

# **ALPS - A Novel Approach for the Classification of Android Applications**

Christiane Schwarzl B.Sc.

## **Master's Thesis**

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

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

Graz, 12 Nov 2020



## **AFFIDAVIT**

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

---

Date, Signature



## Abstract

The Android platform's increasing popularity also led to a growing number of applications in Google Play. With such a considerable number of applications, it becomes challenging for users to discover specific apps relevant to their interests and needs, as well as for developers to reach their intended user base. Furthermore, it also provides additional challenges for managing the app store and the apps it offers. To provide a better overview, Google Play introduced various app categories. However, currently, the categorization is performed manually by the app developers, which leads to the possibility of misclassification. While manual categorization by experts could improve this, it would be highly time-consuming due to the substantial number of apps. On the other hand, automated approaches generally focus on detecting malicious applications.

In this thesis, we introduce a novel approach for the automated classification of Android applications based on context-sensitive features extracted from the applications' disassembled code. Our approach applies state-of-the-art techniques from the fields of natural language processing and deep learning to effectively analyze and classify Android applications based on their code. We generate code sequences based on the disassembled application's abstract syntax trees, enabling us to utilize the code's underlying structural information. By creating context-sensitive vector mappings on these sequences, we create highly abstract features that can capture an app's source code's essence far better than simply using an app's instructions as features. We compare two deep learning techniques commonly used for classifying sequence data. While our first model applies long short-term memory cells, our second model is based on convolutional neural network layers.

Our implementation, named Android Language Processing System (ALPS), consists of a Java-based component for disassembly and feature generation, as well as a Python-based component for model training and evaluation. Additionally, we create an interactive dashboard that enables users to upload Android applications and receive category predictions and their probabilities based on our best performing model.

To evaluate our approach, we compile a dataset of 45,960 Android applications from 8 representative Google Play categories. We use 70% of the data for training, 15% as a validation set for early stopping, and evaluate our models on the remaining 15% of the data. We compare our models using various metrics and demonstrate that ALPS provides a meaningful approach to classifying Android applications.

**Keywords:** Android, Classification, Deep Learning, Natural Language Processing, Abstract Syntax Tree



## Kurzfassung

Die zunehmende Popularität der Android-Plattform führte auch zu einer wachsenden Zahl von Anwendungen in Google Play. Bei einer so beträchtlichen Anzahl von Applikationen wird es für die Nutzer schwierig, Anwendungen zu entdecken, die ihren Interessen und Bedürfnissen entsprechen, und für die Entwickler, ihre beabsichtigte Nutzerbasis zu erreichen. Darüber hinaus bietet es auch zusätzliche Herausforderungen für die Verwaltung des App-Stores und der angebotenen Anwendungen. Um einen besseren Überblick zu bieten, führte Google Play verschiedene App-Kategorien ein. Derzeit erfolgt die Kategorisierung jedoch manuell durch die App-Entwickler, was zu der Möglichkeit einer Fehlerklassifizierung führt. Eine manuelle Kategorisierung durch Experten könnte dies zwar verbessern, wäre aber aufgrund der beträchtlichen Anzahl von Applikationen sehr zeitaufwändig. Automatisierte Ansätze, hingegen, fokussieren sich in der Regel auf die Erkennung bössartiger Anwendungen.

In dieser Arbeit stellen wir einen neuartigen Ansatz für die automatisierte Klassifizierung von Android-Anwendungen vor, der auf kontextsensitiven Merkmalen basiert, die aus dem disassemblierten Code der Anwendungen extrahiert werden. Unser Ansatz wendet modernste Techniken aus den Bereichen des Natural Language Processings und des Deep Learnings an, um Android-Anwendungen auf der Grundlage ihres Codes effektiv zu analysieren und zu klassifizieren. Wir generieren Code-Sequenzen auf der Grundlage der Abstract Syntax Trees der disassemblierten Anwendung, dies ermöglicht, dass wir die dem Code zugrunde liegenden Strukturinformationen nutzen können. Durch die Erstellung kontextsensitiver Vektorzuordnungen auf diesen Sequenzen erstellen wir hochgradig abstrakte Features, die das Wesen des Quellcodes einer Anwendung viel besser erfassen können als die simple Verwendung der Instruktionen einer Applikation als Features. Wir vergleichen zwei Deep Learning Techniken, die üblicherweise zur Klassifizierung von Sequenzdaten verwendet werden. Während unser erstes Modell mit Long Short-Term Memory Zellen arbeitet, basiert unser zweites Modell auf Convolutional Neural Network Layers.

Unsere Implementierung, genannt Android Language Processing System (ALPS), besteht aus einer Java-basierten Komponente zur Disassemblierung und Feature-Generierung sowie einer Python-basierten Komponente zum Modelltraining und -evaluierung. Darüber hinaus erstellen wir ein interaktives Dashboard, mit dem Benutzer Android-Anwendungen hochladen und Kategorievorhersagen und deren Wahrscheinlichkeiten auf der Grundlage unseres besten Modells erhalten können.

Um unseren Ansatz zu evaluieren, erstellen wir einen Datensatz von 45.960 Android-Anwendungen aus 8 repräsentativen Google Play-Kategorien. Wir verwenden 70% der Daten für das Training, 15% als Validierungssatz für Early Stopping und evaluieren unsere Modelle anhand der restlichen 15% der Daten. Wir vergleichen unsere Modelle anhand verschiedener Metriken und zeigen, dass ALPS einen sinnvollen Ansatz zur Klassifizierung von Android-Anwendungen bietet.

**Schlüsselwörter:** Android, Klassifikation, Deep Learning, Natural Language Processing, Abstract Syntax Tree





# Acknowledgements

First, I would like to thank my supervisor Johannes Feichtner for his exceptional guidance. Without his assistance, motivation, and enthusiasm, I would not have achieved this feat. I want to thank him for ensuring I stay on track, always providing quick and insightful feedback, enduring my unorthodox work style, and never losing patience with me even when I face all kinds of challenges.

I'm thankful for all my friends and colleagues who supported me through all kinds of ups and downs while studying and writing my thesis. My special thanks go to Şeyda, Urška, Sindy, Elli, Oliver, Lukas, Stefan, Richard, and Georg.

Finally, I want to thank my boyfriend, David. I cannot put into words how grateful I am for his love and support.

Christiane Schwarzl



# Contents

<b>Contents</b>	<b>iv</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Listings</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Outline . . . . .	3
<b>2 Background</b>	<b>5</b>
2.1 Android . . . . .	5
2.1.1 Android Applications . . . . .	5
2.2 Smali & Baksmali . . . . .	7
2.2.1 Registers . . . . .	7
2.2.2 Class File Structure . . . . .	8
2.2.3 Abstract Syntax Tree . . . . .	9
2.3 Machine Learning . . . . .	10
2.3.1 Text Classification . . . . .	11
2.3.2 Word Embedding . . . . .	12
2.3.3 Artificial Neural Networks . . . . .	12
2.3.4 Recurrent Neural Networks (RNN) . . . . .	12
2.3.5 Convolutional Neural Networks . . . . .	14
2.3.6 Word Embedding Layer . . . . .	16
2.3.7 Regularization . . . . .	17
2.3.8 Activation Functions . . . . .	18
<b>3 Related Work</b>	<b>19</b>
3.1 Android App Classification . . . . .	19
3.2 Natural Language Processing for Source Code . . . . .	20

<b>4</b>	<b>Approach</b>	<b>23</b>
4.1	Code2seq Approach . . . . .	23
4.2	Feature Generation . . . . .	26
4.3	Model Architecture . . . . .	29
4.3.1	Dataset Generation . . . . .	29
4.3.2	Model Generation . . . . .	31
4.3.3	Model 1: RNN with LSTM cells . . . . .	33
4.3.4	Model 2: CNN . . . . .	34
<b>5</b>	<b>Implementation</b>	<b>37</b>
5.1	Architecture . . . . .	37
5.2	baksmali: Path Generation from APK . . . . .	38
5.3	feature: Path & Dictionary Generation . . . . .	44
5.4	training: Model Training . . . . .	44
5.4.1	config: Configuration . . . . .	44
5.4.2	dataset: Dataset Generation . . . . .	45
5.4.3	plotting: Plotting Functions . . . . .	46
5.4.4	models: Model Creation & Training . . . . .	46
5.5	Dashboard . . . . .	47
<b>6</b>	<b>Results</b>	<b>49</b>
6.1	Dataset . . . . .	49
6.2	Configuration . . . . .	51
6.3	Models . . . . .	52
6.3.1	LSTM model . . . . .	54
6.3.2	CNN model . . . . .	55
6.3.3	Evaluation . . . . .	57
6.4	Dashboard . . . . .	58
6.5	Discussion . . . . .	59
<b>7</b>	<b>Conclusion</b>	<b>63</b>
	<b>Bibliography</b>	<b>65</b>

# List of Figures

2.1	Example for a class file structure of Smali files. . . . .	8
2.2	Example for an Abstract Syntax Tree generated from Smali code . . . . .	9
2.3	Overview of machine learning techniques with examples. . . . .	10
2.4	Training and evaluation process of text classification. . . . .	11
2.5	Example of a Recurrent neural network that predicts the next word of a sentence. . . . .	13
2.6	Overview of an LSTM cell. . . . .	14
2.7	Example of a convolution of the sentence "I like ice cream" with a kernel $w$ of size 2. . . . .	15
2.8	Example of a classification with 1 CNN layer and 4 kernels, 2 kernel of size 3 and 2 kernels of size 2. . . . .	16
2.9	Different approaches for solving a text classification task. . . . .	17
2.10	Example for early stopping, where an additional validation set is used to avoid overfitting. . . . .	17
4.1	Flow Chart of our approach: Steps in yellow: Android processing, orange: source code processing, red: code2seq + text classification . . . . .	23
4.2	Example by Alon et al. [3] of two Java method that have the same functionality but different implementations. . . . .	24
4.3	Overview of the parts of the model architecture proposed by Alon et al.[3] . . . . .	25
4.4	Example structure for an APK file. The files used for our approach are the AndroidManifest.xml and the classes.dex. They are marked in bold. . . . .	27
4.5	Example for two AST root paths. The first is marked in orange and the second is marked in red. Both represent a leaf node connecting to the root of the method. . . . .	28
4.6	Example for two AST paths. The first path is marked in orange and the second path is marked in red. . . . .	28
4.7	Flow Chart of the model training: The steps in yellow mark the pre-processing steps, the steps in orange mark the model training and evaluation, and the steps in red mark evaluation and plotting. . . . .	29
4.8	Flow Chart of model evaluation: The steps in yellow mark the pre-processing steps, the steps in orange mark the model evaluation, and the steps in red mark result representation. . . . .	30
4.9	Overview of the parts of the model architecture that is shared between both models. . . . .	32
4.11	Overview of the Model Architecture where an RNN with LSTM cells is used for processing the paths. . . . .	33
4.12	Overview of the Model Architecture where convolutional layers are used for processing the paths. . . . .	35
4.10	Overview of how the dimensions of the input changes while generating the model. . . . .	36
5.1	Overview of the components of our implementation. . . . .	37

5.2	Flow chart showing the steps for disassembling an APK file for a given class and generating its AST tree. . . . .	39
5.3	AST Generation . . . . .	40
5.4	AST Path Generation . . . . .	41
5.5	This Figure shows the generation of all tree root paths for the methods of a given Smali files. The recursion step is marked in blue. . . . .	42
5.6	AST Tree Root Path Generation . . . . .	43
6.1	The distribution of our eight representative class for the training dataset, the validation dataset, and the test dataset. . . . .	50
6.2	Training and validation metrics for the LSTM model. . . . .	54
6.3	Confusion matrices of test set for the LSTM model. . . . .	55
6.4	Training and validation metrics for the CNN model. . . . .	56
6.5	Confusion matrices of the test for the CNN model. . . . .	56
6.6	Visualization of the correct prediction of an Android application. . . . .	58
6.7	Dashboard a correct prediction for an Android application. . . . .	60
6.8	Visualization of the incorrect prediction of an Android application. . . . .	61

# List of Tables

2.1	Overview of the different app components . . . . .	6
2.2	Definition of primitive data types in smali . . . . .	7
2.3	Definition of reference types in smali . . . . .	7
2.4	Class file structure . . . . .	8
2.5	Overview of different activation functions . . . . .	18
4.1	Example for the lookup for path tokens. . . . .	30
6.1	Class weights for each of the categories. . . . .	51
6.2	Configuration values for the different configuration setting. . . . .	52
6.3	Dimensions and number of parameters for all layers that are shared between our models. . . . .	53
6.4	Comparison of the test accuracies of the LSTM model and the CNN model. . . . .	57





# List of Listings

2.1	Example for a Manifest XML file. . . . .	6
2.2	Smali Syntax: Method call . . . . .	8
2.3	Snippet of a Smali file generated from a real Android Application. . . . .	9



# Chapter 1

## Introduction

Mobile applications have become a hugely profitable market over the last years, and Android has the largest share with 87%<sup>1</sup>. To install and manage Android applications, Google provides their app market *Google Play*<sup>2</sup> which is shipped by default with Android devices. There are currently 2.96 million applications available on Google Play<sup>3</sup>. With such a massive number of applications, it becomes difficult for users to find applications that fit their needs best. Moreover, it is challenging for developers to reach users and gain popularity. With such a large number of available apps, it is also not easy to manage the store, keep track of all offered apps, and handle new applications. Google has to make sure to pick applications that have the potential for popularity and cater to their customers' diverse needs.

To help with this problem, Google Play introduced 33 categories for applications and 17 categories for games. Currently, developers decide their application's category without support from Google Play. However, it can be challenging to choose from 33 categories, especially if an app has many functionalities, and can lead to misclassification. We could solve this problem by providing assistance in selecting the best category based on which apps are currently on Google Play. Providing manual assistance is complicated, with the sheer number of applications. Therefore, we provide an approach that can automatically classify Android applications, which we have named the Android Language Processing System (ALPS).

ALPS can not only assist developers in choosing the correct categories but is also helpful for Google Play. It can be used to investigate the quality of the current categories and which apps they contain. For example, if two categories are often confused with each other, they could be merged into one category if they're too similar. On the other hand, if one category is often misclassified with different categories, it might be too diverse. It could then be split into multiple categories or integrated with other classes. Our approach can also be used to assign categories automatically, which would remove the requirement for manual assignment. This automation would result in a more consistent classification and help users find the applications in finding useful apps.

Furthermore, ALPS can be used to help detect malicious Android applications. For malware detection, it is applied to malware datasets instead of Android applications from the Google Play. Instead of classifying Google Play categories, it would decide whether an app is malign or benign. Another possible application is the classification of malware families, which would provide information on whether the app is malicious and what type of malware it is.

Common features for Android app classification are the permissions that an app requires because they represent access to essential system features, such as the camera or file system. The assumption behind

---

<sup>1</sup><https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/>

<sup>2</sup><https://play.google.com/store/apps>

<sup>3</sup><https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>

using permissions is that similar applications require similar permissions. For example, most messaging applications might require the camera permission to take pictures and the file system to send saved files. However, one crucial aspect that should be considered is that developers set permissions by themselves. Moreover, there is no indication of how vital permission is for an app. It might be critical to its operation, only required once, optional, or not at all. Furthermore, permissions might not necessarily correlate with application categories. A majority of applications might need similar permissions, such as the camera, file system, or location, limiting the set of significant permissions. Similarly, some permissions might be too rare to be substantial, limiting an already small group of permissions.

Another possibility is to use the application's source code as a base for the classification. The source code captures an application's functionality in more detail than permissions, as these are limited. However, source code is also significantly more extensive than a list of permissions. So it is essential to detect the code parts that define the app's functionality the most. Most other approaches work on a function or code snippet base. To our knowledge, there are currently no approaches that use an Android app's source code to defer information such as its category. Therefore, we want to investigate whether an app's source code provides a useful feature for classification.

Our approach investigates whether an app's source code provides meaningful features for classification. We abstract the source code to represent its functionalities to find patterns among similar apps. To create this abstraction, we use word embeddings introduced by Mikolov et al. [20]. Their purpose is to abstract words from natural text to its meaning. This technique maps terms to vectors such that words occurring in a similar context are mapped to similar vectors. In our approach, we treat source code as natural text and apply a similar approach to receive word embeddings for the app's code. Since Android apps are compiled to Dalvik byte code and packaged into APK files, we need to unpack them again and convert them to a better readable form for humans and thus closer to natural text. We could decompile the Dalvik byte code to Java code, but most decompilers provide limited Dalvik byte code support. Instead, we use the Baksmali<sup>4</sup> disassembler, which decompiles Dalvik byte code Smali code. Smali code is specifically designed for the Android runtime environment and supports the Dalvik byte code's full functionality.

One possibility would be to treat the entire code as one large document and apply doc2vec, introduced by Mikolov et al [17]. It extends on the original process by also generating context-sensitive vectors for entire documents. However, converting Smali code is not a straightforward task since it is object-oriented, which means that the code is separated into classes with fields, methods, etc. The order of the class' components is irrelevant. Since doc2vec takes a word's context into account to generate word embeddings, the ordering of classes and methods can substantially impact the resulting embeddings and, therefore, our results. Instead, we utilize abstract syntax trees (AST), which are a tree representation of the source code. Moreover, the AST provides us with additional information about the code elements such as their type, e.g., method, variable, method call. Using the abstract syntax tree as input for a model ensures that the proper context is chosen. However, most algorithms are not designed to take tree structures as input, especially not when the size and structure vary enormously.

Alon et al. [3] face the same issues in their approach where they predict a function's name based on the function's code's AST. They solved this issue by taking all paths that lead from one leaf to another leaf of the AST tree as input to the model. We take a similar approach by generating all possible paths for each Smali code method and combining them as input to a deep learning model to predict the app's class. To ensure the best performance, we compare different model architectures to find the one which best suits our task. We will use embedding layers, fully connected layers, recurrent layers, convolutional layers, and dropout layers for the model architecture. To train these models, we create a dataset of apps from the official Google Play provided by the AndroZoo project[2] and the PlayDrone project[28].

This thesis describes our approach, the Android Language Processing System (ALPS), which can

---

<sup>4</sup><https://github.com/JesusFreke/smali>

effectively analyze and classify Android applications using modern techniques from deep learning and natural language processing. ALPS does not only investigate a new problem, but it does so using a novel approach. Most related works focus on particular parts of an Android app like Android API calls or permissions. We, on the other hand, look at entire classes to receive meaningful features. We do so by adapting Alon et al. [3]’s approach for our classification problem and combining it with two different models - an LSTM and a CNN model. Furthermore, we provide an interactive interface where users can upload new applications to receive predictions.

## 1.1 Outline

In this section, we describe the outline of this thesis. Chapter 2 provides the background knowledge that is the basis for our approach. This chapter is divided into three main sections: Android, Smali, and Machine Learning. The first section provides an overview of the Android system with a focus on Android applications. We give more detail on the Android Manifest XML file and the Dalvik byte code as they are essential components of an Android app to our approach. In the second section, we introduce the Smali language and its characteristics. Furthermore, we describe abstract syntax trees in this section as we generate our features from them. In the last section, we introduce the basic concepts of machine learning. Moreover, we provide further detail on text classification, a supervised classification approach on natural language processing. Word embeddings, which are commonly used in natural language processing, are also defined in this section. We continue with introducing artificial neural networks and two of its types from natural language processing: recurrent neural networks and convolutional neural networks. Additionally, we describe how word embeddings are combined with artificial neural networks. Finally, we present an overview of different regularization techniques and compare various activation functions that are commonly used for neural networks.

Chapter 3 describes works that are related to our problem area and approach. Similar works are classified into two sections. The first section describes methods on Android app classification, which can be further divided into works with different features and approaches, works with similar approaches, and works with similar features. The second section covers works on natural language processing using source code as input. These approaches range from general text classification tasks, like assigning names and descriptions to functions or predicting the source code author.

The following Chapter 4 provides an overview of our approach, which is influenced by Alon et al.’s approach [4]. That is why we begin with a detailed description of their approach and then describe which changes we make to adapt to our problem. Next, we explain how we generate our features and dataset. Lastly, we define the architecture of our machine learning models that use for classification.

In Chapter 5, we introduce the implementation of our approach and highlight its features and functionality. Furthermore, we describe its architecture which is divided into the following components: `baksmali`, `feature`, `models`, and `dashboard`. We provide a detailed description of our model’s input generation based on the input APK file in the first two sections. The input includes the generation of AST paths and lookup dictionaries for the path and terminal tokens. The input description is followed by explaining our dataset generation from our input and our model’s preprocessing steps. In the last section, we describe our dashboard implementation, which provides an interface for generating prediction based on a given APK file.

Chapter 6 provides a detailed description of our results, which starts with an overview of the dataset that we create. We continue with the parameter settings that we use to achieve our results. Next, we give a description and comparison of the results of our two models, the LSTM model, and the CNN model. Subsequently, we illustrate the results of our dashboard and provide several examples. We conclude this chapter with a discussion of our results.

We conclude this thesis in Chapter 7. In this chapter, we summarize our work and discuss the contribution of our approach. Finally, we provide suggestions for future work.



## Chapter 2

# Background

In this chapter, we describe background information that is necessary for this thesis. Section 2.1 provides an introduction to the Android platform with a focus on the Dalvik bytecode and the Android Manifest XML file. The following Section 2.2 describes Baksmali, a disassembler that converts Dalvik bytecode to Smali language, a language designed to provide an easier to understand representation of the Dalvik bytecode. The disassembler makes use of abstract syntax trees, which are described in the same section. Section 2.3 gives an overview of machine learning, a set of data-driven algorithms that perform tasks without being explicitly programmed. It provides more in-depth information about techniques that are geared towards processing natural text such as word embeddings. The section then continues with artificial neural networks, a subset of machine learning techniques, and different variations such as recurrent neural networks and convolutional neural networks.

### 2.1 Android

Android is a mobile operating system (OS) developed by Google and with a market share of 88% by far the most popular mobile operating systems on the market<sup>1</sup>. It is primarily deployed on mobile devices. However, derivatives are also available for televisions, wearables, and more. Android is continuously updated with major releases occurring every year. These updates provide new features, optimizations, and bug fixes. The code is open-source and licensed under the Apache license, allowing users to modify and distribute it freely. Because of this, it has become common practice for mobile manufacturers to ship their devices with their own Android derivative.

#### 2.1.1 Android Applications

In addition to the OS, devices come typically installed with several of Google's proprietary software such as Gmail and Google Calendar. It is also possible to create custom Android applications that can then be published in App Stores such as Google Playstore, which is usually pre-installed on Android devices.<sup>2</sup>.

Code for Android apps is typically written in Java, which is then compiled to Dalvik bytecode. This compiled code is saved in DEX files e.g., `classes.dex` and packaged into Android Package files (APK). The Android Runtime (ART) and its predecessor, the Dalvik Virtual Machine, are responsible for executing Dalvik bytecode, which was created specifically for the Android OS and designed to have a small memory footprint to be run even under considerable hardware limitations. The Virtual Machine follows a register-based architecture, unlike the stack-based Java Virtual Machine. However, like Java, Dalvik

---

<sup>1</sup><https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/>

<sup>2</sup><https://play.google.com/store/apps>

makes use of frames to store data during method invocations. These frames are created when a method is invoked and destroyed after it is completed. They are fixed in size upon creation and consist of the number of registers defined by the method and the data required to execute the method.

Apart from Dalvik bytecode, the APK also contains an Android Manifest XML file. It describes vital information about the Android app, such as its package name, main components, required permissions, required hardware, and software. The main components include activities, services, broadcast receivers, and providers. Activities represent screens of the applications; they render the screen's layout and handle user interactions for this screen. For example, a note app might have an activity to list all notes and a different activity for writing a note. Services, on the other hand, do not have a UI. Instead, they run in the background to execute long-running operations. A typical example would be playing music in the background. Broadcast receivers handle broadcast messages from the system or other applications. An example of a broadcast from the system would be that the battery is low. Content providers manage access to a central data repository so that different applications can access it. Table 2.1 summarizes of the previously described components. Listing 2.1 provides an example for a `AndroidManifest.xml` file.

Java class	XML element	Description
Activity	<activity>	Represents screen in app
Service	<service>	For longer-running operations without user interaction
BroadcastReceiver	<receiver>	Handles broadcast messages
ContentProvider	<provider>	Provides content to app from a central repository

**Table 2.1:** Overview of the different app components

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.example.app"
4     android:versionCode="1"
5     android:versionName="1.0" >
6     <uses-sdk android:minSdkVersion="16" android:targetSdkVersion="26">
7     </uses-sdk>
8     <uses-feature android:name="android.hardware.location.gps" android:required="
9         false">
10    </uses-feature>
11    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION">
12    </uses-permission>
13    <application android:name="com.example.app.activities.ExampleApplication" ... >
14        <activity android:name="com.example.app.MainActivity" ... >
15        </activity>
16        ...
17        <provider android:authorities="com.example.app.debug" android:name="android.
18            support.v4.content.FileProvider" ... >
19        </provider>
20        ...
21        <receiver android:name="com.example.app.notification.
22            ReminderNotificationReceiver">
23        </receiver>
24        ...
25        <service android:name="com.example.app.notification.SnoozeAlarmService">
26        </service>
27    </application>
28 </manifest>

```

**Listing 2.1:** Example for a Manifest XML file.



## 2.2 Smali & Baksmali

Smali/Baksmali is an assembler/disassembler suite for the DEX format. Baksmali converts DEX files to Smali code, which was designed to be a humanly readable version of the bytecode. It supports the full functionality of the DEX format and is partly based on the syntax of *Jasmin*<sup>3</sup> and *dedexer*<sup>4</sup> which fulfill a similar purpose as Smali. Dalvik bytecode has two classes of types, primitive types and reference types. Objects and arrays are reference types. All other types are primitive types, such as integers, floats, and strings. For primitive types, Smali uses the same representation as Dalvik bytecode, which is a single letter abbreviation. While Table 2.2 lists all primitive types and their abbreviations, Table 2.3 lists all reference types.

Type	Smali
void	V
boolean	Z
byte	B
short	S
char	C
int	I
long (64 bits)	J
float	F
double (64 bits)	D

**Table 2.2:** Definition of primitive data types in smali

Type	Smali	Java
object	Lpackage/name/ObjectName;	package.name.ObjectName
array	[I	int[]
methods	Lpackage/name/ObjectName;->MethodName(III)Z Lpackage/name/ObjectName;-> method(I[[IILjava/lang/String;)Ljava/lang/String;	void MethodName(int, int, int) String method(int, int[], String)
fields	Lpackage/name/ObjectName;->FieldName:Ljava/lang/String;	String fieldName;

**Table 2.3:** Definition of reference types in smali

### 2.2.1 Registers

Smali introduces two directives to specify the required number of registers in a method. The first one is the registers directive, which is defined as `.registers <n>` in Smali where `<n>` denotes the total number of registers in the method. The second one is the locals directive, it is written as `.locals <n>` where

<sup>3</sup><http://jasmin.sourceforge.net/>

<sup>4</sup><http://dedexer.sourceforge.net/>

<n> denotes the number of registers that are not a parameter. When addressing the registers, the same distinction is made, parameter registers have the prefix v and non-parameter registers p.

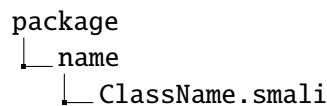
When a method is invoked, the parameters are placed in the last registers. The first parameter of a non-static method is always the reference to the object of the method. The method call example from Listing 2.2 has 2 integer parameters and the additional reference to the object, so if this method required 5 registers, it would require 3 parameter registers and 2 local registers.

```
1 LMyObject; ->callThis(II)V
```

**Listing 2.2:** Smali Syntax: Method call

### 2.2.2 Class File Structure

Smali class files begin with a full class definition where the package defines the folder structure of the output and the class name, which is the name of the smali file. For example, Lpackage/name/ClassName; would result in the folder structure depicted in Figure 2.1.



**Figure 2.1:** Example for a class file structure of Smali files.

After the class definition, a superclass or implemented interfaces may be defined. This is followed by the definition of class members, such as fields and methods. The Smali syntax of all before mentioned elements can be found in Table 2.4. Listing 2.3 provides an example for a Smali class file.

Type	Smali
Class definition	.class <modifiers>Lpackage/name/ClassName;
Super class	.super Lpackage/name/SuperClassName;
Interface	.implements Lpackage/name/InterfaceName;
Fields	.field <modifiers> fieldName
Methods	.method <modifiers> methodName(<parameters>)<return type>

**Table 2.4:** Class file structure

```

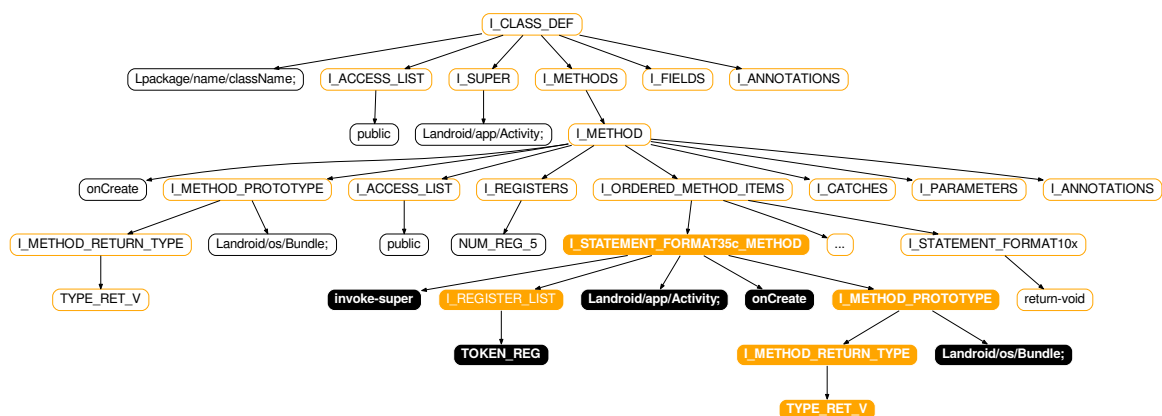
1 .class public Lpackage/name/className;
2 .super Landroid/app/Activity;
3
4
5 .method public onCreate(Landroid/os/Bundle;)V
6   .registers 5
7
8   invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
9
10  ...
11
12  return-void
13 .end method

```

**Listing 2.3:** Snippet of a Smali file generated from a real Android Application.

### 2.2.3 Abstract Syntax Tree

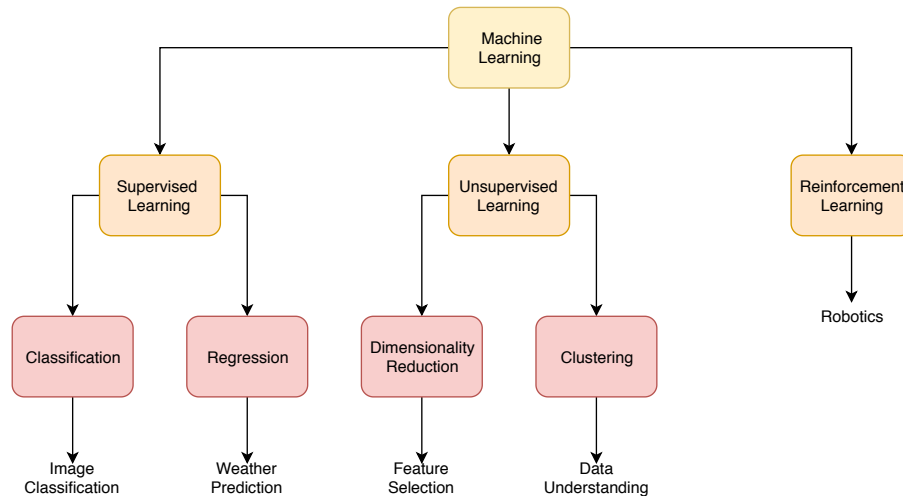
Abstract syntax trees (AST) provide a tree representation of the structure of source code and are commonly used by compilers. The trees generally focus on semantically important parts of the code, ignoring characters like semicolons and parentheses. The leaves of an abstract syntax tree are typically values defined by the programmer such as variable names and are called *terminals*. Nodes that are not leaves are referred to as *non-terminals* and are typically elements of the programming language such as expressions and conditional statements. Abstract Syntax Trees are also utilized by Smali, for example, when assembling Smali files into a DEX file. Figure 2.2 represents the AST for the code depicted in Listing 2.3. The method call in Line 8 is emphasized in the AST with a colored background for better visibility.



**Figure 2.2:** Example for an Abstract Syntax Tree generated from Smali code

## 2.3 Machine Learning

Machine learning describes a set of algorithms and techniques intended to perform tasks without being explicitly programmed. Instead, these algorithms are provided with data of interest, which they use to build mathematical models by inferring patterns within the data. These models can be used to gain deeper knowledge about the provided data or to make predictions about new, unseen data. As depicted in Figure 2.3, machine learning tasks can be roughly divided into three categories: supervised learning, unsupervised learning, and reinforcement learning.



**Figure 2.3:** Overview of machine learning techniques with examples.

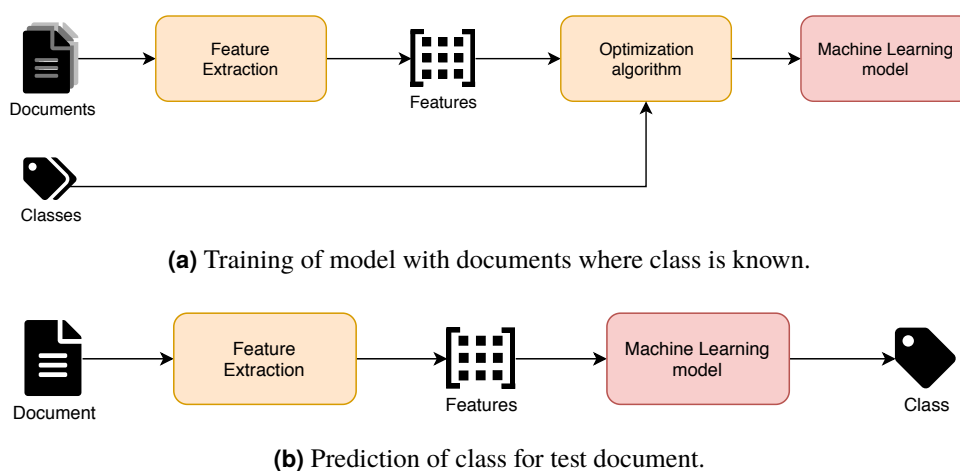
In *supervised learning*, given the input data  $X, Y$ , the goal is to approximate a function  $f$  such that the output of  $f(X)$  matches  $Y$  as closely as possible. This function can be used to predict the output  $y$  for input variables  $x$ , where  $y$  is unknown. It is not directly specified but learned by computing the loss of the function using an error function such as the *cross-entropy loss function*. This loss is used to update the function such that the loss is minimized. These steps are repeated until certain termination criteria are met, such as a minimum change in loss. A typical example of a task that is solved by supervised learning is the recognition of handwritten digits. This is a common problem for the automatic sorting of letters by zip code. Given a set of images of handwritten digits and the digits they represent, a function is learned to predict digits for new, unseen images. More precisely, this is considered a classification task. Here, data is assigned to classes, in contrast to a regression task where continuous values are predicted based on training data. An example would be to predict a house's price based on its size and location.

In *unsupervised learning*, targets  $Y$  are not provided for the training set. This can be the case when manually labeling the training is infeasible, or not enough is known about the dataset. A common method is cluster analysis, where data is assembled into groups based on some similarity measure. Clustering might be used to see if strong patterns exist within the data. For example, if a store wants to find out if there are groups with similar shopping habits among their customer base.

In *reinforcement learning*, rewards are maximized that are provided for different actions. For example, when training a model to win at a certain game, the model might receive positive rewards for winning the game and zero or negative rewards for losing.

### 2.3.1 Text Classification

The goal of text classification tasks is to assign classes to natural text based on its content, which makes them supervised learning problems. Furthermore, they are part of *natural language processing* (NLP), which is the general term when dealing with the analysis of natural text. An example of a text classification task would be to automatically classify news articles into a section such as sports or culture. One approach would be to search for certain keywords that might indicate the nature of the question. However, often, keywords are unknown or too many to define them manually. Furthermore, articles might be misclassified if they contain keywords that are not relevant to the topic. For example, an article might talk about a charity ball that a lot of professional sports players attended and consequently be classified as a sports piece. Instead, a machine learning model could be trained to classify these posts by minimizing the misclassification of posts where the category is known. This model can then be used to estimate classes for articles where the category is unknown. This process is visualized in Figure 2.4.



**Figure 2.4:** Training and evaluation process of text classification.

Text classification deals with natural text. However, for most machine learning techniques, it is preferable or even necessary for the input to be numeric. Different approaches exist to transform words into numbers. A simple approach is to map each word to an index, e.g., "sports" is mapped to 1, "football" is mapped to 2. A drawback of this approach is that some algorithms might assume similarity between words with similar indices. If such a relationship does not exist, *one hot encoding* can be applied. Here, the index is further mapped to a binary sequence of zeros and a one at the position of the integer index, e.g., "sports" = 1 = 100. With this approach, all words are considered equally important, making it a popular choice for text classes, but for the words in the documents, this can be a drawback. A term might only appear one time in one document and would be considered equal to more representative terms. *Term frequency* (TF) can be used to add importance to terms. It represents the number of occurrences of a term in the dataset. But if a term occurs in every single document, it would have high term frequency but also provide little indication about an article's class. That is why *term frequency–inverse document frequency* (TFIDF) introduces inverse document frequency, the inverse fraction of all documents that contain the term. It is multiplied with the term frequency. A term has a high TFIDF score if the term often occurs in a few documents and a low score if it rarely occurs or occurs in most documents. The drawback of this approach is that, depending on the dataset, similar terms might not be mapped to a similar score. For example, a common term in sports articles might be "soccer," so soccer has a high TFIDF. But one article uses "football" instead of soccer. Since it only occurs a few times, it might receive a small score even though it describes the same sport.

### 2.3.2 Word Embedding

Word embeddings, also sometimes referred to as word vectors, provide a different approach for vector space representations. In contrast to TFIDF, mappings are not computed based on frequencies but learned using machine learning methods such as *neural networks*. Here, the context of a term is used to generate its numerical representation. So, if two terms appear in a similar context, they are mapped to similar values. A common approach for generating word vectors is *word2vec*, which was introduced by Mikolov et al. [20]. In this approach, word embeddings are learned by training a neural network to predict a term based on its context. For example, given the sentence "Yesterday I baked an apple pie", the embedding for "pie" is trained by providing the model with "Yesterday I baked an apple", which predicts the subsequent word. The predicted word is then compared with the actual word to receive the prediction loss. The loss is used for the optimization of the model.

### 2.3.3 Artificial Neural Networks

Artificial neural networks (ANN) describes a subset of machine learning that was inspired by biological neural networks in brains. Like the biological counterparts, they are based on the idea of connecting simple elements to solve complex problems. These simple elements are called neurons and are made up of weights  $w$  and biases  $b$ . Each neuron receives input data and computes the output by a linear function. A non-linear function, called *activation function*, is applied to the output of the linear function, as defined in Equation 2.1.

$$\hat{y} = \sigma\left(\sum_{i=1}^n xw + b\right) \quad (2.1)$$

The activation function adds non-linearity to the output and confines the output within a certain range. Activation functions are described more in detail in Subsection 2.3.8. Neurons are structured into layers with typically one input layer, one or more hidden layers, and an output layer. Combined, this is referred to as the network. Often, the previous layer has only connections to the next layer, but variations such as residual neural networks exist.

Artificial neural networks learn by adapting the values of its neuron's weights such that the loss is minimized. This is typically done using *backpropagation* by computing the gradients of the loss with respect to each weight using the chain rule, starting with the last layer, and continuing in reverse order of the layers.

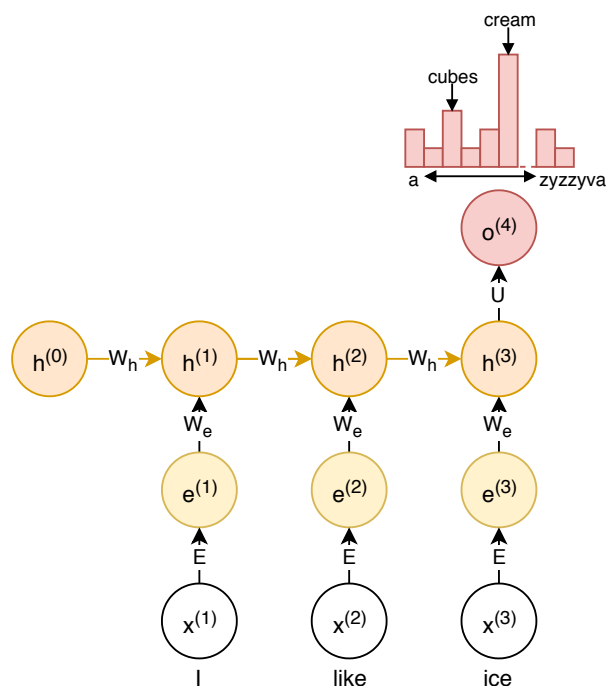
### 2.3.4 Recurrent Neural Networks (RNN)

Artificial neural networks usually treat data samples as independent of each other, but when data samples represent e.g., words of sentences, they are not independent. To overcome this, recurrent neural networks (RNN) change the design of the neurons by introducing a hidden state  $h^{(t)}$  that takes previous samples into account. This hidden state is used to compute the output  $o^{(t)}$  of the neuron. The update rules for the hidden state and the output are defined in Equation 2.2 and Equation 2.3 respectively.

$$h^{(t)} = \sigma(W_h h^{(t-1)} + W_x x^{(t)} + b_1) \quad (2.2)$$

$$o^{(t)} = \text{softmax}(U h^{(t)} + b_2) \quad (2.3)$$

The same weights are applied to every element of the sequence, meaning it can process any input length without increasing the model size. Figure 2.5 provides an example of an RNN that predicts the next words of a sentence using the previous. When working with words as input, the additional step of looking up a word's embedding is necessary. After that, the hidden state of the neuron is updated, and the output is



**Figure 2.5:** Example of a Recurrent neural network that predicts the next word of a sentence.

generated. An output is generated after each update. However, when predicting succeeding words, only the output of the last word is taken into account since it also encodes information about the previous words.

As previously described, neural networks make use of backpropagation, and RNNs are no exception to that. The loss of the first word of a sentence depends on the loss of the succeeding words. However, this may cause problems when the gradient of the loss of the last words is small. Then the gradient becomes smaller and smaller for preceding words. This is known as the *vanishing gradient problem*. If this is the case, model weights may only be updated with respect to close-by effects. Long-term dependencies would not be taken into account, and model weight updates might not fit the data best.

Hochreiter et al. [12] propose *long short-term memory* (LSTM), a type of RNN that should solve the vanishing gradient problem by introducing a separate cell state to store long-term information. The LSTM can erase, write, and read information using so-called gates. The *input gate*  $i^{(t)}$  decides how much of the current input  $x^{(t)}$  is added. The *forget gate*  $f^{(t)}$  controls what is kept from the previous cell state  $c^{(t-1)}$ , and the *output gate*  $o^{(t)}$  defines what parts from the cell  $c^{(t)}$  are added to the hidden state  $h^{(t)}$ . The *cell state*  $c^{(t)}$  is updated by forgetting some content from the past and adding some new input. The *hidden state*  $h^{(t)}$ , i.e., the output of the LSTM cell is updated using the new cell state. The update rules are defined in Equation 2.4 for the input gate, in Equation 2.5 for the forget gate, and Equation 2.6 for the output gate. The update rules for the cell state and its content is defined in Equation 2.7 and Equation 2.8. Equation 2.9 provides the update rule for the hidden state. Figure 2.6 provides an overview of an LSTM cell and the dependencies of the gates and states.

The LSTM architecture improves the ability of RNNs to preserve information over many time steps, but the vanishing gradient problem might still occur. Additionally, recurrent neural networks have their own set of problems because of their architecture. The first one being when capturing words without previous context e.g., the first word of a document. Since the state of a cell is computed based on previous states, if no previous state exists, such words might not be captured correctly. Another problem occurs in classification because only the last cell output of a sentence or a document is considered; it might capture too much of the last words and cause misclassification.

$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i) \quad (2.4)$$

$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f) \quad (2.5)$$

$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o) \quad (2.6)$$

$$\tilde{c}^{(t)} = \tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c) \quad (2.7)$$

$$c^{(t)} = f^{(t)} \cdot c^{(t-1)} + i^{(t)} \cdot \tilde{c}^{(t)} \quad (2.8)$$

$$h^{(t)} = o^{(t)} \cdot \tanh c^{(t)} \quad (2.9)$$

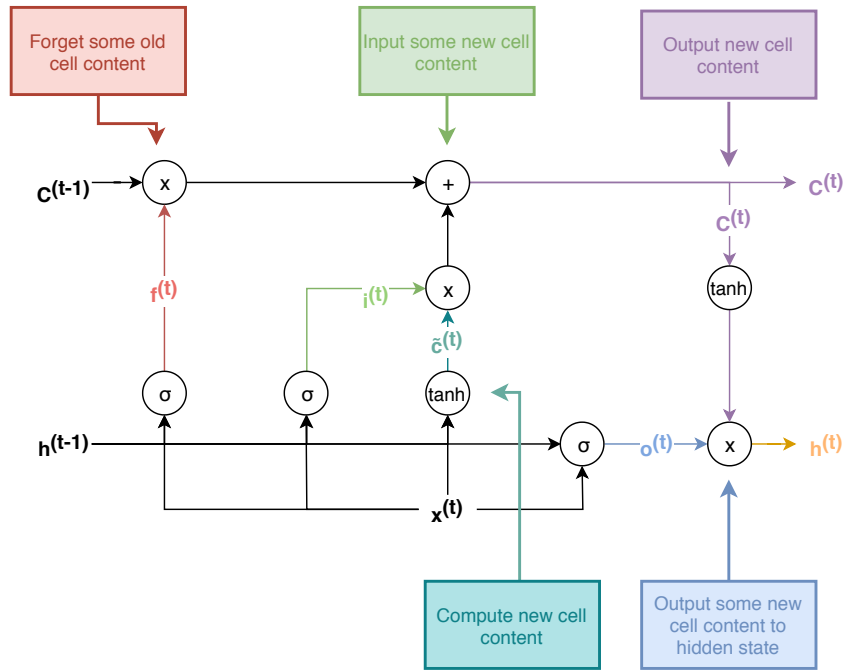


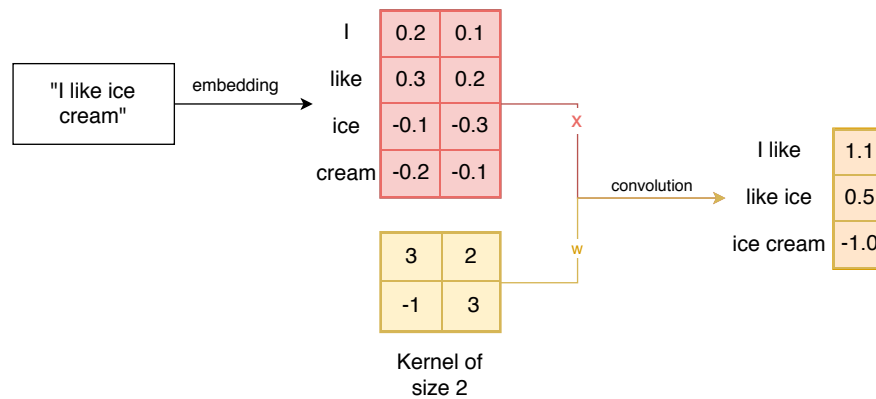
Figure 2.6: Overview of an LSTM cell.

### 2.3.5 Convolutional Neural Networks

As mentioned in the previous paragraph, recurrent neural networks can have certain disadvantages because of the way sequences are processed. In order to prevent these, convolutional neural networks (CNN) follow a different approach. Instead of computing states for entire sequences, they look at local neighborhoods within the text by computing vectors for all possible subsequences of a sequence. For example, for the sentence "I like ice cream", possible subsequences would be "I like", "like ice", "ice cream", "I like ice", "like ice cream". These vectors are computed based on an operation called *convolution*, which is defined in the one dimensional, discrete case in Equation 2.10 where  $x$  represents the input and  $w$  the so-called kernel. Figure 2.7 provides an example for the application of convolution for the sentence "I like ice cream" with a kernel of size 2. At first, the embedding vector for each word is looked up in the embedding matrix, which is then used to compute the convolution resulting in values for 3 pairs "I like", "like ice", and "ice cream".

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \quad (2.10)$$





**Figure 2.7:** Example of a convolution of the sentence "I like ice cream" with a kernel  $w$  of size 2.

The motivation behind convolutional neural networks is three-fold. The first point is reducing the number of parameters compared to traditional fully connected neural network layers, which is done by making the kernel smaller than the input. The second point is increasing the efficiency by parameter sharing, i.e., using the same parameters for more than one function in the layer. Last but not least, CNN layers are equivariant, meaning that if the input is moved in time, the same output is generated, and it is shifted in time as well. This attribute enables one to, e.g., detect all similar edges in an image.

Besides the kernel size, convolution layers have additional parameters, such as the *stride*, which defines the amount of movement between application of the filter to the input. The default stride is 1 which describe a shift of 1. If the value is increased, less convolution applications are performed, resulting in a down-sampling of the output. Another parameter is *padding*. It can be applied to the input, if the kernel does not fix the input. A simple approach would be to just ignore the parts of the input where the kernel does not fit. Another approach is to add zeros at the corners.

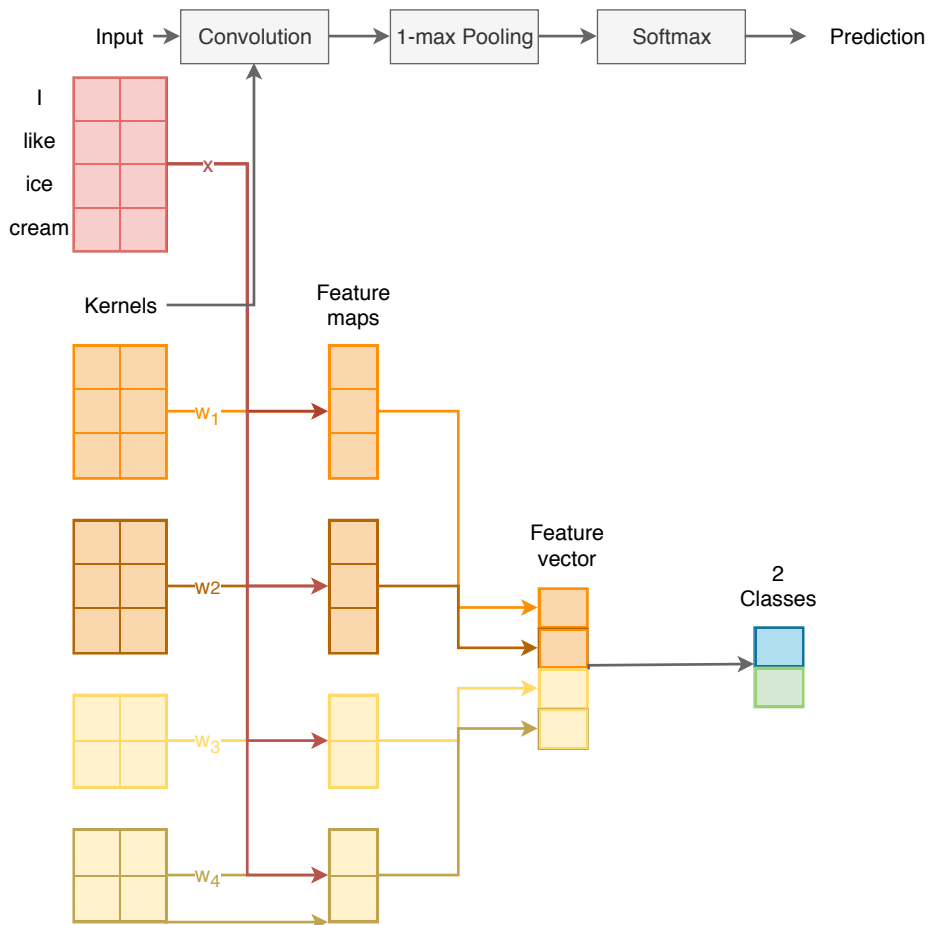
As with standard neural networks, non linearity functions are also applied in CNN layers. The most common function that is utilized for CNNs is the Rectified Linear Unit. As defined in Equation 2.11, it outputs either 0 or the input  $x$  if it is positive. Compared to other activation functions, it has the advantage that it does not saturate in the positive area, it is computed very efficiently, and it converges much faster than e.g. sigmoid in practice. However, it might still saturate, if the input of the function is negative. To solve this, *Leaky Rectified Linear Units* keep a small part of the negative input value, as defined in Equation 2.12.

$$f(x) = \max(0, x) \quad (2.11)$$

$$f(x) = \max(0.01x, x) \quad (2.12)$$

After the application of non-linear function, is it common to apply a *pooling layer*. Pooling reduces the dimensionality of the input by down-sampling by applying functions to neighborhoods of the input. They help to make the output invariant to small changes in the input while still retaining vital information. There are three popular choices for pooling functions: max pooling, average pooling, and sum pooling. *Max pooling* returns the maximum value of the neighborhood, *average pooling* returns the average value, and *sum pooling* returns the sum.

Typically a CNN has 3 different stages, the first one being the convolution state, where several convolutions are applied, resulting in a set of linear activations. Nonlinear activation functions are applied to this set in the detector stage. Finally, in the pooling stage, pooling functions are applied to neighborhoods



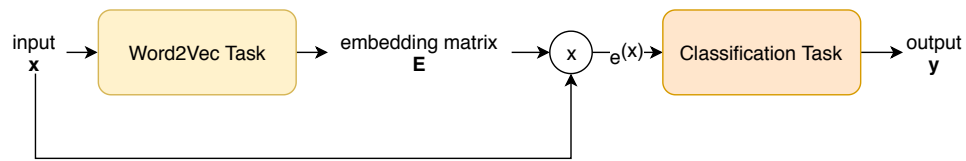
**Figure 2.8:** Example of a classification with 1 CNN layer and 4 kernels, 2 kernel of size 3 and 2 kernels of size 2.

of the output. It is common to use *dropout* as regularization to further prevent the development of co-dependencies of neurons and is explained further in Subsection 2.3.7.

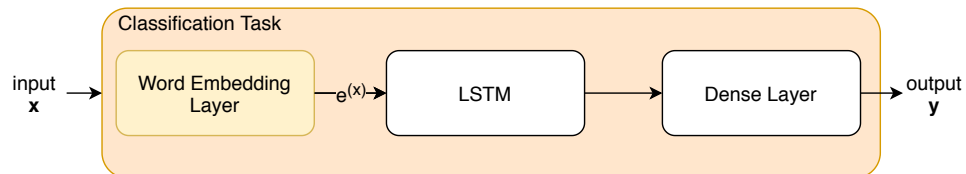
Figure 2.8 illustrates an architecture for sentence classification introduced by Yoon [15] using only one CNN layer with 4 kernels, followed by one 1-max pooling layer and a final softmax layer.

### 2.3.6 Word Embedding Layer

As described in Subsection 2.3.2, word embeddings are a common way of mapping words to vectors. When utilizing these embeddings for, e.g., a sentence classification task, two approaches exist. The first approach would be to train the embeddings using existing algorithms such as *word2vec* [20] or use already trained embeddings. Apply this embedding matrix to the dataset before and input it to an e.g., LSTM layer. Through this approach, similar words are mapped to similar embeddings if they appear in a similar context. However, some words may be more important for classification than others, even if they have a similar meaning. For example, one might want to classify sports articles into different sports. With this approach, all ball sports might have similar word embeddings, possibly leading to inaccurate classification results. Another approach would be to train the embeddings as part of the overall classification task using word embedding layers, meaning that words appearing in the same classes receive similar embeddings. A popular approach is a combination where pre-training embeddings are used as initialization for word embedding layers and further train them as part of the classification task. Figure 2.9 gives an overview of the two approaches which previously mentioned.



(a) Text classification by solving 2 different tasks: word2vec task and classification task

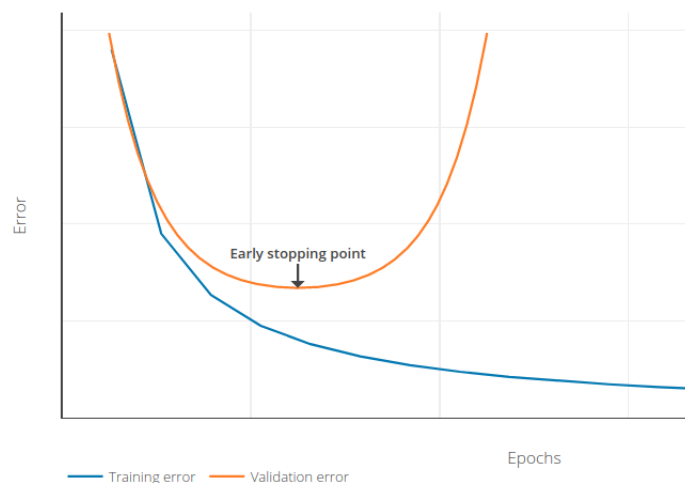


(b) Solving text classification with 1 task by adding word embedding layer to text classification model.

**Figure 2.9:** Different approaches for solving a text classification task.

### 2.3.7 Regularization

The goal that all machine learning models share is to generalize and give predictions about new, previously unseen data. This is done by fitting a model to training data. However, if a model fits too closely to the training data, it can impact its ability to generalize. This is referred to as *overfitting*. That is why *regularization* methods are applied while training. One such regularization method is *early stopping*. The idea behind early stopping is to use a *validation set*, in addition to a training set. After each epoch, the model's performance is evaluated on the validation set. If it has improved, the current model is saved, and the next epoch is started. However, if it has not improved, training is stopped, and the previous model is used. Figure 2.10 shows an example for early stopping.



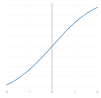
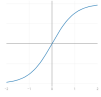
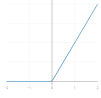
**Figure 2.10:** Example for early stopping, where an additional validation set is used to avoid overfitting.

Another regularization method specifically for neural networks is *dropout*. When training neural networks, some units may learn to fix the mistakes of other units. This problem is referred to as co-adaption and leads to overfitting on the training data. Dropout aims to overcome this by randomly ignoring a certain amount of units while training. Ignoring some units forces the remaining units to take on more or less

responsibility and leads to better generalization. Which neurons are dropped, is randomly chosen during training using a Bernoulli random variable.

### 2.3.8 Activation Functions

Activation functions add non-linearity to the output of a neuron and confine the output within a specific range. One of the most well-known activation function is the *logistic* function, which squashes input numbers to a range of  $[0, 1]$ . It has declined in popularity because it has some disadvantages. Namely that saturated neurons kill the gradient, that it is not zero-centered, and that it is computationally expensive.

Name	Plot	Equation
Logistic		$\sigma(x) = \frac{1}{1+e^{-x}}$
TanH		$\sigma(x) = \tanh(x)$
ReLU		$\sigma(x) = \max(0, x)$
Softmax		$\sigma(x) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$

**Table 2.5:** Overview of different activation functions

Another common activation function is the *hyperbolic tangent* function. It squashes the input to a range of  $[-1, 1]$ , meaning it is zero-centered, unlike the logistic function. However, the gradient is still killed when the neuron is saturated. To improve this issue, *rectified linear units* (ReLU) can be used. They do not saturate in the positive area, are very computationally efficient, and converge must faster than the logistic function and the hyperbolic tangent. That is why rectified linear units have become a popular choice in recent years. All of these functions take one number as input. The *softmax* function, on the other hand, takes a vector of real-valued numbers as input and outputs a probability distribution, which makes it a popular choice for classification tasks. Table 2.5 lists all of the equations and plots of the previously mentioned functions.

## Chapter 3

# Related Work

Analysis of Android applications, especially the detection of malicious applications, has been a popular area of research. Section 3.1 describes related research, mainly focusing on malware detection. While similar to our task, malware detection proposes slightly different challenges. That is why we have also looked into approaches that deal with source code in general. These are described in Section 3.2.

### 3.1 Android App Classification

Related work to the classification of Android applications is primarily focused on the detection of malicious applications. While some challenges and approaches might differ to our task, some of the ideas prove useful for us, as well. Most approaches make use of static or dynamic analysis or a combination of both. Static analysis tends to focus on which permissions are required by the application and which API calls are performed in the source code.

Arp et al. [5] make use of a multitude of static features. From the `AndroidManifest.xml`, they retrieve the required hardware components, requested permissions, the app's components, and intents. From the source code, suspicious API calls, permissions, and network addresses are retrieved. All these features are combined in a binary vector where each dimension represents if a specific feature is present or not. For example, if the camera permission was requested, then the camera permission dimension is set to 1. This feature vector is then used to train a linear *support vector machine*. A linear SVM learns a hyperplane that separates two classes with a maximal margin. It is trained with 5,560 known malware apps and 123,453 apps from different app stores, which are assumed to be benign. With this approach, an accuracy of 94% for binary classification and an average of 93% for classifying the different malware classes. Despite its high accuracy and computational efficiency, it might not fully capture an application's nature since it does not include dynamic analysis. That is why Yuan et al. [30] introduce DroidDetector, which includes dynamic behavior by monitoring different applications actions using *DroidBox* [16], a sandbox that can execute taint analysis and monitor a variety of app actions such as sending data over a network and sending SMS messages. This data is combined with required permissions and sensitive API calls in a binary feature vector, similar to Arp et al.'s approach. This data is used to train a *deep belief network* (DBN), a popular approach in the early days of deep learning. A DBN is created by stacking *restricted Boltzmann machines*, which are 2 layer networks with one visible layer and one hidden layer that are connected in both directions. This model is trained with 20,000 benign apps and 1,760 malicious apps. Using this approach, they achieve 96.76% accuracy for binary classification. However, the low amount of malicious applications might not be enough to represent Android malware fully.

Karbab et al. [14], focus on a smaller feature set, mainly API method calls, but have a significantly larger malware set of 33,006 applications, in addition to 37,627 benign apps. The API calls are extracted from the source, and word2vec is used to train word embeddings for each API call. These embeddings are the

input for a binary classification network with one convolutional layer, one max-pooling layer, and two fully connected layers. With this approach, they achieve an F1 score of 96% to 98%, depending on the setup.

While most approaches focus on API method calls when it comes to analyzing source code, McLaughlin et al. [19] analyze raw opcode sequences instead. The opcodes are attained by disassembling the application using *baksmali* and removing the operands. One-hot vectors are assigned to each opcode and concatenated to sequences. The model for training is a binary classification network with an embedding layer, a convolutional layer, a fully connected layer, and an output layer. They test their approach on three different datasets, a "small dataset" with 863 benign and 1,260 malicious apps, a "large dataset" with 3,627 benign and 2,475 malicious apps, and a "very large dataset" with 9,268 benign and 9,902 malicious apps. They achieve an F1 score of 97%, 78%, and 86% for the "small dataset", the "large dataset", and the "very large dataset," respectively.

All of the previously described approaches achieve high accuracy or F1 score. However, as the evaluation of McLaughlin et al. shows, the performance widely differs depending on the quality of the dataset. Furthermore, the relatively small size of these datasets restricts the possibilities for more complex architectures that may require a larger dataset. That is why we decided to focus on the classification of Android applications into categories instead, as the amount of available Android apps with categories is vastly larger. Thus, we are able to get a broader picture of the nature of Android apps.

### 3.2 Natural Language Processing for Source Code

This section discusses works that process source code in general, not specifically Android source code. Their tasks range from assigning names to functions and generating descriptions for source code to predicting the original author of a code snippet. These approaches tend to focus on the nature of source code rather than security-related aspects, like in section 3.1.

One such example is introduced by Allamanis et al. [1], where the goal is to predict short, descriptive names for source code snippets with the sequence of code sub-tokens as input and a sequence of natural text sub-tokens as output. Sub-token refers to the separate words that a token is comprised of, e.g., a variable may be called "isPresent", then it's sub-tokens would be "is" and "present". They do so using convolutional layers to compute attention and generate method names and combine this with an RNN to process the sequences. Attention is a popular mechanism to enable a model to put more focus on certain features. The dataset includes 10 popular Java projects and results in an F1 score of 59.6% and exactly matches the method name 34% of the time. Rather than predicting a method's name, Iyer et al. [13] generate sentences that describe code snippets and queries using an LSTM with attention. Training data is retrieved from StackOverflow posts where code snippets in responses are taken as data and the post's title as the label. After cleaning the data, this results in 66,014 C# pairs and 32,337 SQL pairs. The code snippets are split into tokens and represented as one-hot vectors. These vectors are then looked up in a token embedding matrix that is learned with the model. The output is generated using a recurrent neural network with LSTM cells and attention. The output is also encoded using a word embedding matrix, which is also learned with the model. For evaluation they report *METEOR* [7] and *BLEU-4* [22] scores. While *METEOR* measures how well the model captures content from references in the output, *BLEU-4* measures the average n-gram precision. The setup results in a *METEOR* score of 12.3 and a *BLEU-4* score of 20.5 for C# code snippets and a *METEOR* score of 10.9 and *BLEU-4* score of 18.4 for SQL queries, outperforming previous approaches.

The previous approaches have treated source code as a sequence of tokens. However, this has been argued to not provide the best representation of context since the order of statement does not always reflect direct dependence. For example, a variable may be defined in the first line in a function but only be referenced in the last line. Depending on the size of the function, this relationship may vanish while training. That is why Mou et al. [21] make use of structural information from abstract syntax trees, which are described in Section 2.2. To process the AST using machine learning, they introduce *tree-based*

*convolutional neural networks* (TBCNN). Rather than embedding the code tokens, they create their own notion of embeddings for the nodes of the AST, where the parent's node should capture a weighted combination of its children's embedding. After embedding the input, they apply tree-based convolution to each node, which applies convolution to nodes within a certain fixed-depth window of the graph. Then, pooling is applied to the output tree of the convolution layer. Here, the maximum value in each dimension is taken. A fully connected layer and a softmax layer are applied to the output of the pooling layer. When classifying source code into 104 programming problems with 500 programs per class and a total of 52,000 samples, this approach reaches an accuracy of 94%. Ben-Nun et al. [8] also use graphs as the base for their approach. However, they do not use abstract syntax trees, but data and control flow graphs. These graphs represent data dependence and execution dependence of code, respectively. They introduce *contextual flow graphs* (XFG), which combine these two graphs into one, arguing that this provides the best representation of dependence and context. From this graph, neighboring statement pairs are generated and are used to train a *skip-gram model* [20] to generate embeddings for the code tokens. One of the evaluation methods performed is the classification of source code into classes with the same dataset as Mou et al. [21] and achieve slightly better results with an accuracy of 94.83%, compared to 94%

Besides predicting programming problem classes and method names, another prevalent task is to predict correct programmers. One such example is the work of Dauber et al. [11], where they assign programmers to code snippets to showcase that authorship attribution may be a threat to the privacy of programmers who prefer to contribute code anonymously. They focus on small code snippets with an average of only 4.5 lines of code to highlight that these small snippets may already be sufficient to predict the programmer. Their approach is to generate the abstract syntax tree of the code snippet and extract nodes and node bigrams and use these to train a *random forest*. The dataset contains code snippets from 106 programmers with at least 150 samples each. With this approach, they are able to assign the correct programmer given only a single sample with an accuracy of 73%, with two samples this increases to 95% and to 99% with 15 samples. Caliskan et al. [9] follow a similar goal, but in their case, they are predicting a programmer from an executable binary. Their approach is to disassemble and decompile the binary, generate the abstract syntax tree and context flow graph to extract lexical and flow features from the graphs. Features include AST n-grams, tokenized instruction traces, integer types, words unigrams, basic blocks, and names of library and internal functions. These features are extracted from the AST, the CFG, the disassembled code, or the decompiled code and results in 705,000 features. Before training a *random forest* for classification, dimensionality reduction methods are used to reduce the number of features from 705,000 to 53. This model is trained using 900 compiled C++ binaries from 100 programmers and achieves an accuracy of 96%.

Extracting features from graphs tend to focus on unigrams, bigrams, or n-grams. Alon et al. [4], however, follow a different approach which they name *code2vec*. Their goal is to assign a name to a method given its body by creating syntactic paths between leaves of the method's abstract syntax tree. They define the triplet of a start AST node, a final AST node, and a path connecting these AST nodes as *path context*. Embeddings are applied for the nodes and the paths. The token and path vocabulary are learned jointly with the rest of the network. After applying the embedding, the resulting vectors are concatenated and input to a fully connected layer that learns to combine these 3 vectors into one vector. Attention mechanisms are applied, which learn to score the different path contexts of a code snippet. All path contexts of a method are summarized weighted by the attention vector. This resulting vector is multiplied with each possible tag from the output vocabulary, and the softmax function is applied to receive a probability distribution of the most likely output tags. The dataset for this model is comprised of 10,072 Java GitHub repositories with almost 15 million samples and is able to achieve an F1 score of 58.4%. This approach is extended by Alon et al. [3] as *code2seq* where an encoder-decoder architecture is replaced as the model. The generation of the paths is the same as with *code2vec*. However, instead of treating the path as one element, it is treated as a sequence. Embedding is applied to each of the path's elements and input to an LSTM. The output of the LSTM is combined with the embeddings for the start and end node and is input to a fully connected layer. The output of the fully connected layer is the encoder's representation of a path context.

After generating this representation for all path contexts of a code sample, the decoder then generates the output sequence while attending over the path contexts. Evaluation is performed on two different tasks. The first task is to predict a method's name based on its body, the same as with `code2vec`. Three datasets are used of different sizes, all being Java projects from GitHub, where the largest dataset contains over 16 million samples. With this setup, they achieve an F1 score of 59.19% for the largest dataset, slightly outperforming `code2vec`. The second task is to generate descriptions for C# code snippet. They use the same dataset as *CodeNN* [13] which contains 66,105 pairs of questions and answers from StackOverflow. They achieve a BLEU score of 23.04 with their approach, which is 2.51 points above CodeNN.

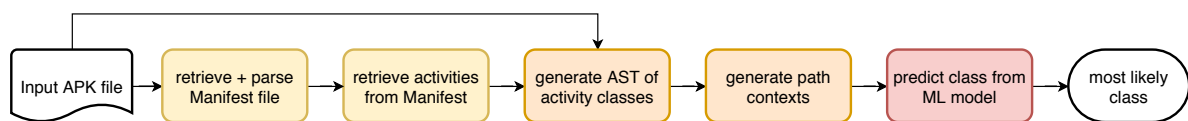
Code2Seq provides a novel approach for extracting features from an abstract syntax tree. Typically only n-grams, such as uni-grams or bi-grams are used from the AST, which strongly limits the context. Approaches with a larger context generally take source code as a sequence of tokens, similar to how natural text processed. However, as previously described, this analogy does not always work. One can introduce a variable at the beginning of a function and only reference it at the very end, so the context is lost, again. Abstract syntax trees, on the other hand, provide a sense of hierarchy. For example, all child nodes of an `if`-statement belong to the `if` statement. With token sequences, this cannot be identified unless additional processing is applied. This is reflected in the trend of using features from graphs, which we also want to make use of in our approach. However, traditional machine learning methods are not geared towards processing trees and generating n-grams does not leverage enough of the context, in our opinion. That is why we see Alon et al. *code2seq*'s code embeddings as a potential candidate to represent Android source code in a meaningful way and want to investigate this. For this purpose, we adapt their model for our classification purposes. Android applications tend to be larger than samples used in `code2seq`, so we make adaptations to handle larger samples while also reducing the size by selecting essential parts of the application. For this, we have taken inspiration from Arp et al. [5] in Section 3.1. Our approach is described more in detail in Chapter 4.



## Chapter 4

# Approach

Our approach combines Alon et al. [3]’s architecture of representing source code as AST paths with pre-processing mechanisms for Android APK files and approaches from natural language processing. It can be divided into the Android processing, the feature generation step, and the machine learning step. Figure 4.1 provides an overview of the steps, where the Android processing steps are marked in yellow, the feature generation steps are marked in orange, and the machine learning step is marked in red. Section 4.1 discusses the approach of Alon et al. in further detail. Section 4.2 provides an overview of the mechanisms applied to the Android APK file and the steps necessary to generate the feature for our machine learning model. Section 4.3 describes the architecture of our two models: the LSTM model and the CNN model.



**Figure 4.1:** Flow Chart of our approach: Steps in yellow: Android processing, orange: source code processing, red: code2seq + text classification

### 4.1 Code2seq Approach

Our approach is strongly influenced by Alon et al.’s [3] approach. In this section, we describe their approach further in detail to highlight the similarities and difference between our approaches.

The first step of code2seq is to generate the abstract syntax tree for the code snippet. The leaves of the resulting tree are called terminals and usually represent values that the user has defined, such as identifiers. The nodes that are within the tree and not leaves are called non-terminals. They typically represent a fixed set of instructions from the programming language, such as expressions or statements.

As previously mentioned, there is no trivial solution to use trees for training neural networks since they typically expect fixed-sized matrices and not variable-sized tree structures. There is, however, a type of neural network geared towards processing sequences, which are called *recurrent neural networks*. These networks are described in detail in Section 2.3.4. To make use of recurrent neural networks, Alon et al. generate sequences from the abstract syntax tree, so-called *paths*. These structures are the paths between all pairs of terminals from an AST and are represented as sequences of non-terminal nodes connecting two terminals.

Their reasoning behind this way of representing source code is that two pieces of source code may have the same functionality but different implementations. If they were represented as sequences of tokens,

the similarity might be overlooked. However, they observe that while having different implementations, similar code pieces share similar syntactic paths. Figure 4.2 provides an example of two Java methods that have the same functionality but different implementation. Even with these differences, similarities can be found in paths generated from the method’s abstract syntax tree. For example, the path in red and yellow are identical. The path in orange is also identical besides the "ForStmt" node instead of the "DoStmt" node. This example shows that syntactic paths can provide effective representation for source code. This form of representation considerably reduces the variance from the different implementations and can, therefore, provides a basis for detecting patterns.

Because the amount of paths varies between different code snippets, Alon et al. sample a fixed number of paths so that all code snippets have the same size. The paths are sampled again after each training iteration to apply regularization and therefore avoid overfitting. If the number of paths is smaller than the fixed amount, they add padding to ensure that all samples have equal size.

```

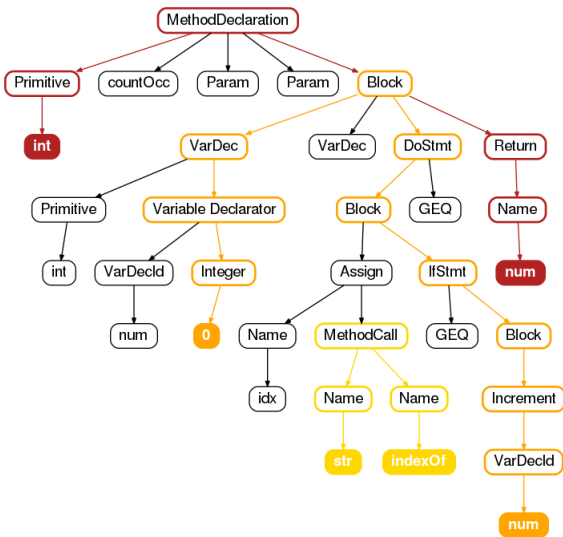
int countOcc(String str, char ch) {
  int num = 0;
  int idx = -1;
  do {
    idx = str.indexOf(ch, idx + 1);
    if (idx >= 0) {
      num++;
    }
  } while (idx >= 0);
  return num;
}

int countOcc(String src, char val) {
  int count = 0;
  for (int i = 0;
       i < src.length(); i++) {
    if (src.charAt(i) == val) {
      count++;
    }
  }
  return count;
}

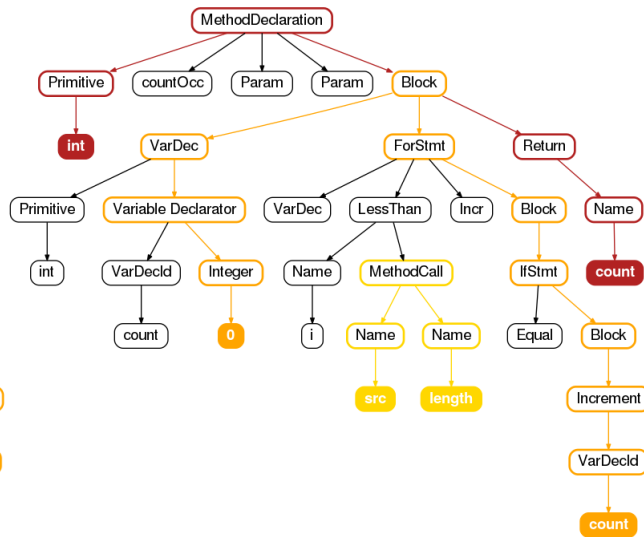
```

(a) Method 1: Java code

(b) Method 2: Java code



(c) Method 1: partial AST with paths

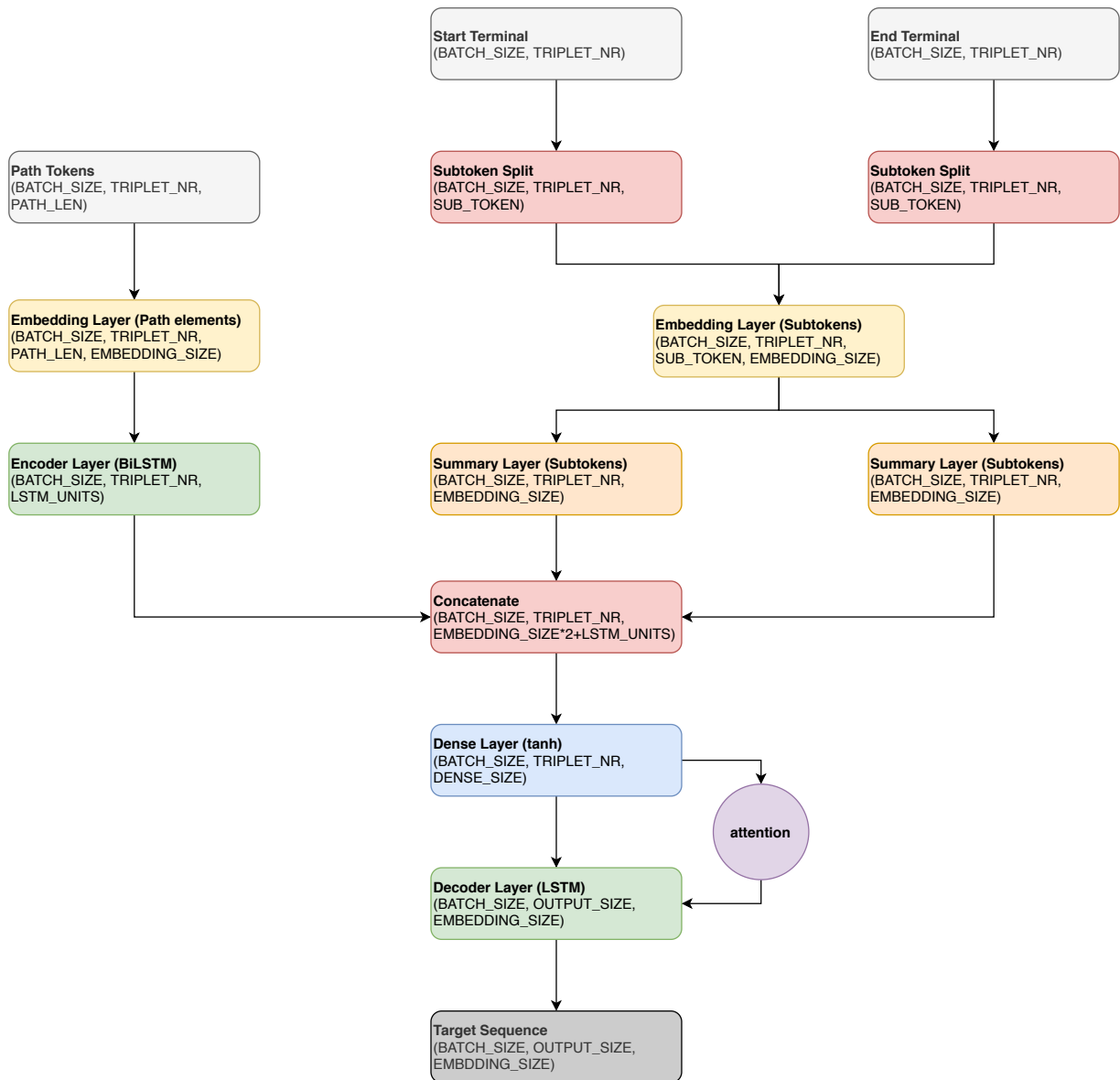


(d) Method 2: partial AST with paths

**Figure 4.2:** Example by Alon et al. [3] of two Java method that have the same functionality but different implementations.

The model of code2seq follows an encoder-decoder architecture which is common for neural machine translation, similar to Sutskever et al. [25], Cho et al. [10], Bahdnav et al. [6], Luong et al. [18], and Vaswani et al. [27]. The main difference between code2seq and the other approaches is that the encoder of

code2seq does not read the input as a sequence of tokens but as a sequence of elements from syntactical paths. This input sequence is then mapped to a continuous representation by the encoder. The decoder generates a sequence of output tokens based on this continuous representation and previous output tokens. Figure 4.3 provides an overview of the model architecture.



**Figure 4.3:** Overview of the parts of the model architecture proposed by Alon et al.[3]

As previously described, the goal of the encoder is to map the input to a continuous representation. Alon et al. achieve this by, at first, splitting each path into its elements and representing the elements using a learned embedding matrix. This matrix is shared for all path elements and trained as part of the overall task. The sequence is encoded using the final states of a bi-directional recurrent neural network with LSTM cells. The start and end terminal, on the other hand, are split into their subtokens. For example, `ArrayList` is split into `Array` and `List`. Afterward, each subtoken is represented by a learned embedding matrix that is shared between all terminal subtokens. The goal of splitting up the tokens into sub-tokens is to reduce the number of possible terminals. Consequently, this reduces the size of the embedding matrix. Each terminal is represented by the sum of its subtoken's embedding vectors. This

process helps to ensure that similar tokens are mapped to similar vectors. For example, `ArrayList` would receive a similar representation than `LinkedList` because they share the `List` subtoken. In the next step, the representation for the path and the terminals are combined by concatenating them and applying a fully connected layer with a hyperbolic tangent function as activation function, which is described in Section 2.3.8. The goal of this layer is to condense the representation of the terminal-path triplet to a combined representation.

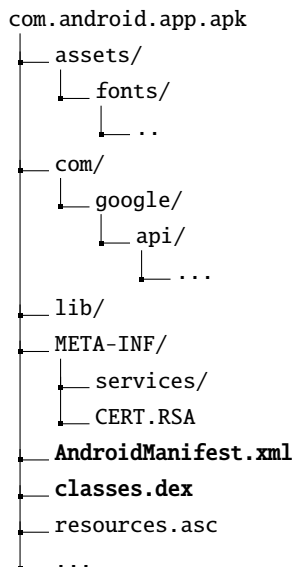
The combined representation is the input to the decoder. The decoder generates output sequences based on the combined representation using a recurrent neural network with LSTM cells and attention. Attention is applied by computing a *context vector* over the elements of the combined representation for each time step when decoding. This context vector and current state of the decoder's LSTM is used to predict the current output of the token. Attention is applied to overcome the issue of vanishing long-range dependencies in an encoder-decoder structure. Without attention, the decoder generates an output sequence solely based on the last state of the encoder. This can cause a loss of long-range dependencies. This problem can be overcome using attention because then the decoder can attend to various parts of the input.

The approach of Alon et al. is geared towards generating descriptive sequences based on source code snippets. Our approach, on the other hand, is focusing on classifying Android applications. Therefore, we need to make alterations to the `cod2seq` architecture to fulfill our requirements. The first alteration is the feature generation step because we are provided with an Android APK file and not the source code directly. That is why we need to introduce additional steps to get the source code. We describe this process in Section 4.2. The generation of the dataset mostly overlaps with the `code2seq` approach. Overlapping steps include the reading of the files, processing the tokens, and embedding the values. However, we extend the approach to consider class imbalance and implement early stopping for regularization. Furthermore, we make adaptations to the `code2seq` architecture to support classification and explore two different ways of processing the paths. We provide more detail about our model generation process and architecture changes in Section 4.3.

## 4.2 Feature Generation

The input to our approach is an Android application's APK file, which is a package file that contains the app's compiled code and additional files such as resources and a Manifest XML file. Figure 4.4 provides an example of the contents of an APK file. Because this is a package file, we cannot directly input it into our machine learning model. That is why we unpack the APK file to retrieve the compiled source code of the Android app. This source code often contains hundreds to thousands of classes. Especially with the increasing power of smartphones, the size of Android apps is getting bigger, and with it also the size of the source code. Using the entirety of the source code would severely slow down the training process of our model and possibly exceed memory constraints. Furthermore, some parts of the code have a lot more relevance to the distinct behavior of an application than others. Other parts may not even be used in the app. If the model focuses on these parts, this could impact the model's ability to generalize. To overcome these obstacles, we choose to focus on the source code of *activity* classes. As described in Section 2.1.1, activity classes represent the screen in an application and handle user interactions for this screen. This guarantees that the source code that we are looking at is actually called within the app. Moreover, it ensures that the content is meaningful because they represent what the user can do within the app.

To retrieve the activities, we could disassemble the entire source code using `baksmali` and look for all classes that extend the `activity` class. However, disassembling the full source code creates a considerable overhead in processing time, especially since we make use of only a particular subset of the code. Fortunately, `baksmali` has an optional argument, where we can provide a list of classes, and `baksmali` only disassemble these classes. So if we provide a list of all activity classes, `baksmali` only disassembles these classes from the source code. All activities are defined in the Manifest XML file. The Manifest XML file is also packaged with the source code in the APK file. So by unpacking the APK file,



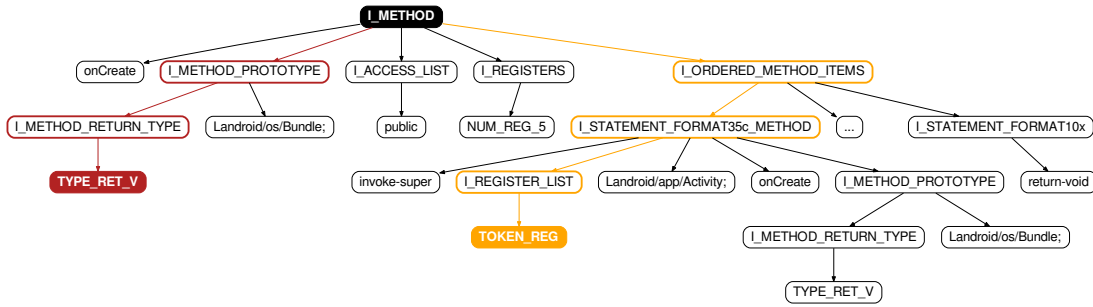
**Figure 4.4:** Example structure for an APK file. The files used for our approach are the `AndroidManifest.xml` and the `classes.dex`. They are marked in bold.

we retrieve the Manifest XML file, additionally to the source code. We then parse the retrieved Manifest XML file and extract all activity entries. Listing 2.1 contains an example for a Manifest XML file. Here we can see that activities are defined by the `activity` tag, and the name of the class is defined in the `android:name` attribute of the tag. Some entries may contain the full class name, including its package. However, some entries only contain the name of the class without any package. If this is the case, the class' package is the app's root package, and then the root package has to be added. The root package is defined in the root element of the Manifest XML file with the attribute `package`. So, in this case, we retrieve the root package and combine it with the class name. We need to full package name because the class name alone is not a unique identifier; only the full name is. That is why baksmali requires the full name for disassembly.

After retrieving the activities, we generate the abstract syntax tree for each activity class using a modified version of `baksmali`, which is described in Section 2.2. Traditionally the AST is only generated in memory in `baksmali`. We extend `baksmali` to save the abstract syntax trees to files. Furthermore, we implement that certain node values are replaced with specific place holders. For example, all registers are replaced with the value `"TOKEN_REG"`. The reasoning behind this is that, to our case, it does not matter which register is used, only the information that a register is used is relevant. So, to help the model generalize better, we replace it.

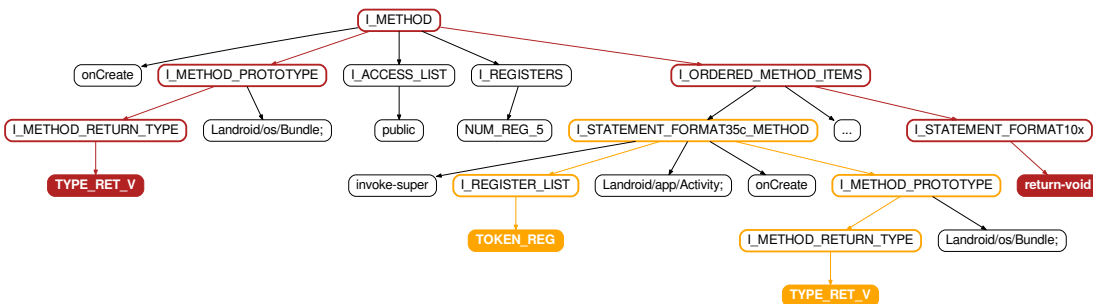
For each AST, paths between all pairs of leaf nodes are generated and stored in files. Alon et al. do not clarify in their work, how they generate the paths. That is why we propose the following process: Firstly, we create and store the path of each leaf node to its method's root. We refer to these paths as *tree root paths*. Examples for tree root paths are provided in Figure 4.5. The orange path represents the tree root path for the terminal `"TYPE_RET_V"`, and the red path represents the tree root path for the terminal `"TOKEN_REG"`. We use these tree root paths to generate the AST paths, as Alon et al. [3] call them. These AST paths represent the shortest path between two leaf nodes. For each pair of leaf nodes, we find the first common element of the two leaf's tree root paths. Figure 4.6 provides an example for two possible AST paths. The first AST path connects the two terminals `"TYPE_RET_V"` and `"return-void"`. The path includes the root of the method, which means that their representative tree root paths only shares the root element. The second path connects the terminals `"TOKEN_REG"` and `"TYPE_RET_V"`. The common element between the terminals' tree root paths is `"I_STATEMENT_FORMAT35c_METHOD"` because both are part of the same call statement in the method.

From baksmali, we receive the abstract syntax tree for each activity class separately. One could paths for all leaves of each class, including class fields, methods, annotations, etc. However, the size and amount of classes often depend on the style of the programmer. An activity could be fully implemented in one class, or it could split into two or more classes with helper or utils classes. The same holds for class fields. Instead, we focus on methods, because most of an app’s behavior is defined within its methods. Since Android methods act mostly independently of each other and their class, we generate paths only within the context of methods.



**Figure 4.5:** Example for two AST root paths. The first is marked in orange and the second is marked in red. Both represent a leaf node connecting to the root of the method.

Depending on the size of the Android application, this approach can result in a vast amount of paths. That is why only paths of a certain minimum length are considered for training our model. We choose longer paths because shorter paths are typically also encoded in longer paths, therefore not providing additional meaning. For example, the path between "TOKEN\_REG" and "Landroid/app/Activity;" would be "I\_REGISTER\_LIST"- "I\_STATEMENT\_FORMAT35c\_METHOD", which is already included in the orange path. All of these generated paths are stored in path files, which contain the start terminal, the end terminal, and the path that connects them.



**Figure 4.6:** Example for two AST paths. The first path is marked in orange and the second path is marked in red.

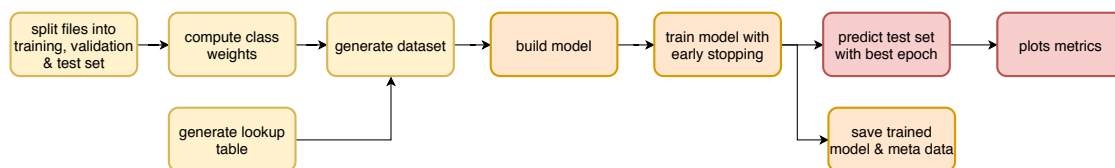
To create the code embeddings like Alon et al. [3] describe, we need to create a lookup table for all tokens. More specifically, we need one lookup table for the start and end terminals and one lookup table for the path elements. This is important because embedding layers require numbers as input, which they map to embedding vectors. So before embedding the values, we need to create dictionaries where each

token is mapped to a natural number. Alon et al. do not specify how they generate these dictionaries in their work. That is why we propose our own approach: Firstly, we create an empty dictionary for each of the lookup tables. For each path file, we get all unique start and end terminal and count how many times they occur. If the token is not yet part of the dictionary, we add it and its count to the dictionary. If the token is already present in the dictionary, we add the count from the current path file to the already existing count. Furthermore, we split the paths into tokens and add them to the path token dictionary in the same manner as the terminals. In summary, we are generating the term frequency of each terminal and path token, and store them in separate dictionary files.

## 4.3 Model Architecture

We train our dataset on two different models with slightly different architecture for processing the paths. For the first model, an RNN with LSTM cells is used to generate a representation for the AST paths. It is described more in detail in Section 4.3.3. Section 4.3.4 provides information about the second model, which uses a combination of convolutional layers and pooling layers to create a representation for the paths. Note that both models follow the same steps for processing the terminal tokens. Before going into more detail about these models, we describe in Section 4.3.1 how we generate the dataset both models use for training based on the previously created path files. Both the dataset generation and terminal processing are heavily inspired by Alon et al. [3].

### 4.3.1 Dataset Generation



**Figure 4.7:** Flow Chart of the model training: The steps in yellow mark the pre-processing steps, the steps in orange mark the model training and evaluation, and the steps in red mark evaluation and plotting.

To enable our machine learning model to process our input data, we need to generate a dataset from our training data. The steps to generate the dataset starts with creating the lookup tables for the terminal and the path tokens. These lookup tables map each token to a unique index and are required for generating the code embeddings because neural networks are geared towards processing numbers and not text, as described in Section 2.3.

The lookup tables are created based on the dictionaries that are described in Section 4.2. The dictionary files are parsed and all terminal tokens and path tokens are extracted. If their term frequency is higher than a given threshold, an index is assigned, and the token and its index are added to their respective lookup table. Table 4.1 lists the most common path tokens, their frequency, and the index that they were mapped to, according to our approach. The <PAD> and <UNK> token do not have a frequency attached because they are added by us manually. The <PAD> token represents all elements that were added as padding if the shape of the input is too small. The <UNK> token represents all unknown path tokens. For example, if a new, previously unseen app contains a path token that is not part of the training set, it is replaced with the <UNK> token. While the dictionary generation is our approach, we have drawn inspiration for the <UNK> and <PAD> token from the implementation of code2seq<sup>1</sup>.

<sup>1</sup><https://github.com/tech-sr1/code2seq>

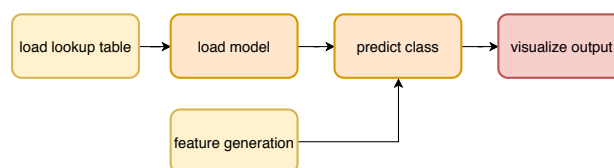
Token	Frequency	Index
<PAD>		0
<UNK>		1
I_ORDERED_METHOD_ITEMS	689319949	2
I_METHOD	436300192	3
I_STATEMENT_FORMAT35c_METHOD	372918668	4
I_METHOD_PROTOTYPE	274002693	5
I_ANNOTATION	118679367	6
I_ANNOTATIONS	118605856	7
I_SUBANNOTATION	116790404	8
I_ANNOTATION_ELEMENT	97448683	9
I_ENCODED_ARRAY	77313964	10
I_CATCHES	59724864	11
I_CATCH	52396025	12
I_LABEL	43420331	13

**Table 4.1:** Example for the lookup for path tokens.

The next step is to collect all path files of the apps used for training and evaluation. These files are split into a training, validation, and test set. We use the training dataset for training our model, the validation set for regularization, and the test set for evaluating the model performance. Since the number of apps within each class might differ, we compute the ratio of how much they differ, i.e., the class weights. This is important because otherwise, the model might end up merely predicting the majority class. To avoid this, we include class weights in the training process so that the imbalance is taken into consideration.

The three datasets are generated in the same manner, namely by extracting the paths and the terminals from each path file. If the number of paths is too small, padding is added to ensure that all samples have the same shape. If the number of paths is larger than the expected shape, paths are randomly drawn from the path file for training, in the same manner as proposed by Alon et al [3]. After padding and sampling, the index of each of the terminals is extracted from the terminal lookup table. If the terminal does not exist in the lookup table, it is replaced with the <UNK> token. The paths are split into their tokens and replaced with the indices from the path token lookup table. The table lookup is performed in the same manner as with the terminals. The longest path in the dataset represents the maximum path length.

After generating the dataset, the model is built and trained using the training dataset. The validation dataset is used to prevent overfitting of the model on the training data and for early stopping, which is described in Section 2.3.8. The trained model is then evaluated on the test set. The test set is not involved in the training stage and therefore provides an independent measure for the performance of the model on new, unseen data. This cannot be done using the training or validation set since they are directly or indirectly involved in the training process. Plots are generated based on the metrics of each of the datasets. Lastly, the lookup tables and weights are saved for future purposes.



**Figure 4.8:** Flow Chart of model evaluation: The steps in yellow mark the pre-processing steps, the steps in orange mark the model evaluation, and the steps in red mark result representation.



When applying the previously trained model on new applications, firstly, the feature generation steps are applied to the new apps to generate the path files. The lookup tables used by the trained model are loaded. It is essential that the same lookup tables are used. Otherwise, different indices might be assigned to the tokens. Because the model only recognizes the indices and not the tokens, it would misinterpret the input and lead to an incorrect prediction. The next step is to load the model. Loading the model can either be done by saving the entire model and loading it or just saving the model's weights. If only the weights are saved, the model has to be built again. Afterward, weights are loaded into the new model. Just saving the weights, requires more steps but has the advantage of saving memory space. Because the memory cost outweighs the additional time to re-build the model, we opt for saving only the weights. After loading the model, a dataset that contains the new applications is generated in the same manner as the test set. The model can then make predictions for each of the applications in the dataset. Lastly, the outcome of each prediction is visualized using an interactive dashboard.

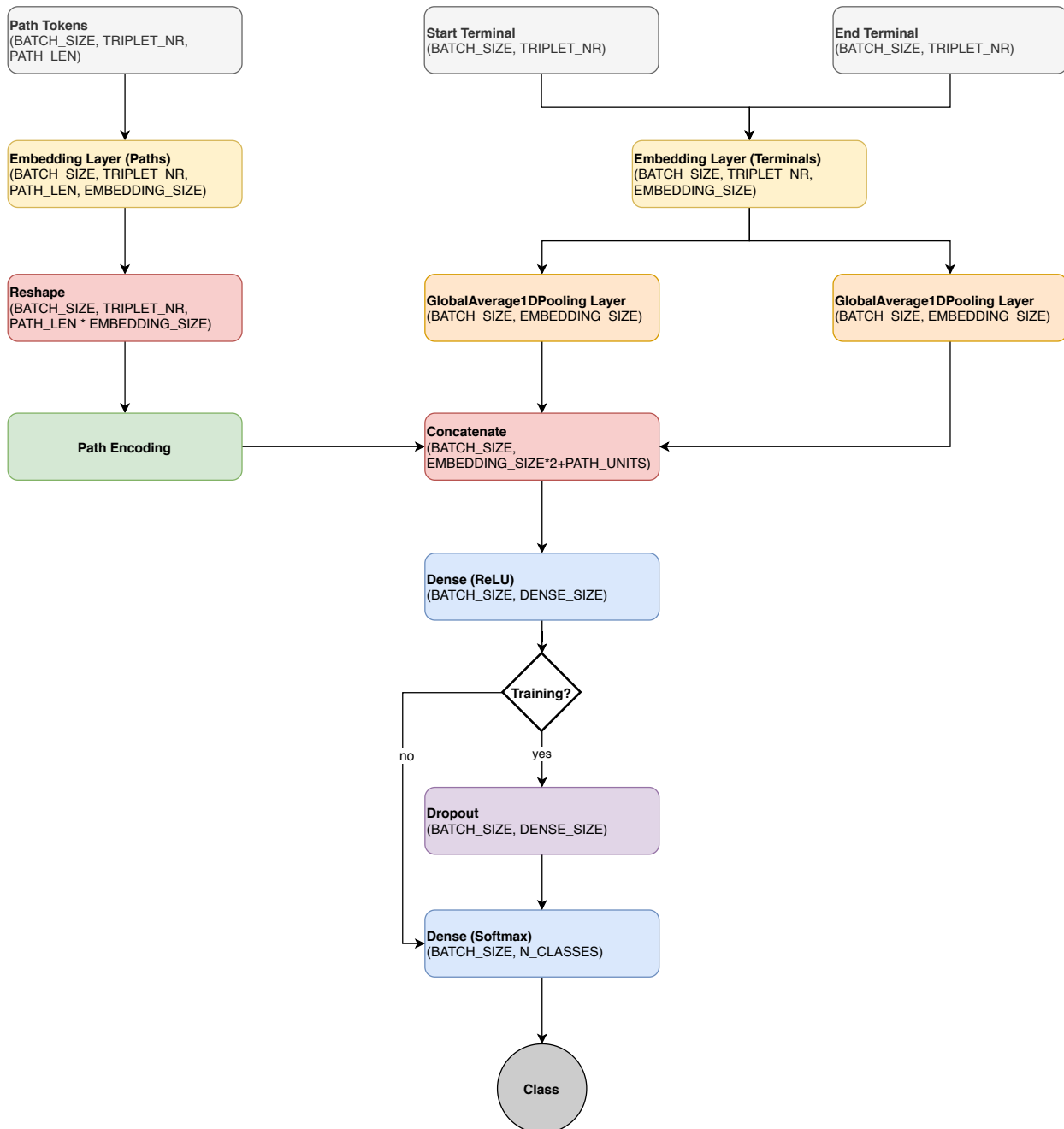
### 4.3.2 Model Generation

In this thesis, we compare two different models to see which best fits our task of classifying Android applications. Both models draw inspiration from code2seq [3], especially when it comes to the encoding of the tokens. As previously described, this starts with the generation of the dataset and lookup tables. These lookup tables are used to map the terminals and path tokens to indices. This step is necessary to generate the embeddings, which are trained using embedding layers, one for the terminals, and one for the path tokens.

The embedding layers are randomly initialized and trained as part of the classification tasks. The embedding layers map the index from the lookup tables to high-dimension embedding vectors, such that similar tokens are mapped to similar vectors. In our case, this means that tokens that typically occur in one class should be mapped to similar vectors. Since embedding layers map an integer value to a vector, the outcome of the layers receives an additional dimension. In order to further process the paths, they are reshaped to fit the following layers. Which layers are applied next, depends on the models.

The first model makes use of recurrent neural networks with LSTM cells, similar to code2seq. Recurrent neural networks are a popular tool for dealing with sequential data, such as text or in our case paths. The second model makes use of convolutional layers, which is inspired by the field of sentence classification. While recurrent neural networks have the advantage of capturing long-range dependencies within the data, convolutional neural networks can identify specific patterns in the sample independent of their position. This pattern recognition makes them useful for classification because they can detect key phrases that are common for a particular class. However, it is not always the case that CNNs outperform RNNs for classification. Tang et al. [26] show that in their case, RNNs also provide excellent performance on document-level sentiment classification. Yin et al. [29] provide a comparative study on the performance of CNNs and RNNs for different NLP tasks. In their work, recurrent neural networks with *gated recurrent units* outperform convolution neural networks for text classification tasks. However, we cannot know whether this is also the case for our setup because we are working with source code and not natural text. That is why we train our dataset on two models, one with an RNN and one with a CNN, to see which performs best.

The task of both is to generate a common representation for all paths of a sample. The common representations for all start and end terminals of samples are generated using an embedding layer for the terminals. Same as with the paths, the indices of the terminals are mapped to high-dimensional vectors. In this case, however, pooling layers are used to generate a common representation for all start terminals and for all end terminals. These pooling layers compute the average of each dimension of the embedding vectors. So for each dimension of the embedding vector, the average between all terminals of a sample is computed. The outcomes are concatenated with the representation of all paths. A dense layer is used to learn a denser representation of the terminal-path triplet to reduce the size of the outcome. A rectified linear unit is used for activation. It is described more in detail in Section 2.3.8. Reducing the size in a



**Figure 4.9:** Overview of the parts of the model architecture that is shared between both models.

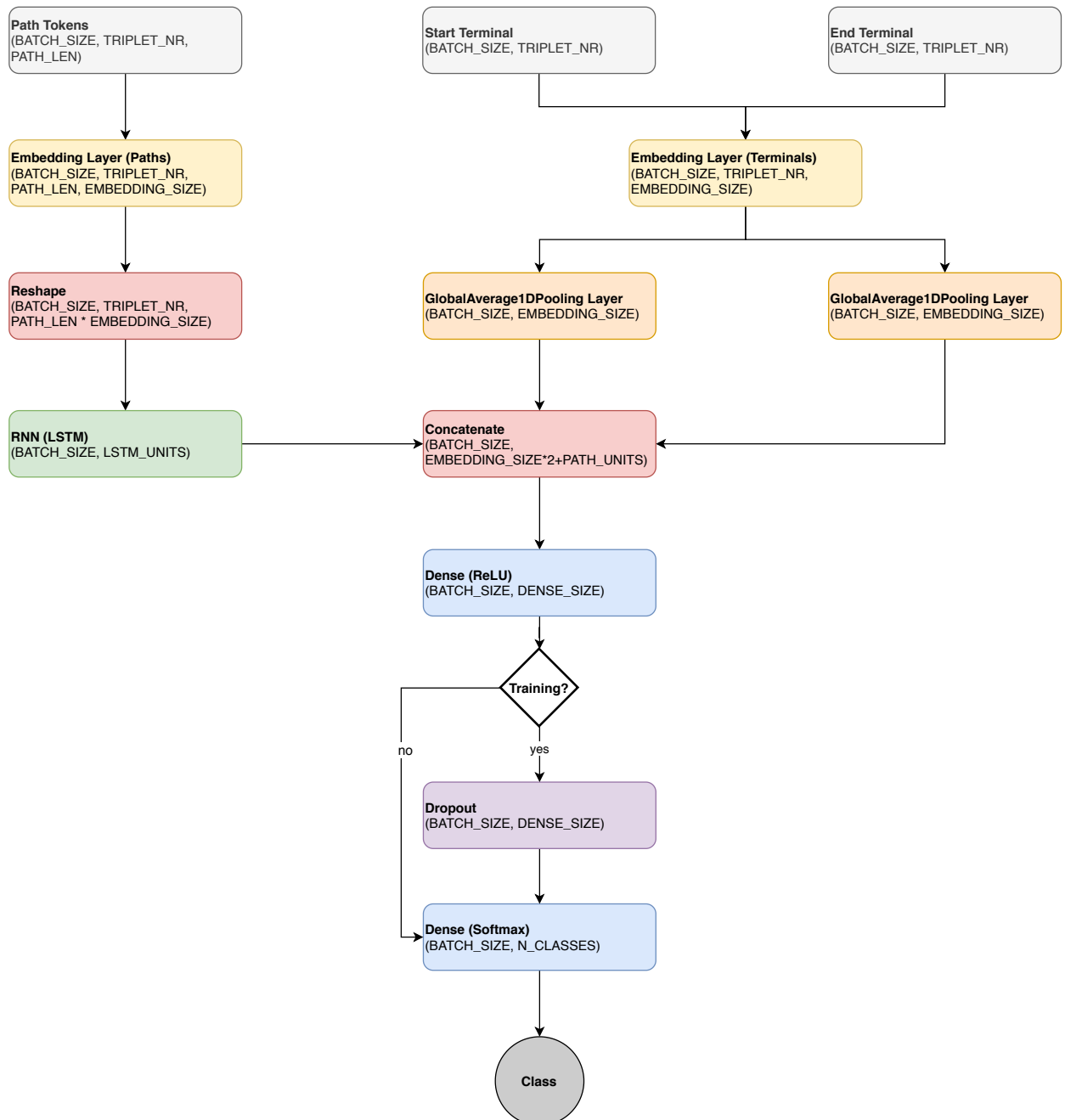
separate layer helps the last layer, to generalize better.

The most significant difference between our approach and code2seq is the task. Code2seq’s goal is to generate descriptions of small pieces of code, and our task is to classify large pieces of code, i.e., Android applications. That is why both of our models process the trained representation of the paths and terminals differently than code2seq. Namely, our last layer is a dense layer with a softmax activation function, which generates a probability distribution for the given amount of classes. The class with the highest probability represents the model’s prediction for an application’s class. Figure 4.9 provides an overview of all the described layers and how they are combined in our model. Figure 4.10, on the other hand, gives an overview of how the dimension of the paths and terminals changes after applying different layers.

The error of misclassification is computed by the *cross-entropy error function*, the most common error

function for classification. To optimize the model, the optimization algorithm *adaptive moment estimation* (Adam) is used. It is based on *gradient descent*, a common optimization algorithm for training neural networks. To avoid overfitting, we employ two regularization techniques: early stopping and dropout. Both are described in Section 2.3.7.

### 4.3.3 Model 1: RNN with LSTM cells



**Figure 4.11:** Overview of the Model Architecture where an RNN with LSTM cells is used for processing the paths.

Recurrent neural networks are a popular choice for processing text because it learns patterns across time, making it perfect for capturing sequential data. Section 2.3.4 describes in detail, how recurrent neural networks are able to process sequential data.

This model is inspired by code2seq, which also uses a recurrent neural network with long-short term memory cells to learn the representation of the AST paths, as described in Section 4.1. However, because the tasks differ between code2seq and our approach, we have made slight changes to the architecture. For instance, code2seq's model has an encoder-decoder architecture. The encoder-decoder architecture is a combination of two RNNs that are connected. The first RNN, the encoder, encodes the input text into a vector representation, and the second RNN, the decoder, takes the output of the encoder and generates output text from the input. This model is very popular with machine translation, but also with generating descriptions from source code, which is the goal of code2seq. This is described in further detail in Section 4.1.

Since our goal is classification and not text generation, the decoder RNN is unnecessary and is therefore replaced by a softmax dense layer, which generates an output distribution for the classification classes. Furthermore, code2seq uses bi-directional LSTM cells for the RNNs. The purpose of bi-directional LSTMs is to also encode sequences in reverse order to combat the loss of context from early sequence elements. However, this significantly increases the number of parameters that need to be trained, making training the model vastly slower and increasing the amount of memory required for training. Space is not a large problem for code2seq, because it operates on methods. We, on the other hand, train on entire applications, which are significantly larger than just one method. That is why we replace the bi-directional LSTM cells with uni-directional LSTM cells to reduce the number of parameters that need to be learned and, therefore, the time and memory to train the model. Figure 4.11 gives an overview of the model's architecture.

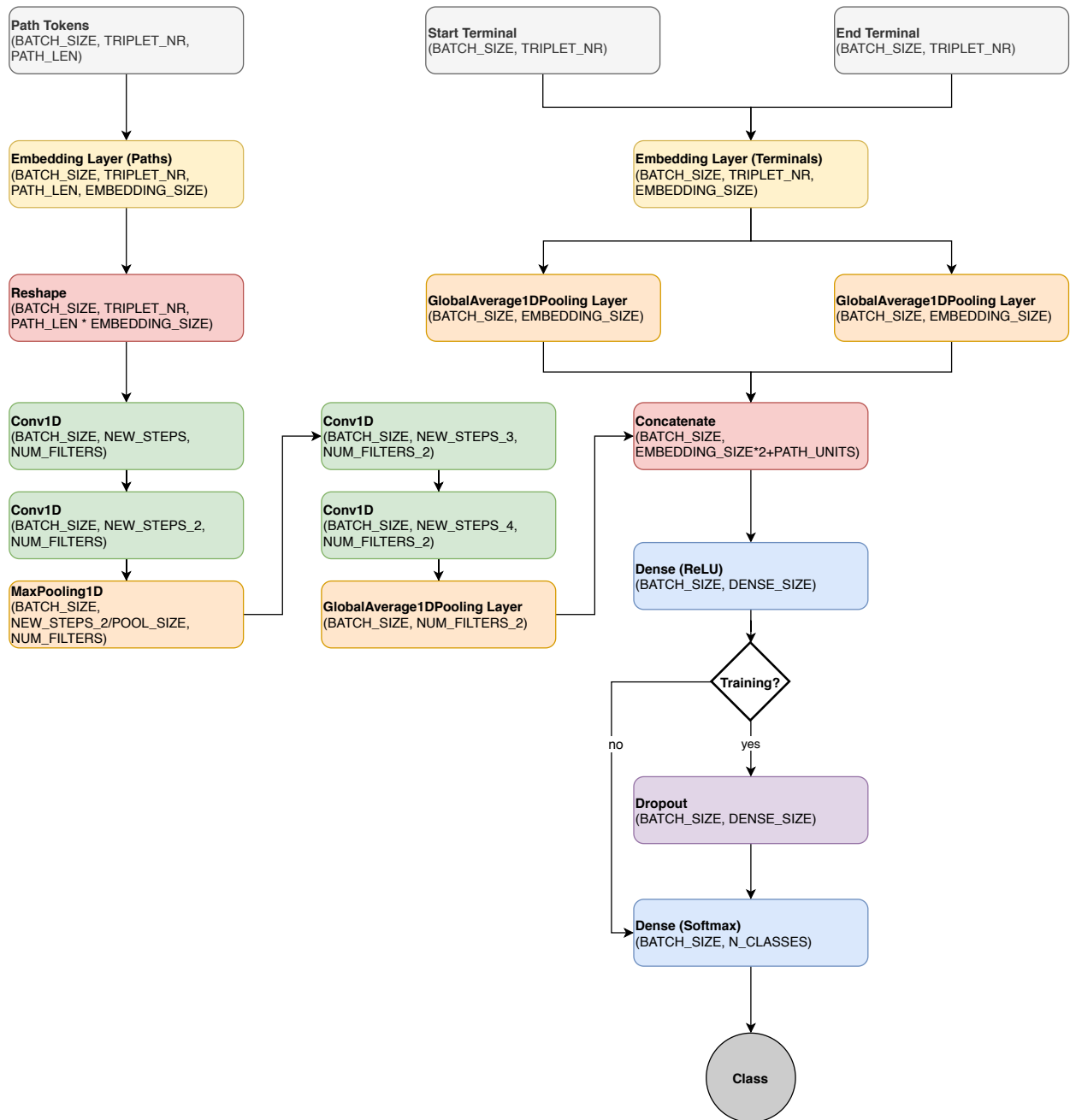
#### 4.3.4 Model 2: CNN

Convolutional neural networks are typically used in computer vision. Recently, they have also become a popular choice for certain natural language tasks. Recurrent neural networks learn patterns across time and are therefore able to capture long-range dependencies. Convolutional neural networks, on the other hand, learn to find patterns independent of position within a sample. They can detect patterns in text, independent of the position. CNNs are a popular choice for sentiment analysis or text classification since CNN can detect specific key phrases in the text that are typical for a particular class. Another reason for choosing CNNs over RNNs is they are much faster because they are implemented on the hardware level of the GPU. That is why we propose a model that makes use of CNNs instead of RNNs for processing the paths.

There are many different possibilities of how many convolution layers and pooling layers can be used and how they are combined. The simplest model would be one convolutional layer and one pooling layer. It can be tricky to find a model that is powerful enough to find significant patterns within the data and keeping it simple enough to avoid overfitting on the training data. While keeping this in mind, we propose a model of two consecutive convolutional layers and one max-pooling layer. The max-pooling layer is followed by two consecutive convolutional layers and a global average pooling layer. Figure 4.12 shows the different layers of the CNN model.

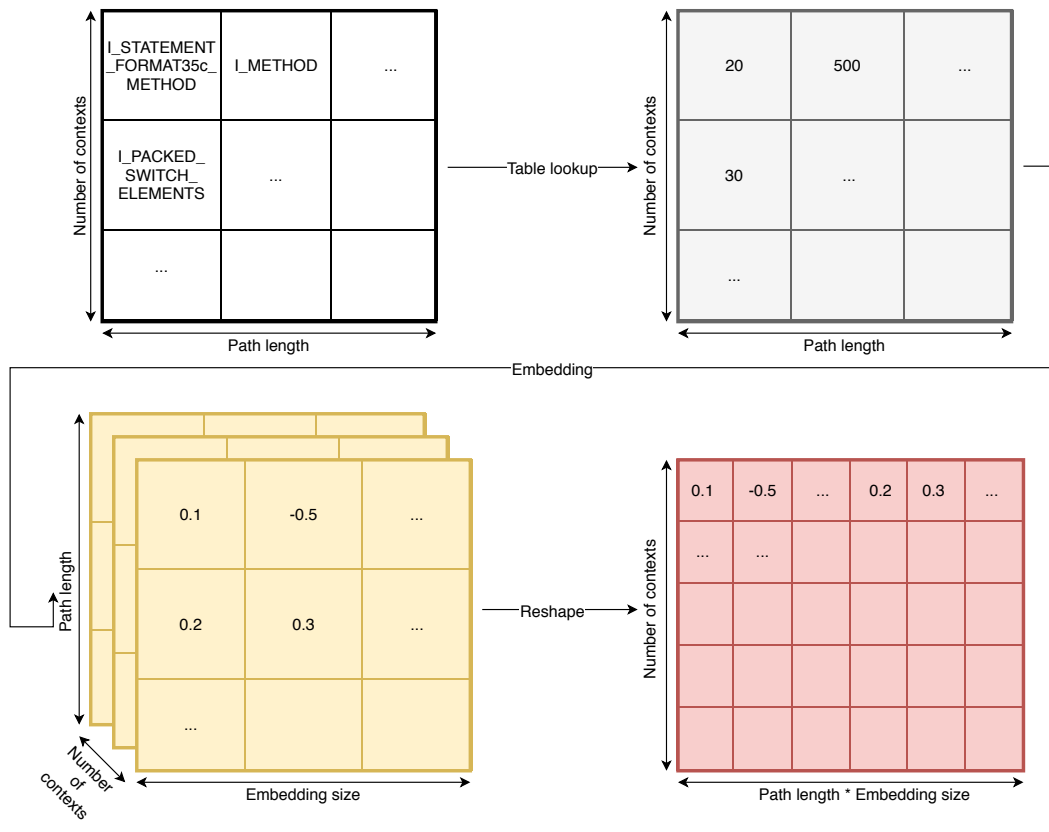
The purpose of the first two layers is to find specific patterns within the paths. They consist of `NUM_FILTER` filters of a given filter size, `FILTER_SIZE`. Each filter learns one feature in the neighborhood of `FILTER_SIZE`. So the first layer learns `NUM_FILTER` features from the input, which leads to an output shape of `(NEW_STEPS, NUM_FILTERS)` where `NEW_STEPS` represents the outcome of each filter, which is typically smaller than the input size unless a filter of size 1 was chosen. How the output is generated is described in Section 2.3.5. The output of the first layer is the input to the second CNN layer, which again consists of `NUM_FILTER` filter of size `FILTER_SIZE`. The purpose of this layer is to learn features based on the output of the first layer. Stacking multiple layers enables the model to learn more complex features and, thus, detect more complex patterns. The max-pooling layer summarizes these findings by only choosing the maximum value within a specified window of size `POOL_SIZE`. The purpose of this layer is to prevent overfitting and reduce the complexity of the output. The output is reduced to `1/POOL_SIZE`

of its original shape.

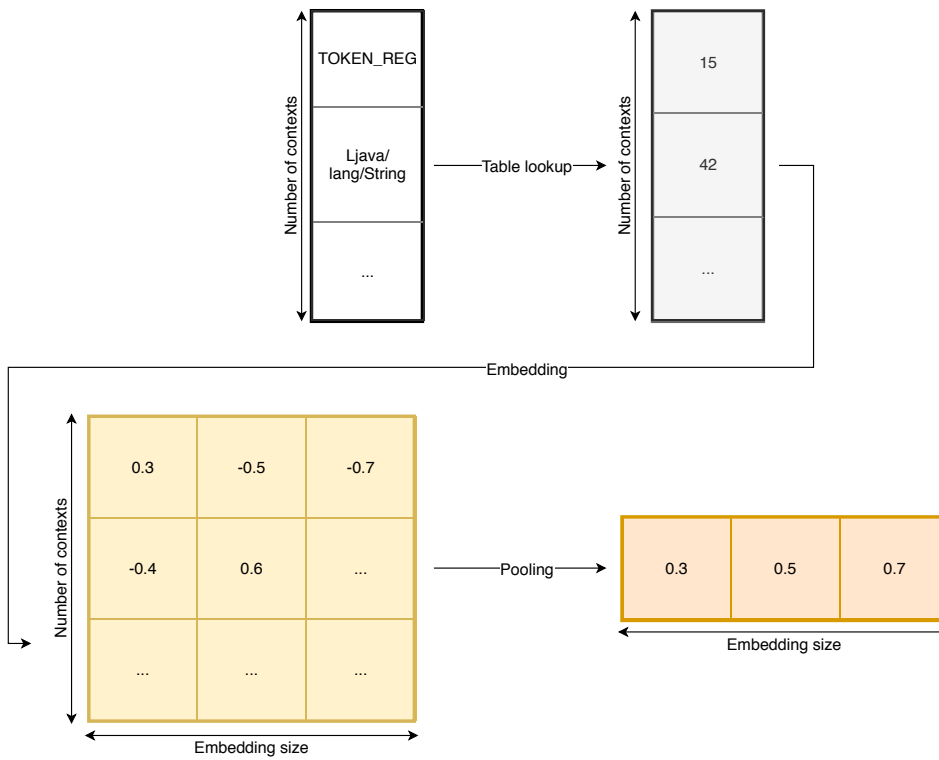


**Figure 4.12:** Overview of the Model Architecture where convolutional layers are used for processing the paths.

Another two convolutional layers look for patterns in the outcome of the max-pooling layer. Both layers consist of NUM\_FILTER\_2 filter of size FILTER\_SIZE\_2. Again, their purpose is to extract NUM\_FILTER\_2 features from their input. The outcome is input to an average-pooling layer. Similar to the max-pooling layer, its purpose is to prevent overfitting. However, no maximum value is chosen, but each filter is summarized by computing its average value. This reduces the outcome to a vector of NUM\_FILTER\_2 values. This outcome is combined with the representation for the start and end terminal.



(a) Dimension changes of the paths



(b) Dimension changes of the terminals

Figure 4.10: Overview of how the dimensions of the input changes while generating the model.

## Chapter 5

# Implementation

In this chapter, we describe the implementation of our approach. We call our application ALPS, Android Language Processing System. It supports the creation of AST trees and paths, the generation of datasets, and the training of an LSTM or CNN model. Furthermore, a trained model can be loaded and applied to a provided Android app. The result of the model is visualized in an interactive dashboard. Figure 5.1 provides an overview of the components of our implementation.

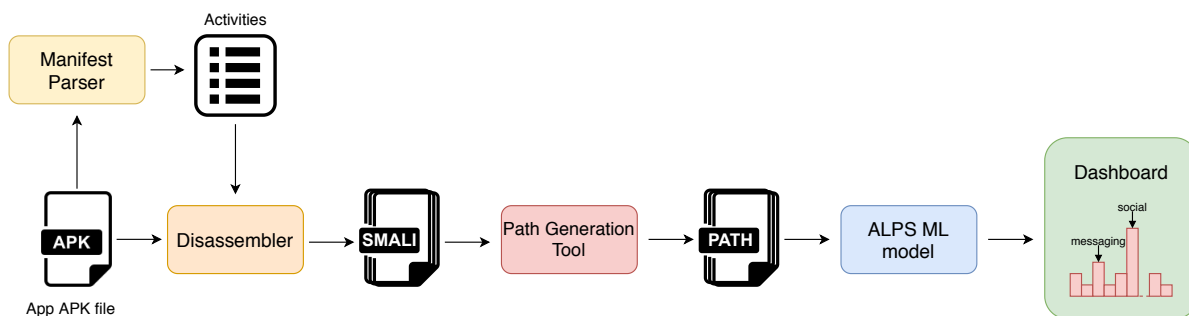


Figure 5.1: Overview of the components of our implementation.

### 5.1 Architecture

This section provides an overview of the ALPS architecture. It can be primarily broken into a Java component and a Python component. The Java component consists of a `baksmali` module and a path extraction module. The Python component, on the other hand, consists of a feature module, a models module, and a dashboard module. The Java component is described in further detail in the following listing.

`baksmali` this module is a modified fork of `baksmali`<sup>1</sup> project that consists of multiple modules, namely the `smali` module (assembler for Smali to dex) and the `baksmali` module (disassembler for dex to Smali). Our modifications include the generation of an AST during disassembly of an APK file and the creation of AST paths.

`path generation` is our extension to the `baksmali` project to support the creation of AST paths from a given AST, which is provided by the disassembler.

<sup>1</sup><https://github.com/JesusFreke/smali>

`path extraction` parses the Manifest XML file from an APK file and retrieves the activity classes and calls `baksmali` for the retrieved classes.

The Python component, on the other hand, consists of a feature module, a models module, and a dashboard module. The following listing describes these modules in greater detail.

`feature` calls the Java path extraction tool to generate the path files from input APK files and generates the dictionary files for the token lookup tables.

`models` contains everything connected to the model, such as loading the dataset, building and training the model, different hyper-parameters configurations, and loading a previously trained model.

`dashboard` enables the user to upload new applications, receive a prediction based on a previously trained model, and visualize the outcome interactively.

## 5.2 baksmali: Path Generation from APK

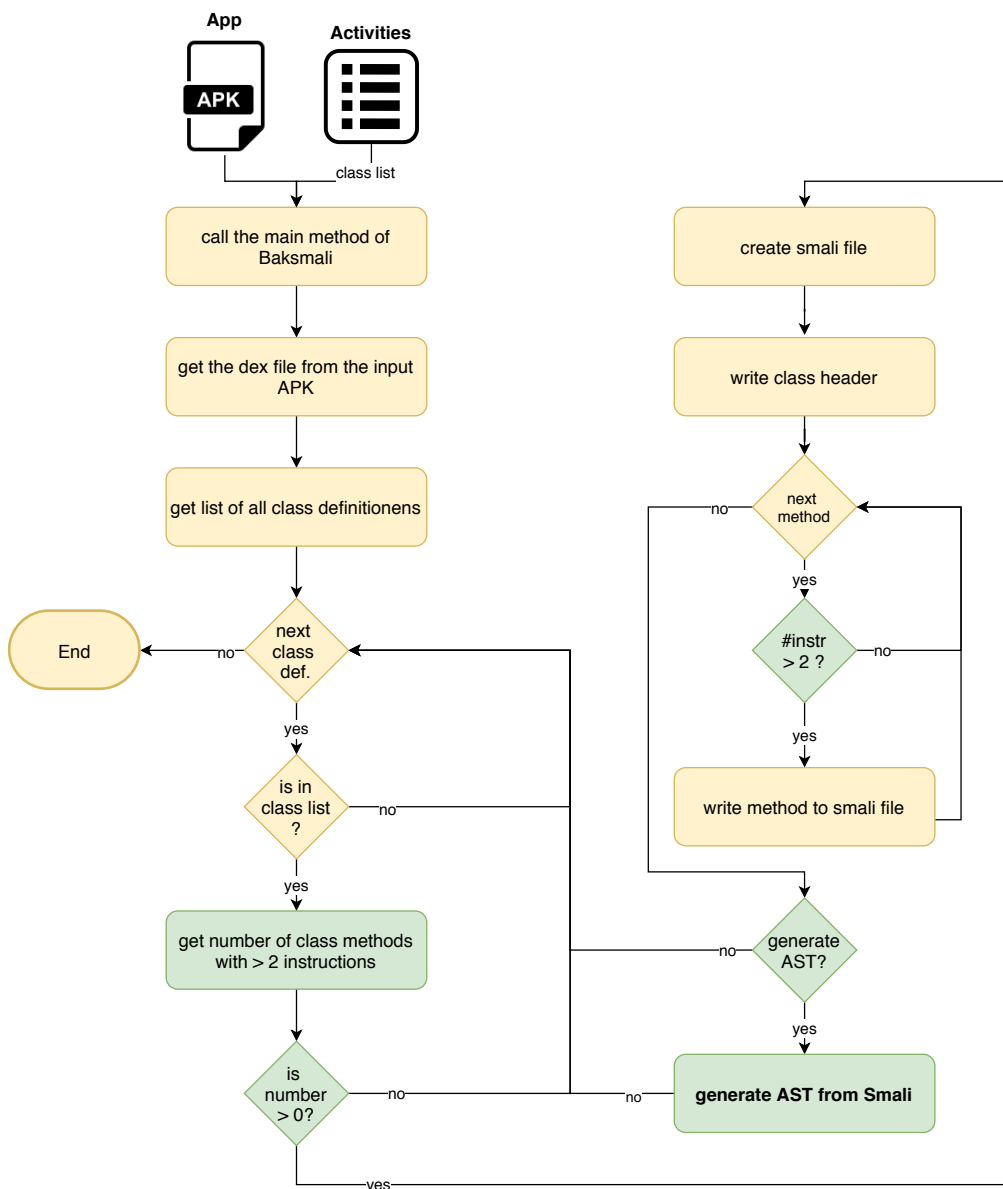
The following section describes our implementation of the generation of the AST paths based on an Android APK file. This functionality is realized in the Java component of ALPS. As previously described, this module consists of two modules, `path extraction` and `baksmali`. The `baksmali` module also contains the `path generation` sub-module.

The `path extraction` module extracts and parses the Manifest XML file from a provided APK file. Given the Manifest XML file, it retrieves the list of activity classes. Afterward, it calls the modified `baksmali` module with the APK file, the previously extracted activity classes, and some hyper-parameters such as the minimum path length. This is described in further detail in Section 4.2.

The `baksmali` project consists of multiple sub-modules. We focus on the sub-modules that are relevant for the ALPS project, which include the `baksmali` and the `smali` module. The `baksmali` sub-module handles the disassembly of Dalvik byte code to Smali code. We use this sub-module to generate the smali code for the provided activity classes. The `smali` sub-module, on the other hand, handles the assembly of Smali code to Dalvik byte code. During assembly, it generates an AST tree of the provided Smali code. We make use of this functionality to create the AST tree for our disassembled classes. Furthermore, we extend this sub-module with a `path generation` sub-module that generates AST paths based on a provided AST tree.

Figure 5.2 represents the steps that are performed to disassemble specific classes from an APK file using `baksmali`. These steps are marked in yellow. Furthermore, it shows the steps to generate an AST tree. The AST generation is an extension implemented by us and is marked in green. The process is triggered by calling the main method of `baksmali`. Parameters include the path to an APK file and a list of classes, in our case activity classes. `Baksmali` then retrieves the DEX file from the APK, which contains the compiled source code of an Android application. From this, it retrieves the list of all class definitions and iterates over it. For each class definition, a process is created for disassembly to build a speed-up in the disassembly process. Each process verifies whether its class is part of the input class list if one was provided. If the class is not part of the input list, it is skipped, as we only want to disassemble the classes that have been provided by the class input list. However, if the class definition is part of the input list, we retrieve the number of significant methods within the current class. We define a method as significant if it has more than two instructions. Methods with less or equal than two instructions typically carry little meaning for the overall application, such as getter and setter functions. That is why we ignore them for our purposes. If the current class contains no significant method, it is not disassembled. Otherwise, a Smali file is created, and the Smali class header is written to the file. For our purposes, we only look at methods. Therefore we only write significant methods to the Smali file. Next, if an AST should be created,





**Figure 5.2:** Flow chart showing the steps for disassembling an APK file for a given class and generating its AST tree.

we generate the AST from the Smali file, as described in Figure 5.3. Otherwise, we look at the next class definition.

Figure 5.3 provides an overview of the steps to generate an AST tree. The implementation is located in the `smali` module. A Smali file is supplied as a parameter. In our case, this is the previously generated Smali file from the disassembly process. First, the Smali file is read into memory, and the Smali lexer and parser are applied to the file. The lexer and parser are both generated using the *ANTLR*<sup>2</sup> project, which is a tool that creates parsers from a provided language grammar. These parsers can build and walk parse trees. We utilize this ability to generate the parse tree of the Smali input file. The generated parse tree can be stored in the DOT format, a graph description language. Typically this done using *ANTLR*'s `DOTGenerator` class. However, this is mainly intended for viewing purposes and not further processing. That is why it

<sup>2</sup><https://www.antlr.org/>

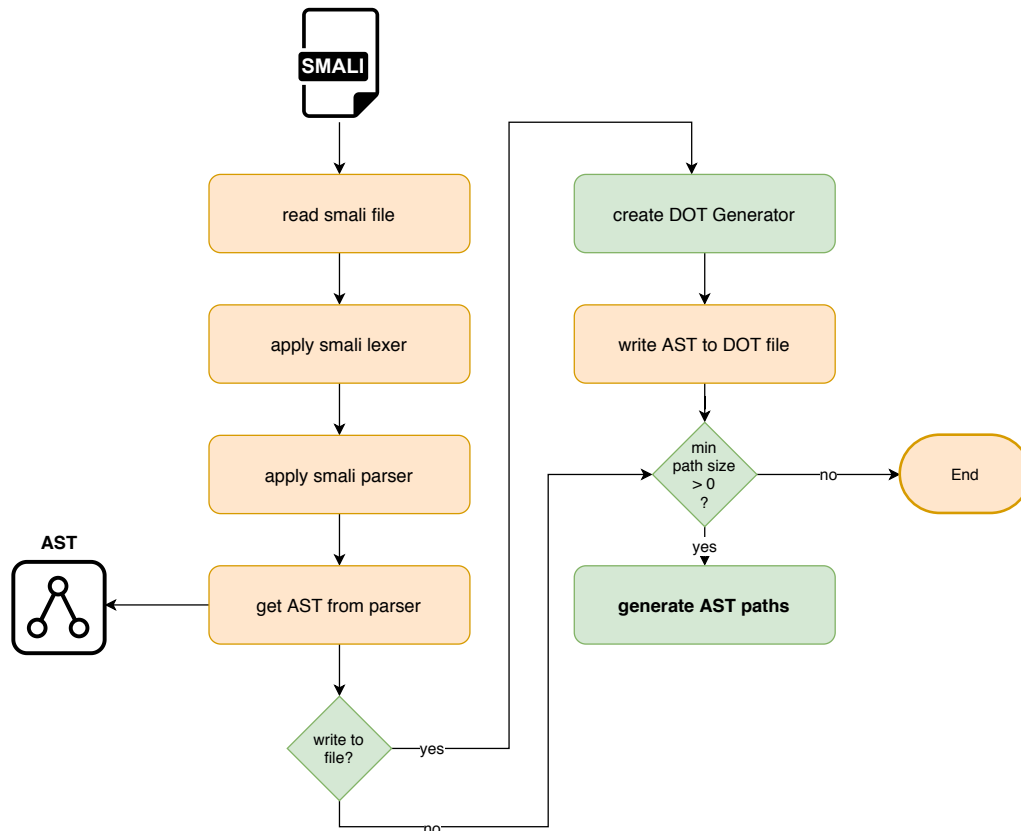


Figure 5.3: AST Generation

shortens strings to a fixed length. However, this might lead to two different strings being interpreted as the same. For example, two expressions call two different methods from the same class. But the class name is too long, so the method strings are cut off and end up being the same. This string shortening could lead to incorrect results in our model. Therefore, we implement the `FullStringDOTGenerator` class, which extends `DOTGenerator` to always write full strings. Furthermore, it replaces tokens that represent specific registers with the value `TOKEN_REG`. We replace these values because they represent little meaning in the overall functionality of an application. Using the `FullStringDOTGenerator`, we write the AST to a DOT file. The next step is to generate AST paths based on the previously generated AST if a minimum path size was defined. This step is visualized in Figure 5.4.

As described in Section 4.2, we only generate paths for class methods as they define most of an application's behavior. That is why the first step is to retrieve the `I_METHODS` node, the root node of all methods in the AST. The next step is to generate the tree root paths, which are all paths from the AST's leaves to its method's root element, depicted by the `I_METHOD` token. This process is depicted in Figure 5.5. Using this list of tree root paths, we compare all paths with each other, given they are different and come from the same method. If two different paths are part of the same method, we find the shortest path between the paths' leaf node. This process is depicted in Figure 5.6. If the resulting path is longer than the provided minimum path length, it is added to the path list, which is returned by the method.

The steps for generating tree root paths are depicted in Figure 5.5. Our implementation creates all tree root paths by searching depth-first for leaf nodes and then generates the path recursively bottom-up. We call the tree root path algorithm with the node `I_METHODS` as the node and `NULL` as the method. Each node that is not a leaf calls the tree root path function for each of its children as the node parameter and its method as the method parameter until a leaf node is detected. The tree root path algorithm starts by initializing a list for the created tree root paths. If the current node is a leaf node, we create a new path

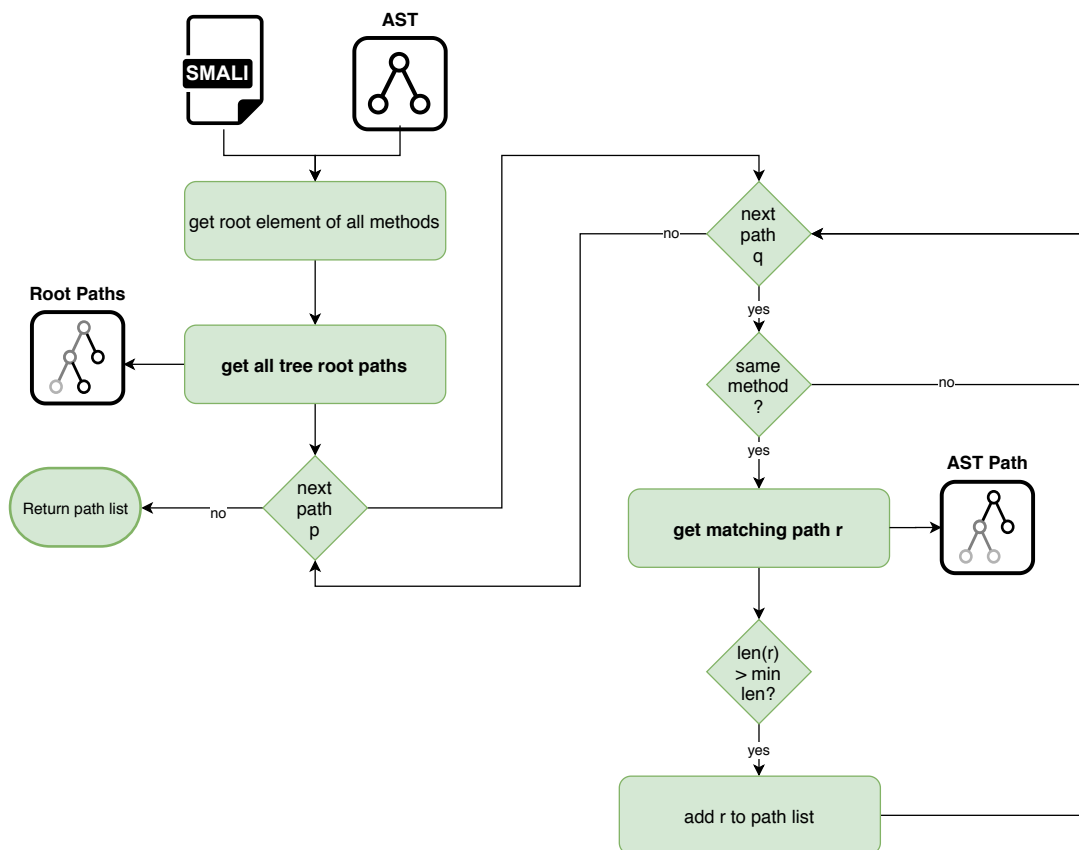
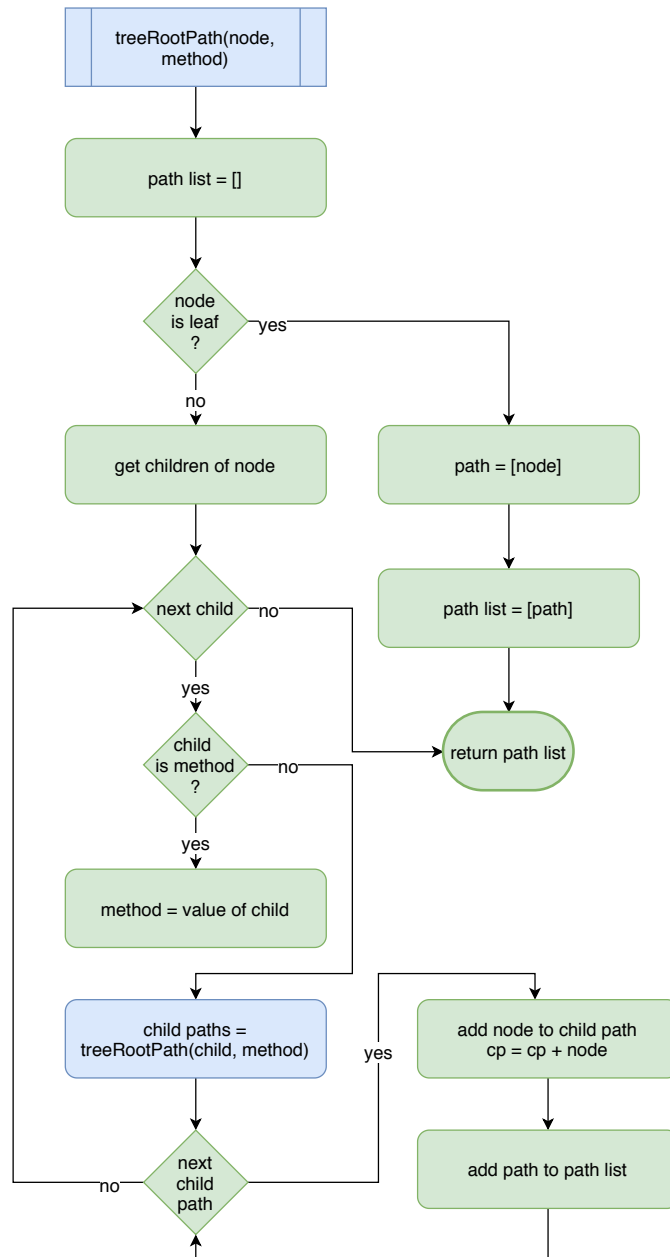


Figure 5.4: AST Path Generation

with the current node as its element, add the path to the path list and return it. If the current node is not a leaf node, we iterate through its children. If the current child is a method, we set it as the current method. Otherwise, we call the tree root path function with the current child and the current method name and receive all child paths. For example, if the current child is the parent of one leaf node, this function call returns a list of one path with one element, the leaf node. We then add the current node to this path and add the path to the path list. This process is performed for all of the current node’s children. We return the current path list to the current node’s parent and repeat the process until all nodes have been visited.

The matching of two tree root paths to receive an AST path is visualized in Figure 5.6. Given two tree root paths  $p$  and  $q$ , we generate the AST path, i.e., the shortest path between two leaf nodes, by finding the first matching elements between the two root paths. By the nature of our tree root path algorithm, the nodes of tree root paths are sorted bottom-up, i.e., the first element is always a leaf node, and the last node is the method node. So, by choosing the first matching element of two tree root paths, we receive the deepest matching node and, therefore, the shortest path between two leaf nodes. We begin by creating an empty AST path. If the tree root paths are not from the same method or if they are the same path, we return the empty AST path. Otherwise, we get the first element of the path  $p$  and path  $q$ . If they are not the same element, we choose the next element of path  $q$ . We repeat this for the next node of  $p$  until we find a match. If we cannot find a match, we return the empty AST path. However, if a match was found, we retrieve the partial paths of  $p$  and  $q$  from their leaf to the matching element. We reverse the nodes of path  $q$  and combine the partial path of  $p$  and the reversed partial path of  $q$  to receive the AST path.



**Figure 5.5:** This Figure shows the generation of all tree root paths for the methods of a given Smali files. The recursion step is marked in blue.

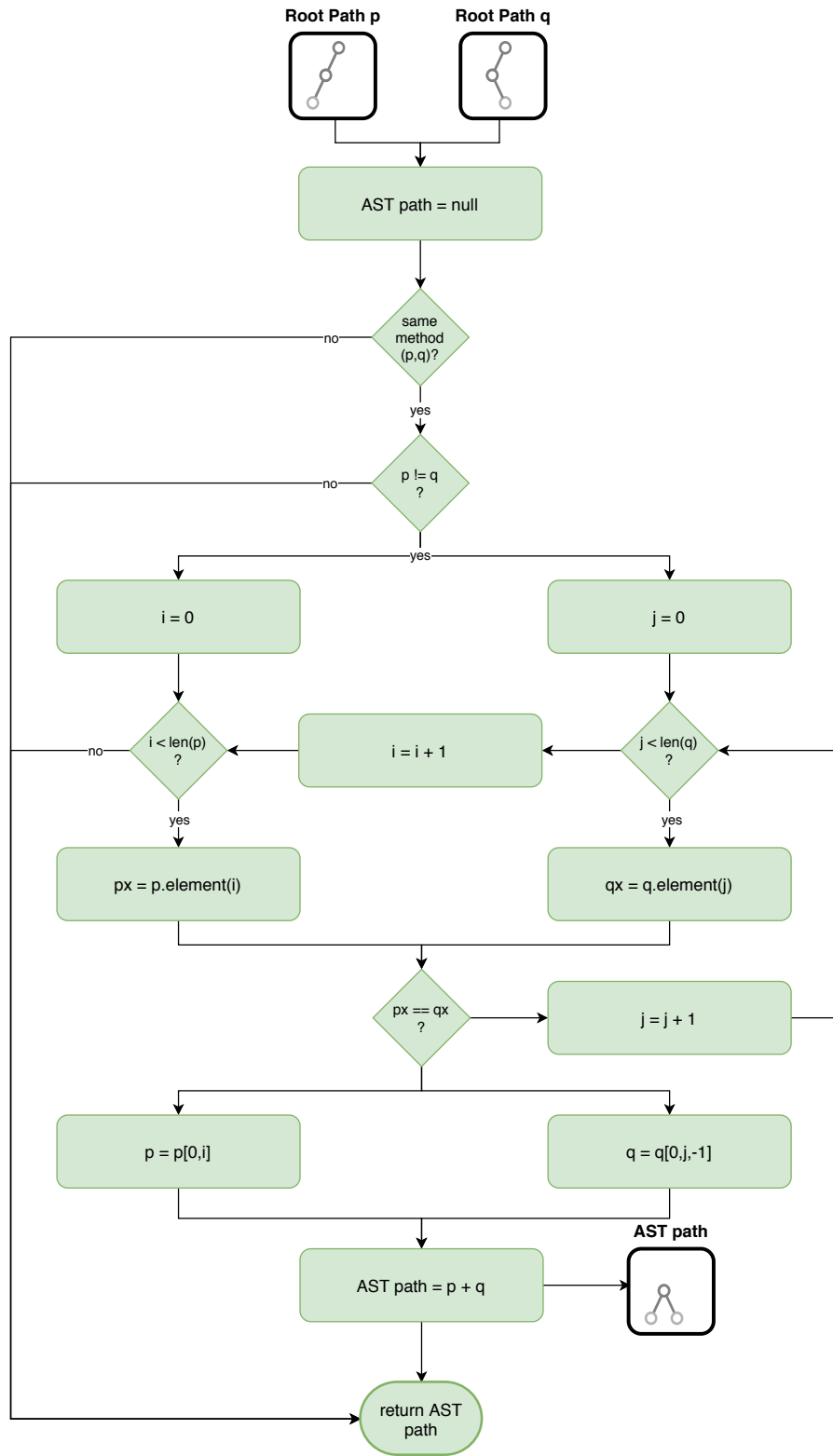


Figure 5.6: AST Tree Root Path Generation

### 5.3 feature: Path & Dictionary Generation

The feature module serves two functionalities, to call the Java path generation tool efficiently, and to generate the dictionaries used for the token lookup tables. The Java path generation module is designed to create the paths for one Android APK file. However, to train our model properly, we need to generate paths for a large number of APK files for the different categories. That is why the Python path generation module makes use of multi-processing. For each APK file in the provided input directory, a process is created. Each process calls the Java path generation tool with the provided APK file, retrieves the generated output files, and logs any errors that might have occurred.

The path files alone do not suffice to train our model. We also need to create token dictionaries for the lookup hash tables of our model. Again, we make use of multi-processing. For each of the provided categories, we create a process that iterates over the path files of Android apps from that category. We read each path file into a *pandas*<sup>3</sup> data frame. This data structure is optimized for data analysis and manipulation and enables us to retrieve the frequency of each token in a simple and fast manner. After retrieving the terminal tokens and path tokens, we sort them by their frequency and store them in dictionary files.

### 5.4 training: Model Training

The training module serves the purpose of training our machine learning models. The responsibilities are divided into the following sub-modules.

`config` contains all model hyper-parameters and configuration variables.

`dataset` creates and initializes the lookup tables and creates the training, validation, and test set.

`models` can build, train, and load the LSTM and CNN models.

`plotting` creates plots for various metrics such as the confusion matrix.

#### 5.4.1 config: Configuration

The `config` sub-module consists of a `Config` class that holds all hyper-parameters and additional variables such as the path file location. Furthermore, the module contains several instances of the `Config` class, which covers various training and deployment scenarios. The `Config` class holds the following parameters.

`CATEGORY_LIST` represents the list of classes to consider for model training.

`MAX_TRIPLET_NUMBER` provides the maximum number of triplets considered for each application.

`EMBEDDING_SIZE` is the dimension of the token embeddings generated while training.

`BATCH_SIZE` gives the number of samples to consider for each training batch.

`NUM_EPOCHS` represents the maximum number of training epochs.

`APPS_PER_CATEGORY` provides the number of apps to consider for training for each category.

`TRAINING_SPLIT` is the percentage of apps used for training.

`VALIDATION_SPLIT` is the percentage of remaining apps used for validation; the remaining apps are used for testing.

`SHUFFLE_SIZE` gives buffer size used for shuffling the training dataset to prevent overfitting.

---

<sup>3</sup><https://pandas.pydata.org>

`DENSE_LAYER_SIZE` represents the number of units for the dense layer that combines the path representation with the terminal representation.

`MIN_TOKEN_THRESHOLD` is the minimum number that a token has to occur to be included in the lookup table.

`SCENARIO_NAME` provides the name of the output folder where all required data is stored i.e., the model weights, config values, training logs, evaluation plots, dictionary.

`EARLY_STOPPING_PATIENCE` gives the number of epochs without improvement, after which training will be stopped.

`DROPOUT_RATE` is the fraction of the input units to drop while training.

`MAX_PATH_LENGTH` represents the maximum length of paths considered for training.

`SAVE_MODEL` is a boolean variable that defines whether the current model should be saved to file.

`INPUT_DIRECTORY` provides the directory where the path files are stored.

`DICTIONARY_FILES` gives the path to the input dictionary file used for the lookup tables.

We are working with multiple deployment environments with varying computational power. Therefore, we create an instance for each of these environments. For each setting, the configuration is adapted to their capabilities and architectures.

`DevConfig` config for locally testing the functionality of the implementation with less epochs and reduced unit sizes

`ClusterConfig` config for training the model on a GPU cluster node with more epochs and larger unit size but a small batch size due to limited memory

`CloudConfig` config for the Google Cloud Platform VM Instance<sup>4</sup> with a higher number of epochs, unit size, and batch size

## 5.4.2 dataset: Dataset Generation

We use the Tensorflow Dataset API<sup>5</sup> to process our samples for training. This API simplifies the process of reading the input files, transform the data for preprocessing, and iterating the dataset. The `dataset` sub-module is responsible for creating our datasets using this API. It is also responsible for splitting the datasets into training, validation, and test sets, creating the token lookup tables and generating the iterators for our datasets.

The `TableGenerator` class, which is contained in the `dataset` sub-module, handles the creation of the lookup tables. For the terminal lookup table and the path token table, we read the provided dictionaries which contain the token values and their frequency in our dataset. Based on this data, we generate dictionaries that map the token values to a unique index using our approach described in Section 4.2. In addition to the terminal token table and the path token table, we also create a lookup for the output labels that map the category names to a numeric label. To do so, we enumerate the input category list and map the category value to the current index. For the lookup tables we use Tensorflow and use `Tensorflow KeyValueTypeTensorInitializer` to initialize the tables with the token values and indices from

---

<sup>4</sup><https://cloud.google.com/>

<sup>5</sup>[https://www.tensorflow.org/api\\_docs/python/tf/data/Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset)

the previously generated dictionaries. The use of these classes for the lookup tables is inspired by Alon et al. [3].

The `dataset` sub-module also manages the retrieval of all input files and their categories based on multiple directory sources. These input files and their categories are combined into a list of pairs and shuffle. After shuffling, the list is split into files for training, validation, and testing. The ratio for the training test set is based on the `config.TRAINING_SPLIT` value. For example, if `config.TRAINING_SPLIT` is 0.7, then 70% of all files are assigned to the training set. The `config.VALIDATION_SPLIT` defines the ratio of validation to test files. For example, if `config.VALIDATION_SPLIT` is 0.5, it means that 50% of the remaining files that are not part of the training set are assigned to the validation set, and the remaining 50% are assigned to the test set. In our example, this would result in a 70%-15%-15% split for training, validation, and testing.

The main task of the `dataset` sub-module is the generation of Tensorflow Datasets. These datasets are created based on the provided list of input files for the training set, the validation set, and the test set. These input files contain all terminal-path triplets for one application. For each of the datasets, we map a function for processing the datasets. In the dataset processing function, we read each of their input files into a *pandas* DataFrame and add padding if necessary. In the next step, we shuffle the input and randomly select `config.MAX_TRIPLET_NUMBER` of path-terminal triplets and split the input into start terminal, end terminal, path tokens, and labels. We use the `TableGenerator` to look up the index for the tokens and the label and return the indices. After mapping this function to the datasets, we repeat the training and validation size to increase the number of samples. Furthermore, we divide the training and validation dataset into batches based on the `batch_size` configuration value. Lastly, we enable prefetching for the datasets, which enables Tensorflow to prepare the following elements of the dataset while the current element is processed. Prefetching the data, reduce the time required for training the model, but also requires more memory.

### 5.4.3 plotting: Plotting Functions

The `plotting` sub-module is responsible for creating plots for various performance measures. To create these plots we make use of the *matplotlib*<sup>6</sup> visualization library. One of the plots visualizes the training and validation losses provided by the Keras fit function as line plots. The same function also provides us with training and validation accuracy, which we visualize as a line plot. Besides plotting training metrics, we also visualize the confusion matrix for the outcome of the test set. Based on the predictions of the model and the true labels, we compute the confusion matrix using the *scikit-learn*<sup>7</sup> library and plot it using *matplotlib*.

### 5.4.4 models: Model Creation & Training

The `models` sub-module contains all functionality to do with our LSTM and CNN model, which includes building and training the models, loading a trained model, and retrieving predictions from a trained model.

Building the models is divided into three parts, the components that are shared between the models, the LSTM specific components, and the CNN specific components. This is already described in detail in Sections 4.3.2 to 4.3.4. To create the LSTM and CNN model, we use various Keras layers and combine them to a Keras Model. Figures 4.9, 4.11, and 4.12 provide an overview of the Keras layers that are created when building the LSTM and the CNN model.

Both models are trained in the same manner. It starts with defining which `config` instance is used. Then, we use the `dataset` sub-module to generates the lookup hash tables, retrieve the input path files,

---

<sup>6</sup><https://matplotlib.org/>

<sup>7</sup><https://scikit-learn.org/stable/>



and generate the training, validation, and test set. We use the input files to compute the class distribution and class weights, as defined in Section 4.3.1. Afterwards, we use the `models` sub-module to build either the LSTM or CNN model which returns a `Keras Model` instance. We configure this model for training with `Tensorflow AdamOptimizer` as optimizer, cross entropy error as loss function and accuracy as optimization metric. We print a summary of the model for logging purposes. Then, we add `Keras EarlyStopping` as backend with validation loss as monitoring metric and `config.EARLY_STOPPING_PATIENCE` as patience. We create a `Keras ModelCheckpoint` that save the weights of the best epoch. Before training, we create a model iterator for the training set and the validation set using the `dataset` sub-module. We train the model using the `model.fit_generator` function with the iterators for the training and validation set, the previously computed class weights, `config.NUM_EPOCHS` as epochs, and early stopping and model checkpoint as callbacks. After training, we plot the loss and accuracy of the training and validation set using the `plotting` sub-module. Then, we create an iterator for the test set, load the weights of the best training epoch, and evaluate the module using the `model.evaluate_generator` as generator. As the last step, we get the prediction for the test set using `model.predict_generator` and plot the confusion matrix given the test labels and the predicted outcome of the model using the `plotting` sub-module.

The `models` sub-module further contains the functionality to load a previously trained model and its environment. This process starts by loading the configuration used for training using the `config` sub-module and loading the lookup tables using the `dataset` sub-module. Next, we build the model, either the LSTM model or the CNN model, and loading the weights using the `models` sub-module. Then we create the dataset and its iterator for the input files using the `dataset` sub-module. Finally, we get the prediction for the input file from the model. This process requires path files as input. That is why we implement a wrapper function that takes APK files as input, calls the `feature` module to generate path files, triggers the previously described process, and returns the predictions.

## 5.5 Dashboard

The dashboard provides an interface for generating the prediction of a provided APK file. This APK file can be uploaded to our server via drag and drop. After the upload has finished, we generate the paths for the provided APK file. We load a previously trained model and retrieve the prediction for the provided app, as previously described. Furthermore, we find additional information about the application from the official Google PlayStore<sup>8</sup>. We use the Python *requests*<sup>9</sup> package to retrieve the PlayStore web page for the provided APK file, which is typically `https://play.google.com/store/apps/details?id=com.example.app` where the `id` is the app package of the APK. We parse the page content of the web page using the *BeautifulSoup*<sup>10</sup> package and retrieve the title, the genre, the number of downloads, and the developer of the app. We present this information in our dashboard using the *Dash*<sup>11</sup> framework and we visualize our predictions using the *Plotly*<sup>12</sup> visualization library.

---

<sup>8</sup><https://play.google.com/store>

<sup>9</sup><https://requests.readthedocs.io/en/master/>

<sup>10</sup><https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

<sup>11</sup><https://plotly.com/dash/>

<sup>12</sup><https://plotly.com/>



# Chapter 6

## Results

In this chapter, we describe the results we have achieved with our approach and implementation. At first, we describe the dataset that we have created. We discuss how we have obtained this dataset and its characteristics. Next, we explain which configurations we have used to achieve our results. Based on these configurations, we described the performance we have achieved for our LSTM and CNN model and which hyper-parameter we have chosen to receive these results. We compare the results of our models and describe which model fits best to which circumstance. Lastly, we describe our dashboard, which provides an interface to our trained model.

### 6.1 Dataset

For our task of predicting Android app categories, it is necessary to retrieve a representative number of APK files for the different categories to achieve good performance for our models. APK files cannot be easily retrieved from the Google PlayStore<sup>1</sup>. That is why, Viennot et al. [28] introduce *PlayDrone*<sup>2</sup>, a crawler for the Google PlayStore to download APK files and metadata. The *Internet Archive*<sup>3</sup> provides a collection of Android apps and metadata which is generated using PlayDrone. From this collection, we have retrieved the APK files for our dataset. In addition to the InternetArchive's collection, we also retrieve APK files from the AndroZoo project [2], which is another collection of Android applications.

From the *Internet Archive* Android app collection we have retrieved 45,960 Android application, where 32,171 apps were used for training, 6,895 for validation, and 6,894 for testing. These numbers correlate to a 70% – 15% – 15% split for training, validation, and testing. The distribution of samples is visualized in Figure 6.1. Here we can see that the largest class is *Entertainment* with 11,489 applications, followed by *Business* with 9,773 apps. Next is *Shopping* with 6,871 samples and *Medical* with 4,883 samples. These classes are followed by *Maps and Navigation* with 4,092 apps and *Weather* with 3,628 apps. The smallest classes are *Comics* with 3,402 applications and *Libraries and Demo* with 2,701 applications.

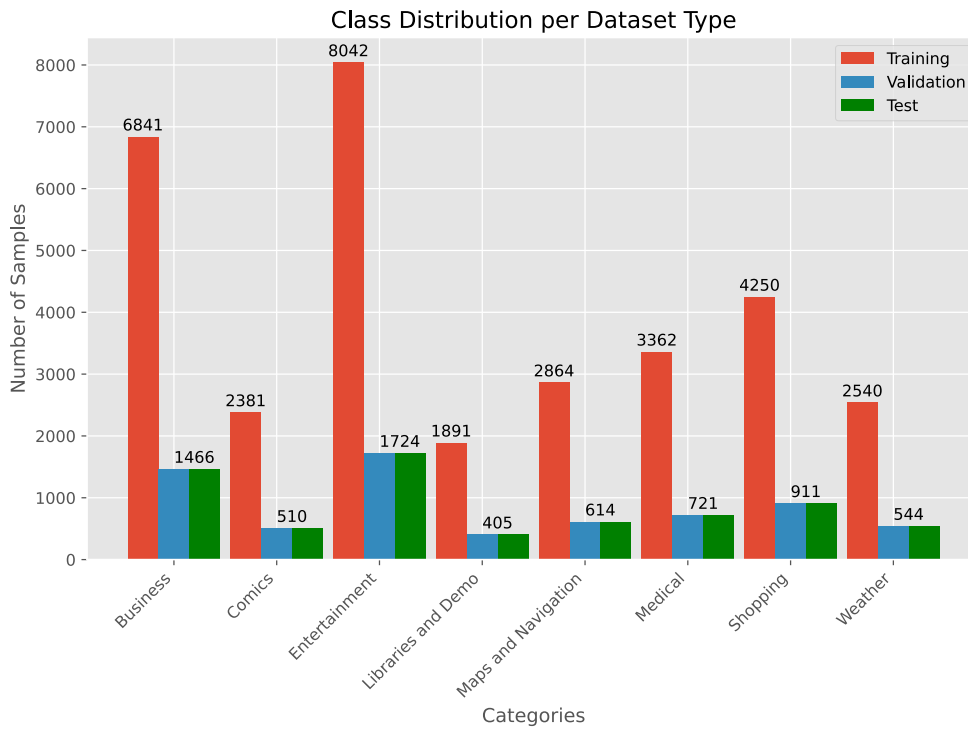
The Google PlayStore has 59 different categories where 17 categories represent games, and 9 categories represent family-friendly apps. The apps in the family-friendly categories contain a sub-set of applications that follow some additional restrictions. As these applications can be found in other categories, we do not use them as classification labels. Furthermore, we have chosen to exclude games from our dataset. Instead, we focus on "traditional" applications. We do not think it would be sensible to mix games and apps, as their code might differ considerably, and our model would mainly learn how to distinguish an Android game from an Android application. We have chosen to focus on 8 categories from the 33 remaining categories.

---

<sup>1</sup><https://play.google.com/store/apps>

<sup>2</sup><http://systems.cs.columbia.edu/projects/playdrone/>

<sup>3</sup>[https://archive.org/details/android\\_apps](https://archive.org/details/android_apps)



**Figure 6.1:** The distribution of our eight representative class for the training dataset, the validation dataset, and the test dataset.

The categories are *Business*, *Comics*, *Entertainment*, *Libraries and Demo*, *Maps and Navigation*, *Medical*, *Shopping*, and *Weather*. We chose these categories because they seem relatively distinct from each other. Furthermore, they also differ in popularity, as we wanted to test our approach with different sample sizes and not only on the most popular categories.

According to the official Android developer guide<sup>4</sup>, the *Business* category contains applications for document editing and reading, package tracking, accessing remote desktops, email management, and job searching. Examples for *Business* apps are "Microsoft Teams"<sup>5</sup> and "LinkedIn"<sup>6</sup>. Applications in the *Comics* category, on the other hand, are typically comic players or comic titles such as "WEBTOON"<sup>7</sup>. *Entertainment* apps, like "Twitch"<sup>8</sup>, are intended for streaming videos, movies, TV, or interactive entertainment. The *Library and Demo* category is for software libraries and technical demos, such as "Barcode Scanner"<sup>9</sup>. The *Maps and Navigation* category contains apps for navigation, GPS tracking, mapping, transit, and public transportation. An example for a *Maps and Navigation* app would be the "ÖBB Scotty"<sup>10</sup> app. *Medical* apps, like the "Stopp Corona"<sup>11</sup>, are apps for drug and clinical references, calculators, handbooks for health-care providers, medical journals and news. Applications in the *Shopping* category are intended for online shopping, auctions, coupons, price comparison, grocery lists, and product

<sup>4</sup><https://support.google.com/googleplay/android-developer/answer/113475>

<sup>5</sup><https://play.google.com/store/apps/details?id=com.microsoft.teams>

<sup>6</sup><https://play.google.com/store/apps/details?id=com.linkedin.android>

<sup>7</sup><https://play.google.com/store/apps/details?id=com.naver.linewebtoon>

<sup>8</sup><https://play.google.com/store/apps/details?id=tv.twitch.android.app>

<sup>9</sup><https://play.google.com/store/apps/details?id=com.manateeworks.barcodescanners>

<sup>10</sup><https://play.google.com/store/apps/details?id=de.hafas.android.oebb>

<sup>11</sup><https://play.google.com/store/apps/details?id=at.rotekreuz.stopcorona>

reviews. Examples for *Shopping* apps include "Amazon Shopping"<sup>12</sup> and "Lidl"<sup>13</sup>. The last category that we look at, is the *Weather* category which intended for weather reports, such as "Weather forecast"<sup>14</sup>.

Note that developers choose an app's category by themselves, which means that there is a certain chance that an application was assigned to an inappropriate category. For example, we found an Android game called "IGI Jungle Commando 3D Shooter" in the *Comics* category. Therefore, there might be a pre-existing error in the dataset. This error limits the possible performance of our model. However, our trained model can assist in finding such misclassified applications.

In addition to our model dataset, we have retrieved the path and terminal tokens for Android apps from all app categories, excluding games. The number of path tokens is confined to the grammatical elements of the Smali language. That is why we only have 113 values. This limitation also restricts the number of valid paths, which is 1,893. On the other hand, terminal values can be chosen more freely, as they also include variable values. This absence of restrictions leads to a much higher number of possible choices. That is why we restrict the token by having to occur at least 20 times in all applications. This limitation reduces the number of possible values from 3,789,993 to 1,707,822.

Table 6.1 describes the class weights that we compute for each category. These values weigh the loss function such that classes with fewer training instances are given more weight to make up for the class imbalance. Therefore, they are inverse proportionate to the size of the class. Hence, the smallest class, i.e., *Libraries and Demo*, has the largest weight, and the largest class, *Entertainment*, has the smallest weight.

Category	Class Weight
Business	4.7027
Comics	13.5115
Entertainment	4.0004
Libraries and Demo	17.0127
Maps and Navigation	11.2329
Medical	9.5690
Shopping	7.5696
Weather	12.6657

**Table 6.1:** Class weights for each of the categories.

## 6.2 Configuration

As described in Section 5.4.1, we use three different environments for training our models. These environments differ largely in processing power and memory size. To make the best use of them, we have created a different training configuration for each of them. The values for the different configurations are listed in Table 6.2.

The first configuration is `DevConfig`. It is mainly for locally testing the functionality of the training code. That is why we only use a subset of 1,000 apps for training and train only for 3 epochs. Furthermore, we reduce the shuffle size to 10, the embedding size to 10, and the dense layer size to 10. It is also unnecessary to store the model for this configuration as it is not used for the final evaluation.

<sup>12</sup><https://play.google.com/store/apps/details?id=com.amazon.mShop.android.shopping>

<sup>13</sup><https://play.google.com/store/apps/details?id=de.sec.mobile>

<sup>14</sup><https://play.google.com/store/apps/details?id=com.chanel.weather.forecast.accu>

The `ClusterConfig` is targeted for an environment that is more powerful than the `DevConfig`. It can provide an initial estimation of the performance of a model. That is why all samples are used for training. However, training with all applications significantly increases the required memory consumption. Therefore, the batch size is reduced to 1. The batch size does not influence the performance of a model, only the time it takes to train it. We increase the number of epochs to 15, the embedding size to 50, and the dense layer size to 50 so that we receive a more powerful model.

The `CloudConfig` is the configuration for an environment that is geared towards training deep learning models. It is used for the final training of the model. It trains for 30 epochs unless early stopping is enforced which happens after 5 epochs of no improvement in the validation loss. For better training efficiency, it used a batch size of 200. It also trains the most powerful model out of the three configurations, with an embedding size of 128, and a dense layer size of 128.

Parameter	DevConfig	ClusterConfig	CloudConfig
APPS_PER_CATEGORY	1000	All	All
NUM_EPOCHS	3	15	30
BATCH_SIZE	3	1	200
MAX_TRIPLET_NUMBER	1000	1000	1000
EMBEDDING_SIZE	10	50	128
DENSE_LAYER_SIZE	10	50	128
TOKEN_MIN_THRESHOLD	20	20	20
EARLY_STOPPING_PATIENCE	3	3	5
DROPOUT_RATE	0.2	0.2	0.2
MAX_PATH_LEN	15	15	15
SAVE_MODEL	False	True	True
VAL_TEST_SPLIT	0.5	0.5	0.5
TRAIN_DATASET_SIZE	0.7	0.7	0.7
SHUFFLE_SIZE	10	10	600

**Table 6.2:** Configuration values for the different configuration setting.

### 6.3 Models

In this section, we describe our final training configuration for our models. Then, we define the layers and learnable parameters that are shared between our models. Afterward, we describe the layers, weights, and performance of each of our models. We continue by comparing the results which we were able to achieve for our models. Lastly, we give an overview of our dashboard and provide some examples.

For the final training of both our models, we used the `CloudConfig`. From Table 6.2, we can see that this means that we trained for a maximum of 30 epochs, had a batch size of 50, and considered 1000 context triplets per application. Furthermore, we had an embedding size and dense layer size of 128, a shuffle size of 600, and a minimum token occurrence of 20. We also had a dropout rate of 0.2, early stopping patience of 5, a maximum path length of 15, and a shuffle size of 600.

The dimensions of all layers that occur in both models for the final training run are described in Table 6.3. The colors correspond to our layer color coding from Figure 4.9. The first dimension represents the `BATCH_SIZE` config. The None value represents that this dimension is variable. So one can choose a different batch size for training, validation, and testing. The second dimension of layers 1-3 and 5-7 represents the `MAX_TRIPLET_NUMBER` config, which is 1,000 in our case. This value represents the

maximum number of terminal-path triplets considered for each application, as described in Section 5.4.1. The third dimension of the input layer and the embedding layer is `PATH_LEN`, i.e., the maximum path length. So the path input layer consists of `BATCH_SIZE` apps, where each app is represented by 1,000 paths of length 15. The path embedding layer then looks up the embedding for each path value, which is 128. We reshape the (15, 128) path embedding matrix to  $(15 * 128) = (1,920)$ . The outcome shape of the path encoding layer varies for the LSTM model and the CNN model.

The input layers for the start and end terminal both consist of 1,000 terminal per application. The terminal embedding layer is applied to both terminal layers as they belong to the same dictionaries. Each terminal value is mapped to a 128-dimensional vector. A global average pooling layer is applied to both embedded terminals. This computes the average embedding values of the 1,000 terminal values for each embedding dimension per application, leading to an output shape of  $(BATCH\_SIZE, 1,000)$ . The outcome of the path encoding layer and the two pooling layers is concatenating, reuniting the triplet. This concatenation leads to an output shape of  $(128 * 2 + PATH\_UNITS)$  where `PATH_UNITS` represents the output shape of the final path encoding layer. The encoded terminal-path token is condensed to a 128-dimensional vector for each sample using a dense layer. During training, a dropout layer is applied to prevent overfitting. This layer randomly sets input units to 0, which means it does not affect the shape, i.e.,  $(BATCH\_SIZE, 128)$ , of the outcome. A final dense layer is applied to the dropout outcome, which maps the input to a probability distribution for our 8 classes.

Nr.	Layer	Type	Output Shape	Param #
1	input_path	InputLayer	(None, 1000, 15)	0
2	path_embedding	Embedding	(None, 1000, 15, 128)	14,720
3	reshape	Reshape	(None, 1000, 1920)	0
4	path_encoding		(None, PATH_UNITS)	
5	input_terminal_start	InputLayer	(None, 1000)	0
6	input_terminal_end	InputLayer	(None, 1000)	0
7	terminal_embedding	Embedding	(None, 1000, 128)	266,518,144
8	start_token_pooling	GlobalAvg1DPooling	(None, 128)	0
9	end_token_pooling	GlobalAvg1DPooling	(None, 128)	0
10	concatenate	Concatenate	(None, 256+PATH_UNITS)	0
11	combined_dense	Dense	(None, 128)	128*(257+ PATH_UNITS)
12	combined_dropout	Dropout	(None, 128)	0
13	combined_out	Dense	(None, 8)	1,032
<b>LSTM</b>				
4A	path_lstm	LSTM	(None, 200)	1,696,800
<b>CNN</b>				
4Ba	path_conv1	Conv1D	(None, 990, 100)	2,112,100
4Bb	path_conv2	Conv1D	(None, 980, 100)	110,100
4Bc	path_pooling1	MaxPooling1D	(None, 326, 100)	0
4Bd	path_conv3	Conv1D	(None, 316, 160)	176,160
4Be	path_conv4	Conv1D	(None, 306, 160)	281,760
4Bf	path_pooling2	GlobalAverage1DPooling	(None, 160)	0

**Table 6.3:** Dimensions and number of parameters for all layers that are shared between our models.

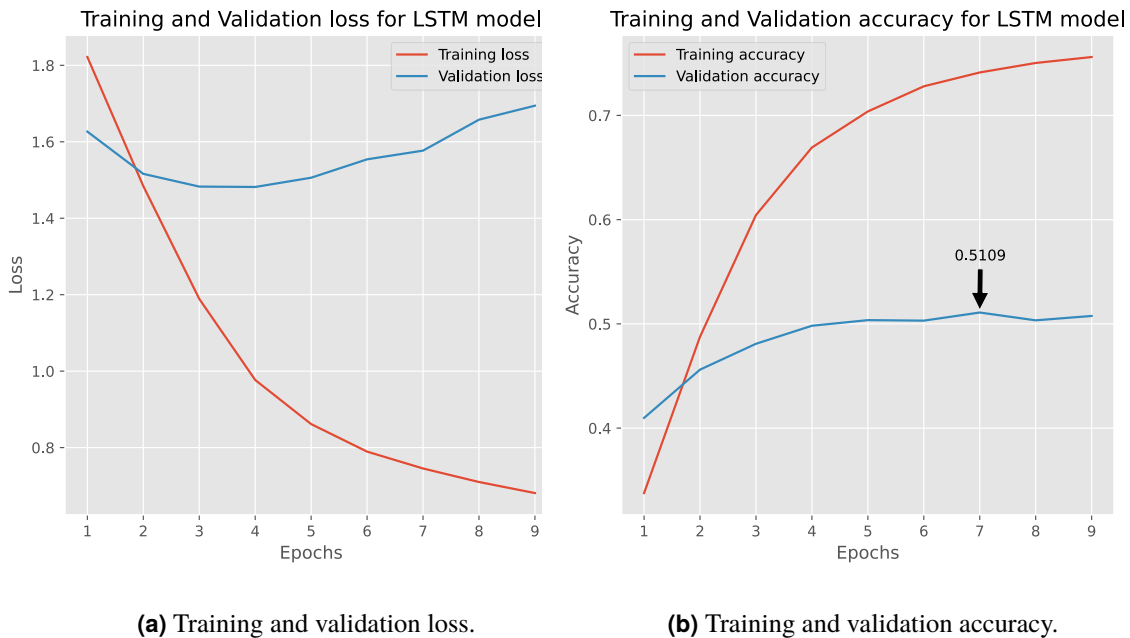
Both models share 4 layers trained, the path embedding layer, the terminal embedding layer, the combined dense layer, and the output dense layer. We have 113 unique path elements and two default values, the padding and unknown token. Each element is mapped to a vector of length 128, which leads

to  $(113 + 2) * 128 = 14,720$  learnable parameters. For the terminal embedding, we have a much higher number of trainable parameters, because of the high number of 1,707,822 unique values. With the addition of the two default values, we have  $(1,707,822 + 2) * 128 = 266,518,144$  learnable parameters. the combined dense layer with 58,496 learnable parameters, and the combined output layer with 1,032 learnable parameters. For the combined dense layer, we have  $256 + \text{PATH\_UNITS}$  as input, 128 units, and 1 additional unit for the bias, which leads to  $128 * (256 + \text{PATH\_UNITS} + 1)$  learnable parameters. The output layer has 8 units for the 8 classes, a bias, and has 128 as input dimension from the dropout layer. Therefore, the number of learnable parameters is  $(128 + 1) * 9 = 1,032$ .

### 6.3.1 LSTM model

In the following paragraphs, we describe the dimensions and weights of the layers that are unique to the LSTM model. Additionally, we provide an overview of the performance of the LSTM model, given our chosen hyper-parameters. We evaluate the performance in terms of loss and accuracy for each of our datasets. Furthermore, we describe the confusion matrices we have received based on our test dataset.

For the path encoding, we use one LSTM layer with 200 units. The LSTM layer is described in Table 6.3 as layer 4A. The outcome of this layer is the output value for each unit, which leads to an output shape of  $(\text{BATCH\_SIZE}, 200)$ . As described in Section 2.3.4, an LSTM cell has 4 gates. Each of these gates has 2 weight matrices,  $U \in \mathbb{R}^{n \times m}$  and  $W \in \mathbb{R}^{n \times n}$ , and a bias vector  $b \in \mathbb{R}^n$ , where  $n$  is the input size and  $m$  is the output size. This means that the number of learnable parameters is  $4(nm + n^2 + n)$ . The input to the LSTM layer are 1,000 sequences of length 1,920, so  $m = 1920$ . Our LSTM layers consists of 200 units, therefore  $n = 200$ . So in our case, the number of learnable parameters is  $4(nm + n^2 + n) = 4(200 \cdot 1920 + 200^2 + 200) = 1,696,800$ .



**Figure 6.2:** Training and validation metrics for the LSTM model.

Figure 6.2 shows the loss and accuracy of the training and validation set of our LSTM model. Figure 6.2a shows the loss for the training set and the validation set. While the training loss is steadily decreasing, the



validation loss quickly reaches a local minimum. Figure 6.2b, on the other hand, plots the accuracy of the training set and the validation set. Similar to the loss, the accuracy of the training set is steadily increasing, while the accuracy of the validation set reaches a plateau at around 0.5. The reason behind this is that the model starts to overfit on the training set. That is why we use a validation set to avoid this issue. It can be seen that the best epoch in terms of validation accuracy is epoch 7 with a validation accuracy of 0.5109, a training accuracy of 0.7412, a validation loss of 1.5769, and a training loss of 0.7454. Using the weights from epoch 7, we achieve a loss of 1.6019 and an accuracy of 0.5113.

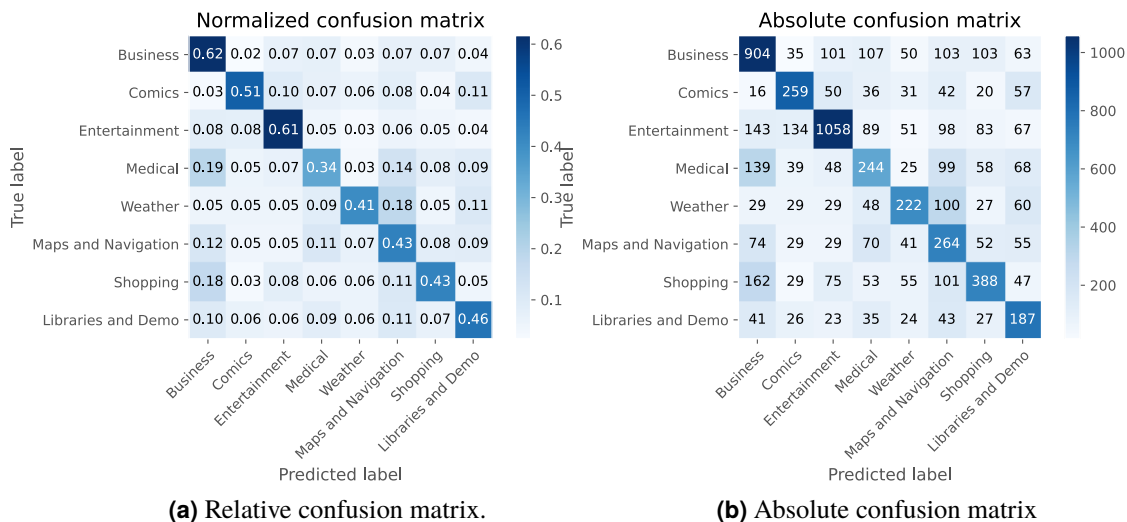


Figure 6.3: Confusion matrices of test set for the LSTM model.

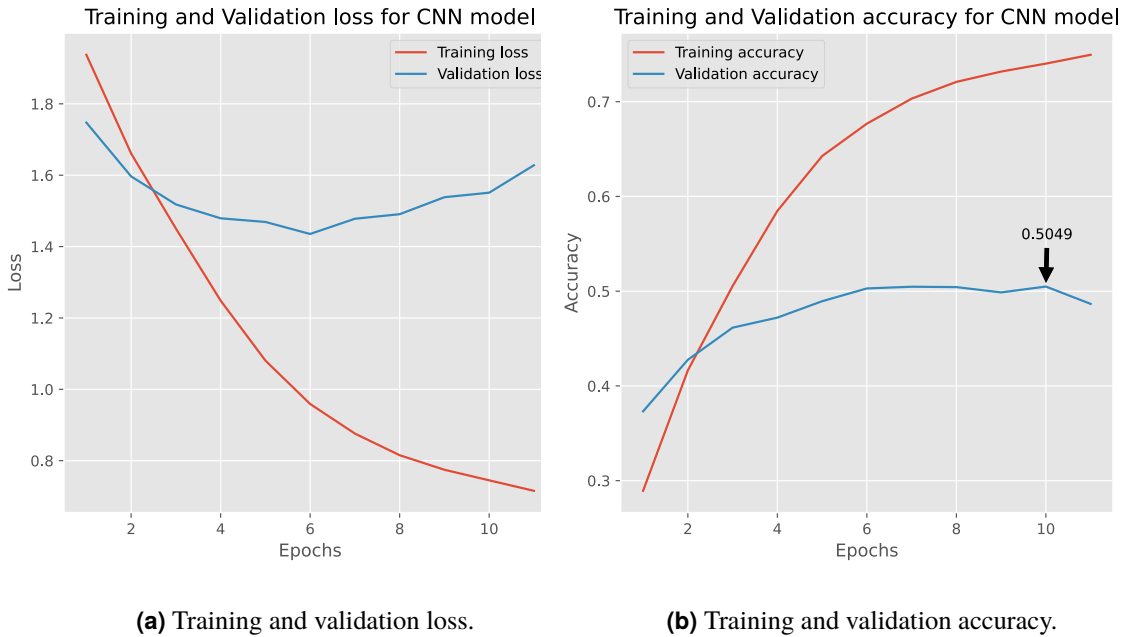
In Figure 6.3, two confusion matrices are shown. Figure 6.3a shows the relative prediction distribution per class. Figure 6.3b shows the absolute confusion matrix, where each value represents the absolute number of test samples that were classified correctly or incorrectly. The *Business* category is assigned correctly the most with 62% or 904 samples. Next is the *Entertainment* category with 61% or 1,058 correctly predicted samples. Note that, for both classes, the predictions for other classes are relatively small, which means that there is no single category with which they are typically misclassified. The *Medical* category, on the other hand, is only predicted correctly in 34% of the cases. They are incorrectly predicted as *Business* in 19% or 139 test cases. However, *Business* is only misclassified as *Medical* in 7% of the cases. So, the two classes are not necessarily similar. It could just be the cases that the *Medical* class is not very unique, and the model assigns *Medical* applications to a larger class.

### 6.3.2 CNN model

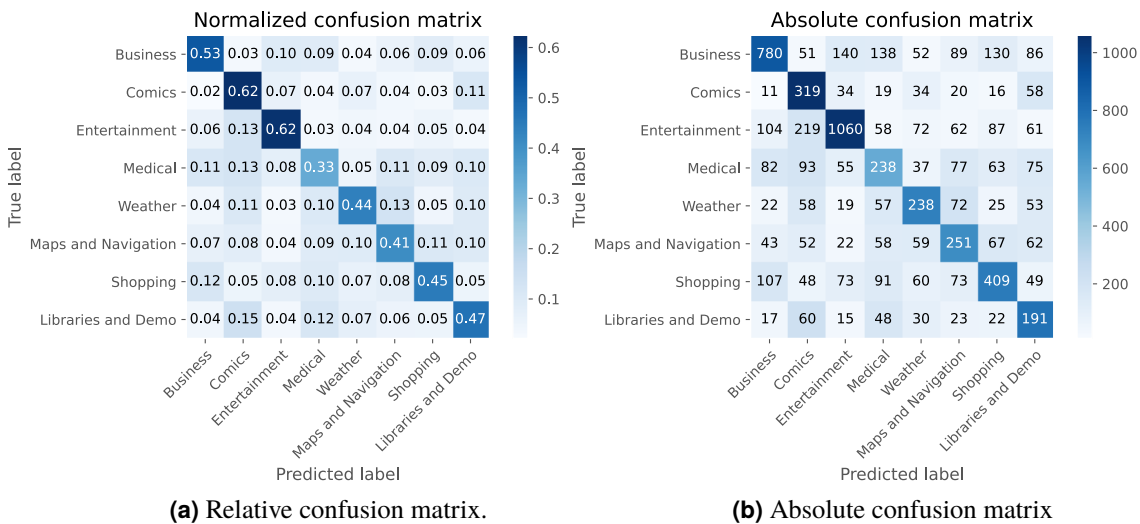
Similar to the previous section, we provide an overview of the layers and weights that are only part of this section’s CNN model. We continue by describing the performance of our CNN model, in terms of loss and accuracy. Lastly, we discuss the confusion matrices for the test dataset.

For the CNN model, we have two one-dimensional convolutional layers with 100 filters and a kernel size of 10. By applying this kernel to the path elements, we receive an output shape of (990, 100). Each filter has the shape (11, 1920), where 11 represents the window size and 1920 the path length. Therefore, including the bias, each filter consists of  $11 * 1920 + 1 = 21,121$  learnable parameters and  $21,121 * 100 = 2,112,100$  in total. After applying another convolutional layer with the same parameters, we receive an output shape of (980, 100). This layer consists of  $(100 * 11 + 1) * 100 = 110,100$  learnable parameters. We then apply a

maximum pooling layer with a pooling window of 3, which leaves us with an output shape of (326, 100). Next, two convolutional layers are applied with 160 filters and a kernel of 10, which changes the output to (306, 160). The first convolutional layer contains  $(100 * 11 + 1) * 160 = 176,160$  weights and the second layer contains  $(160 * 11 + 1) * 160 = 281,760$  weights. Lastly, the average for each filter is calculated, leaving us with 160 values.



**Figure 6.4:** Training and validation metrics for the CNN model.



**Figure 6.5:** Confusion matrices of the test for the CNN model.

Figure 6.4 shows the training and validation metrics for the CNN model. In Figure 6.4a, we can see the training and validation loss. While the training loss is steadily decreasing, the validation loss reaches its

local minimum reasonably quickly. The same holds for the accuracy, as can be seen in Figure 6.4b. To avoid overfitting on the training set, we make use of early stopping. We can see that epoch 6 provides the best validation loss with a value of 1.4353. The training accuracy for this epoch is 0.6768, the training loss is 0.9589, and the validation accuracy is 0.5029. Using the weights from this epoch, we achieve an accuracy of 0.5045 and a loss of 1.5537 for the test set.

Figure 6.5 shows two confusion matrices, one confusion matrix with relative values and one with absolute values. For the CNN model, the classes with the highest accuracy are the *Comics* class with 62% or 319 correctly classified samples and the *Entertainment* class with 62% or 1,060 correctly classified samples. The *Comics* class is predicted incorrectly instead of the *Entertainment* class in 13% of the cases, i.e., for 219 samples. However, the opposite case does not occur as often with only 7% or 34 samples. Similar to the LSTM model, the *Medical* category has the lowest accuracy, with 33% of the cases and 238 samples. However, in contrast to the LSTM model, it is incorrectly predicted as *Business* for only 6% or 204 test cases. Instead, *Medical* apps are assigned to the *Entertainment* class in 13% of the cases.

### 6.3.3 Evaluation

Table 6.4 provides an overview of the distribution of our test and the accuracies our models achieve on the test set. The first column shows the number of samples for each category of our test set. In summary, our test set consists of 6,895 samples. The second column provides a distribution of the classes in the test set. We can see that the largest class is *Entertainment* with 25% of the samples. *Libraries and Demo*, on the other hand, is the smallest calls with 5.84% of the test set. The next column provides the per-class accuracy of the LSTM model. It shows that given an application, what is the probability that our model will assign it to this class. The class with the highest probability is the *Business* class with 61.66% accuracy, followed by *Entertainment* with 61.40%. These classes are also the largest. The LSTM model seems to generalize better with a larger variety of samples. The per-class accuracy of the CNN model, on the other hand, shows that the class with the highest probability is not the *Business* class, but the *Comics* class with 62.43%. The *Comics* class makes up only 7.4% of the test set. The CNN model might be able to generalize better with a smaller sample size than the LSTM model. For both models, the class with the lowest per-class accuracy is the *Medical* class with 33.89% accuracy with the LSTM model and 33.06% accuracy with the CNN model. From Figure 6.3 and Figure 6.5, we can see that in both models the *Comics* class is incorrectly assigned to the *Libraries and Demo* class in 11% of the cases. This common misclassification indicates that the class might not contain as many distinct features as the other classes.

	Distribution		LSTM		CNN	
	absolute	relative	class	total	class	total
Business	1,466	0.2126	<b>0.6166</b>	0.1311	0.5321	0.1131
Comics	510	0.0740	0.5068	0.0375	<b>0.6243</b>	0.0462
Entertainment	1,724	0.2500	0.6140	0.1535	0.6152	0.1538
Libraries and Demo	405	0.0587	0.4606	0.0271	0.4704	0.0276
Maps and Navigation	614	0.0891	0.4300	0.0383	0.4088	0.0364
Medical	721	0.1046	0.3389	0.0354	0.3306	0.0346
Shopping	911	0.1321	0.4264	0.0563	0.4495	0.0594
Weather	544	0.0789	0.4081	0.0322	0.4375	0.0345
	6,895	0.1250	<b>0.4752</b>	<b>0.5113</b>	<b>0.4835</b>	<b>0.5045</b>

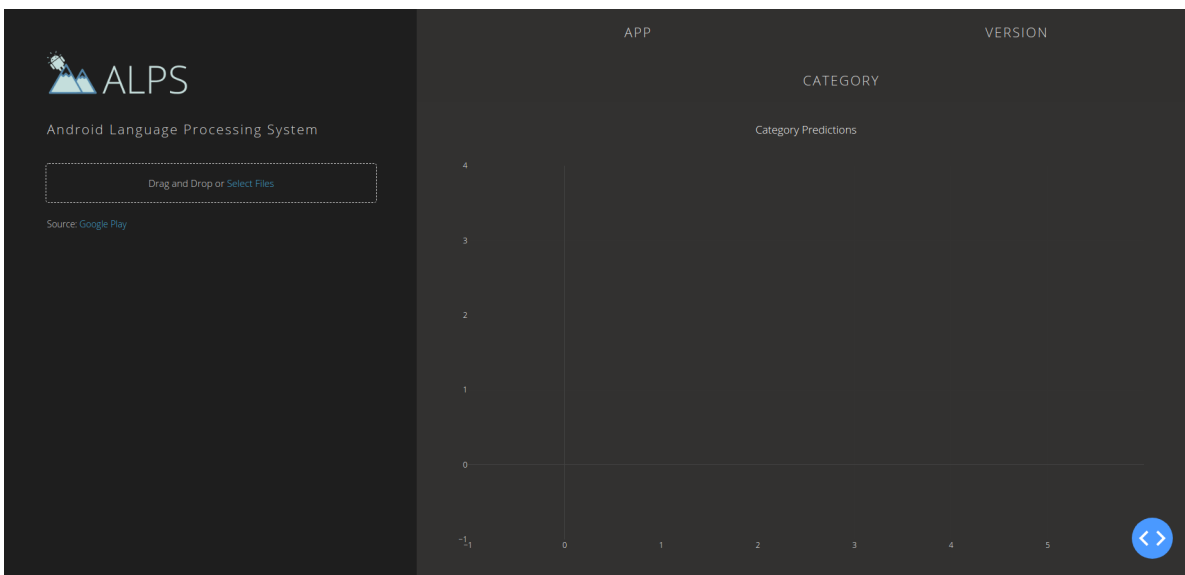
**Table 6.4:** Comparison of the test accuracies of the LSTM model and the CNN model.

If we chose a class randomly, we would have an average accuracy of 12.5%. Since our classes are not distributed equally, we could also choose the largest class, i.e., *Entertainment*. This approach would

give us an accuracy of 25%, since the *Entertainment* makes up 25% of our dataset. On average, our LSTM model has a per-class accuracy of 47.52%, and our CNN model has a per-class accuracy of 48.35%. However, this assumes an equal class distribution. If we include the class distribution, the LSTM has an overall accuracy of 51.13%, and the CNN model has an accuracy of 50.45%. In any case, both our models perform better than random guessing. For the per-class accuracy, the CNN model performs better than the LSTM model. However, for the overall accuracy, the LSTM slightly outperforms the CNN model. The reason behind this is that the LSTM model provides the best accuracy for the most common classes. Therefore, the question of which model is better depends on the problem. Is it more important than a high overall accuracy is achieved, then the LSTM model would be the better candidate. However, if high per-class accuracy is favored, then the CNN model preferable.

## 6.4 Dashboard

To show our approach's usability, we have created an interface so that the user can quickly receive predictions for an application of their choice. The user can upload an APK file to the server using drag and drop or using a file browser. The tool then generates the paths from the given APK file, loads the latest CNN model, process the paths to get a prediction for the outcome, and visualizes the probability distribution in an interactive bar chart. Additionally, it retrieves meta-information of the provided application from its Google Play<sup>15</sup> store entry, if it exists, and visualizes them in a data table. This information includes the developer's name, title, category, and the number of downloads.



**Figure 6.6:** Visualization of the correct prediction of an Android application.

As described in Section 5.5, we use the *Dash*<sup>16</sup> framework to create our dashboard. This framework is open source and on the most popular libraries for data visualization and user interfaces. Dash apps run on a Flask<sup>17</sup> server, which means it can easily be accessed by others without any additional setup. For the layout and styling, we use a pre-existing CSS file<sup>18</sup> in addition to the style sheets provided by the Dash

<sup>15</sup><https://play.google.com/store>

<sup>16</sup><https://plotly.com/dash/>

<sup>17</sup><https://flask.palletsprojects.com/en/1.1.x/>

<sup>18</sup><https://codepen.io/chriddyp/pen/bWLwgP.css>

framework. For the color palette of the bar chart, we use the Material UI color palette<sup>19</sup>, as they provide a good contrast to the dark background.

Figure 6.6 shows the dashboard without a prediction of an Android app. On the top, we display the package name and version of the provided APK file. We extract this information from the Manifest XML file. Using the package name, we retrieve the Google Play store entry, if it exists. It might not always be available, as the application might not be available in the PlayStore, or it could have been removed from it. If the entry exists, we retrieve its assigned category for comparison and display it below the title and version. As previously described, we also retrieve the developer's name, title, category, and the number of downloads. We display this information on the left-hand side as a data table.

We can see the dashboard with a correct prediction in Figure 6.7. The model has been applied to the application `com.kbzk.android.weather`. From the data table, we can see that this application's title is "KBZK STORM Tracker Weather App" and was developed by "The E.W. Scripps Company". It was downloaded over 1,000 times, and is a *Weather* app. On the right-hand side, we can see the bar chart of the prediction outcome. The bar chart shows us that the model assigned this application to the *Weather* category without any doubt.

However, the model is not always correct in its predictions, as shown in Figure 6.8. In this examples, we use the application `com.senior_safety_phone` for our model. From the data table, we can see that this application's title is "Senior Safety Phone - Big Icons Launcher" and was developed by "Deskshare, Inc". It was downloaded over 10,000 times and was assigned to the *Medical* category. However, the model provides only a 23.38% probability part of the *Medical* category. Instead, it assigns it to the *Entertainment* category with 64.27%. However, given the possible error in our dataset, we cannot know whether the *Medical* is the best category for the application. Maybe, the *Entertainment* category contains more applications that are similar to `com.senior_safety_phone`.

## 6.5 Discussion

The goal of our approach was to provide an automated method to classify Android applications based on their source code. We wanted to determine whether the source code is even distinct enough for classification and showcase its potential. We have developed and implemented a Deep Learning-based approach to test this theory. Furthermore, we have created a dataset of 45,960 Android applications to train and evaluate our approach.

As described in Section 6.3.3, both of our models perform better than the baseline of random choice, which means that they were able to deduce relevant information for classification. Therefore, we can say that Android source code is varied enough between the different categories to assign applications into their specified classes in 47.52% – 51.14% of the cases, depending on the model and accuracy measure. We verified our models' performance on previously unseen test data, ensuring that these results represent our model's ability to generalize.

It is hard to provide a single reason for the upper-performance limit of our models. Possible causes include that the dataset's size is too small, the source code variance is too small, or classes are too similar or varied. This upper limit could also represent the pre-existing error within our dataset because developers choose their application's category themselves.

In summary, we conclude that it is possible to classify Android applications based on their source code. Our approach is able to showcase its potential. However, there are still possibilities to improve this approach.

---

<sup>19</sup><http://materialuicolors.co/>

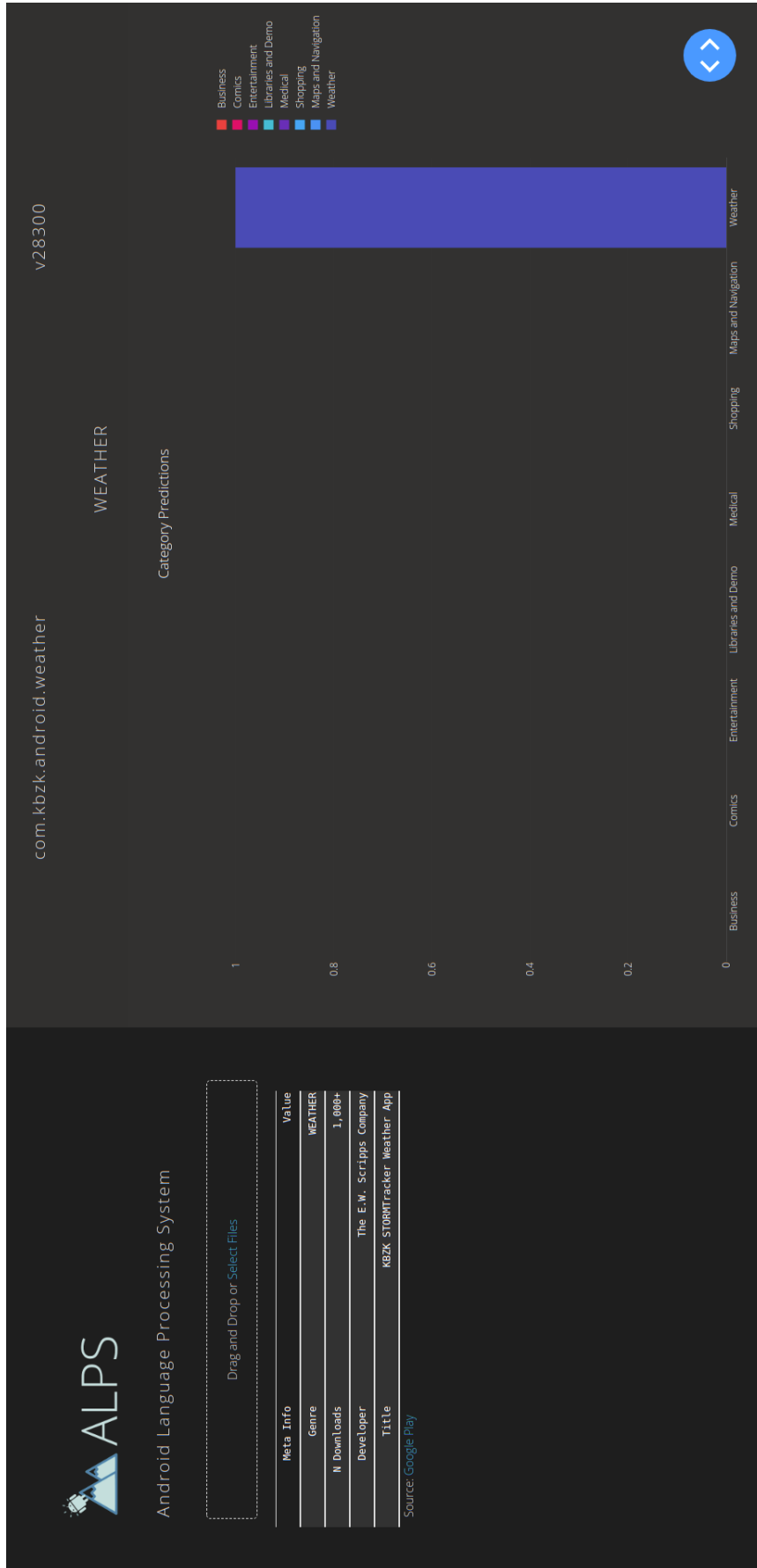


Figure 6.7: Dashboard a correct prediction for an Android application.

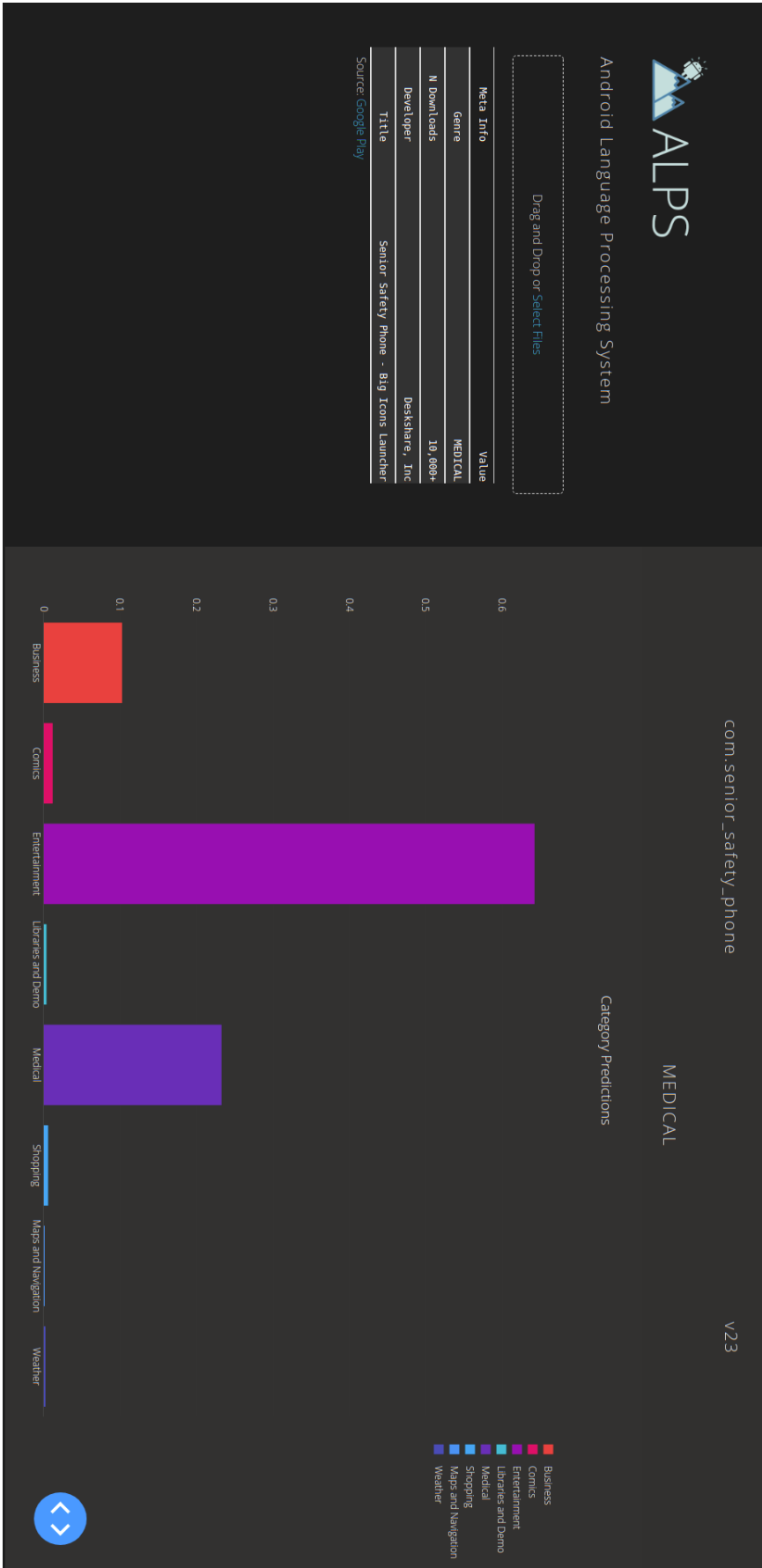


Figure 6.8: Visualization of the incorrect prediction of an Android application.





## Chapter 7

# Conclusion

Google Play plays a pivotal role in the popularity of Android applications. However, with the increasing number of applications, it has become challenging for users to find applications and, consequently, for developers to reach their intended user base. To help with this problem, Google has introduced 33 app categories, representing the different types of applications available on Google Play. Which category an app belongs to is currently chosen by the app's developers. However, it is not always trivial to select the most relevant category and can lead to misclassification.

In this thesis, we presented our novel classification approach - Android Language Processing System (ALPS), which can automatically classify Android applications into Google Play categories. Unlike most related work, which focus on permissions and API calls, our approach generates features from the app's source code. We do so by combining natural language processing techniques with source code processing and applying it to Android app classification.

As a first step, our approach filtered an app's Dalvik byte code for activity classes by leveraging information from its Manifest XML file. We retrieved all activity classes from the Dalvik byte code and disassembled them using the Smali framework. We extended the framework with the generation of AST paths introduced by Alon et al. [3]. Furthermore, we retrieved all unique path and terminal tokens and their frequency within the dataset. Using this information, we generated our terminal-path dataset and trained two machine learning models, an LSTM model, and a CNN model. In our first model, we combined word embeddings with a long-short term memory approach to process the path sequences. On the other hand, in the second model, we used a convolutional layer to process the sequence information. We trained both models on a specific training dataset, applied early stopping using a validation dataset, and performed evaluation on a separate test dataset. Based on the word embeddings, our models predicted an app's Google Play category and provided feedback on the most likely class. We evaluated our approach in terms of accuracy and loss. Furthermore, we provided visualization of the results in plotting the training and validation accuracy and loss. Additionally, we generated and visualized the confusion matrix.

The resulting framework, ALPS, can be used for multiple purposes. It can retrieve an app's Manifest XML file, parse it, and return its activity classes. Using the Smali framework, it can disassemble an app's activity classes and generate abstract syntax trees. ALPS can create AST paths, terminal-path triplets, and lookup dictionaries that contain all unique terminal or path elements and their frequencies. The primary purpose, however, is the classification of Android applications. ALPS can train models with two different architecture and evaluate them using a test dataset. The outcome can be evaluated using various metrics and plots that are provided by our framework. Additionally, ALPS includes a dashboard where a user can upload an Android APK file, which retrieves our CNN model's prediction and presents the outcome in an interactive chart.

We compiled a dataset containing 45,960 Android applications from 8 representative Google Play categories to evaluate our approach. After applying our path generation module to each of the applications,

we used generated features to train our LSTM and CNN models. Based on our test set, the LSTM model achieved an overall accuracy of 51.13%, which slightly outperformed the CNN model with an overall accuracy of 50.45%. On the other hand, the CNN model provided better results for the per-class accuracy with 48.35%, compared to the LSTM model with 47.52%. Therefore, which model provided better performance depends on which accuracy is favored, overall, or per-class.

The following list describes ideas for future work and possible improvements:

- **Create a larger dataset.** Our dataset contains 45,960 applications from 8 different categories. While it is a significant dataset size, the split into a training set, validation set, and test set reduces the number of training samples by 30%, impacting the model's performance. Moreover, the classes are not equally distributed. For example, while the *Business* class has 9,773 samples, the *Libraries and Demo* consists of only 2,701 samples. This invariance could lead to lower per-class performance in classes with fewer samples. By increasing the dataset size, we might increase the overall accuracy and the per-class of our model.
- **Train model with all Google Play categories.** For our evaluation, we chose 8 categories to verify that our approach leads to meaningful results. The next step is to create a dataset for all 33 categories and train our model with this dataset to provide predictions for all classes. The outcome can also be used to evaluate the quality of the current status of Google Play.
- **Train model on malware dataset.** In this thesis, we evaluated our approach by classifying Android applications into their Google Play categories. However, our model could also be trained to detect malware applications by replacing our dataset with a dataset of known malware applications. It could either predict whether an application is malicious or not by using a combination of known benign and malicious apps or predict the malware family of a malicious application.
- **Clean dataset using clustering methods.** Currently, in our dataset, the Google Play category assignment is considered as ground truth. However, these assignments are performed manually by an app's developers. Therefore, there might be incorrect assignments that could lead to lower performance. If the number of incorrectly assigned is large enough, it could also lead to a model that does not fully represent Google Play's category definition. Removing applications that are outliers to their category could solve both problems. Manual outlier detection would be very time consuming due to the dataset and feature size. Alternatively, clustering methods could be used to detect outliers, which could then be removed from the dataset.
- **Use tree-based convolution to process AST.** In our approach, we generate terminal-path triplets from a class' AST. While these triplets retain some structural information from the AST, we do not capture it fully. To leverage the entire AST, we could utilize tree-based convolutional neural networks, as introduced by Mou et al. [21], to process the abstract syntax tree. Using this approach would empower our model to grasp an app's source code even better. Furthermore, it would remove the overhead of the path generation.
- **Include dynamic analysis.** At the moment, we focus solely on the source code of an Android application. Including dynamic analysis could provide additional, useful information about an application and further improve our model's performance. While retrieving execution traces manually from each app in the dataset is a cumbersome task, Android app sandboxes be utilized to automate this task.

In this thesis, we presented a novel approach for Android app classification, and with ALPS, we provide a modular implementation of this approach. For evaluation, we created a dataset of 45,960 Android applications and trained two different models. Based on our approach and data set, we were able to verify that terminal-path triplets provide meaningful features for Android app classification.

# Bibliography

- [1] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. *A Convolutional Attention Network for Extreme Summarization of Source Code*. Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. Volume 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pages 2091–2100 (cited on page 20).
- [2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. *AndroZoo: collecting millions of Android apps for the research community*. Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016. Edited by Miryung Kim, Romain Robbes, and Christian Bird. ACM, 2016, pages 468–471. doi:10.1145/2901739.2903508. <https://doi.org/10.1145/2901739.2903508> (cited on pages 2, 49).
- [3] Uri Alon, Shaked Brody, Omer Levy, and Eran 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 pages 2–3, 21, 23–25, 27–31, 46, 63).
- [4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. *code2vec: learning distributed representations of code*. Proc. ACM Program. Lang. 3 (2019), 40:1–40:29 (cited on pages 3, 21).
- [5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad 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 pages 19, 22).
- [6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. 2015 (cited on page 24).
- [7] Satanjeev Banerjee and Alon Lavie. *METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments*. Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005. Association for Computational Linguistics, 2005, pages 65–72 (cited on page 20).
- [8] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. *Neural Code Comprehension: A Learnable Representation of Code Semantics*. Neural Information Processing Systems – NIPS 2018. 2018, pages 3589–3601 (cited on page 21).
- [9] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard E. Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. *When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries*. Network and Distributed System Security Symposium – NDSS 2018. The Internet Society, 2018 (cited on page 21).
- [10] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 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 1724–1734 (cited on page 24).
- [11] Edwin Dauber, Aylin Caliskan, Richard E. Harang, and Rachel Greenstadt. *Git blame who?: stylistic authorship attribution of small, incomplete source code fragments*. Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. ACM, 2018, pages 356–357. ISBN 978-1-4503-5663-3 (cited on page 21).
- [12] Sepp Hochreiter and Jürgen Schmidhuber. *Long Short-Term Memory*. *Neural Comput.* 9 (1997), pages 1735–1780 (cited on page 13).
- [13] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. *Summarizing Source Code using a Neural Attention Model*. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics, 2016. ISBN 978-1-945626-00-5 (cited on pages 20, 22).
- [14] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. *Android Malware Detection using Deep Learning on API Method Sequences*. CoRR abs/1712.08996 (2017) (cited on page 19).
- [15] Yoon 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 (cited on page 16).
- [16] Patrik Lantz. *DroidBox: Android Application Sandbox*. 2011. <https://www.honeynet.org/projects/active/droidbox/> (cited on page 19).
- [17] Quoc V. Le and Tomas Mikolov. *Distributed Representations of Sentences and Documents*. Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014. Volume 32. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pages 1188–1196. <http://proceedings.mlr.press/v32/le14.html> (cited on page 2).
- [18] Thang Luong, Hieu Pham, and Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation*. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015. The Association for Computational Linguistics, 2015, pages 1412–1421 (cited on page 24).
- [19] Niall McLaughlin, Jesús Martínez del Rincón, BooJoong Kang, Suleiman Y. Yerima, Paul C. Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickle, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. *Deep Android Malware Detection*. Conference on Data and Application Security and Privacy – CODASPY 2017. ACM, 2017, pages 301–308. ISBN 978-1-4503-4523-1 (cited on page 20).
- [20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. *Distributed Representations of Words and Phrases and their Compositionality*. Neural Information Processing Systems – NIPS 2013. 2013, pages 3111–3119 (cited on pages 2, 12, 16, 21).
- [21] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. *Convolutional Neural Networks over Tree Structures for Programming Language Processing*. Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA. AAAI Press, 2016, pages 1287–1293. ISBN 978-1-57735-760-5 (cited on pages 20–21, 64).
- [22] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. *Bleu: a Method for Automatic Evaluation of Machine Translation*. Proceedings of the 40th Annual Meeting of the Association for

- Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA. ACL, 2002, pages 311–318 (cited on page 20).
- [23] *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.
- [24] *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*. The Association for Computational Linguistics. ISBN 978-1-941643-32-7.
- [25] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. Neural Information Processing Systems – NIPS 2014. 2014, pages 3104–3112 (cited on page 24).
- [26] Duyu Tang, Bing Qin, and Ting Liu. *Document Modeling with Gated Recurrent Neural Network for Sentiment Classification*. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015. The Association for Computational Linguistics, 2015, pages 1422–1432 (cited on page 31).
- [27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention is All you Need*. Neural Information Processing Systems – NIPS 2017. 2017, pages 5998–6008 (cited on page 24).
- [28] Nicolas Viennot, Edward Garcia, and Jason 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 pages 2, 49).
- [29] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. *Comparative Study of CNN and RNN for Natural Language Processing*. CoRR abs/1702.01923 (2017) (cited on page 31).
- [30] Yuan Zhenlong, Lu Yongqiang, and Xue Yibo. *Droiddetector: Android Malware Characterization and Detection using Deep Learning*. Tsinghua Science and Technology 21.1 (2016), pages 114–123 (cited on page 19).