio**. The predicted description words with the highest values were *music player* for resource identifiers and *music* for string constants. By looking at the top input influences, we can see that the two different network models make their decision based on reasonable inputs. These input tokens affect the output in a way that is easily comprehensible and verifiable by humans. We noticed that for many samples, the information gained from the model explanation works very well for predictions based on resource identifiers and string constants.

In contrast, however, the workings of the neural network for API calls are hard to explain with SHAP. Although we found many apps where the method-based model gives good descriptions, the SHAP values were hard for humans to interpret. For example, the Vevo app above (Figure 5.7) has *video* as its top

(i) Resource Identifiers

Description Token(s)	Input Tokens	SHAP
music player	artist	0.0122
	album	0.0107
	playlist	0.0071
	art	0.0034
	lyrics	0.0032

(ii) String Constants

Description Token(s)	Input Tokens	SHAP
music	playlist	0.0321
	song	0.0230
	stations	0.0125
	songs	0.0096
	tracks	0.0039

Table 5.8: SHAP model explanation: Applying the algorithm for the app *Slacker Radio* on particular description words returns meaningful input tokens that caused the prediction.

predicted term. The top SHAP values for it return Android activity manipulation methods unspecific to multimedia applications. We assume that in cases like these, library implementations tend to consist of a particular set of methods, working as a fingerprint for video-related purposes. In some cases, however, the SHAP explanations also give meaningful insights here, e.g., for the predicted description word *shake*, we could trace the `SensorManager` class among the top input features that caused the prediction. The API calls model consists of a very high number of input features that the network wires in a way that in most cases, is difficult to understand for humans.

Our evaluation shows that although it is difficult to measure if a predicted description accurately depicts the app's purpose, the description inference system learned plausible relations during training. Based on several sample apps from the test set, we can conclude that the predicted descriptive fragments are very expressive in many cases. Their accuracy and plausibility vary between the three input types, with the inference based on resources identifiers leading to the best results. For API calls, it is hard for humans to comprehend the model's decisions, while for resource identifiers and string constants, our qualitative analysis showed meaningful correlations. All told, the three neural networks, in general, yield plausible descriptions for most apps in our test set.

5.5 Summary

The second contribution of this thesis presented in this chapter was a system that comes up with a rough textual portrait of an Android app. We began by outlining our approach, including preprocessing APK files, neural network training, and explaining the predictions made. The approach was evaluated on a real-world app set to demonstrate its practical application. In what follows, we briefly summarize this chapter.

For the inference of app description phrases, we extracted three types of information from the APKs: The identifiers of static resources (XML files), the values of string constants in both XML files and the source code, and the method calls made to Android APIs and Java classes. Identifiers and string constants were split into alphanumeric tokens, and method calls were referred to with their method signature, including class/package. Each of these three entity types was represented via TF-IDF, meaning we considered how often one element occurs per app and how often it occurs across all apps. Based on these three resulting one-dimensional vectors, we trained three different dense neural networks. Their targets were TF-IDF-modeled descriptions. The output could thus be interpreted as a list of descriptive phrases. The entries of this list sorted in descending order gave the most relevant phrases. For these top outputs, we used a model explanation algorithm to interpret which inputs caused them. Our system thus used static and code resources of an app to infer significant words of the description.

Afterward, we evaluated the system based on overall measured performance as well as by visualizing and examining the result for particular apps. First, we compared the performance of the three different machine learning models with their tree different input entity types. We saw that inferring description phrases work best via resources identifiers, followed by string constants. The inference via API calls performed worst. Our individual evaluations of apps using word clouds emphasized these findings. Deriving descriptions from tokenized resource identifiers yields more specific information than from API calls. Due to the generalization of neural networks and the description TF-IDF model not understanding synonyms, the performance metrics had to be taken with a grain of salt. Despite the use of a noisy dataset, in overall terms the system found concise, accurate depictions for many apps. With the use of the model explanation framework SHAP, incorrect predictions could be discovered. In some cases, the model prediction explained the purpose even better than the app's original description. The evaluation showed the generalizing nature of neural networks trained on a large, diverse sample set and underlined the effectiveness of our approach.

Depending on the app and its underlying source code and static resources, the neural network model found meaningful descriptions for many apps. As part of the app publishing assistance system, it can thus be used as a quick reference for drafting or verifying app descriptions. The evaluation confirms the beneficial value of our description inference system for improving the app descriptions that users see in Google Play.

Chapter 6

Textual Permission Clarity

After clustering apps and inferring parts of the description text, in the third main chapter, we address the relationship between an app's description and the permission set it requests. We start by motivating our work on a textual permission clarity system. Then, we explain our approach for preprocessing tuples of description text and permission set. Thereupon we describe the convolutional neural network we use to estimate the clarity. The final section of this chapter is an evaluation of our approach to apps the model has not seen during training. We show that our system finds text fragments that point to permission usage, it identifies missing permission explanations, and it finds other interesting, more general correlations between permissions and common use cases. We conclude that it can be of great use for improving app descriptions in regard to permission usage.

6.1 Motivation

When smartphone users search for an app in Google Play, a concise depiction of its functionality is crucial, especially concerning app permissions. Description texts are, together with screenshots, app title, and reviews, relevant information for users before installing the app. A user must comprehend not only what features the application has, but also the implication on their privacy. Due to the large pool of personal data, a smartphone may disclose, the user must be briefed about the potential exposure.

In general, access to privacy-critical information, e.g., location, contacts, or personal files, is protected by the Android permission system. Any text describing an application should thus contain direct or indirect references to the permissions the application requests. Ideally, a short, yet concise part is provided that includes sentences which directly describe how its requested dangerous permissions are used. Humans may or may not read the text before downloading an app. However, a well-drafted description text ensures that users can make reasonable decisions at install-time in regard to app functionality and protecting their privacy.

Consequently, our goal is to obtain a system that processes a description text in human language. Based on this text, the system should give a score for each permission. A better score corresponds to a higher likelihood that the app requests the permission. Additionally, the system must provide information which parts of the description texts cause the score. Ultimately, the predictions are compared to the actual permissions to draw conclusions. The discrepancy between these two sets, the prediction based on the text and the actual permissions, is also security-relevant.

The outlined textual permission clarity system can serve multiple purposes. Our system gives a score for each permission group. At the same time, app markets, i.e., Google Play, also show the set of requested permissions before downloading. Based on this additional information, a user might not want to download an app. Furthermore, the system highlights which parts of the text affect the score for a certain permission, e.g., the word *photo* is likely to trigger permission groups for camera and external storage. App publishers can use this information to improve the quality of their description texts. Ultimately, our approach can also be used to find common correlations between permissions and certain words or phrases. Relations such as the word photo for the camera permissions are apparent. Other permissions are insufficiently described, as such a system can show. An automated system that finds word correlations based on real-world samples can gather such information. We consider these three use cases, i.e., pre-download evaluation, description quality improvement, and data-driven evaluation of frequent permission usage, substantial incentives for building our textual permission clarity system.

6.2 Approach

Our system is supposed to find a relationship between text and permissions based on a large sample dataset. Subsequently, we first cover the preprocessing steps of apps and descriptions, which differs from the actions we undertook in the previous clustering chapter in several aspects. Then, we can present that data to a convolutional neural network architecture equipped for text classification. To understand why an app estimates a score, we use LIME to comprehend the decisions our model makes, meaning the correspondence between permissions and words.

We differentiate between two phases, the training phase, and the prediction phase. Permissions and descriptions are transformed to be processable by the network in the preprocessing step. During the training phase, as shown in Figure 6.1, the model learns correlations between parts of description texts and actual permissions of the app. The CNN uses backpropagation to adjust its internal parameters to predict the app's permissions correctly. The training continues until the performance plateaus or decreases, i.e., the quality of our permission scores does not improve anymore. We then use the model with the best performance in training for the prediction phase.

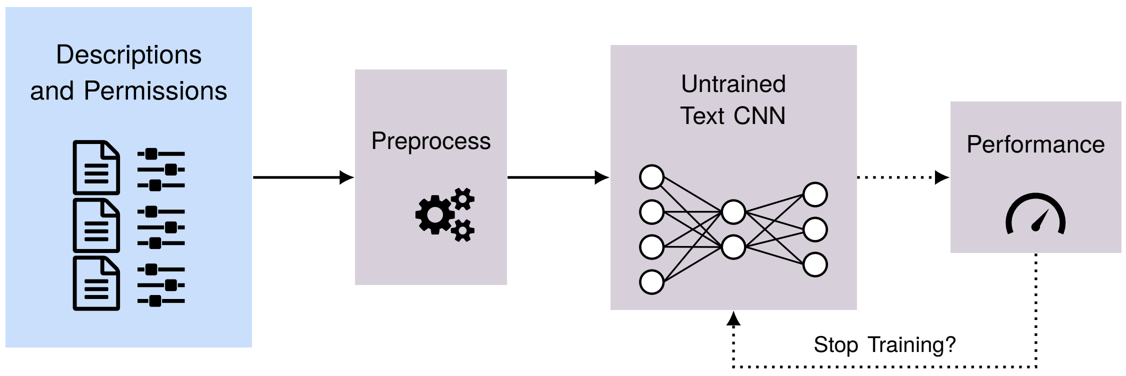


Figure 6.1: Textual permission clarity system: training phase.

In the prediction phase, the system receives app descriptions not seen during training. We depict the prediction phase in Figure 6.2. The description text receives the same preprocessing steps as during the training phase. The now-trained network outputs scores whether the description reveals hints about permissions, based on the knowledge gained in the training phase. Additionally, we use LIME for network interpretation to estimate which words in the description cause the scores. We thus obtain a probability for an app to request a certain permission, as well as which parts of the text affect this score.

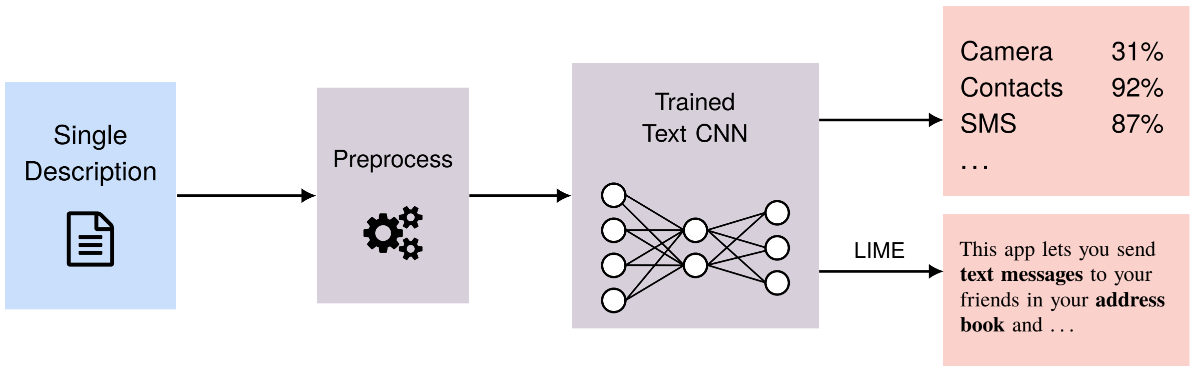

Single Description
Preprocess
Trained Text CNN
Camera 31%
Contacts 92%
SMS 87%
...
LIME
This app lets you send **text messages** to your friends in your **address book** and ...

Figure 6.2: Textual permission clarity system: prediction phase.

6.2.1 Preprocessing

To be able to train a neural network, adequate and efficient data representation is crucial. Subsequently, we cover the preprocessing work our neural network requires. First, we deal with the processing of the description texts, in particular, the tokenization steps and the use of pre-trained word embeddings. Then, we transform the grouped permissions list to vectors. The preprocessing steps that we subsequently describe are applied to all samples, regardless of whether they are used for the training phase or the prediction phase.

Descriptions

First and foremost, we need to find a way to transform the description texts consisting of character arrays into tokens. Unlike the bag-of-words approach with TF-IDF in the previous chapters, we now maintain the word order. With the use of a pre-trained word embedding model, we translate the description tokens to embedding vectors with a trial-and-error lookup strategy. We ultimately receive dense matrices of a fixed size for each description.

The pre-trained embedding model consists of key/value dictionaries, with the key being a word or word combination, and the value being the corresponding embedding vector. Many embedding models are not only trained on alpha-numeric tokens, but also contain entities like hyphenated expressions, web addresses, or brands and trademarks (including special characters). The neural network has to utilize the objects in the pre-trained embedding model properly.

The tokenization strategy thus has to adapt to the available dictionary keys. We show our approach to this problem in Figure 6.3. Initially, we strip the description of all formatting done with HTML tags. We receive a raw description string. For splitting the string into tokens, we analyze the character set of our pre-trained embeddings. A set of all particular characters available in the dictionary's keys is created, which we call alphabet. The keys, i.e., the words and word groups, mainly consists of alphanumeric ASCII characters, but also of non-alphanumeric and Unicode characters. After obtaining the full alphabet, we use its inverse as a delimiter for tokenization. Consequently, the raw string is split by every character that does not occur in the alphabet, which leaves us with an initial set of tokens.

After acquiring this set of tokens, we start the lookup process. For each token in the initial set, we perform the following steps: First, we check whether the full token exists in the dictionary. If it does, we use the embedding e_i corresponding to the entry (word) at dictionary position i. If not, we attempt to clean and split it. First, we remove the leading and trailing non-alphanumeric characters and perform another lookup. In case of another dictionary-miss, we split the cleaned token into subtokens, with all remaining non-alphanumeric as delimiters. We perform separate lookups for all subtokens. Our goal is to match as many tokens against the pre-trained embedding as possible.

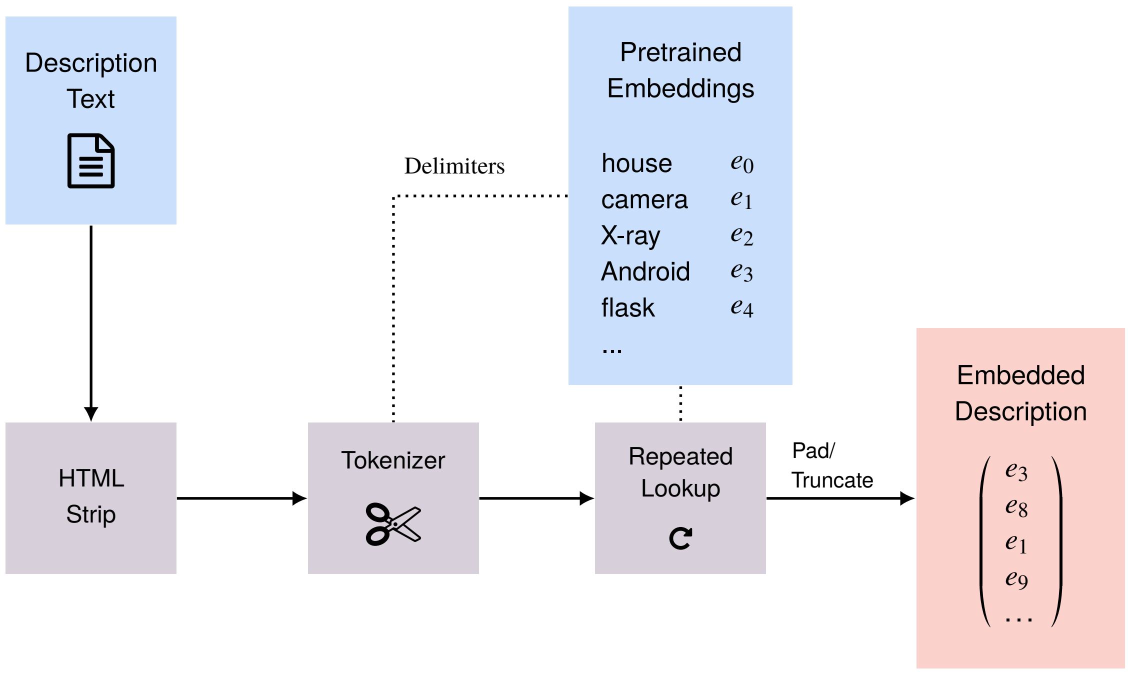


Figure 6.3: Tokenization and word embedding lookup for the permission perceptibility system: the text is split into tokens using all characters that do *not* occur in the pre-trained embedding dictionary. On a lookup-miss, the token is partitioned further.

A limitation of the token number, i.e., the description text length to be processed by the network, is necessary. The reason is that the convolutional neural network can only handle a fixed-size input. Descriptions practically have arbitrary length, and thus, we need to set an upper limit. We look at the dataset and choose an upper limit in a way so that the majority of the descriptions can be fully processed. In particular, we set the maximum description length in a way that around 95% of descriptions can be processed as until the last token. This is a threshold we learned by experience which depends on the dataset itself and the number of tokens that can be represented via embeddings. For texts larger than that, we truncate them, and for the ones shorter, we add padding tokens. Ultimately, we obtain a matrix of constant size with the row indices corresponding to the description tokens, and each row containing an embedding vector from the pre-trained word embedding model. The matrix is a transformed representation of an app's description that we can subsequently use as machine learning model input.

Permissions

The permissions of each app also have to be represented in a way that a neural network can handle them. We focus on critical, privacy-concerning permissions, so-called dangerous permissions, as described in the background chapter. Custom permissions apart from the ones in the SDK are also not processed. The grouped permissions are then represented in binary.

Each app comes with a list of `<uses-permission>` tags defined in the manifest file. In a first step, we extract all the permission identifiers. Then, we only keep the ones that are in dangerous groups (Android 6.0 or higher). We group these permissions according to the SDK docs. This leaves us with nine permission groups.[1] There may be apps in the dataset that target or support phones running an OS

[1]CALENDAR, CALL_LOG, CAMERA, CONTACTS, LOCATION, MICROPHONE, PHONE, SMS, and STORAGE. SENSORS is omitted because it is not used by any app in our sample set.

version below Android 6.0, i.e., API level 22 or lower. However, grouping their permissions still subsumes privacy-related ones. We thus represent all single permissions of an app via runtime permission groups.

These permission groups are then transformed into multi-hot encoded vectors. For each app, we check whether it requests one or more single permissions within a group. If it does, then the corresponding column of the vector is set to 1, and 0 otherwise. As a result, we obtain a 1-by-9 binary vector that portrays the app's permissions. We use these vectors as targets for the machine learning model.

6.2.2 Network Model

After finding a suitable representation for description texts and permission sets, we can build our neural network. We create a convolutional neural network (CNN) to make use of the local properties of our textual input. Related work for processing text with comparably simple CNNs yielded powerful results due to the structural similarities to images. The comparably efficient training process, as described in the background chapter, adds to our decision for CNNs. Our model performs a multi-label classification task, with each label corresponding to a permission group. Subsequently, we describe the general architecture of our network in regard to filters, layers, and other hyperparameters.

As a general architectural decision, we opted for a CNN for several reasons. Unlike recurrent architectures (e.g., LSTM cells), CNNs are faster to train and more straightforward to interpret. The convolutional architecture captures the impact of single words, word groups, and phrases. If we only wanted to examine the existence of certain words or set of words, machine learning would not be required. However, CNNs do not only find local patterns, but also interpret their co-occurrence. In our case, this means that decisions are made, especially in regard to the combination of words, i.e., if certain words occur together and how relevant each of them is for a certain permission. On the other hand, the filters allow for better generalization. Since we use word embeddings, the 1-d filters do not remember single words. A particular filter thus ideally captures groups of words that have similar properties in the word embedding subspace, e.g., one filter could capture "photo" and "picture," as well as their plurals. These properties emphasize the use of CNNs for our text multi-label classification problem.

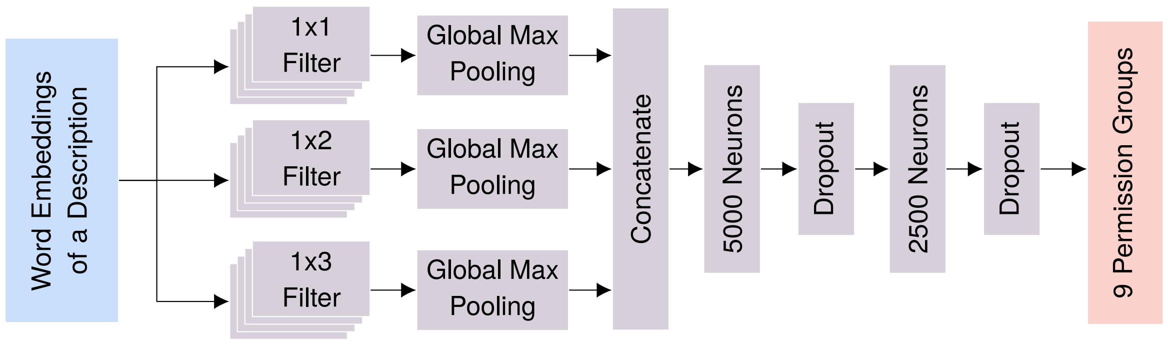


Figure 6.4: CNN architecture to process description texts (embedded) and get the permission likelihood.

Figure 6.4 shows the full neural network in detail. We use the text CNN by Kim et al.[25] as summarized in the background chapter as a basis and modify it to some extent. The network input is a $d \times m$ matrix, where d stands for the embedding size and m for the maximum number of tokens (embeddings) we process per description. Then, the network applies 1-d convolutions with different kernel sizes: 1, 2, and 3, i.e., filters that can cover one to three words at a time. For each of these three sizes, we allocate 1024 filters to be trained. Each of the three filter bundles is subsequently reduced through a global max-pooling layer. We are left with $3x1024$ max-values, which are concatenated so they can be processed by the successive

1-d dense layers. For regularization, we apply a dropout of 0.2 after each of the two dense layers.[2] The output layer, consisting of 9 neurons that match the nine permission groups, gives then values between 0 and 1 for each of them. Table 6.1 lists the additional hyperparameters. We used random search to find good parameters for the number of layers, the dense neurons, and the dropout.

Optimizer	Adam, $\eta = 0.0001$
Batch Size	32
Weight Initialization	Glorot (Uniform)
Loss Function	Binary Cross-Entropy
Filters	Padding: Same
	Stride: 1
Hidden Activations	ReLU
Final Activations	Sigmoid
Early Stopping	Patience: 6 epochs
	Delta: 2% F_β-score (macro)

Table 6.1: Neural network hyperparameters for the permission perceptibility CNN.

6.2.3 Training

The training phase of the previously described network model requires some additional parameters to be set up. Regarding the metrics, we weigh the loss function to combat class imbalance. For the same reason, we also scale individual performance metrics. Regarding the samples, our dataset is divided into three sets. This allows us to adopt early stopping to find a good final training state that balances overfitting and underfitting. We briefly outline these points and their relevance to the learning process.

Class imbalance is a common problem in machine learning. In our system, the unequal distribution of labels results from the differences in permission usage between apps. The grouped permissions of an app heavily depend on the purpose of an app, as well as their co-occurrence. Some groups tend to occur together more often. Additionally, some permissions are required by fewer apps than others. As we have nine network outputs, i.e., permission groups, which can occur in any combination, the impact of class imbalances is even higher.

We aim to partially even out the imbalance by using class weights, as shown in Equation 6.1. To this end, we first need to find the actual distribution. We add up the binary 1-by-9 permission vectors $\mathbf{p}_i$ of all samples to obtain a sum vector $\mathbf{s}$. We divide the inverse of the sum vector by the total number of samples. Additionally, we normalize the vector via diving by 9, so the non-weighted loss and the class-weighted loss can be compared directly. We receive a 1-by-9 vector $\mathbf{c}$ that counter-balances the overall class distribution. It is multiplied with the actual value of the loss (error value) that is back-propagated during training for every single output. Consequently, for a class with lower sample support, the extent of the error is increased, and decreased for higher support. Although this balancing mechanism increases the importance of particular samples over others, it reduces the magnitude of class-dependent overfitting.

[2]20% of the weight connections are chosen randomly and set to zero in each training iteration.

$$
\begin{aligned}
\mathbf{p}_i &= \begin{bmatrix} p_{i,0} & p_{i,1} & \dots & p_{i,8} \end{bmatrix} && \text{for } 0 \le i < N_{\text{samples}} \\
\mathbf{s} &= \sum_{N_{\text{samples}}} \mathbf{p}_i \\
\mathbf{c} &= \frac{N_{\text{samples}}}{\mathbf{s}} \cdot \frac{1}{9}
\end{aligned}
\tag{6.1}
$$

As an essential requirement, our system must be able to work with real-world descriptions. This requirement means that a portion of the texts does not contain sufficient information for each permission group we have. For these samples, humans cannot identify permission usage in the description, and thus, the machine learning model should not attempt to find meaningless correlations. This obstacle, in general, is challenging to tackle, as the training set does not contain particular information of a description's quality. We at least want to address this problem through a weighed F-score and use the β parameter. We set it to 0.5, which causes the precision to be treated more important than the recall. Thereby, we prefer models that make more confident predictions.

While the previously described class-weighing approach focuses on the loss function, i.e., the back-propagation of the error during training, class imbalance must be taken into account for performance metrics as well. We use precision, recall, and F_β-score to measure the performance for each class. For the sake of comparing different architectures and training states, a single numeric value for a trained model is required. Thus, we calculate macro and micro averages over all classes. We use the macro F_β-score for comparing as it factors in the class imbalance through the support of true positives per class.

In regard to model overfitting, we split the samples into sets and adopt early stopping. We partition the dataset into training data, validation data, and test data. Most samples are put into the training set, which is used for back-propagating the error. The performance on the validation set is calculated after every epoch to assess how well the model generalizes. The test set is not used during training. All set affinities are decided randomly, and the order within a set is permuted after every epoch as well. We aim to find a good epoch count for training without using a fixed epoch count. As stated, the validation set's performance is constantly monitored after every epoch. In the case of performance decrease, training continues for six more epochs if another local minimum can be found later. If this is not the case, the weights are restored to the previously best-performing state. That weight configuration is the final state of the trained network.

6.2.4 Prediction Explanation

Besides the likelihood of permission groups for a description text, we also want to gain knowledge about the words leading to the predicted permission scores. In order to interpret how our CNN makes a prediction, we use the model explainer LIME to find relevant parts of the input. We calculate a value for each word that shows its significance for the output. Ultimately, we get nine heatmaps for the description, one for every permission group.

LIME treats our machine learning classifier as a black box and evaluates variations of a particular input sample. The input to be evaluated, meaning the description text, is split into its tokens. The LIME algorithm then generates similar test samples: For each test sample, random tokens are removed from the original sample. The impact of each change is then accumulated per output label (i.e., permission group). LIME needs two parameters. First, the number of features to be varied, which we set to the number of tokens a sample has. Second, the number of random samples to be generated, whereby a higher number computes more different combinations. We empirically decided for 100 times as a trade-off between computation time and result accuracy. The algorithm can then calculate the token-related significance of the model's decisions. In case a token occurs multiple times, LIME removes all occurrences. This unique-feature treatment makes sense in our case because of the global pooling layer that merges the information coming from all CNN-filters. Eventually, we are left with several impact scores, one list for each permission group, with a sub-list consisting of values for each word in the description regarding that group.

The impact scores generated by LIME are negative and positive decimal numbers. Depending on the exact model and the input, the minimum and maximum values of these impact scores can have arbitrary absolute values, e.g., -3.7 for a negative impact of a single token on the result and 2.4 for a positive impact. A negative impact means that the removal of a word causes a more confident prediction. Thus, we need to set a lower boundary and normalize the LIME values. Additionally, we need to take the actual prediction of our model into account. Our neural network outputs sigmoid values between 0 and 1 for each permission group, which refers to the confidence of a decision. For example, if one class output is close to 0.3, i.e., a confidence of only 30% that the app requires that permission, the normalized LIME score also has to be weighed lower in comparison to, e.g., 90%.

$$\begin{aligned} \widetilde{L}_{m,j} &= \frac{L_{m,j}}{\max_k \left(L_{m,k}\right)} \qquad \text{for k ... } 0 \le j < N \\ h_{m,j} &= \widetilde{L}_{m,j} \cdot p_m \end{aligned} \tag{6.2}$$

Consequently, we use the simple normalization and scaling formula in Equation 6.2. For each output label m we have the estimated probability y_m, and N tokens in the description. LIME outputs the impact scores $L_{m,j}$ for each token j and each permission m. First, we zero-out all negative values and then divide each of these impact scores by the maximum value for that sample and label m. Then, these values $\widetilde{L}_{m,j}$ are scaled by the probability y_m for permission m. The result is a heat value $h_{m,j}$ for each word between 0 and the prediction for that permission m.

Hence, $h_{m,j}$ can be used as a heatmap over the text. It shows the relative impact each token has on the CNN's prediction for a specific permission group. A shortcoming of LIME is, however, that it only alters individual, single words. Therefore, we cannot evaluate our filter kernels that cover two and three tokens. Despite this downside, LIME can still give meaningful information about why the model decides for a particular permission group probability. In the evaluation section, we use the heatmaps to effectively differentiate between good and bad correlations that the system learns.

6.3 Dataset

For the training and evaluation of our permission perceptibility model, we need a pre-trained word embedding model and sample apps. For the embeddings, we compare two different word embedding architectures: Word2vec and GloVe. Like in chapter 4, we use apps from the PlayDrone dataset. However, we remove samples that would hamper training. Finally, we list some statistical properties of the obtained samples to show in advance which kind of data we use for the evaluation.

Since our approach requires descriptions to be provided as word embedding vectors, we need to choose a pre-trained model. Thus, we compare two different embedding methods: a prediction-based and a count-based one. The first one is Word2vec, where we use a model trained on three text corpora at once: Wikipedia articles, news articles and crawled websites.[3] The second embedding model we make use of is GloVe. There we use a model published by the authors which was trained on Wikipedia articles.[4] The

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

[4]http://nlp.stanford.edu/projects/glove/

resulting embedding vectors for both models have dimensionality 300. Table 6.2 shows some properties of the two models. These word-vector pairs are used for tokenization (preprocessing) and word representation (neural network input).

	Word2vec	**GloVe**
Architecture	Predictive	Co-occurance
Text corpora trained on	Wikipedia, UMBC, and statmt.org	Wikipedia
Total corpus words	≈ 16 billion	≈ 5 billion
Output embeddings	999,994	400,000
Characters	Lowercase	Lowercase

Table 6.2: Word embedding model properties.

As for the descriptions, we remove particular samples. All text we process has to be in English. Additionally, we set an upper limit on the length. Instead of using a character-based metric, we count the embeddable tokens, i.e., the number of tokens for each description after preprocessing. All descriptions are preprocessed as described in section 6.2.1 and transformed into a list of embeddings. Therefore, all tokens that do not exist in the pre-trained embeddings dictionary disappear. We take the length of the resulting list and remove all descriptions that have 20 or fewer embeddable description tokens.

The whole dataset is split into three subsets. The smallest one, the test set, contains samples we do not use at all during the training phase: the same 1,000 apps as in the previous chapter. Training and validation set are made up of the remaining apps, which are split so that randomly 20% are part of the latter. This partitioning scheme is required to both prevent overfitting of our machine learning model and give meaningful, commonly known examples in the subsequent evaluation. Table 6.3 lists the number of apps.

English descriptions	81,803 apps
20+ embeddable tokens	77,758 apps
Training and validation set	76,758 apps
Test set	1,000 apps

Table 6.3: App dataset properties, GloVe embeddings.

6.4 Evaluation

On the following pages, we evaluate our textual permission clarity system. We begin by comparing the performance of GloVe vs. Word2vec. Then, we have a brief look at the performance metrics and discuss their meaningfulness. We continue with a closer look at specific examples, where we analyze their word-to-permission correspondence and compare predictions and actual permissions. Apart from showing specific description texts, we also list words that we found to have a high impact on the model prediction. Finally, we give a general overview in which cases the model performs well and where it does not.

For comparing the influence of the two word embedding models, we use the same network architecture. The embedding sets we use have dimensionality 300 and are normalized, which allows us to only switch the underlying word-to-vector mapping of the system, i.e., its input before training. There is the possibility that different architectures could better exploit and handle the mathematical properties of GloVe and Word2vec, respectively. However, we estimate the impact of this limitation comparably low. Hence, we only do not modify the parameters like, e.g., neurons, dropout, or layer count further and only switch the embedding vector mapping here.

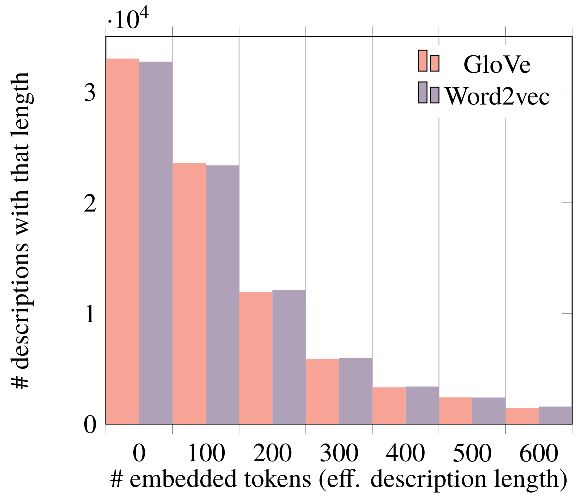


Figure 6.5: Histogram of the description length after embedding with the pre-trained GloVe and Word2vec models.

First, we want to check how well the two embedding models cover descriptions, which is part of the preprocessing step. We perform tokenization and repeated lookups, as previously described. Figure 6.5 shows the number of embeddings found, i.e., the effective description length the network sees, and how often descriptions with a certain length occur. As seen in the histogram, both create embedded descriptions of similar length: For GloVe, the resulting median length is 126 and Word2vec it is 127. Both embedding models are able to cover the texts to a similar extent. In regard to how well our textual permission clarity system can harness the pre-trained embeddings, there is no significant difference between GloVe and Word2vec for the obtained description lengths.

	GloVe	**Word2vec**
Precision	81%	76%
Recall	55%	54%
F_β-score	77%	70%

Table 6.4: Performance comparison for the two different embedding models GloVe and Word2vec. Scores are macro-averaged over all labels and over five folds.

After embedding the descriptions and analyzing the lookup rate, we can train our CNN. Table 6.4 compares the performance of both embedding models on the test set. These evaluations are done using 5-fold cross-validation and averaging the single values over the five folds. We can see that the two models compare very similar for the recall. Precision and F_β are better for the GloVe embeddings. We performed several more runs and saw that the differences could be slightly larger or smaller by a few percent, depending on the randomization of samples and network initialization. Thus, we conclude that the choice whether to use GloVe or Word2vec for our system does not affect the performance substantially, but GloVe performs marginally better, judging by the numbers. For the following results, we only use the GloVe-trained network models.

	Precision	**Recall**	$F_{0.5}$**-score**	**Support**
Ext. Storage	93%	76%	89%	699
Calendar	55%	24%	42%	36
Call Log	70%	44%	62%	108
Camera	71%	56%	67%	199
Contacts	80%	65%	76%	369
Location	82%	56%	74%	327
Microphone	83%	64%	78%	125
Phone	73%	68%	72%	440
SMS	83%	52%	73%	100
Macro Average	77%	56%	70%	
Micro Average	81%	65%	77%	

Table 6.5: Performance of our system on the test set, GloVe embeddings, averaged over five folds.

Next, we want to evaluate the performance of our trained CNN model in-depth. Table 6.5 shows the test set results.[5] The last column shows the support, i.e., the number of true positives the model captured. As mentioned before, the varying support for each permission group is influenced by the unequal distribution of permission groups in apps in general. As we assumed, the recall column altogether has lower values compared to the precision column. This can be explained by the fact that we do not have labeled samples, but also many low-quality descriptions. Some descriptions miss permission-related phrases, and therefore, the model cannot predict a positive result. Hence, we focus on precision scores.

The precision values range between 71% and 93%. We can see a tendency for some permissions. The macro precision average is at 77%, and certain permission groups perform significantly better or worse than that. Calendar is worst, while camera, phone, and call log are also on the lower end. On the opposite, external storage has the highest precision, concurrently with the highest support. Although the scores suggest that some permissions are more clearly identifiable in a description, a detailed look is necessary before drawing concrete conclusions. Despite the numeric results, we can only make vague assumptions on the model's performance in practice. Due to our use of real-world descriptions and their qualitative shortcomings, we need to analyze samples and predictions in detail. A high precision score can also mean that the textual correlations found during training are not reasonable and not helpful for humans to identify permissions. Since neural networks always learn correlation-based, we have to gain knowledge which correlations the model has learned and whether they are meaningful or coincidental.

Hence, we want to demonstrate the result on single samples. We select three apps and show interesting findings: a messaging app, a video editing app, and a food delivery app. For the sake of brevity and readability, we only show relevant fragments of their descriptions, which are overlaid with the LIME heatmaps for certain permission groups.[6] The words are formatted via the heat values in a way so that the background color's opacity corresponds with the LIME impact score, i.e., a darker color means a higher significance for the prediction. Subsequently, we have a look at the description words our system deems significant, as well as how accurate the prediction is compared to the actual permission.

[5]All single values are averaged independently across the folds.

[6]We do not omit any sentences relevant for permissions.

Snapchat

Actual vs. Inferred Permissions

	Requested	Prediction
Camera	yes	90%
Ext. Storage	yes	90%
Microphone	yes	77%
Contacts	yes	46%
Phone	yes	39%
Location	yes	8%
SMS	yes	1%
Call Log	no	1%
Calendar	no	0%

Camera

Enjoy fast and fun mobile conversation! Snap a photo or a video, add a caption, and send it to a friend. They'll view it, laugh, and then the Snap disappears from the screen – unless they take a screenshot! You can also add a Snap to your Story with one tap to share your day with all of your friends. [...] if you're both Here, simply press and hold to share live video - and Chat face-to-face! Happy Snapping! *** Please note: even though Snaps, Chats, and Stories are deleted from our servers after they expire, we cannot prevent recipient(s) from capturing and saving the message by taking a screenshot or using an image capture device.

Ext. Storage

Enjoy fast and fun mobile conversation! Snap a photo or a video, add a caption, and send it to a friend. They'll view it, laugh, and then the Snap disappears from the screen – unless they take a screenshot! You can also add a Snap to your Story with one tap to share your day with all of your friends. [...] if you're both Here, simply press and hold to share live video - and Chat face-to-face! Happy Snapping! *** Please note: even though Snaps, Chats, and Stories are deleted from our servers after they expire, we cannot prevent recipient(s) from capturing and saving the message by taking a screenshot or using an image capture device.

Microphone

Enjoy fast and fun mobile conversation! Snap a photo or a video, add a caption, and send it to a friend. They'll view it, laugh, and then the Snap disappears from the screen – unless they take a screenshot! You can also add a Snap to your Story with one tap to share your day with all of your friends. [...] if you're both Here, simply press and hold to share live video - and Chat face-to-face! Happy Snapping! *** Please note: even though Snaps, Chats, and Stories are deleted from our servers after they expire, we cannot prevent recipient(s) from capturing and saving the message by taking a screenshot or using an image capture device.

Figure 6.6: *Snapchat*: prediction for each permission group based on the description and LIME heatmap overlays for selected permission groups.

Snapchat is a social messenger that allows sending pictures, so-called snaps, as well as sending text messages, sharing one's location and making calls. Figure 6.6 shows the permission prediction our system makes, compared to the permissions an app actually requests, and heatmap overlays for three permission groups. For the camera permission, the words *snap*, *photo*, *video*, and *capturing* have high impact. Similar words are important for the external storage permission, which also relies on the words *share*. Sharing files often necessitates access to the phone's storage, e.g., for file pickers or saving larger files. The prediction scores for these two permissions of 90% are thus legitimate. For the microphone permission, a prediction score of 77% is given with a large influence from the words *chat*, *video*, and *capture*. This value would likely be higher if words like *voice* or *audio* occurred in the text. In contrast, the SMS and location permissions cannot be predicted based on the description. The absence of relevant words justifies the stated predictions of 8% and 1%, respectively. The contacts and phone permissions are under 50%, also due to insufficiently describing them. For this description text, the presence of significant words for the camera, microphone, and external storage permissions can be confirmed, while the use of other permissions is insufficiently mentioned.

AndroMedia is a video editor. Its description that we evaluate in Figure 6.7 contains information relevant for this kind of multimedia apps. The necessity of the external storage permission is captured by the words *export*, *audio*, *files*, and *save*. The app also requests the location, microphone, and phone permissions, with no mentioning of their usage in the text whatsoever. There is also no apparent practical

AndroMedia

Actual vs. Inferred Permissions

	Requested	Prediction	
Ext. Storage	yes	90%	
Phone	yes	27%	⚠
Microphone	yes	17%	⚠
Location	yes	1%	⚠
Contacts	no	10%	
Camera	no	1%	
Calendar	no	0%	
Call Log	no	0%	
SMS	no	0%	

⚠ Prediction deviates strongly.

External Storage

AndroMedia Video Editor Video Editing App For Android Platform AndroMedia is unique Video Editing App For Android Platform Designed to be intuitive to use, AndroMedia is fully featured video editing program for creating professional looking videos in minutes. Making movies has never been easier. * Export movies in standard definition or HD (320p 480p 720p) * Drag and drop video clips for easy video editing * Trim and Combine both video and audio files in two different editor. * Apply effects and transitions and more * Overlay title clips for captions and movie credits * Apply Crop and Ken Burns effects to your video tracks * Apply FadeIn and FadeOut effect to your audio tracks * Supports MP4,MOV,JPG,PNG,MP3,WAV file formats * Save login credentials to upload videos directly to YouTube from AndroMedia * Easy to use layout

Figure 6.7: *AndroMedia*: prediction for each permission group based on the description and LIME heatmap overlay for the external storage permission group.

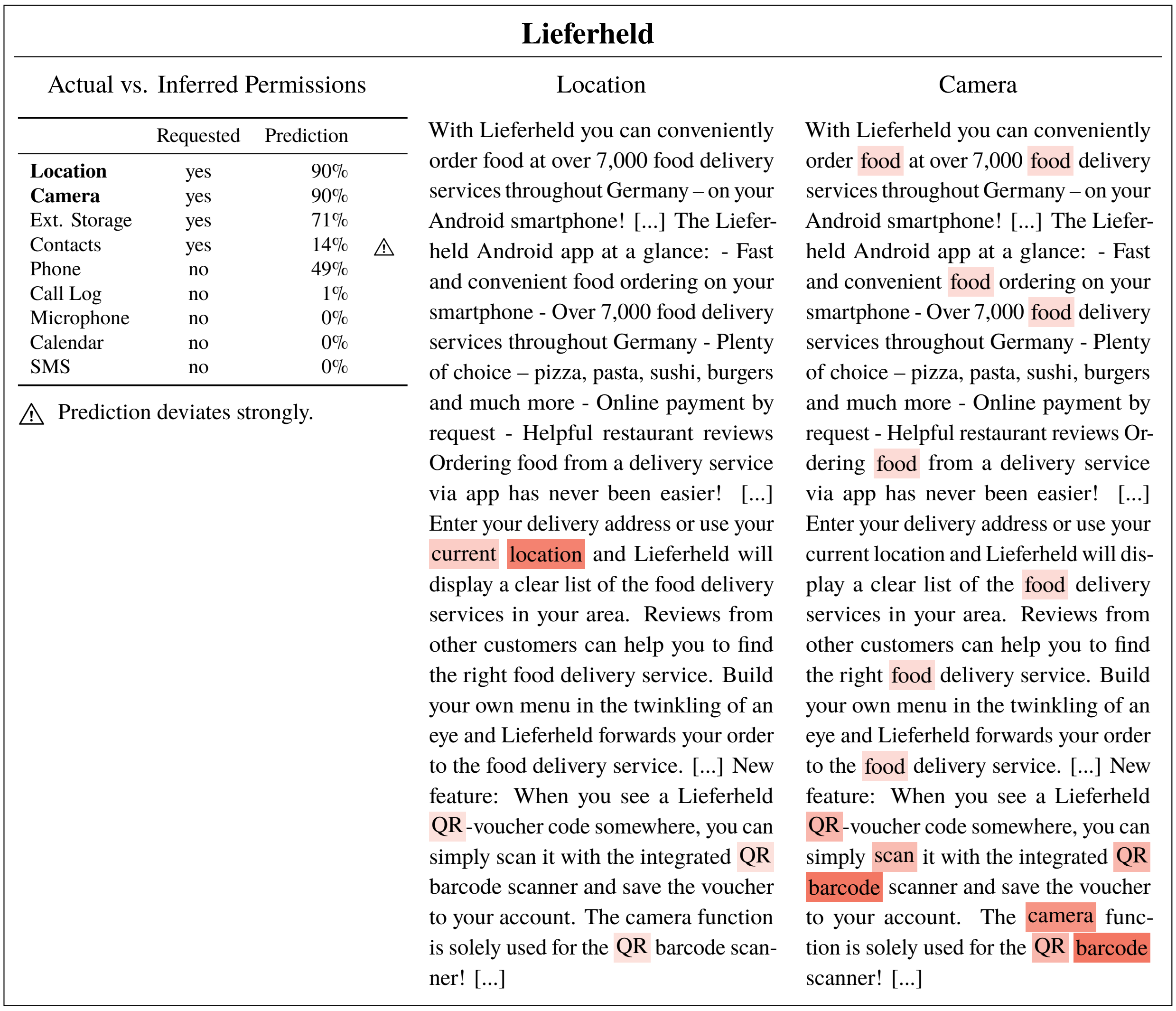
reason for video editing apps to require these privacy-critical permissions. Although the app might provide the promoted features, the three non-mentioned permissions should alarm users. In case there is no malicious intent, the developer is encouraged to either add information to the description or to remove that functionality from the source code. This example shows that our textual permission clarity system can identify critical discrepancies between described app purpose and permission usage.

Lieferheld is a German food delivery platform, with the app description given in Figure 6.8. Our system successfully identifies the word *location*, in combination with *current*, to predict the location permission at 90%. As these words occur next to each other and presumably exist multiple times in the dataset, it is likely that a 1-by-2 or 1-by-3 filter of the CNN has learned this combination.[7] The camera permission is mainly triggered by *camera*, *barcode*, and *QR* (*scanner*). This relation shows that there are many apps in our dataset that mention the words *barcode* and *QR* (*codes*), which tend to require the camera permission even more likely than apps that solely contain the word *camera* in their description. Although the external storage permission was predicted with 71%, there are no hints on its usage in the text. We attribute this to the higher co-occurrence likelihood of external storage and camera. The contacts permission's usage was also not reflected in the description. Our analysis of the Lieferheld app thus also shows sufficient expression for certain permissions, but a lack for others. These should be either addressed by the app publishers or be of concern for the end-users.

Furthermore, we want to evaluate which words have a high impact for each permission group. Although the dense layer of the CNN makes the predictions because of word combinations, frequently occurring words can be telling for the network's decision process. First, we gather all words across all descriptions. Then, we sort them by how often they occur and how high their LIME impact sore is. We do this separately for each permission group. Table 6.6 lists the most significant tokens per permission group. These words have a comparably high impact for true positives (LIME score >70%) and occur multiple times. The word lists demonstrate that the system captures significant words that make sense for a human reader. Certain correlations are apparent, e.g., the word *calendar* for the calendar permission or *map* for the location permission. Other words show correlation-based tendencies that are less evident:

- Weather apps tend to require the user's location, which makes *weather* a trigger word for the location

[7]Due to the way the LIME algorithm is implemented, i.e., removing features one by one, we can only analyze the impact of single words, not their co-occurrence.



Lieferheld

Actual vs. Inferred Permissions

	Requested	Prediction	
Location	yes	90%	
Camera	yes	90%	
Ext. Storage	yes	71%	
Contacts	yes	14%	⚠
Phone	no	49%	
Call Log	no	1%	
Microphone	no	0%	
Calendar	no	0%	
SMS	no	0%	

⚠ Prediction deviates strongly.

Location

With Lieferheld you can conveniently order food at over 7,000 food delivery services throughout Germany – on your Android smartphone! [...] The Lieferheld Android app at a glance: - Fast and convenient food ordering on your smartphone - Over 7,000 food delivery services throughout Germany - Plenty of choice – pizza, pasta, sushi, burgers and much more - Online payment by request - Helpful restaurant reviews Ordering food from a delivery service via app has never been easier! [...] Enter your delivery address or use your current location and Lieferheld will display a clear list of the food delivery services in your area. Reviews from other customers can help you to find the right food delivery service. Build your own menu in the twinkling of an eye and Lieferheld forwards your order to the food delivery service. [...] New feature: When you see a Lieferheld QR-voucher code somewhere, you can simply scan it with the integrated QR barcode scanner and save the voucher to your account. The camera function is solely used for the QR barcode scanner! [...]

Camera

With Lieferheld you can conveniently order food at over 7,000 food delivery services throughout Germany – on your Android smartphone! [...] The Lieferheld Android app at a glance: - Fast and convenient food ordering on your smartphone - Over 7,000 food delivery services throughout Germany - Plenty of choice – pizza, pasta, sushi, burgers and much more - Online payment by request - Helpful restaurant reviews Ordering food from a delivery service via app has never been easier! [...] Enter your delivery address or use your current location and Lieferheld will display a clear list of the food delivery services in your area. Reviews from other customers can help you to find the right food delivery service. Build your own menu in the twinkling of an eye and Lieferheld forwards your order to the food delivery service. [...] New feature: When you see a Lieferheld QR-voucher code somewhere, you can simply scan it with the integrated QR barcode scanner and save the voucher to your account. The camera function is solely used for the QR barcode scanner! [...]

Figure 6.8: *Lieferheld*: prediction for all permission groups based on the description and LIME heatmap overlay for the location and camera permission groups.

permission.

- The contacts permission is often used for *sync*hronizing the user's address book with remote services, and also for setting different *ringtones* for single contacts.
- The broadest spectrum of words can be found for the external storage permission. Words like *screenshot*, *photos*, or especially *files* are reasonable correlations as writing or reading them mostly requires that permission.

While these less-evident findings might be reasonably evident to developers and experienced users, their link to a particular permission might not be so obvious for other users. Links like weather to the location permission also show how the correlation-based learning effects. Our qualitative analyses show that certain categories like camera or location learned clearer, more understandable links. Better and more frequent textual reflections most likely cause this clarity. A similar clarity also counts for the calendar and microphone permissions, which tend to have a narrower spectrum of use cases and a lower class support, leading to more similar descriptions. The external storage permission lies on the other end of this spectrum, as it the most frequently requested permission (support, Table 6.5) and commonly lacks good reflections. Similar effects occur for the contacts, phone, and call log permissions, that partially rely on the same trigger words. These overlaps show that description writers often do not differentiate sufficiently when they describe these three permission groups.

Working with noisy samples that have not been manually labeled inevitably causes these effects. We do think that our approach can be of great benefit in practice, but also want to point out that the system should solely be used as assistance and not included in any decision pipeline without human monitoring. In general, however, on the one hand, it can help to identify shortcomings in the reflection of permission usage, and on the other hand, show common usages for permissions and their overlaps.

Contacts	**Location**	**Phone**
contact(s)	gps	call(s)
call(s)	map(s)	ringtones
ringtone(s)	near(est)	voice
friends	location	personalized
sync	weather	phone
Ext. Storage	**Camera**	**Microphone**
photo(s)	qr	microphone
files	barcode	walkie / talkie
sync	scanning	record
mp3	photo	voice
voicemail	flash(light)	calls
Call Log	**SMS**	**Calendar**
contact(s)	sms	calendar
call(s)	text	
phone	message	
sms	call	

Table 6.6: Words per permission group that occur more than once and have high impact according to their normalized LIME values. Some permission groups contain fewer than 5 words due to their low class support in the test set.

6.5 Summary

In this chapter, we showed our approach to detect an app's permissions based on its description. We created a convolutional neural network model that finds relations between elements of the description text and run-time permissions groups. Additionally, we included functionality to show the impact of certain parts of the description on the prediction. The model was evaluated using a real-world dataset to verify the reflection of permission usages and to find word influences per permission, both sample-based and on a larger scale.

The approach consisted of three parts, preprocessing, neural network model, and application of a model explanation framework. We processed the description texts by splitting them into tokens that occur in the pre-trained word embeddings model. Therefore, we used a lookup procedure that repeatedly checked if tokens existed in the embeddings model, and if not, split them and repeated the lookup on subtokens. Then, we used a convolutional neural network with 1-d filters. These filters captured the meaning of words and short phrases in the text. The CNN, with its filters and dense layers, aimed to find correlations between these word embeddings, representing the description, and the permission groups an app actually requested. After training the network, we used the model explanation algorithm LIME to obtain more detailed information on how the model makes its decisions. LIME's application outputted per-word relevance scores for each permission group per description sample. Altogether, we created a convolutional architecture for text classification and explained which description parts lead to specific permission probabilities.

Thereupon we evaluated this approach from several angles. First, we compared two different pre-trained embedding models. We used GloVe and Word2vec corpora. Our findings showed that GloVe performs better to some extent. Then, we had a look at the metric-based performance evaluation for the single permission groups. Due to the noisy dataset we used, i.e., real-word descriptions that, to some extent, do not contain sufficient permission-related phrases, we deemed this metric-based evaluation partially unfit. To find out whether the network correlations were meaningful, we assessed which words caused particular permission predictions. First, we demonstrated the application of our system based on single apps. On the one hand, we use textual highlighting for visual identification of the relevant words. On the other hand, we also used the permission prediction and text highlighting to assess insufficient descriptions better. We showed based on particular apps that our system finds textual shortcomings. Second, we analyzed frequent words that triggered the prediction of individual permissions. Due to the noisy dataset, however, we also found non-intuitive word influences that lead to correct predictions without the words being meaningful permission identifiers for humans. The results of our evaluation showed that, in general, our models found meaningful relations that could help to improve or verify the corresponding descriptions.

Overall, we created a system that yields valuable information for enhancing app descriptions. We demonstrated that a multi-label text classification CNN can be used to identify relevant description parts and predict whether the text points towards a particular permission usage. Our approach identifies incoherence between actual permissions of an app and its description, which can reveal permission abuse. Hence, our textual permission clarity system can be utilized to help publishers and end-users to assess an app's impact on privacy and also improve app description quality. Our evaluation has shown that this third contribution in regard to app description assessments is a vital part of an app publishing assistance system.

Chapter 7

Conclusion

In this thesis, we presented an Android app publishing assistance system that aims to improve app description quality in Google Play. In case an app is about to be published to the store, the assistance system provides information to make the description more comprehensible and accurate for the end-user. Our work consisted of three contributions. First, we used non-linear clustering and dimensionality reduction to compare the app to be examined with others in regard to descriptions and permissions. Second, we created a model that predicts description fragments for the app via its install package. Also, third, we proposed another model that is capable of detecting absent permission explanations in a description text. In what follows, we recapitulate our three approaches and their evaluation.

We began with app-relatedness in Chapter 4, *Clustering based on Descriptions and Permissions*. Data clustering commonly refers to the assignment of labels to data points using a similarity measure. While this distinction makes sense for certain aspects of smartphone apps, like the app's category in Google Play, we mainly wanted to work with measuring relative similarity. For an app that needed to be examined, we aimed to find and visualize its neighbors concerning the description text and the permission set. Positioning a new app in the already existing app universe was thus supposed to gain fundamental knowledge about the app's description quality and its permission usage.

For calculating app similarity and dissimilarity, our approach consisted of autoencoders and t-SNE. We trained two undercomplete autoencoders, one for permissions and one for descriptions. The fine-grained permission set was represented via one-hot encoding, the description texts via TF-IDF. After training, we extracted the latent space encoding for each sample. Then, we applied t-SNE dimensionality reduction. Eventually, this gave us two 2-d representations from these separate processes, one from the app's permissions, and one from the app's descriptions. We could then use these 2-d spaces to make distance calculations and visualize them.

Our evaluation of these non-linear transformations of permissions and descriptions showed several things. First, we looked at descriptions in regard to similarity. A comparison with PCA showed that t-SNE performed significantly better in positioning apps closer in regard to their category. Apart from their store category assignments, manually assessing the neighboring apps also emphasized that t-SNE can more accurately identify description relatedness. Second, we evaluated apps via their permission dissimilarity. We looked at two app groups in the dataset, anti-malware apps and web browsers. Their low-dimensional representations showed that the system captured the differences in permission sets, assigned individual permissions different importance, and understood which permissions were related. Based on the non-linear low-dimensional transformation of permissions, we could easily find groups of apps and spot outliers. The identification of outliers showed to be a good starting point for further investigations. Based on our evaluation, we concluded that our clustering approach was a valuable first angle for our app publishing assistance system's analyses.

Furthermore, Chapter 5 proposed a machine learning model for *Description Inference*. On account of the rising number of apps being published, a concise impression about what the app does, i.e., its core features, is an integral part of its pre-publication analysis. On the one side, smartphone apps are described in a way to distinguish themselves from their competitors by emphasizing app-specific advantages, and by adopting corporate wording. On the other side, apps, in general, share lots of features. A relatable, concise expression of core app features generated by a machine learning model was thus what we found necessary for app publishers to picture their app descriptions more accurately.

To infer such a concise description, we looked at three different information sources. We preprocessed application packages to get neural network inputs for three separate models: Two types of static resources in the app, namely string constants and XML resource identifiers, and the calls to the Android and Java APIs. We represented all of these inputs occurrence-based via TF-IDF. The model output, i.e., the training target, was a TF-IDF representation of the corresponding store description. The three different networks we trained outputted three ranked lists of tokens that should describe a given app's core features.

We then evaluated the three models with unseen samples to infer descriptions. Overall, we saw that the networks depicted many apps accurately. The generalizing nature of neural networks often yielded phrases that described core app functionality correctly but did not appear literally in the actual description text. Depending on the actual app's implementation, not all three models gave equally good results, with resource identifiers giving the best performance. In general, however, our qualitative analysis showed that for most apps, we could infer a valid result. Trained and evaluated on a broad spectrum of sample apps, we demonstrated that our description inference system was capable of generating meaningful, concise description phrases.

The third and last contribution of this thesis dealt with the explanation of permission-related functionality in the description text. Our model for *Textual Permission Clarity* in Chapter 6 tried to find a sample-based correlation between certain parts of the description text and the permission groups an app requests. As part of the app publishing assistance system, its purpose was to give a score of how well a description reflected these permissions. In case the score was low, meaning a certain permission could not be positively predicted from the text, the publisher should reason its use better. As permissions are one of the core concepts in Android to protect the user's privacy, their accurate depiction an app's store description is pivotal.

On the basis of this idea, we built a multi-label convolutional neural network for text classification. We preprocessed raw description texts so we could represent the embeddings that we matched from a pre-trained word embedding model. The neural network, which had different 1-d filter sizes and several hidden layers, gave a percentage for each of the nine dangerous Android permission groups. In addition to the mere predictions, we employed a model explanation algorithm to find the words in the description that caused the scores. Upon training, we could evaluate the system on unseen test apps.

Our evaluation showed that the system overall learned very reasonable relations between permission description in human language and the request of particular permission groups. First, we analyzed the description of several apps to gain knowledge of what words in the description caused a permission likeliness. Many relations, like the words *QR* or *barcode* for the camera permission, were intuitive to us. We also noticed that there is a gray area between what counted as obvious causality and what was a correlation coming from low-quality samples. A more experienced user might understand the link between barcode-scanning features and the camera, while others might not. Apart from showing the existence of permission-related phrases, we also demonstrated that their absence could be irritating. A discrepancy between requested permission and prediction could thus arise from anything between the publisher's negligence to intentional misinformation. We demonstrated that our textual permission clarity system could identify these shortcomings.

In a nutshell, this thesis contained three contributions that aim to enhance the description quality of apps in Google Play. Given an app is to be published, the solutions of this app publishing assistance system focused on three critical areas: finding related apps, estimating the app's core functionality, and assessing the permission reflection in the description. Our evaluation of numerous real-world apps showed that the practical application of our system could significantly enhance the portrayal of new Android apps in Google Play. Its application could provide end-users with description texts that have higher feature accuracy, better text conciseness, and clearer information about permissions.

Appendix A

Cross–Platform App Detection

Framework Name	**Directory**
Adobe AIR	`com/adobe/air`
	`air/com`
Apache Cordova	`org/apache/cordova`
Xamarin	`mono/`
React Native	`com/facebook/react`
Adobe Phonegap	`com/phonegap`
Appcelerator	`org/appcelerator`
	`ti/modules`
	`org/appcelerator`
Adobe Flash	`com/adobe/flashruntime`
Kawa	`gnu/kawa`
	`gnu/q2`
Unity	`com/unity3d`
AppMK	`com/appmk`
Apache Thrift	`org/apache/thrift`
OpenTK	`opentk/platform`
	`opentk_1_0/platform`
App.Yet	`com/appyet`
AndEngine	`org/andengine`
Conduit	`com/conduit/app/core`
AppPlant / Cordova	`de/appplant/cordova`
UbiKod	`com/ubikod`
AppMakr	`com/appmakr`
AndEngine	`org/anddev/andengine`
Mozilla JS Library	`org/mozilla/javascript`

Table A.1: Detection of cross–platform apps by checking the source code tree layout against the tree layouts of known frameworks.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: A System for Large-Scale Machine Learning*, Symposium on Operating Systems Design and Implementation – OSDI 2016, USENIX Association, 2016, pages 265–283 (cited on page 18).

[2] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang, *Clustering Mobile Apps Based on Mined Textual Features*, Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016, ACM, 2016, 38:1–38:10, ISBN 978-1-4503-4427-2 (cited on pages 3, 25).

[3] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton, *Suggesting accurate method and class names*, Foundations of Software Engineering – FSE 2015, ACM, 2015, pages 38–49, ISBN 978-1-4503-3675-8 (cited on page 3).

[4] U. Alon, S. Brody, O. Levy, and E. Yahav, *code2seq: Generating Sequences from Structured Representations of Code*, 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, OpenReview.net, 2019 (cited on page 3).

[5] A. Aminordin, M. F. Abdollah, R. Yusof, and R. Ahmad, *Preliminary findings: Revising developer guideline using word frequency for identifying apps miscategorization*, Proceedings of the Second International Conference on the Future of ASEAN (ICoFA) 2017–Volume 2, Springer, 2018, pages 123–131 (cited on page 25).

[6] P. Andriotis, M. A. Sasse, and G. Stringhini, *Permissions snapshots: Assessing users' adaptation to the Android runtime permission model*, Workshop on Information Forensics and Security – WIFS 2016, IEEE, 2016, pages 1–6, ISBN 978-1-5090-1138-4 (cited on page 11).

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, *DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket*, Network and Distributed System Security Symposium – NDSS 2014, The Internet Society, 2014 (cited on page 1).

[8] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, *Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T)*, 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, IEEE Computer Society, 2015, pages 669–679, ISBN 978-1-5090-0025-8 (cited on page 12).

[9] Y. Bengio, R. Ducharme, and P. Vincent, *A Neural Probabilistic Language Model*, Neural Information Processing Systems – NIPS 2000, MIT Press, 2000, pages 932–938 (cited on page 15).

[10] J. Bergstra and Y. Bengio, *Random Search for Hyper-Parameter Optimization*, Journal of Machine Learning Research, volume 13, pages 281–305, 2012 (cited on page 17).

[11] T. Brants, *Natural Language Processing in Information Retrieval*, Computational Linguistics in the Netherlands 2003, December 19, Centre for Dutch Language and Speech, University of Antwerp, series Antwerp papers in linguistics, volume 111, University of Antwerp, 2003 (cited on page 14).

[12] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, *EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework*, Network and Distributed System Security Symposium – NDSS 2015, The Internet Society, 2015 (cited on pages 12, 47).

[13] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, *Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale*, USENIX Security Symposium 2015, USENIX Association, 2015, pages 659–674 (cited on page 1).

[14] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, *Natural Language Processing (Almost) from Scratch*, Journal of Machine Learning Research, volume 12, pages 2493–2537, 2011 (cited on page 16).

[15] Y. Feng, L. Chen, A. Zheng, C. Gao, and Z. Zheng, *Ac-net: Assessing the consistency of description and permission in android apps*, IEEE Access, volume 7, pages 57 829–57 842, 2019 (cited on pages 4, 26).

[16] B. Gelman, B. Hoyle, J. Moore, J. Saxe, and D. Slater, *A Language-Agnostic Model for Semantic Source Code Labeling*, CoRR, volume abs/1906.01032, 2019 (cited on page 3).

[17] H. Gonzalez, A. A. Kadir, N. Stakhanova, A. J. Alzahrani, and A. A. Ghorbani, *Exploring reverse engineering symptoms in Android apps*, European Workshop on System Security – EUROSEC 2015, ACM, 2015, 7:1–7:7, ISBN 978-1-4503-3479-2 (cited on page 9).

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org` (cited on pages 17–18, 20, 22–23, 30).

[19] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, *RiskRanker: scalable and accurate zero-day android malware detection*, Mobile Systems – MobiSys 2012, ACM, 2012, pages 281–294, ISBN 978-1-4503-1301-8 (cited on page 1).

[20] G. E. Hinton and S. T. Roweis, *Stochastic Neighbor Embedding*, Neural Information Processing Systems – NIPS 2002, MIT Press, 2002, pages 833–840, ISBN 0-262-02550-7 (cited on page 21).

[21] K. Hornik, *Approximation capabilities of multilayer feedforward networks*, Neural Networks, volume 4, pages 251–257, 1991 (cited on page 22).

[22] A. G. Jivani *et al.*, *A comparative study of stemming algorithms*, Int. J. Comp. Tech. Appl, volume 2, number 6, pages 1930–1938, 2011 (cited on page 13).

[23] K. S. Jones, *A statistical interpretation of term specificity and its application in retrieval*, Journal of Documentation, volume 60, pages 493–502, 2004 (cited on page 14).

[24] H. J. Kelley, *Gradient theory of optimal flight paths*, Ars Journal, volume 30, number 10, pages 947–954, 1960 (cited on page 17).

[25] Y. Kim, *Convolutional Neural Networks for Sentence Classification*, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL, ACL, 2014, pages 1746–1751, ISBN 978-1-937284-96-1 (cited on pages 19, 63).

[26] E. Kowalczyk, A. M. Memon, and M. B. Cohen, *Piecing together app behavior from multiple artifacts: A case study*, 26th IEEE International Symposium on Software Reliability Engineering,

ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015, IEEE Computer Society, 2015, pages 438–449, ISBN 978-1-5090-0406-5 (cited on page 26).

[27] N. Kriegeskorte, *Deep neural networks: A new framework for modeling biological vision and brain information processing*, Annual Review of Vision Science, volume 1, number 1, pages 417–446, 2015. doi:10.1146/annurev-vision-082114-035447 (cited on page 19).

[28] K. Kuznetsov, V. Avdiienko, A. Gorla, and A. Zeller, *Checking app user interfaces against app descriptions*, Proceedings of the International Workshop on App Market Analytics, WAMA@SIGSOFT FSE, Seattle, WA, USA, November 14, 2016, ACM, 2016, pages 1–7, ISBN 978-1-4503-4398-5 (cited on pages 3, 26).

[29] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, *Backpropagation Applied to Handwritten Zip Code Recognition*, Neural Computation, volume 1, pages 541–551, 1989 (cited on page 18).

[30] B. Li, Y. Zhang, J. Li, W. Yang, and D. Gu, *AppSpear: Automating the hidden-code extraction and reassembling of packed android malware*, Journal of Systems and Software, volume 140, pages 3–16, 2018 (cited on page 9).

[31] Q. Li and X. Li, *Android Malware Detection Based on Static Analysis of Characteristic Tree*, 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2015, Xi'an, China, September 17-19, 2015, IEEE Computer Society, 2015, pages 84–91, ISBN 978-1-4673-9200-6 (cited on page 9).

[32] Y. Li, J. Jang, X. Hu, and X. Ou, *Android Malware Clustering Through Malicious Payload Mining*, Recent Advances in Intrusion Detection – RAID 2017, series LNCS, volume 10453, Springer, 2017, pages 192–214, ISBN 978-3-319-66331-9 (cited on page 3).

[33] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, *ANDRUBIS - 1, 000, 000 Apps Later: A View on Current Android Malware Behaviors*, Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS@ESORICS 2014, Wroclaw, Poland, September 11, 2014, IEEE, 2014, pages 3–17, ISBN 978-1-4799-8308-7 (cited on page 3).

[34] Z. C. Lipton, *The Mythos of Model Interpretability*, CoRR, volume abs/1606.03490, 2016 (cited on page 24).

[35] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie, *A Large-Scale Empirical Study on Android Runtime-Permission Rationale Messages*, 2018 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2018, Lisbon, Portugal, October 1-4, 2018, IEEE Computer Society, 2018, pages 137–146, ISBN 978-1-5386-4235-1 (cited on page 11).

[36] D. Low, *Protecting java code via code obfuscation*, Crossroads, volume 4, number 3, pages 21–23, 1998 (cited on page 10).

[37] D. L. B. Lulu and T. Kuflik, *Functionality-based clustering using short textual description: helping users to find apps installed on their mobile device*, 18th International Conference on Intelligent User Interfaces, IUI 2013, Santa Monica, CA, USA, March 19-22, 2013, ACM, 2013, pages 297–306, ISBN 978-1-4503-1965-2 (cited on page 3).

[38] S. M. Lundberg and S. Lee, *A Unified Approach to Interpreting Model Predictions*, Neural Information Processing Systems – NIPS 2017, 2017, pages 4768–4777 (cited on page 24).

[39] L. v. d. Maaten and G. Hinton, *Visualizing data using t-sne*, Journal of machine learning research, volume 9, number Nov, pages 2579–2605, 2008 (cited on page 20).

[40] V. Markovtsev and E. Kant, *Topic modeling of public repositories at scale using names in source code*, CoRR, volume abs/1704.00135, 2017 (cited on page 3).

[41] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, *The Android Platform Security Model*, CoRR, volume abs/1904.05572, 2019 (cited on page 10).

[42] O. Melamud, D. McClosky, S. Patwardhan, and M. Bansal, *The Role of Context Types and Dimensionality in Learning Word Embeddings*, NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016, The Association for Computational Linguistics, 2016, pages 1030–1040, ISBN 978-1-941643-91-4 (cited on page 17).

[43] G. Meng, Y. Xue, J. K. Siow, T. Su, A. Narayanan, and Y. Liu, *AndroVault: Constructing Knowledge Graph from Millions of Android Apps for Automated Analysis*, CoRR, volume abs/1711.07451, 2017 (cited on page 26).

[44] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, *Distributed Representations of Words and Phrases and their Compositionality*, Neural Information Processing Systems – NIPS 2013, 2013, pages 3111–3119 (cited on page 15).

[45] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, *WHYPER: Towards Automating Risk Assessment of Mobile Applications*, USENIX Security Symposium 2013, USENIX Association, 2013, pages 527–542, ISBN 978-1-931971-03-4 (cited on pages 4, 26).

[46] J. Pennington, R. Socher, and C. D. Manning, *Glove: Global Vectors for Word Representation*, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL, ACL, 2014, pages 1532–1543, ISBN 978-1-937284-96-1 (cited on page 16).

[47] M. F. Porter, *An algorithm for suffix stripping*, Program, volume 14, pages 130–137, 1980 (cited on page 29).

[48] *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, ACL, ISBN 978-1-937284-96-1.

[49] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, *AutoCog: Measuring the Description-to-permission Fidelity in Android Applications*, Conference on Computer and Communications Security – CCS 2014, ACM, 2014, pages 1354–1365, ISBN 978-1-4503-2957-6 (cited on pages 4, 25).

[50] J. Ramos *et al.*, *Using tf-idf to determine word relevance in document queries*, Proceedings of the first instructional conference on machine learning, Piscataway, NJ, volume 242, 2003, pages 133–142 (cited on page 14).

[51] M. T. Ribeiro, S. Singh, and C. Guestrin, *Model-Agnostic Interpretability of Machine Learning*, CoRR, volume abs/1606.05386, 2016 (cited on page 24).

[52] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain.* Psychological review, volume 65, number 6, page 386, 1958 (cited on page 17).

[53] B. G. Ryder, *Constructing the Call Graph of a Program*, IEEE Trans. Software Eng., volume 5, pages 216–226, 1979 (cited on page 12).

[54] A. A. Samra and O. A. Ghanem, *Analysis of Clustering Technique in Android Malware Detection*, Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2013, Taichung, Taiwan, July 3-5, 2013, IEEE Computer Society, 2013, pages 729–733, ISBN 978-0-7695-4974-3 (cited on page 3).

[55] C. Song, F. Liu, Y. Huang, L. Wang, and T. Tan, *Auto-encoder Based Data Clustering*, Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications - 18th Iberoamerican Congress, CIARP 2013, Havana, Cuba, November 20-23, 2013, Proceedings, Part I, series LNCS, volume 8258, Springer, 2013, pages 117–124, ISBN 978-3-642-41821-1 (cited on page 20).

[56] T. Takahashi and T. Ban, *Android application analysis using machine learning techniques*, in *AI in Cybersecurity*, Springer, 2019, pages 181–205 (cited on page 26).

[57] B. Trstenjak, S. Mikac, and D. Donko, *Knn with tf-idf based framework for text categorization*, Procedia Engineering, volume 69, pages 1356–1364, 2014, 24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013, ISSN 1877-7058. doi:https://doi.org/10.1016/j.proeng.2014.03.129. `http://www.sciencedirect.com/science/article/pii/S1877705814003750` (cited on page 15).

[58] S. Vakulenko, O. Müller, and J. vom Brocke, *Enriching iTunes App Store Categories via Topic Modeling*, Proceedings of the International Conference on Information Systems - Building a Better World through Information Systems, ICIS 2014, Auckland, New Zealand, December 14-17, 2014, Association for Information Systems, 2014, ISBN 978-0-615-15788-7 (cited on page 3).

[59] M. L. Vásquez, A. Holtzhauer, and D. Poshyvanyk, *On automatically detecting similar Android apps*, International Conference on Program Comprehension – ICPC 2016, IEEE Computer Society, 2016, pages 1–10, ISBN 978-1-5090-1428-6 (cited on page 25).

[60] N. Viennot, E. Garcia, and J. Nieh, *A measurement study of google play*, Measurement and Modeling of Computer Systems – SIGMETRICS 2014, ACM, 2014, pages 221–233, ISBN 978-1-4503-2789-3 (cited on page 31).

[61] S. Vijayarani, M. J. Ilamathi, and M. Nithya, *Preprocessing techniques for text mining-an overview*, International Journal of Computer Science & Communication Networks, volume 5, number 1, pages 7–16, 2015 (cited on page 13).

[62] J. J. Webster and C. Kit, *Tokenization As The Initial Phase In NLP*, 14th International Conference on Computational Linguistics, COLING 1992, Nantes, France, August 23-28, 1992, 1992, pages 1106–1110 (cited on page 13).

[63] M. Weiser, *Program Slicing*, IEEE Trans. Software Eng., volume 10, pages 352–357, 1984 (cited on page 12).

[64] W. Yin, K. Kann, M. Yu, and H. Schütze, *Comparative Study of CNN and RNN for Natural Language Processing*, CoRR, volume abs/1702.01923, 2017 (cited on page 19).