POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Mobile Testing Framework Exploiting Machine Learning and NLP

Supervisors

Prof. Luca ARDITO

Prof. Maurizio MORISIO

Dr. Riccardo COPPOLA

Candidate

AYDA TANIK

JULY 2021

Summary

The aim of this thesis is to create a new automated android testing framework supported by machine learning and NLP. The android automated framework provides a user to write simple and adaptive test scripts. The thesis work is divided into three main phases. The first part includes application classification using different feature vectors by supporting machine learning and NLP and in this way proves that the same type of applications uses similar features and code characteristics. In the second part, activity classification was applied and inferred that the same type of activities can have the same UI elements and can follow a similar UI design pattern. Based on these two parts it has been developed an automated android testing framework that users can write test suites with human-understandable and simple syntax. Each test suite can have more than one test case and each one can apply to the specified activity types. Test cases can re-use similar activity types without needing any change. Also, test suites can re-use in apps that have similar application types. To summarize, The android testing framework can reduce the time needed for writing UI testing with simple commands and the scripts can be easily understandable for all developers.

Acknowledgements

Firstly, I would like to deeply express my gratitude to my supervisor, Prof. Luca Ardito, Prof. Maurizio Moriso and Dr. Riccardo Coppola, and also I would like to thanks to Simone Leonardi for giving me valuable ideas and suggestions during my thesis work.

I am so grateful to my family for supporting me and encouraging me during my studies. Special thanks to my mother Ayça, my father Hakan, my sister Beliz and also my uncle Akın.

Finally, I would like to thanks to all my friends and specially thanks to Gözde, Deniz, Simay, Sevgim and Berfin who always believe in me and supporting me in the difficulty moments.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
: Artificial Intelligence

NLP
: Natural Language Processing

DEX
: Dalvik Executable File

UI
: User Interface

XML
: Extensible Markup Language

Chapter 1

Introduction

This chapter introduces the thesis main goals and structure, providing implementation details.

1.1 Thesis Aim and Goals

Android UI testing is an essential task for testing user interaction with the user interface of the android application while developing an android application. Since users interact with the user interface for executing the desired tasks when using an android app, the role of the UI testing is ensuring every function and UI element are working as expected. UI testing guarantees the usability, accessibility, and consistency of apps. However, UI testing can be a costly and time-consuming job with repetitive tasks for developers. The automated android framework can be implemented for solving these issues. When a developer implements an android application, generally the developer follows some design and functional patterns. Based on this information and some features, applications can be categorized and also similar activity types can be combined together according to some structural behaviors. The main goal is the thesis to create an Android Testing Framework that is providing a user to write simple and adaptive test scripts and can be re-used in apps that have similar characteristics code level without changing test scripts structure. The Android Testing Framework ensures the reuse of existing test scripts for the same type of application and activity category. Thanks to machine learning and NLP, Android application categories and activity were classified by supporting machine learning algorithms and natural language processing. To sum up, The framework can execute UI testing and can be reused at a similar code level of apps by writing simple human-readable test scripts.

1.2 Thesis Organization

The tasks can be divided into three main sections. We can see the APK classification steps explained in detail in Figure 1.1 and also Figure 1.2 shows Activity classification steps.

- **APK Classification**

  Chapter 2 covers data preparation steps for APK classification. It describes collecting a set of useful features and an APK data set that will be used for the classification procedure. It also covers feature extraction with the NLP step that is generating different feature vectors using collected features.

  Chapter 3 describes APK classification steps. There are preliminary operations that are pre-processing for classification and it describes classification steps in detail. At the end of the chapter, there is a comparison of results which compare to results of different feature vectors.

- **Activity Classification**

  Chapter 4 presents data preparation steps for activity classification. It mentions activity structure and also describes activity types. It includes the creation of an activity data set. The other aspect of this chapter is creating feature vectors which will be used in chapter 5 for classification.

  Chapter 5 describes activity classification steps. Similar to the APK classification, There are preliminary operations that are pre-processing for classification and explain classification steps. This chapter ends with a comparison of the results of different feature vectors.

- **Android Testing Framework**

  Chapter 6 covers Android Testing Framework. It describes the structure of the framework and how our framework is working. Commands of the framework are explaining support with examples.

  Chapter 7 covers the evaluation of the Testing framework. After the detailed explained framework in the previous chapter, This chapter shows some evaluation procedure that applied to the framework.

  Chapter 8 presents the conclusion of our work and future works.

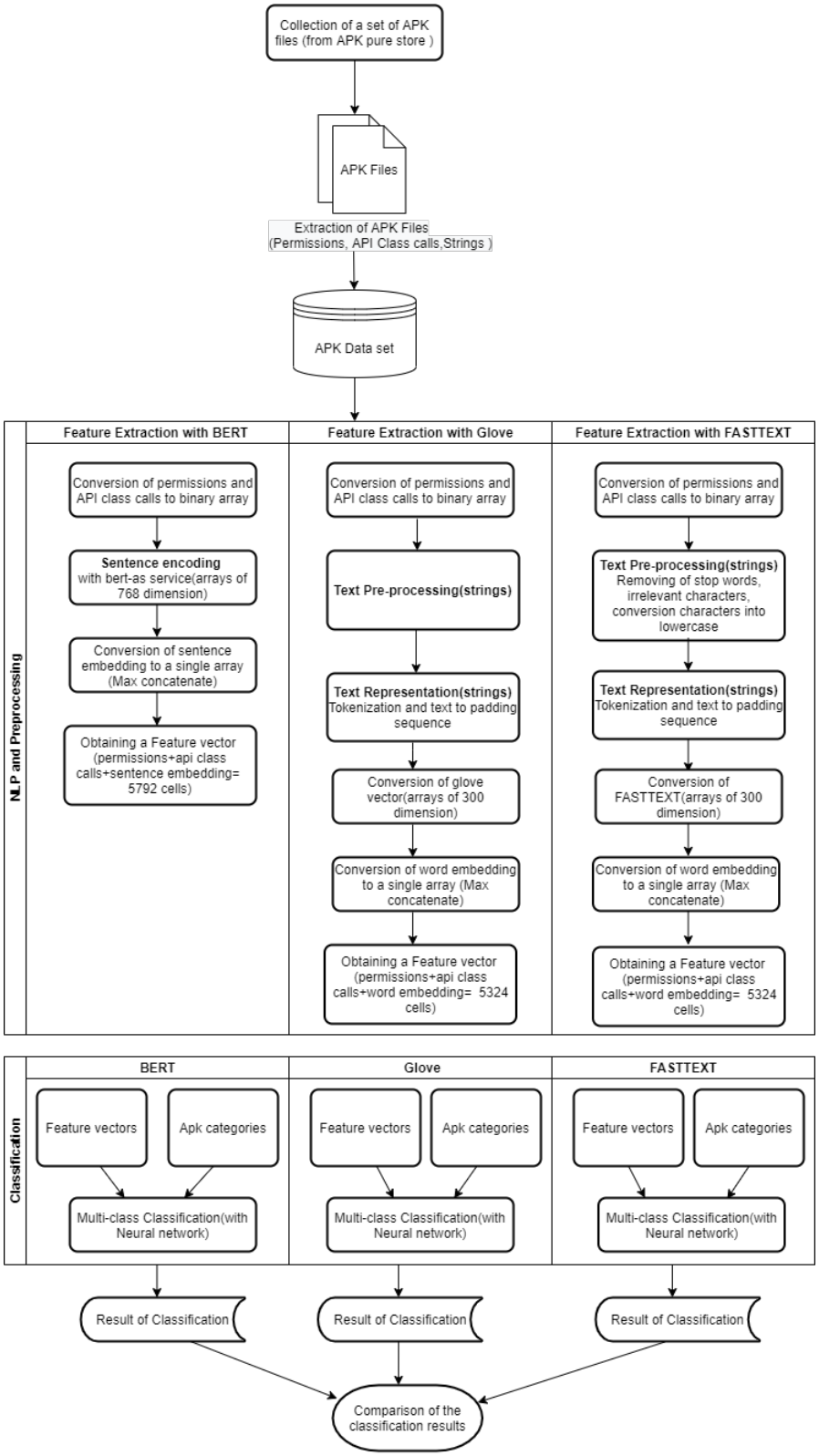

Figure 1.1: APK classification steps by using three different feature vectors

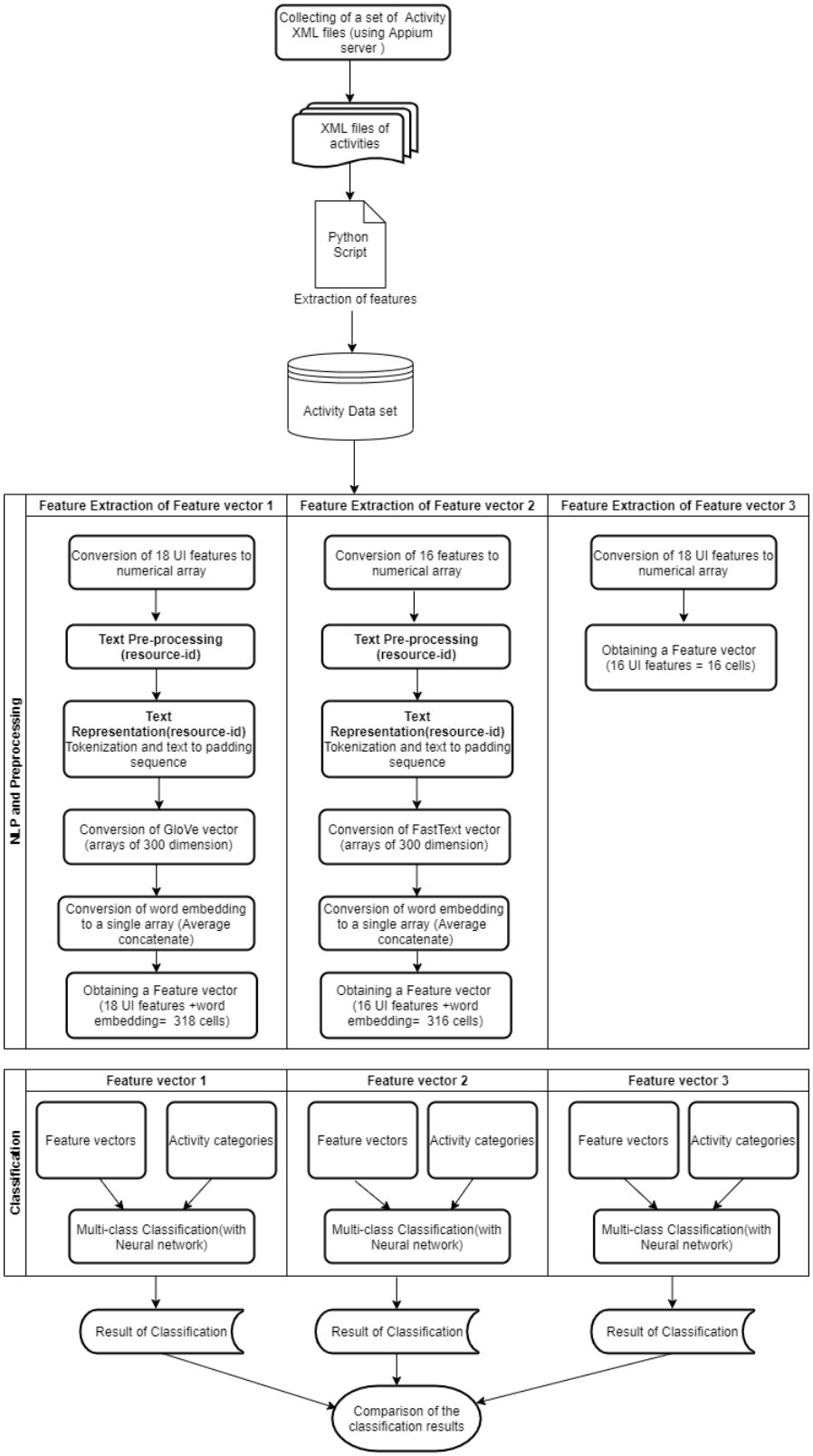

Figure 1.2: Activity classification steps by using three different feature vectors

1.3 Implementation details

Most of the work on machine learning and NLP was developed using the ***Python*** language and with the support of ***Scikit-Learn*** and **Keras** libraries. Three pre-processed models were used for generation of feature vectors in the NLP step. Bert-Base, Multilingual Cased for BERT feature vector, GloVe Common Crawl 300 dimensional model for Glove, and lastly Fasttext 300 dimensional pre-trained model for Fasttext vector was used.

On the android testing part, The automated testing tool was implemented using the ***Python*** language and ***ANTLR*** (Another Tool for Language Recognition) tool. The work was done with *Android 10* running and testing on a real device which is *Huawei nuova 5T.*

Chapter 2

Data Preparation for APK Classification

This chapter includes an explanation of APK file structure in detail. Starting from that point it covers how to collect features and create a data set. After the creation of the data set, we generate three different feature vectors using NLP techniques which will be used in the next chapter.

2.1 Overview

Nowadays, Mobile devices have a significant role in our lives. According to statistics in January 2021, android is the worldwide mobile operating system leader with a 71.93% share [1]. Google Play stores include a large number of apps for android users. Users are able to download 3.04 million apps [2]. In order to manage these huge numbers of apps, apps are divided into categories. Separating apps into categories provides huge convenience for Android users and developers. Categories help the users to discover, make comparisons and find a suitable app for performing desirable tasks. As an example, if the user wants to download photo editing tools, it can be discovered in the Photography category. Users can search easily for the specific type of app if they don't know the exact name of the application. On the other hand, developers can standardize some functionalities, UI designs in their applications like the same categorical apps. It can be said that the same categorical apps can have the same functionality behaviours. Our aspect is getting the useful app's features that can be used in classification without accessing the source code of the app(Black box). If the same categorical apps have similar functionality, the APK file is also affected by the same functionalities. To sum up, our work can begin with an examination of the APK file.

2.2 APK File Structure

The android package kit(APK) file is a package file format that includes all application information, source code to install and distribute an application on the android system. To release and share android applications, developers have to create an apk file. After the implementation application, an APK file can be created in Android Studio. When the user of the android system wants to download the android application from the google play store, basically the android system downloads the APK file in the background. There are a lot of websites to download APK files directly. ApkPure [3] and APKMirror [4] are examples of these websites. For obtaining features of APK, we need to decompile the APK file. **ApkTool** [5] was used for decompiling the APK file.

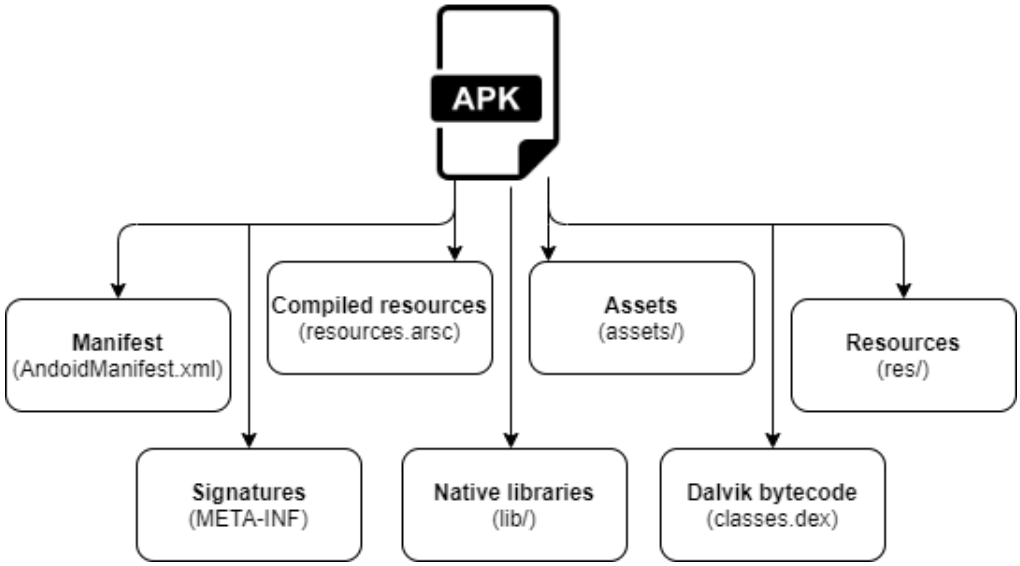

Figure 2.1: APK File Structure

1. **Resources(res) :** The folder contains app's resources(strings, values, drawable, layout folders) in XML format.
2. **Native Libraries(lib) :** The folder is optional and contains platform dependent compiled code. There are folders for different CPU architectures.
3. **Assests(assets) :** The folder contains application information and assets. Images, music, backgrounds, forecast colours can be replaced inside of this folder.

4. **Manifest(AndroidManifest.xml) :** It is a mandatory XML file which is each application must include this file.
   Manifest file includes basic information about the app such as package name app components, permissions, device compatibility, file conventions.

5. **Signatures(META-INF) :** This folder contains CERT.RSA and CERT.SF files that include information about the digital signature of the app and also contain MANIFEST.MF file. It stores metadata information about the app.

6. **Dalvik bytecode(classes.dex) :** The file is a Dalvik Executable file containing source code and all java libraries that app uses.

7. **Compiled Resources(resources.arsc) :** It contains compiled resources of the app.

2.2.1 Android Manifest File

After the de-complication of the APK file, our first aim is to examine the app manifest file[6] which is one of the essential files in the APK file. It is a mandatory file and must be present in every apk file. If the AndroidManifest.xml file does not set up correctly, it can lead to different problems. When the developer initiates a project in the android studio platform, the AndroidManifest.xml file is automatically created by the platform. The file includes the app's package name, hardware and software features of the app, permissions, components of the app which are activities, services, broadcast receivers, and content providers. As we need to mention about manifest file structure in detail, there is **<manifest>** element which is the root of the XML file. Manifest element includes some attributes that describe general information about the app. For instance, the package attribute describes the package name of the app. Since manifest file is a root element, there are other elements that are the sub element of the manifest. The sub-elements of the manifest are in the following list.

- **Application:** One of the sub elements of manifest element is <application>. Each manifest file can have only one application element. Activity, Intent-filter, Action and Category can present in the application element as a sub element of application.

- **Uses-features:** It is provided to specify hardware and software requirements for the app. When an app requires specific hardware or software components, it can add uses-features for requesting to use specific components. As an example if an application needs to support multi-touch then the following line should be added to the manifest file.
  <uses-features android:name=”android.hardware.touchscreen.multitouch”/>

- **Uses-sdk:** For specifying which platform the app will be available, it is needed to use <uses-sdk> and indicate minSdkVersion of the app.

- **Permissions:** Permissions support to protect the user's privacy. Every android application must request permission for accessing and using the significant device features. The requisitions can be made by using and adding an <uses-permission> element to the manifest file. For instance, the application needs to reach the camera the following line can be added.
  <uses-permission android:name="android.permission.CAMERA"/>
  There are many types of permission that developers can use when they are implementing their apps. If applications need to reach calendar features, READ_CALENDAR and WRITE_CALENDAR permissions can be used.

In AndroClass[7], Authors have used permissions as a feature of classification. They have extracted permissions from the manifest file to use their classification, and we also examined the permissions on different applications. As the need to give an example, Applications in the maps navigation apps category, generally have ACCESS_COARSE _LOCATION and ACCESS_FINE_LOCATION. In addition to this example, Applications in the music and audio category, the Manifest file can contain RECORD_AUDIO and MODIFY_AUDIO_SETTINGS. It is observed that developers who develop the same type of application can use the same type of permissions in their applications and these can help to categorize apps. To sum up, permissions can have an impact on the classification step and decide to utilize permissions from the manifest file.

2.2.2 Dalvik Executable format

In order to run apps on the device, the android device uses a different file format than Java-byte code which is a Dalvik byte-code. For obtaining the Dalvik byte-code several steps can apply. To begin with, the source of the program can be converted into the .class files after that the class files and jar libraries have to be translated to the dex file which includes Dalvik byte-codes. Finally, For executing the app, the Dex file can be translated to the machine-understandable code by Android Runtime(ART). The Dalvik Executable File (.dex) contains the source code of android applications that must be included in each apk file and provide the application to run on the device. However, the .dex file size is limited and the maximum size is 64k. For this reason apk files can have a multiple dex file if methods greater than 65,536. Dex classes are divided into several parts. The figure 2.2 shows parts of the dex file. The *header* contains general information about the file, *strings_ids* include a string identifier for every string that is used. *Type_ids* contains identifiers for all parts, for example classes and arrays. *Proto_ids* contains all method prototypes. *Method_ids* contains all specific user methods and all API.

Class_defs contains all classes that used in apps. When a developer implements apps, either uses specified methods or importing Apps in the same category can use the same API classes. For this, it needs to apply reverse engineering to the dex file and methods can be extracted from the dex file. The second feature that needs to be extracted is API class.

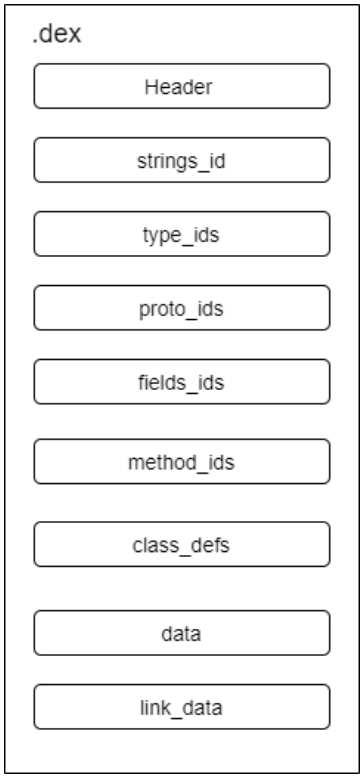

Figure 2.2: Dex File Structure

2.2.3 String XML File

Another important file that we need to examine deeper is the strings XML file that is procuring text data from the app. A string's XML file must be in every apk file and resides in project/res/values. The **<resources>** element is the root of the XML file and the **<string>** element is the sub-element of the resources element which contains textual information. Every <string> element has a name and this name associates as a resource-id. The following example shows an example of string XML file syntax and structure. When the developers need to update the content of some strings, they may simply modify strings from XML files without needing to reach the source code. Since it is providing text data of the app, it includes semantic information about the app.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="name_of_string">String_text</string>
</resources>
```

2.3 Creation of APK Data set

2.3.1 Collecting Set of APK Files

Since our approach is having application information with black-box techniques, apk files can be found simply by reaching the websites that include android apps. As mentioned before, There are many websites for downloading apk files. In our work, It has been selected on the apk Pure website. We have created a simple crawler using the Python programming language for collecting apk files automatically. In apk Pure store mainly there are two sections which are apps and games. In our case, we'll only consider the apps section because generally downloading a game requires more time than apps since their size is larger. In the apps section, categories are divided into 32 types. Since downloading apps is a time-consuming task, we have to specify some restrictions. It has been defined as the upper boundary download size which is 100 MB and in the meantime, some categories are discarded from our experiment. We have collected **15 category types** with app names, the category that apps belong to and an apk file of the app. To summarize, it was collected **3958 apk files** from apk Pure store in total. Figure 2.1 shows the number of apps per category.

2.3.2 Obtaining Set of Features

Using the apk tool, A python script has been developed that takes and decompiling every application that we collected. We can apply **'"apktool d -m "+apk_file_name'** command to every apk file. If the apk file correctly decompiled, new folder will be created for this apk. Apk file can be deleted and it can be started to work with decompiled folder.

- Extraction of the apk files
- Deletion of the old apk files

After obtaining decompiled apk files, we can extract useful features by simply writing with another python script. A second python script was written for obtaining the features from decompiled apk folders.

Category	Number of Apps
Free,Business APP	303
Free,Maps and Navigation APP	287
Free,Food and Drink APP	282
Free,House and Home APP	277
Free,Communication APP	275
Free,Art and Design APP	274
Free,Health and Fitness APP	273
Free,Medical APP	269
Free,Shopping APP	259
Free,Photography APP	257
Free,Entertainment APP	253
Free,Music and Audio APP	246
Free,Social APP	242
Free,Books and Reference APP	231
Free,Education APP	225

Table 2.1: Number of APK files per category in APK Data set

The string XML file can be parsed for getting the value of the string element and also the manifest XML file can be parsed to get the app's permissions by using *xml.etree.ElementTree* module. When we are parsing the strings and manifest file, the first step is reaching the root element of the XML file. After the reaching of the root element, we can iterate sub elements which is **uses-permission-sdk** and **uses-permission-sdk-23** for manifest XML file and **string** for the string XML file. It can apply reverse engineering for taking the API class calls.

- Get the permissions from manifest xml file
- Get the strings from string.xml file
- Extract the API class calls

To summarize, Permissions, API class calls, and strings have been extracted with the python script that is implemented and features are ready for the conversion of machine-understandable format.

2.4 Feature Extraction and NLP

This part covers the generation of three different feature vectors with NLP methods. The feature vectors include useful app features and these are permissions retrieved from the manifest file, API class calls, and strings obtained from strings XML file.

2.4.1 Background

One of the essential problems in machine learning algorithms, it cannot be provided raw text directly to algorithms as an input. In order to apply machine learning algorithms, it is needed to convert textual files into a numerical feature vector. This problem can be solved by using Feature extraction with NLP techniques. Feature extraction techniques in NLP have an important role that deals with transforming textual data to a numerical feature vector that will be used as features for the classification step. ***Bag of words*** is one of the simple methods for feature extraction that counts the words appearing in the sentence. In BoW the first step is taking the input text breaking this text into words. This step is also called tokenization. After the tokenization step, the next step is building a vocabulary. Each sentence that can be called a document can transform into the vectorized format according to counting how many times words are appearing. Each vectorized format length is equal to vocabulary size length. The main drawback of this model is that it can not gain semantic information. Another drawback needs to be considered, when the number of documents increases, at the same time the vocabulary size will increase so this makes a more complex model.There are many feature extraction methods that exist as mentioned before like Bag of words. However, these methods do not take into consideration semantic meanings with the other words. Word embeddings give a solution to this problem.

2.4.2 Word Embeddings

Word embeddings are feature extraction methods for understanding both syntactic and semantic meanings of the words. It is basically transforming the text into n-dimensional space. Each word has a vectorial representation. In vector space, words that have a similar meaning reside closer. Instead of creating word embeddings from scratch, it will use pre-trained word embeddings. Pre-trained word embeddings have created and trained vectors. Some methods of word embeddings are ***Word2Vec*** , ***GloVe*** , ***Fasttext*** , ***Flair embeddings***.

2.4.3 Creating GloVe feature vector

GloVe vector is one of the word embedding methods that use unsupervised learning methods [8]. For using unsupervised methods and learning word representation they take into consideration **word co-occurrence statistics** in a corpus. Pennington gave an example of aspects of **how to extract the meaning of words from the co-occurrence probabilities**. Let's take two words which are k = ice and m = steam. The relationship between these words specifies finding the ratio between co-occurrence probabilities by using several words and let's specify as and n is equal to solid. n has a relationship between ice but has no relationship with steam. According to that ratio, Pkn/Pmn will be large. Also when k is equal to gas, this time Word gas relates with stem but not with ice. To sum up, GloVe vectors are using ratios of co-occurrence probabilities instead of the probability of themselves.

As we mentioned before, permissions can have an impact on the classification and have been extracted using python script. Permissions have to be converted to the machine-understandable format. For converting the binary feature array, we need to obtain all possible permissions. Permissions were taken from android developer official websites[9] and in total there are 165 permissions so the length of permissions binary feature vector is 165. If the permission exists in the current application, it assigns 1 for this permission otherwise assign 0. Similar to permissions, API class calls can be converted into binary feature arrays. API class call feature array includes 4859 class calls which were taken from android developer official websites[10]. If the application uses the API class call, this API class call assigns to the 1, otherwise 0.

When we are converting strings into GloVe vectors, the first task to apply is text pre-processing and representation. Before the conversion of the strings to the Glove feature vector, we have to apply text pre-processing and representation in order to have a more accurate result in the classification section. Text pre-processing and representation is one of the significant and initial tasks of an NLP pipeline for having more useful and meaningful inputs. It can be applied to different text pre-processing according to the data set that has. [2]In the role of text pre-processing paper, the authors mention the importance of applying to the right text-pre-processing task according to the data set that has it, and authors have used lower-cased, lemmatized, and multi-word. There are many text pre-processing NLP techniques, for instance, tokenization, Lemmatization, Stemming, and removal of stop words. To begin with, lemmatization and stemming helps to convert inflected words to the root forms. Inflected words are words which are derived from another word. The main difference between lemmatization and stemming is that lemmatization uses morphological analysis of the words and it converts inflected words to the base form which is called a lemma.

For example, the word "better" can convert to "good" or the word "kept" converts to "keep". On the other hand, stemming converts to the root word by using prefixes and suffixes. As an example, words “playing”, “played”, “plays” can be reduced to “play”. Figure 2.4 shows text prepossessing and representation tasks that we have done.

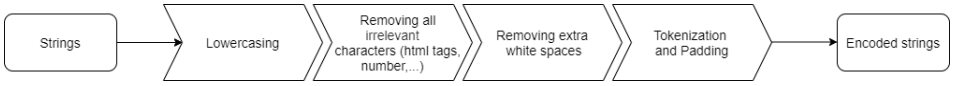

Figure 2.3: Text pre-processing and representation

Our first step is lower-casing. We are avoiding two different dimensions for the same word in vectorized format by converting upper case characters to lowercase characters so We will obtain one vector per word. The next step is removing all irrelevant characters from the strings list by using regular expressions. It has been cleaned with special characters, HTML tags, numbers, and also emojis. After removing special characters, it can be removed extra white spaces. Two examples present that show a pre-processing procedure in the bellow list.

- **’3.Minimum withdrawal amount is 10 points’ -> ’minimum withdrawal amount is points”**
- **’Please tap OK to set up your email first!’ -> ’please tap ok to set up your email first’**

Tokenization is splitting a text into smaller chunks. Firstly, each sentence can be divided into words which are called tokens. After tokenization, sentence words have been encoded into the sequences which map each word to the index of the corresponding word. The examples of conversion of sequences are in the following list.

- **’allow access to camera and device storage to start scanning documents’ -> [62, 163, 1, 37, 104, 88, 60, 1, 26, 170, 93]**
- **’start scanning docs’ -> [26, 170, 319]**

Since all input’s length must be equal for applying the next steps, another approach that must be applied is padding. We needed to find the maximum sequence length for padding. According to the maximum sequence length, it can be added 0 to the end of the sequence in each sequence until it becomes equal to the maximum sequence length. The padding examples of sequences are in the following list.

- **[62, 163, 1, 37, 104, 88, 60, 1, 26, 170, 93] -> [62 163 1 37 104 88 60 1 26 170 93 0 0 0 0 0 0]**
- **[26, 170, 319] -> [26 170 319 0 0 0 0 0 0 0 0 0 0 0 0 0 0]**

As mentioned before, it was used pre-trained word embeddings model and for GloVe vector pre-trained word embeddings 300 dimensional embeddings was downloading from official GloVe websites [1]. Each word is converted to the float numbers. After the conversion, we have obtained a 300 dimensional array. It can be converted to a one dimension array using max concatenation. For obtaining a one single vector, we have concatenated permissions binary feature array, API class calls feature array and word embeddings.

To sum up, We have ended up with 5324 cells which are the concatenation of the three features. The binary feature vector with 165 elements represents the apps permissions, similarly the binary feature vector with 4859 elements represents the apps API class calls and lastly, 300 float values represent max of the word embeddings which is the conversion of the strings.

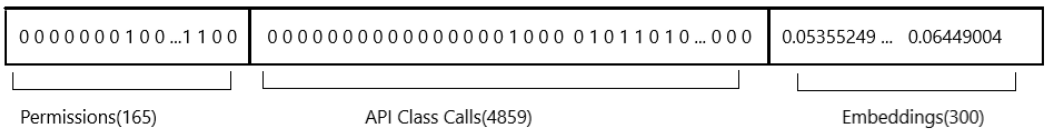

Figure 2.4: Concatenated GloVe Vector

2.4.4 Creating FastText feature vector

FastText[11] is a library for implementing the word embedding technique published by Facebook which each word represents with a bag of character n-gram. The FastText supports both CBOW and skip-gram model usage. Most word embedding models assign a distinct vector for each word without considering the morphology of words but the fastText model takes into account the morphology of words with character-level knowledge. In Enriching Word Vectors with Subword Information paper[12] explains how word vectors can be obtained using subword information. Basically, Fasttext breaks words into n-grams of characters.

[1] `https://nlp.stanford.edu/projects/glove/`

It can be added special symbols < and > to the beginning and end of each word. Let's examine the word *where* while n= 3. The representation of a word where can be *<wh, he, her, ere, re>*. Each n-gram can have a vector representation and each word has the sum of n-grams vector representation. In this way, fastText is supporting character-level knowledge.

We have used 300 dimensional pre-trained fastText model that was downloaded from official fastText website [2]. Permissions and API class calls converted to the binary feature array. The same procedures were applied for permissions and API class calls as we did in the GloVe feature vector. Also for strings, we have followed the same path. The only difference is we have used a pre-trained fastText model. To sum up, we ended up with 5324 cells which are the concatenation of the three arrays.

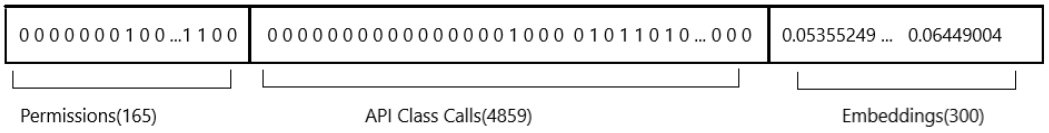

Figure 2.5: Concatenated FastText feature Vector

2.4.5 Creating BERT feature vector

Pre-trained models can be two types which are context-free and contextual representations[13]. GloVe and word2Vec are context-free. As an example, "bank" has the same representation and has created the same word embeddings in "bank account" and "bank of the river". Unlike the other embeddings, BERT is based on a contextual model that creates a representation of a word according to other words in the sentence. BERT is a NLP technique called Bidirectional Encoder Representations from Transformers published by Google[14]. To begin with, OpenAI applies just left-right transformers and ELMO applies the concatenation of left-to-right and right-to-left LSTM but BERT applies bidirectional transformers which means that it uses both left and right content of words. This feature allows you to learn the context of words from both right and left words.

[2] `https://fasttext.cc/docs/en/english-vectors.html`

The main steps in BERT architecture that uses two unsupervised training tasks are the Masked language model(MLM) which randomly masks some tokens and the next sentence prediction is used. In Masked LM 15% of input randomly is chosen as mask tokens in each sentence and after that tries to predict [MASK] tokens instead of the entire sentence. The next task is Next Sentence Prediction that is taken as a pair of sentences A and B and predict that sentence B can follow B or not. It has been used in three different embeddings which are token embeddings, segment embeddings and position embeddings.

- **Token embeddings :** [SEP] tokens are added to at the end of the sentence for separation of sentence and [CLS] tokens can be added to the beginning of the first sentence.
- **Segments embeddings :** It is for distinguishing the different sentences.
- **Segments embeddings :** It is for specifying the position of each token in the sentence.

In implementation it will be used **bert-as-service** [3] which helps to sentence encoding. Bert-as-service uses BERT for encoding sentences and representing sentences into fixed length representations. Similarly to GloVe and Fasttext embeddings, BERT has pre-trained models. As a first step, a pre-trained BERT model must be downloaded and loaded into memory. It will be used BERT-Base Multilingual cased. After the download of the pre-trained model it has to install Bert server and client. When bert-as service starts to run, we can get encoding sentences by writing a couple of lines of codes. Bert-as-services converts each sentence to the 768 dimension using BERT. In order to reduce dimension, we can apply max,sum and average concatenation to the word embeddings. In our case, max concatenation is applied to the word embeddings. Permissions and API class calls are converted the same as Glove and FastText vectors. To summarize,the bert feature vector includes 5792 cells.

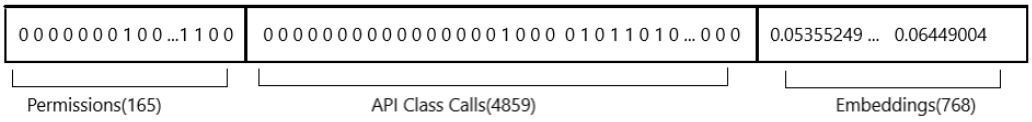

Figure 2.6: Concatenated Bert Vector

[3] `https://github.com/hanxiao/bert-as-service`

Chapter 3

APK Classification

In the previous chapter, we have created three different feature vectors by using NLP techniques. This chapter covers the classification step with three different separate feature vectors and also compares their performance. The deep neural network was chosen as a classifier. Firstly, our first step is pre-processing which is preparing our feature vectors and categories for classification. After the pre-processing, we'll focus on tuning the hyper-parameters and select the best parameters and the model architecture. Finally, we'll conclude this part with a comparison of the classification of models using different feature vectors.

3.1 Building a Model

3.1.1 Data Pre-processing

The first step is the preparation of feature vectors and categories for classification. Since we have more than 2 categories we'll implement multi-class classification. Each application is one sample and is labelled as one of the 15 possible categories. The feature vector that we have obtained in the previous section can have higher and lower absolute values. Having an absolute value of a feature can dominate other features and have a higher impact on the model. To avoid this we will transform the feature vector. Transformation can be done by implementing different techniques. Standardization makes re-scaling the distribution of values that transform standard deviation to 1 and have a mean of zero. Each value is standardized by using the equation 3.1.

$$z = \frac{score - mean}{standarddeviation} \tag{3.1}$$

The other issue we need to consider is converting a categorical variable into a numerical form. A label encoder will be used for the transformation of the categorical variables.

Basically, It assigns to each string a numerical variable starting from 0 and our range will be between 0 to 14. In order to make more reliable predictions, model evaluation should be done with data not used in the training set. We are taking our data set and dividing it into two subsets. One subset is training data which will be used to fit our model. The other subset is test data which will be used to evaluate our model. In this way, we can also specify if our model suffers from underfitting or overfitting. In our data set %80 of data will be used for the training phase and %20 of data will be used for the testing phase.

- Training set includes 80% of the data set and has **3162 samples**.
- Testing set includes 20% of the data set and has **791 samples**.

However dividing our data set into two subset which is called the holdout method is not enough for tuning hyper-parameters and obtaining precise results. We can also use a more reliable cross-validation technique. **Stratified k-Fold Cross Validation** has been used in our case. Stratified k-Fold Cross Validation makes sure that data is splitting equally in each fold. We have picked k as 10 and it is splitting the data set into 10 folds. For each iteration k-1 fold is used for training and the remaining fold is for test. Model will be trained and tested 10 times.

3.1.2 Evaluation Metrics

For evaluation of hyper-parameters and models, Accuracy, precision, and recall have been used. Accuracy is one of the most commonly used evaluation metrics which divide the number of correctly predicted samples by the number of total samples. The accuracy formula is in the following.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.2}$$

If needed to examine deeper and for having more information, precision and recall can be used. Precision indicates how many positive predictions are correct. The number of positive true predictions divided by total positive predictions gives the precision result.

$$precision = \frac{TP}{TP + FP} \tag{3.3}$$

Recall is the number of positive predictions divided by the number of total samples that need to actually belong to the positive predictions.

$$recall = \frac{TP}{TP + FN} \tag{3.4}$$

3.1.3 Tuning Hyper-parameters and Selecting the Best Results

For obtaining the best result, we need to tune hyper-parameters and a naive approach is to try every possible combination of different features. **Grid Search** is a hyper-parameter tuning method and provides a model for each possible combination of hyper-parameters. We can compare the results of each possible combination and select the best result.

The list of hyper-parameters for neural networks:

- ***Number of Units :*** It is a model hyper-parameter that must be specified. 50,100, 200 units have been tried in our case.
- ***Activation Function :*** It transforms the given input value to the output value in each node. Some activation functions are ReLU, Sigmoid, Softsign, Leaky ReLU.
- ***Number of Epochs :*** It is a hyper-parameter that specifies how many times the algorithm needs to run to cover all training data sets. Higher epochs might indicate overfitting.
- ***Batch Size :*** It is a hyper-parameter that specifies how many numbers of samples should be passed through to the network at a time. A larger batch size requires more computational power. The batch size can be one of the three modes. Batch Gradient Descent(Batch mode) where the batch size equals the number of the training data set. Stochastic Gradient Descent(Stochastic mode) where the batch size is equal to one and mini-batch gradient descent(Mini batch mode) where batch size between more than one and less than the number of the training data set. It can be 32,64,128 and 256. A smaller batch size updates weights more frequently.
- ***Optimizer :*** It updates the weight of neurons for minimizing the loss between expected and predicted results. Some available optimizer functions: Adam, Adagrad, SGD, AdaDelta.
- ***Learning rate :*** When the learning rate is set to a low number, the training process could be extremely slow since it makes small updates to the weights. The learning rate range can be between 0 and 1.
- ***Weight initialization :*** It is assigning initial random weights for the neural network. Some of the weight initializers are uniform, normal, and glorot_normal.

- ***Number of hidden layer :*** The hidden layer takes the input from the previous layer and uses the activation function to produce output then passes this output to the next layer.

The first parameter we need to tune is the **Number of units** for the input layer. Since our machine computational power is very low we have provided a small number of units. We have tried 3 different numbers of units starting from 50 units up to 200 units. We obtained the highest (44.562 %) accuracy with 200 units in the model with Bert feature vector and also in the model with GloVe and FastText feature vectors with 200 units gave the best result.

Metrics	**Accuracy**	**Precision**	**Recall**
Units	*Bert Vector*		
50	30.270%+/-6.446%	29.337%+/-8.208%	30.120%+/-6.653%
100	42.602%+/-2.048%	44.690%+/-2.268%	42.574%+/-2.104%
200	**44.562%+/-2.19%**	**45.972%+/-2.106%**	**44.577%+/-2.08%**
Units	*GloVe Vector*		
50	36.433%+/-3.628%	36.949%+/-3.135%	36.298%+/-3.52%
100	40.229%+/-1.849%	40.931%+/-1.969%	40.120%+/-1.824%
200	**41.524%+/-1.600%**	**42.687%+/-1.585%**	**41.484%+/-1.817%**
Units	*FastText Vector*		
50	37.826%+/-2.634%	38.998%+/-2.694%	37.630%+/-2.662%
100	40.544%+/-2.055%	41.562%+/-2.034%	40.396%+/-2.187%
200	**41.968%+/-1.996%**	**42.225%+/-1.752%**	**41.836%+/-1.898%**
Parameters	One dense layer	Batch size is 32	Epoch is 100
Optimizer	Adagrad(lr=0.01)	Activation function	relu

Table 3.1: Results obtained from tuning of number of units with using grid-SearchCV

After specifying the number of units, our next and essential step is tuning the **activation functions**. We mainly focused on non-linear activation functions because linear activation functions have a low power of compute complex parameters. The main reason we do not use linear functions is it is not possible to use back-propagation since the derivation of the linear function is constant.

If we use linear functions, our neural network behaves like a single layer perceptron and this is not a desirable aspect that we would like to perform. Firstly, we can take a look at basic activation functions like sigmoid, ReLU. Sigmoid function is the one of the widely used non-linear functions. It gives output between the range 0 and 1. However sigmoid function during the back-propagation can cause the vanishing gradient problem. The derivative of sigmoid activation function can take a maximum 0.25 [15]. Equation 3.5 shows sigmoid function derivation formulation.

$$f'(x) = \text{sigmoid}(x) \times (1 - \text{sigmoid}(x)) \tag{3.5}$$

We can try to use another commonly used activation function which is ReLU for avoiding vanishing gradient problems. The expression and derivation of ReLU functions are in the equation 3.6 and 3.7.

$$f(x) = x, x > 0 \qquad f(x) = 0, x <= 0 \tag{3.6}$$
$$f'(x) = 1, x >= 0 \qquad f'(x) = 0, x < 0, \tag{3.7}$$

If the input value is negative or equal to zero , it assigns zero as output value. If output of neuron is equal to zero, we can't calculate the derivation of neuron so it can't update the weight during the back-propagation and it is creating a dead neurons which means it can't have a contribution of network training phase. This is called as a dying ReLU issue. As an improvement of ReLU function we have been using Leaky ReLU and ELU activation function. The main different between ReLU and Leaky ReLU, it will return small number rather than returning zero when the input value less than zero which is 0.01 times x. Similar to Leaky ReLU ELU function produces negative values when the input less than zero. [3] ELU has alpha positive integer constant. The equation 3.8 belongs to Leaky ReLU activation function and the equation 3.9 is belongs to ELU functions.

$$f(x) = x, x > 0 \qquad f(x) = 0.01x, x <= 0 \tag{3.8}$$
$$f(x) = x, x > 0 \qquad f(x) = \alpha.(e^x - 1), x <= 0, \tag{3.9}$$

Furthermore, Figure 3.1 shows a graph of the ReLU, Leaky ReLU, and ELU functions. As we mentioned before and the following figure below shows us when the input value is negative or equal to zero, the output of the ReLU function always assign to zero. However, when we use leaky ReLU we can see that the line slopes to the left.

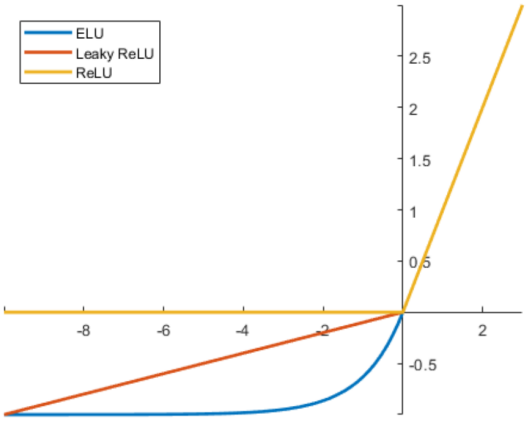

Figure 3.1: The graph of ReLU, Leaky ReLU AND ELU functions, taken from [16]

In addition to these functions, softsign and softplus activation functions are involved in our experiment. They don't cause vanishing gradient problems however they are much slower than ReLU. In our case we have observed that the best performance function is softsign in model with Bert and GloVe feature vectors. In models with Fasttext feature vector sigmoid function gives better accuracy(44.119 %) according to the other activation functions. In sum up, we'll use softsign as an activation function of the input layer in a model with Bert and GloVe feature vector and sigmoid function in Fasttext feature vector.

Metrics	Accuracy	Precision	Recall
	Bert Vector		
Sigmoid	40.069%+/-3.831%	42.133%+/-4.021%	39.875%+/-3.860%
Softplus	44.309%+/-2.389%	46.209%+/-1.916%	44.251% +/- 2.384%
Softsign	**45.668%+/-1.523%**	**47.104%+/-1.395%**	**45.632%+/-1.504%**
ReLU	44.277%+/-1.810%	46.031%+/-2.331%	44.209%+/-1.705%
ELU	45.163%+/-1.890%	47.034%+/-1.58%	44.993%+/-1.934%
Leaky ReLU	43.549%+/-1.808%	46.292%+/-1.766%	43.552 %+/-1.749%
	GloVe Vector		
Sigmoid	39.913%+/-3.421%	40.522%+/-3.109%	39.513%+/-3.449%
Softplus	42.348%+/-1.519%	43.092%+/-1.789%	42.286%+/-1.541%
Softsign	**44.182%+/-1.772%**	**44.893%+/-1.802%**	**44.144%+/- 1.850%**
ReLU	40.988%+/-2.232%	42.141%+/-1.853%	41.014%+/-2.341%
ELU	42.221%+/-2.206%	43.312%+/-2.359%	42.113%+/-2.325%
Leaky ReLU	41.241%+/-1.994%	42.644%+/-1.950%	41.114%+/- 1.860%
	FastText Vector		
Sigmoid	**44.119%+/-2.334%**	**44.658%+/-2.016%**	**44.095% +/-2.276%**
Softplus	40.450%+/-2.491%	41.087%+/- 2.200%	40.325%+/-2.511%
Softsign	43.898%+/-2.386%	44.480%+/-2.300%	43.712% +/-2.244%
ReLU	41.429%+/-2.077%	41.860%+/-1.763%	41.361%+/-2.190%
ELU	41.557%+/-1.918%	41.779%+/-1.998%	41.466%+/- 1.984%
Leaky ReLU	40.829%+/-2.954%	42.101%+/-2.721%	40.675 %+/-2.98%
Parameters	One dense layer	Batch size is 32	Epoch is 100
Optimizer is	Adagrad(lr=0.01)	Number of units	200

Table 3.2: Results obtained from tuning activation functions with using grid-SearchCV

The next parameters are **batch size** and **number of epochs**. We'll use mini-batch Gradient Descent as mentioned before. It divides into a data set less than a total number of samples. If the bath size is large, it may consume a lot of memory and needs more computational power. To sum up, We'll try 32,64,128 batch sizes. Each epoch includes one forward and backward pass of the entire training data set.

In order to learn the pattern of the training data set, the training data set has to cycle more than one time in the neural network so we need more than one epoch. We'll try 100 and 200 epochs in our experimentation.

Metrics	**Accuracy**	**Precision**	**Recall**
BS - Epochs	*Bert Vector*		
32 - 100	45.890%+/-2.035%	47.178%+/-1.626%	45.795%+/-2.078%
32 - 200	**46.807%+/-2.196%**	**47.392%+/-1.974%**	**46.735%+/-2.218%**
64 - 100	45.479%+/-2.979%	48.357%+/-2.968%	45.434%+/-3.051%
64 - 200	46.775%+/-2.004%	47.722%+/-2.067%	46.732%+/-1.945%
128 -100	41.209%+/-2.158%	49.441%+/-2.351%	40.972%+/-2.113%
128 -200	46.174%+/-2.252%	48.535%+/-2.144%	46.193%+/-2.138%
BS - Epochs	*GloVe Vector*		
32 - 100	**44.087%+/-2.044%**	**44.500%+/-1.739%**	**43.973%+/-1.984%**
32 - 200	43.802%+/-1.898%	44.061%+/-1.427%	43.663%+/-1.844%
64 - 100	43.201%+/-1.862%	44.413%+/-1.289%	43.091%+/-1.851%
64 - 200	44.056%+/-2.531%	44.855%+/-2.246%	44.000%+/- 2.539%
128 -100	42.665%+/-2.610%	45.101%+/-2.051%	42.375%+/-2.521%
128 -200	43.707%+/-2.161%	44.421%+/-2.025%	43.601%+/-2.250%
BS - Epochs	*FastText Vector*		
32 - 100	**44.466%+/-1.817%**	**44.977%+/-1.430%**	**44.345% +/-1.742%**
32 - 200	43.771%+/-2.109%	44.019%+/-1.756%	43.661% +/-2.051%
64 - 100	44.404%+/-2.695%	45.195%+/-2.042%	44.288%+/-2.551%
64 - 200	44.023%+/-1.966%	44.773%+/-1.763%	43.884%+/-1.917%
128 -100	42.316%+/-2.417%	44.078%+/-2.311%	42.315%+/- 2.447%
128 -200	43.960%+/-1.916%	44.713%+/-1.611%	43.881%+/-1.985%
Parameters	One dense layer	Activation function	softsign
Optimizer	Adagrad(lr=0.01)	Number of units	200

Table 3.3: Results obtained from tuning batch size and number of epochs with using gridSearchCV

In model with Bert Vector Batch size 32 and 200 epochs gives 46.807 % accuracy which is the best result according to the other parameters. For both models with Glove and Fasttext vectors we'll use 100 epochs and set the batch size to 32.

After the batch size and number of epochs we'll continue with **weight initialization**. To prevent our network from exploding and the problem of vanishing gradient we have involved different weight initialization methods in our experiments which are *uniform*, *normal* and *Glorot_normal*. In weight initialization, a common important point is selecting a value which is closest to the zero. Uniform weight method is to randomly initialize weight from uniform distribution and every value has the same probability. Uniform weight initialization methods will be used in models with Bert and Fasttext vectors. In model with GloVe vector Glorot_normal gave best result with 44.499% accuracy.

Metrics	**Accuracy**	**Precision**	**Recall**
	Bert Vector		
Uniform	**46.806%+/-1.843%**	**48.289%+/-1.613%**	**46.741%+/-1.715%**
Normal	45.826%+/-2.097%	47.034%+/-1.381%	45.758%+/-2.090%
Glorot Normal	46.016%+/-2.428%	48.152%+/-2.428%	46.004% +/-2.375%
	GloVe Vector		
Uniform	43.992%+/-2.131%	44.813%+/-2.070%	43.956%+/-2.131%
Normal	44.372%+/-2.530%	45.148%+/-1.856%	44.312%+/-2.558%
Glorot Normal	**44.499%+/-2.001%**	**45.339%+/-1.686%**	**44.411%+/- 1.975%**
	FastText Vector		
Uniform	**44.054%+/-1.464%**	**44.537%+/-0.866%**	**43.951%+/-1.571%**
Normal	43.803%+/-2.020%	43.992%+/-1.787%	43.781%+/-2.109%
Glorot Normal	43.549%+/-1.896%	44.227%+/-1.377%	43.442%+/-1.824%
Parameters	One dense layer	Batch size is 32	Epoch is 100
Optimizer	Adagrad(lr=0.01)	Number of units	200
Activation	softsign		

Table 3.4: Results obtained from tuning weight initialization with using gridSearchCV

The next step is deciding the **optimizer**. There are many optimization algorithms for reducing the loss in network and obtaining the more accurate results. Four different optimizers were involved in our experiment which are SGD, Adam, Adagrad and Adamax. In all feature vectors, the Adagrad optimizer gave the best score in our experiment. The adagrad[17] is a stochastic optimization algorithm which weights a different learning rate by using past observations.

For parameters with frequently used features it is applying smaller updates and for parameters with infrequently used features it is applying larger updates. The main advantage of the adagrad optimizer is no need to manually tune the learning rate. To summarize,we got the best score with adagrad 46.617% in model with Bert feature vector, 44.782% in model with GloVe feature vector and 44.560% in model with Fasttext feature vector.

Metrics	**Accuracy**	**Precision**	**Recall**
	Bert Vector		
SGD	42.886%+/-2.712%	50.613%+/-3.681%	42.990%+/-2.421%
Adam	40.735%+/-1.946%	42.840%+/-3.469%	40.869%+/-2.066%
Adagrad	**46.617%+/-2.330%**	**48.200%+/-1.900%**	**46.601%+/-2.418%**
Adamax	44.593%+/-2.286%	46.550%+/-2.691%	44.620%+/-2.330%
	GloVe Vector		
SGD	43.202%+/-2.480%	44.526%+/-2.410%	43.068%+/-2.532%
Adam	40.101%+/-2.080%	43.030%+/-2.328%	39.931%+/-2.037%
Adagrad	**44.782%+/-1.518%**	**45.583%+/-1.310%**	**44.733%+/-1.546%**
Adamax	42.473%+/-2.056%	44.523%+/-2.324%	42.194%+/-2.066%
	FastText Vector		
SGD	42.095%+/-2.814%	43.118%+/-2.547%	42.081%+/-2.911%
Adam	39.596%+/-2.485%	42.104% +/- 2.247%	39.451%+/-2.497%
Adagrad	**44.560%+/-1.265%**	**45.119%+/-1.240%**	**44.488%+/-1.300%**
Adamax	42.348%+/-2.131%	43.700%+/-2.275%	42.293%+/-2.063%
Parameters	One dense layer	Batch size is 32	Epoch is 100
Activation	softsign	Number of units	200

Table 3.5: Results obtained from tuning optimizer with using gridSearchCV

After the optimizer was selected, one of the most significant hyper-parameters which is **learning rate** has been tried to our neural network. If the lower value is assigned to the learning rate, the training process can be slow. However, if the learning rate is too high, it can cause divergent behaviors. We have tried 0.001% , 0.01% and 0.1% parameters. As it shows in table 3.6, we can obtain 45.542% accuracy with 0.01% of learning rate in a model with the Bert feature vector. For the model with the GloVe vector at most we have reached 44.150% accuracy with 0.01% of learning rate. Like the other two models, we have obtained the best result with 0.01% learning rate in a model with Fasttext vector. To conclude, it will use the same learning rate which is 0.01% in three neural networks.

Metrics	**Accuracy**	**Precision**	**Recall**
LR	*Bert Vector*		
0.001%	42.759%+/-2.497%	43.661%+/-2.236%	42.638%+/-2.500%
0.01%	**45.542%+/-1.698%**	**47.739%+/-1.733%**	**45.474%+/-1.703%**
0.1%	13.949%+/-2.429%	8.825%+/-2.945%	13.529/- 2.459%
LR	*GloVe Vector*		
0.001%	41.715%+/-2.106%	42.309%+/-1.747%	41.587%+/-2.178%
0.01%	**44.150%+/-2.519%**	**45.111%+/-2.368%**	**44.104%+/-2.543%**
0.1%	16.351%+/-1.322%	12.397%+/-3.060%	15.877%+/-%1.287
LR	*FastText Vector*		
0.001%	40.450%+/-1.981%	41.107%+/-1.968%	40.369%+/-2.070%
0.01%	**44.339%+/-2.763%**	**44.515%+/-2.118%**	**44.286%+/-2.786%**
0.1%	24.226%+/-2.190%	24.478%+/-5.293%	24.069%+/-2.221%
Parameters	One dense layer	Batch size is 32	Epoch is 100
Activation	softsign	Number of units	200
Optimizer	Adagrad		

Table 3.6: Results obtained from tuning learning rate with using gridSearchCV

Another essential point that we need to consider when building a neural network is about avoiding overfitting. To be able to produce better results in our test set, we have capitalized on regularization techniques. There are several regularization techniques such as L1& L2 regularization, Data Augmentation, Dropout, and Early stopping. Data augmentation is a technique that increases the size of the training data set. In our case, we'll not use that regularization technique. Early stopping techniques will be tried after all parameters have been specified. Dropout technique randomly selects and removes the units during the training in every iteration. When removing the units, in and out connections of the units are cut off. **Dropout** can be applied to the input layer and also the hidden layer. In our case, 0.1 ,0.3 and 0.5 dropout rates have been used. Table 3.7 shows us the results of three different models when we applied dropout to the network.

We have specified basic hyper-parameters for three different neural networks.

Metrics	**Accuracy**	**Precision**	**Recall**
Dropout Rate	*Bert Vector*		
0.1	46.205%+/-2.503%	47.655%+/-1.907%	46.124%+/-2.487%
0.3	**46.934%+/-2.796%**	**48.291%+/-2.396%**	**46.853%+/-2.728%**
0.5	46.301%+/-2.319%	48.289%+/-2.111%	46.337%+/-2.433%
Dropout Rate	*GloVe Vector*		
0.1	**43.992%+/-2.316%**	**44.485%+/-2.067%**	**44.008%+/-2.389%**
0.3	43.896%+/-1.383%	44.884%+/-1.147%	43.866%+/-1.391%
0.5	43.991%+/-1.363%	44.793%+/-0.976%	43.973%+/-1.462%
Dropout Rate	*FastText Vector*		
0.1	42.947%+/-2.112%	44.752%+/-2.091%	42.910%+/-2.070%
0.3	43.518%+/-2.006%	44.870%+/-1.446%	43.490%+/-1.967%
0.5	**44.435%+/-2.118%**	**45.673%+/-1.755%**	**44.273%+/-2.171%**
Parameters	One dense layer	Batch size is 32	Epoch is 100
Optimizer	Adagrad(lr=0.01)	Number of units	200
Activation	softsign		

Table 3.7: Results obtained from dropout regularization with using gridSearchCV

After we have set some basic hyper-parameters, we need to specify the **number of hidden layers**. We can define our final network structure by defining the hidden layers. To obtain better results we can add hidden layers between the input and output layer. Table 3.8 shows the tuning of the number of hidden layer results of the model using the Bert feature vector. In Bert feature vector model dense layer with 200 units plus dense layer with 100 units gives the best result which is **46.806%** of accuracy and also in GloVe feature vector model with **44.277%** accuracy dense layer with 200 units plus dense layer with 100 units the best result. The results of the model with the GloVe vector are in table 3.9. Finally, a model with the Fasttext feature vector, like the other feature vectors, has two dense layers and we have obtained **42.980%** accuracy by using a dense layer with 200 units plus a dense layer with 100 units. Table 3.10 shows the tuning of the number of hidden layer results of the model with the Fasttext feature vector.

Metrics	**Accuracy**	**Precision**	**Recall**
	Bert Vector		
Dense(200)	45.668%+/-1.754%	46.395%+/-1.741%	45.630%+/-1.693%
Dense(200) **Dense(100)**	**46.806%+/-2.012%**	**47.660%+/-1.569%**	**46.721%+/-1.995%**
Dense(200) dense(100) dense(20)	42.379%+/-2.807%	43.709%+/-3.119%	42.408%+/-2.882%
Dense(200) dropout(0.3) dense(20)	32.323%+/-3.366%	33.735%+/-2.721%	32.321%+/-3.450%
Parameters	Weight uniform	Batch size is 32	Epoch is 200
Activation	softsign	Number of units	200
Optimizer	Adagrad	Learning rate	0.01

Table 3.8: Results obtained from tuning number of hidden layers with using gridSearchCV

Metrics	**Accuracy**	**Precision**	**Recall**
	GloVe Vector		
Dense(200)	42.410%+/-1.779%	44.038%+/-1.536%	42.310%+/-1.810%
Dense(200) **Dense(100)**	**44.277%+/-1.960%**	**44.724%+/-1.604%**	**44.137%+/-1.952%**
Dense(200) dense(100) dense(20)	43.043%+/-1.999%	43.602%+/-2.111%	42.930%+/-2.089%
Dense(200) dropout(0.1) dense(20)	32.035%+/-3.380%	32.446%+/-3.077%	32.057%+/-3.476%
Parameters	glorot_normal	Batch size is 32	Epoch is 100
Activation	softsign	Number of units	200
Optimizer	Adagrad	Learning rate	0.01

Table 3.9: Results obtained from tuning number of hidden layers with using gridSearchCV

Metrics	**Accuracy**	**Precision**	**Recall**
	Fasttext Vector		
Dense(200)	41.872%+/-%2.163	43.224%+/-1.89%	41.840%+/-2.072%
Dense(200)			
Dense(100	**42.980%+/-1.762%**	**43.579%+/-1.848%**	**42.663%+/-1.951%**
Dense(200)			
Dense(100)			
Dense(20)	29.509%+/-3.223%	29.248%+/-3.386%	29.215%+/-3.145%
Dense(200)			
Dropout(0.5)			
Dense(20)	15.814%+/-1.869%	9.525%+/-3.636%	15.164%+/-1.894%
Parameters	Weight uniform	Batch size is 32	Epoch is 100
Activation	sigmoid	Number of units	200
Optimizer	Adagrad	Learning rate	0.01

Table 3.10: Results obtained from tuning number of hidden layers with using gridSearchCV

The last point that we need to take into consideration is applying an early stopping to avoid overfitting, which is mentioned before. Having a high number of epochs can lead to over-fitting in the neural network and during the training phase, overfitting can occur when the gap between validation loss and training loss starts increasing. Early stopping provides stop training when the validation error increases. Figure 3.2 shows the Bert feature vector model loss chart without using early stopping and with using early stopping. As seen in the left chart in figure 3.2, the model tends to over-fit after the 80 epochs therefore for avoiding over-fitting in our model training phase must be stopped after the 80 epochs, and the validation loss decreased by using the early stopping.

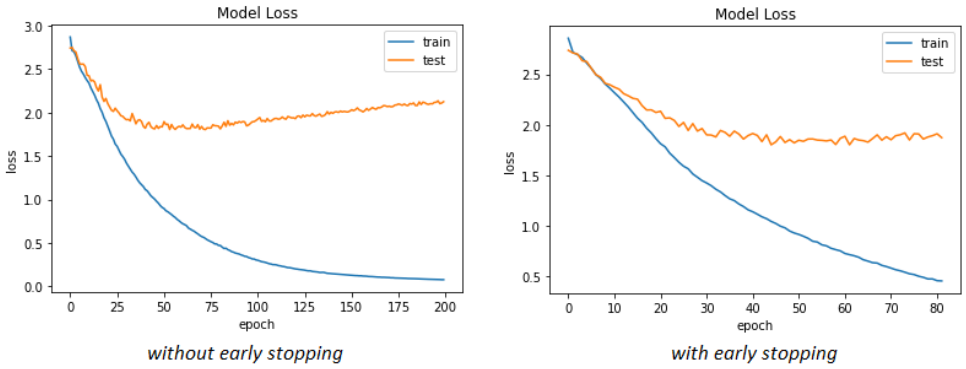

Figure 3.2: Effect of early stopping to the Bert feature vector model loss

The model using the GloVe feature vector has used 100 epochs. If we look at figure 3.3, we can observe that over-fitting starts at epoch 60 so when early stopping is used, the model training stops at epoch 60. In addition to this, the validation loss of the model decreases.

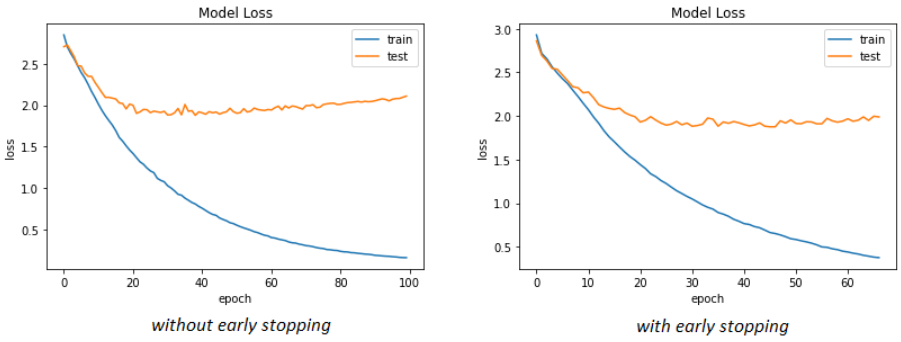

Figure 3.3: Effect of early stopping to the Glove feature vector model loss

Lastly, in the model using the Fasttext feature vector like the Glove feature vector we have used 100 epochs. We have observed that 100 epochs is the optimal number of epochs and does not lead to overfitting. If early stopping applies to the model, we can say that the model does not stop before the 100 epochs.

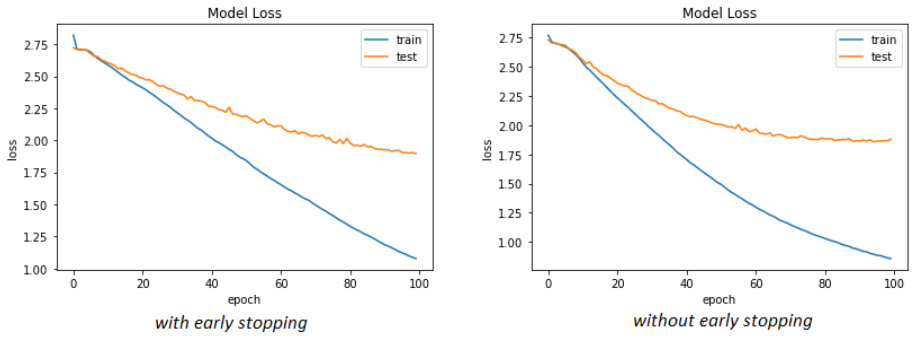

Figure 3.4: Effect of early stopping to the Fasttext feature vector model loss

3.2 Classification Results

To summarize, Three different neural networks were created by using different feature vectors. We have tried to find the best results by tuning the hyper-parameters for three different models. Since implementing the multi-class classification and having more than two label classes, the **Softmax** activation function was used for the output layer and **sparse_categorical_crossentropy** was used as a loss function in three neural networks. The main advantage of sparse_categorical_crossentropy is saving time. In order to use sparse_categorical_crossentropy categorical labels must be converted to the integer value. In our case conversions are done in the pre-processing step. In the model that used the Bert feature vector, the following architecture has been implemented: As an input layer 5792 nodes have been used and go towards the dense layer of 200 units using the softsign activation function and the uniform is assigned as a kernel initializer. After that, the dense layer of 200 units goes into a dense layer with 100 units with the elu activation function and uniform kernel initializer. As an optimizer we have used Adagrad with a 0.01 learning rate. Lastly, the Neural network ends up with an output layer using softmax activation function with 15 nodes. With this architecture we have reached 46.806% accuracy with a 2.012% standard deviation. The recall is 46.721% with a 1.569% standard deviation and precision is 47.660% with a 1.995%. Table 3.11 shows a summary of all hyper-parameters for model with the Bert feature vector.

Hyperparameters	Values
Number of units	200
Activation Function	softsign
Batch size	32
Number of epochs	200
Weight Initialization	uniform
Dropout Regularization	0.3
Optimization Algorithm	Adagrad
Learning rate	0.01
Loss Function	sparse_categorical_crossentropy

Table 3.11: Best hyperparameters of model using Bert feature vector

In the model that used the GloVe feature vector, the following architecture has been implemented: As an input layer 5324 nodes have been used and go towards the dense layer of 200 units using the softsign activation function and glorot_normal is assigned as a kernel initializer. The dense layer of 200 units goes into a dense layer with 100 units with a softsign activation function and a uniform glorot_normal initializer. Adagrad is used with a 0.01 learning rate. Lastly, Neural network ends up with an output layer using softmax activation function with 15 nodes(15 possible categories). We have reached 44.277% accuracy with a 1.952% standard deviation with this architecture. The recall is 44.137% with a 1.569% standard deviation and precision is 44.724% with a 1.604%. We can see that all hyper-parameters of model with GloVe feature vectors in table 3.12.

Hyperparameters	Values
Number of units	200
Activation Function	softsign
Batch size	32
Number of epochs	100
Weight Initialization	glorot_normal
Dropout Regularization	0.1
Optimization Algorithm	Adagrad
Learning rate	0.01
Loss Function	sparse_categorical_crossentropy

Table 3.12: Best hyperparameters of model using GloVe feature vector

Last neural network that we have implemented uses Fasttext feature vector. The architecture of our last neural network is in the following : As an input layer, 5324 nodes have been used to go towards the dense layer of 200 units with a sigmoid activation function and the uniform is assigned as a kernel initializer. The dense layer of 200 units goes into a dense layer of 100 units with a sigmoid activation function and a uniform glorot_normal initializer. In addition to this, Adagrad optimizer was used with a 0.01 learning rate. Lastly, Neural network ends up with an output layer using softmax activation function with 15 nodes(15 possible categories). We have reached 42.980% accuracy with a 1.762% standard deviation with this architecture. The recall is 42.663% with a 1.951% standard deviation and precision is 43.579% with a 1.848%. We can see that all hyper-parameters of model with Fasttext feature vectors in table 3.13.

Hyperparameters	**Values**
Number of units	200
Activation Function	softsign
Batch size	32
Number of epochs	100
Weight Initialization	uniform
Dropout Regularization	0.5
Optimization Algorithm	Adagrad
Learning rate	0.01
Loss Function	sparse_categorical_crossentropy

Table 3.13: Best hyperparameters of model using Fasttext feature vector

3.2.1 Comparison of Results

As needed to compare three different neural networks that we have implemented, the model with Bert feature vector gave the best result with the 46.806% accuracy according to the word embeddings vectors model as seen in figure 3.5. The reason is that Bert learns the context of words from both right and left words. In this way, a model with a bert feature vector can collect more accurate information about the text semantic. To conclude this chapter, We can say that models with sentence embedding vectors give a better result than models with word embedding vectors.

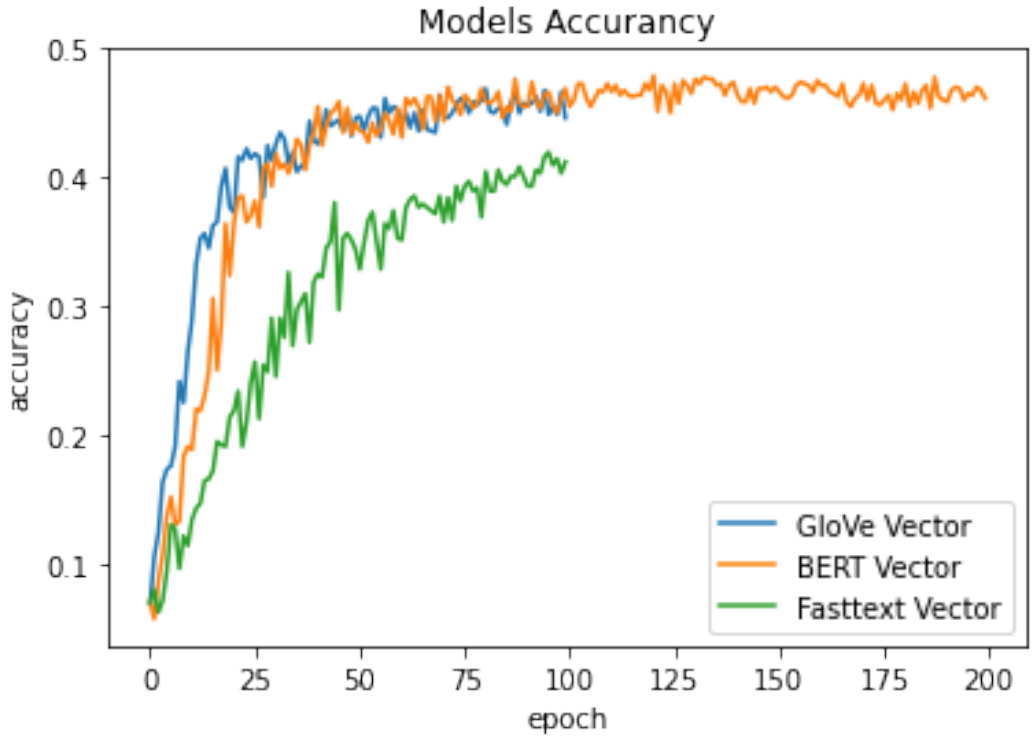

Figure 3.5: Comparison of three different neural network architecture

Chapter 4

Data Preparation for Activity Classification

4.1 Overview

Android applications can have four different **components** [1] and these components are activities, services, broadcast receivers, and content providers. Services handle background operations, hence not providing a user interface. Broadcast receivers provide interactions with other applications or from the system. Content provider servers application data to the other applications and manages to store them in the file system. If the content provider gives permission to the other applications, apps may modify or query the data. Android activities[2] one of the essential components of the Android applications which represent a single screen for handling the user interaction. Each android application may consist of more than one screen which means it has more than one activity. For instance, In a food ordering application, one of the activities may contain a list of restaurants that users can select the restaurant for ordering food and another activity of the same application can contain a login screen.

When a developer creates activities for their android app, A java file and an XML layout file must be created for every activity. To make these activities visible, the developer has to register activities information to the application manifest file. Various UI elements are linked to activity in a hierarchical way such as check boxes, buttons, slider, text fields and these UI elements are statically designated in the XML layout file; however we can create or destroy some UI elements on runtime.

---

[1] `https://developer.android.com/guide/components/fundamentals`

[2] `https://developer.android.com/reference/android/app/Activity`

Android developers may conduct **UI design patterns** and **UI design guidelines** [3] to overcome common problems in the design interface. The design pattern and guidelines are not only significant from the side of the developer but also contribute to conditions of providing a better experience to the user and increasing usability.

In Google I/0 2010 conference[4] Richard Fulcher, Chris Nesladek, Jim Palmer and Christan Robertson presented Android UI design patterns and later on In Google I/0 2019 conference[5] Roman Guy has introduced Declarative UI patterns. Using design patterns provides a common user interface in most applications, ensures user retention and the user does not need to learn new gestures. As an example, if we examine the login activity of the application, from a user point of view it is expected to have input fields for entering login credentials and one login button or in the search page users expect to come with input fields where users can enter keywords of searched items. Our point of view, we may consider that having similar types of activity can have similar design patterns and this can help us in our experiment.

4.2 Activity Structure and UI Elements

In Android applications, each activity possesses multiple UI elements such as Button, Checkbox, Linear Layout. These UI elements have structural and hierarchy behaviours and are divided into groups as different members of the classes.

- **The Android View** is an essential class for UI elements. Android provides buttons, spinner as view objects.
- **The Android View Groups** is a subclass of the View which includes invisible containers for helping the structural design.

When the developer starts to create an XML file for a desirable activity, the Root element which has to be a view or view group object must be chosen as a first step. After specifying the root element, the developer can add essential child elements. These child elements can belong to the view or view group. For instance, Buttons can be added to perform the desired action. An application can take different information from users as an input, in this case the developer can use Edit Text which allows the user to enter text into an application. List View contains a list of elements and provides a display of these elements as a list and also developers can use RecyclerView as an improvement of the list view.

[3] `https://developer.android.com/design`

[4] `https://www.youtube.com/watch?v=M1ZBjlCRfz0`

[5] `https://www.youtube.com/watch?v=VsStyq4Lzxo`

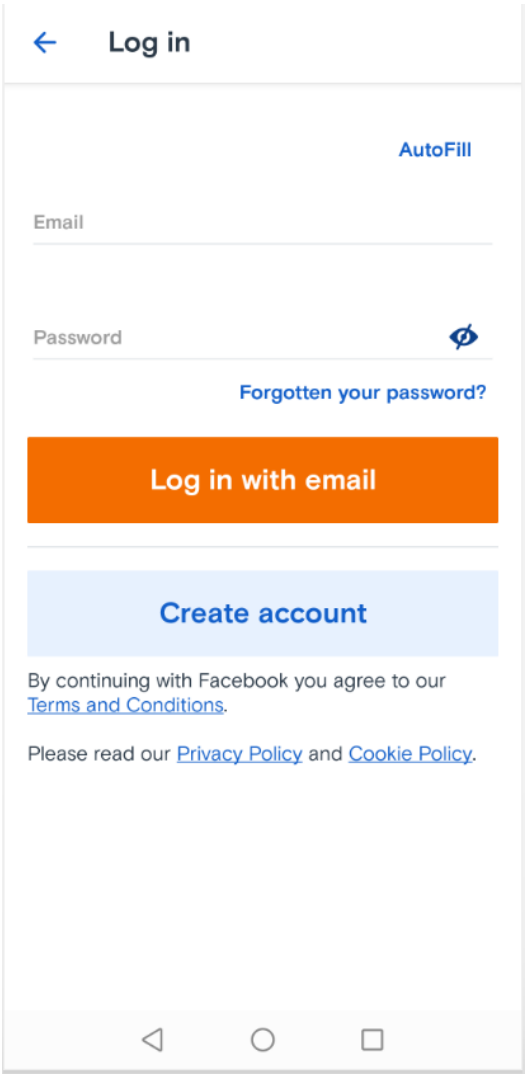

Figure 4.1: Login Activity

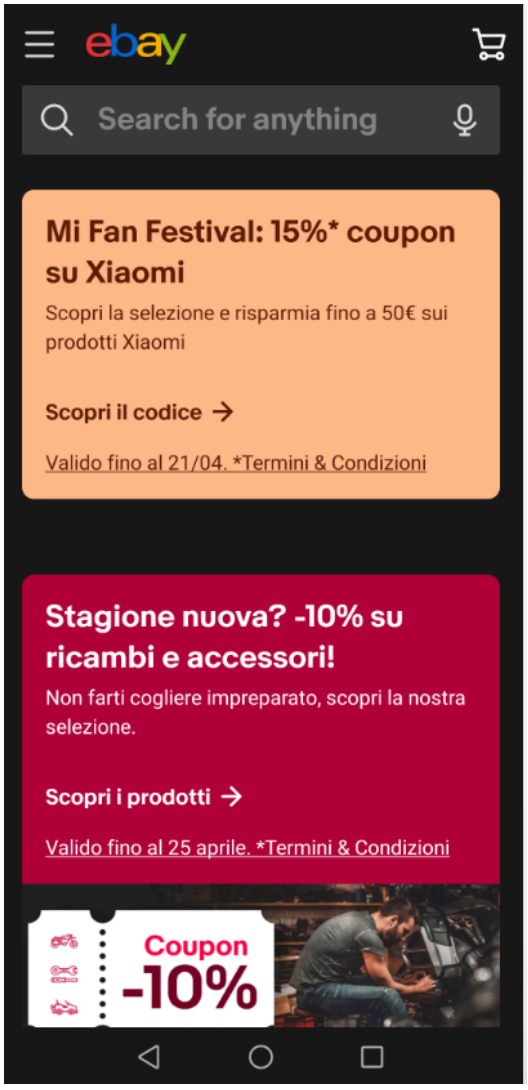

Figure 4.2: Main Activity(ebay)

In figure 4.1 It can see that there are two text fields that the user can give his information for reaching application features and also in these activities it must include at least one button to execute the login action. On the other hand, it can be observed that figure 4.2 is more representative and congested activity is the main activity. The main activity can include the bottom bar or navigation bar. To summarize, developers can use the same UI elements in the same type of activity and follow a specific path for each activity so the next step is to specify the activity types.

4.3 Creation of Activity Data set

4.3.1 Specifying Activity Types

When we create our activity data set the first step is determining the activity types. To begin with, we have examined the Automation of Android Applications Testing Using Machine Learning Activities Classification paper proposed by Rosenfeld, Kardashov and Zang[18] and they have specified 7 types of activities. Our desirable aspect is finding common activity types for all possible categories. To sum up, we have ended up with 8 activity types. In the following list, we can see the activity types in detail.

1. **Splash Activity :** Splash activity is the welcome page of the activity in which the user first interacts with the application while the application is launched. This type of activity is in every application. Splash activity has a simple structure and generally, it just includes the application logo and it can be one ***image View*** or just a simple text View. Also, observed that in some cases developers may add a progressive bar to their implementation.

2. **Login Activity :** Login activity allows a user to reach his application profile by entering his credential information. Mostly it includes two ***edit Text*** which can take username or email and password and also at least one sign in button. Application can offer users different login types as an example login with Facebook. There may be forgotten passwords and create account options. These options can be a button or clickable text view which can navigate to another activity.

3. **Main Activity :** After the application has launched, the main activity is the first activity that the user actually can generally have an idea about the application and can also perform some tasks. The structure of this activity can include ***DrawerLayout*** and the ***bottom navigation bar***. These types of widgets aim to navigate different sections of the application. Usually, a drawer Layout is used when a developer needs to navigate a lot of different sections. The bottom navigation bar contains fewer destination sections. Each destination layout can have an icon and text. The main activity includes a lot of imageView and TextView that can be the advertising of different sections and supported by ***scroll view*** and ***horizontal Scroll View***.

4. **Search Activity :** Edit Text can present a search activity that the user can enter search terms for. After pressing the search button, the list can occur, which is a possible result. The list can be listView or ScrollView.

5. **Settings Activity :** Settings activity appears in almost every application for changing application properties. As an example, users can regulate custom notifications, changing country, and language app themes. Each line can include a switch or toggle button that helps to operate desired features.

6. **Camera Activity :** Camera activity mostly used in photography, shopping, productivity, and tools category of applications.
   Besides the photography category, It's used to scan and find the desired product with the captured image or uploaded image from the gallery. The structure of the camera activity consists of a camera layout and an image button that can be below the screen to capture the image. Other image buttons can be implemented such as an open gallery. Also, some camera activity consists of switching to activate or deactivate some camera properties.

7. **Map Activity :** Map activity consists of map layout and is supported by icons and image buttons. In some cases, it includes edit Text where the user can enter the location wants to find.

8. **Advertisement Activity :** Advertisement activity has a simple structure activity type. Usually it includes a full screen advertisement as a web-view and also it can include one or two buttons for opening or closing the associated advertisement.

4.3.2 Collecting Set of Activity

After we have specified the activity categories, the next step is the collection of a set of activities. Since we have created our data set from scratch, we have to collect activities for our data set. To do that we can use the Appium Desktop. Appium Desktop is an interface that can scan and inspect UI elements of activity and is supported by the Appium server. Basically, it includes the Appium server and Appium inspector. In the next session, Appium architecture will be explained in detail. To launch a new session and view UI elements we need to configure Desired capabilities. The device Name that will be used for testing, platform version, and other information should be added as desired capabilities. The reason that we prefer Appium desktop instead of Uiautomator is that we can obtain all XML sources of currently displayed activity. We have collected a total of 180 activities and these activities categories were selected from the 11 most popular categories in the Google Play Store[19], according to 2020 statistics. After collecting the set of XML sources of activity categories, we have decided to write a python script for extracting the vital and needed features for the activity classification step. Table 4.1 shows the number of the activities per activity type in the activity data set.

Category	Number of Activity
Splash Activity	25
Login Activity	25
Camera Activity	25
Settings Activity	24
Search Activity	23
Main Activity	22
Advertisement Activity	19
Map Activity	17

Table 4.1: Number of Activities per each activity type

4.3.3 Determining a Set of Activity Features

Rosenfeld, Kardashov, and Zang[18] in their work they have split the device screen into three areas as top 20%, middle 60%, and bottom 20%. However, we will try to follow a different approach. Instead of finding the elements specific to the activity category by dividing the screen into some areas, we will be getting the all elements features without dividing the screens and their resource ids and involving these resource ids into the feature vectors by using NLP. As we mentioned before, we need to write a python script to extract all needed features and for this first step is specifying features for extraction. In total we have found 18 useful features for UI elements. These features are listed in the following in detail.

- **Number of checked elements :** Generally, The checked feature is used in checkboxes and switches UI elements. If the option of the element is enabled, isChecked is determined as true. In settings activity can have switches and checkboxes.
- **Number of checkable elements :** Checkable features are similar to checked features. The difference from the checked feature is that no checking option is enabled or disable.
- **Number of clickable elements :** The elements can perform a click action considering a clickable element. As an example, buttons, edit Text, menus can have clickable attributes. In splash and advertisement activity includes less clickable elements compared to other activities.
- **Number of edit Text elements :** Number of edit Text is an important determiner. Usually login, search and map activity comprise edit text elements. One edit Text for searching the desired element in the search activity and

one edit Text for searching the location in the map Activity. If one activity contains two or more than two edit texts we can say that this activity is a login activity.

- **Number of focusable elements :**
- **Number of web View elements :** Web view elements included in advertisement activity.
- **Present of drawer Layout :** Another essential feature is drawer layout. Drawer layout used to implement a navigation drawer. When we implement a navigation drawer, the first step is to add the navigation drawer as the root element. Navigation drawer is used in the main activity for navigating different sections of the application. Checking the present of the drawer layout will distinguish the determiner for our classification.
- **Number of recycler view :** The recycler view that is evaluating the list view allows loading a large list of data into an activity. The search, settings and main activity uses a recycler view.

After specifying numerical values for classification steps, as we mentioned before, we have decided to add textual information in our experiment which can help us get more accurate results in classification. When the developer implements an activity and writes codes, he leaves some clues and information about the type of activity. In android application implementation, developers define resource id when they write a code. As an example, when developer implements image button for the camera activity, as a resource id he may write ***packagename:id/takePhotoButton*** or ***packagename:id/captureButton***. In map activity, for map layout developers may write resource id as ***packagename:id/map_view***. Some essential keywords for activity classification:

- **Splash Activity :** Welcome, welcome msg, loading, onboarding
- **Advertisement Activity :** Ad, advertisement, content
- **Login Activity :** Auth, login, email, password ,forgot ,username, input, edit, auto complete, account, user, Facebook
- **Camera Activity :** flash, button, gallery, camera, preview, capture, on, light, take, picture, scan, area, zoom, view, image, snapshot, scanner
- **Search Activity :** search, title, item, text, auto complete, list, recyclerview, show, all
- **Main Activity :** Home, Main, DrawerLayout, mainactivity layout

- **Settings Activity :** Settings, summary, title, switch
- **Map Activity :** Map, Location ,full map, position, address, pin

4.4 Feature Extraction and NLP

In the previous steps, we have specified the useful features for our activity classification. 18 numerical UI elements features have been extracted and also resource ids of each UI element in the activity have been extracted by using the python script. Having these features, we have created different types of feature vectors. In figure 4.3 we can see the creation steps of the different feature vectors.

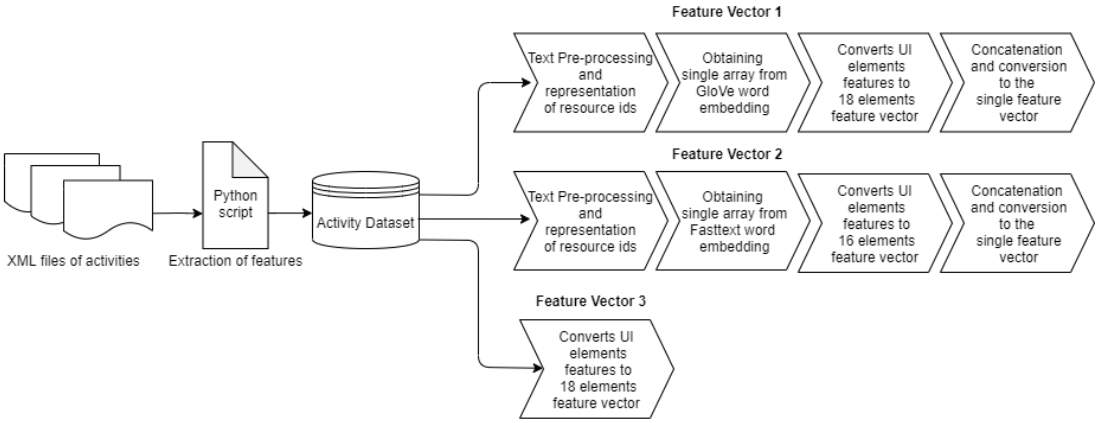

Figure 4.3: Activity feature extraction steps

4.4.1 Generating Feature vector 1

As we did APK feature extraction, The first step is text pre-processing and representation. The steps of text pre-processing and representation are in the following list :

- Separation of elements : Developer writes resource id without using blank space like map_view. To obtain a more accurate result we need to consider map and view as two different words.
- Lower-casing : Upper case elements have been converted to the lower case elements in order to have the same vectorized format.
- Removing special character and numbers : If the resource id includes any special character or number it has been removed from the string.

- Removing irrelevant words : We have created our own stop words list and irrelevant words are removed from the list of resource ids.
- Tokenization and Padding : Same steps have applied like in the apk classification.

After the tokenization and padding we have converted the words to the GloVe word embeddings. We have used pre-trained word embeddings which are 300 dimensional GloVe vectors and each word converted to the float numbers. After the conversion, we have obtained a 300 dimensional array. It can be converted to a one-dimensional array using average concatenation. After the conversion of the resource ids, 18 numerical UI features are converted to the numerical feature array. As a last step, For obtaining a one single vector,we have concatenated 18 elements in the feature array and resource ids array. To sum up, we have ended up with 318 cells of feature vectors.

4.4.2 Generating Feature vector 2

In feature vector 2, Less relevant 2 features are eliminated from the features which are *number of recyclerView elements* and *number of UI elements*. The following list shows the list of features that is used in generation of feature vector.

- Number of checked elements
- Number of checkable elements
- Number of clickable elements
- Number of focusable elements
- Number of focused elements
- Number of long clickable elements
- Number of password elements
- Number of scrollable elements
- Number of selected elements
- Number of image view elements
- Number of button elements
- Number of textView elements
- Number of EditText elements

- Number of DraweLayout elements
- Number of WebView elements
- Number of switch elements

Same steps were followed for the text pre-processing and representation. For word embeddings instead of using GloVe we have decided to use Fasttext pre-trained word embeddings and a 300 dimensional array was obtained. This array was converted to the one dimensional array using the average concatenation. Lastly, the 316 cells of a single feature vector was obtained by concatenation of the two arrays.

4.4.3 Generating Feature vector 3

In feature vector 3, textual information of UI elements was discarded. Only numerical arrays were used in the feature vector 3, to see the impact of using NLP in our classification. To sum up, 18 cells feature vectors were created. These features are listed in the following list.

- Number of checked elements
- Number of checkable elements
- Number of clickable elements
- Number of focusable elements
- Number of focused elements
- Number of long clickable elements
- Number of password elements
- Number of scrollable elements
- Number of selected elements
- Number of image view elements
- Number of button elements
- Number of textView elements
- Number of EditText elements
- Number of DraweLayout elements

- Number of WebView elements
- Number of switch elements
- Number of recyclerView elements
- Number of UI elements

Chapter 5

Activity Classification

In the previous chapter, we have created three different feature vectors by using NLP techniques with different activity features. In this chapter, we'll apply classification steps for three different feature vectors and at the end of the chapter we will compare their results. As in the apk classification, we will use a deep neural network as a classifier.

5.1 Building a Model

5.1.1 Data Pre-processing

As a first step, a pre-processing step can be applied with having feature vectors and categories. One activity is considered as one sample and is labeled one of the 8 possible activity types. Transformation of feature vectors is done by using standardization. After the transformation, for our output variables which are activity types can be converted to the numerical values by using a label encoder. It assigns to each string a numerical variable starting from 0 and our range will be between 0 to 7. We are dividing our activity data set into two subsets. One subset is training data which will be used to fit our model. The other subset is test data which will be used to evaluate our model. In our data set %80 of data will be used for the training phase and %20 of data will be used for the testing phase.

- Training set includes 80% of the data set and has **144 samples**.
- Testing set includes 20% of the data set and has **36 samples**.

We can also use a more reliable cross-validation technique. Stratified k-Fold Cross Validation has been used in our case. We have been picking k as 10 and it is splitting the data set into 10 folds. For each iteration the k-1 fold is used for training and the remaining fold is for testing. Model will be trained and tested 10 times.

5.1.2 Evaluation Metrics

The evaluation metrics we mentioned in chapter 3.1.2 will be used which are accuracy, precision and recall. In addition to this, **confusion matrix** is going to be used. Confusion matrix provides more detailed information about the performance of the classifier. Confusion matrix shows the correct and incorrect predictions with counting the values in per class and It is a table where actual values are represented in the rows and columns for the predicted values.

5.1.3 Tuning Hyper-parameters and Selecting the Best Results

As in the APK file classification, we can tune hyper-parameters for having the best results by **grid search**. To begin with, we have tried 4 different **number of units** starting from 100 units up to 400. In a model with feature vector 1, we have reached 74.333% accuracy with 400 units. In a model with feature vector 2, we have reached the highest accuracy(73.571%) with 200 units and in a model with feature vector 3 at most we have reached 60.429% accuracy. In table 5.1 we can see the results of tuning the number of units in detail. After the number of units, the next parameter we need to tune is **activation functions**. We have decided to involve 5 different activation functions to our experiment which are sigmoid, softsign, relu, elu and leaky relu. For models with feature vector 1 and 2 we have reached the highest accuracy with the relu activation function. Feature vector 1 highest result is 73.534% and at the same time feature vector 2 highest result is 74.190%. Different from the model with feature vector 1 and model with feature vector 2, in feature vector 3 we have decided to use leakyRelu whose accuracy is 60.625%. Table 5.2 shows results of the tuning activation function of three models in detail.

The next parameters are **batch size** and **number of epochs**. We'll use mini-batch Gradient Descent and 32,64,128 batch sizes will be used. 100 and 200 epochs are going to be used in our experimentation. With bath size 62 and epochs 200 it has reached the highest accuracy in models with feature vector 1 and 2 so it has been decided to use bath size 62 and epoch 200 for models with feature vector 1 and feature vector 2.

Model with the feature vector 1 the highest accuracy is 75.619% and the model with feature vector 2 the highest accuracy is 72.953%. Batch size 32 and epochs 200 gave the highest score(59.000%) with a clear difference in the model with feature vector 3. We can see the results of tuning batch size and number of epochs in table 5.3. In addition to these parameters, we need to consider **weight initialization**. In the model with feature vector 1, Both Uniform and glorot_normal weight initializers gave the same accuracy results. It has been picked glorot_normal since it gave better recall results and also for models with feature vector 2 and 3 we have picked glorot_normal weight initializer. If needed to see results in detail, it can be examined in table 5.4 which indicates the tuning weight initialization in detail.

Our next parameter and one of the essential hyper-parameters is **optimizer**. It has been tuned to 4 different optimizers which are SGD, Adam, Adagrad and Adamax. As seen in table 5.5, for models with feature vector 1 and 2, Adamax gave the best results in accuracy, precision and recall. In a model with feature vector 3, we have reached the best accuracy result(55.476%) with Adam optimizer.

Lastly, **Learning rate** parameter has been tuned. In a model with feature vector 1, we have obtained a best result of 74.905% accuracy with 0.01 learning rate. For feature vector 2, we have reached a best result 72.095% accuracy with 0.001 learning rate and in feature vector 3, it has been decided to use 0.01 learning rate since it gave the best accuracy result(63.190%). In table 5.6 we can see the results of tuning learning rate for model with feature vector 1 using Adamax optimizer, Table 5.7 represents results of tuning learning rate for model with feature vector 2 using Adamax optimizer and lastly in table 5.8 we can see the results of tuning learning rate of the model with feature vector 3 using Adam optimizer.

Metrics	**Accuracy**	**Precision**	**Recall**
Units	*Feature Vector 1*		
100	73.429%+/-10.643%	71.979%+/-11.585%	73.750%+/-10.000%
200	72.048%+/-8.765%	70.625%+/-11.765%	71.875%+/-8.501%
300	71.524%+/-13.335%	69.208%+/-15.853%	71.250%+/-13.463%
400	**74.333%+/-11.706%**	**72.792%+/-13.365**%	**73.750%+/-11.456%**
Units	*Feature Vector 2*		
100	71.476%+/-11.065%	70.312%+/-15.346%	70.625%+/-10.843%
200	**73.571%+/-12.105**%	**71.979%+/-13.932%**	**73.125%+/-11.198%**
300	72.143%+/-8.899%	69.792%+/-10.217%	71.875%+/-8.028%
400	72.238%+/-12.461%	71.065%+/-15.125%	71.875%+/-12.885%
Units	*Feature Vector 3*		
100	54.143%+/-4.513%	45.146%+/-6.553%	53.750%+/-4.146%
200	59.667%+/-6.960%	52.333%+/-11.125%	59.375%+/-7.526%
300	**60.429%+/-7.406%**	**54.500%+/-12.903%**	**59.375%+/-7.526%**
400	58.381%+/-8.474%	50.396%+/-9.879%	46.337%+/-7.806%
	One dense layer Adagrad(lr=0.01) Activation	Batch size is 32 Kernel initializer relu	Epoch is 100 normal

Table 5.1: Results obtained from tuning number of units with using gridSearchCV

Metrics	**Accuracy**	**Precision**	**Recall**
	Feature Vector 1		
sigmoid	68.667%+/-8.645%	64.354%+/-9.211%	68.750%+/-8.385%
softsign	72.905%+/-10.187%	71.542%+/-10.087%	73.125%+/-9.703%
relu	**73.524%+/-8.935%**	**71.875%+/-11.112%**	**73.125%+/-8.861%**
elu	72.095%+/-10.271%	70.604%+/-9.412%	71.875%+/-8.949%
Leaky Relu	70.857%+/-11.272%	66.542%+/-13.974%	71.250%+/-12.247%
	Feature Vector 2		
sigmoid	70.667%+/-11.433%	67.896%+/-12.228%	70.625%+/-10.843%
softsign	71.524%+/-11.761%	71.458%+/-10.907%	71.875%+/-11.609%
relu	**74.190%+/-11.488%**	**72.292%+/-13.503%**	**73.750%+/-10.383%**
elu	70.190%+/-12.440%	70.208%+/-14.458%	71.250%+/-11.592%
Leaky Relu	72.810%+/-14.566%	70.000%+/-15.456%	69.375%+/-15.922%
	Feature Vector 3		
sigmoid	47.810%+/-9.426%	35.680%+/-11.466%	45.625%+/-9.708%
softsign	58.333%+/-6.668%	47.812%+/-9.634%	57.500%+/-6.731%
relu	57.619%+/-5.742%	49.854%+/-7.208%	56.875%+/-5.896%
elu	59.714%+/-9.123%	53.583%+/-13.059%	60.000%+/-9.354%
Leaky Relu	**60.625%+/-7.550%**	**53.521%+/-12.431%**	**60.625%+/-8.409%**
	One dense layer Adagrad(lr=0.01)	Batch size is 32 Kernel initializer	Epoch is 100 normal

Table 5.2: Results obtained from tuning activation function with using grid-SearchCV

Metrics	**Accuracy**	**Precision**	**Recall**
BS - Epochs	*Feature Vector 1*		
32 - 100	74.905%+/-8.526%	72.917%+/-10.333%	74.375%+/-9.036%
32 - 200	72.857%+/-12.311%	68.208%+/-13.919%	73.125%+/-11.875%
64 - 100	69.476%+/-8.789%	66.875%+/-12.550%	70.000%+/-9.601%
64 - 200	**75.619%+/-12.248%**	**72.083%+/-10.383%**	**75.625%+/-11.336%**
128 -100	68.048%+/-9.678%	64.271%+/-11.220%	68.125%+/-9.862%
128 -200	72.762%+/-8.826%	68.917%+/-10.016%	71.875%+/-8.028%
BS - Epochs	*Feature Vector 2*		
32 - 100	71.429%+/-11.920%	68.438%+/-13.077%	71.875%+/-11.267%
32 - 200	71.476%+/-8.604%	71.042%+/-11.838%	71.875%+/-8.501%
64 - 100	**72.952%+/-9.306%**	**71.146%+/-14.161%**	**72.500%+/-9.763%**
64 - 200	72.143%+/-11.295%	71.083%+/-13.374%	71.875%+/-10.551%
128 -100	68.667%+/-11.870%	64.625%+/-15.619%	68.750%+/-11.524%
128 -200	70.714%+/-13.499%	70.208%+/-12.185%	71.250%+/-12.562%
BS - Epochs	*Feature Vector 3*		
32 - 100	54.905%+/-8.031%	47.437%+/-9.898%	54.375% +/-7.421%
32 - 200	**59.000%+/-9.016%**	**54.417%+/-14.547%**	**59.375% +/-9.375%**
64 - 100	50.048%+/-5.351%	40.021%+/-7.873%	48.750%+/-6.731%
64 - 200	52.048%+/-9.113%	42.729%+/-9.918%	52.500%+/-9.354%
128 -100	50.619%+/-10.468%	40.899%+/-2.311%	51.875%+/-11.198%
128 -200	52.048%+/-10.041%	41.274%+/-10.101%	50.625%+/-9.036%
Parameters is	One dense layer	Activation function	ReLu
Optimizer	Adagrad(lr=0.01)	Number of units	100

Table 5.3: Results obtained from tuning batch size and number of epochs with using gridSearchCV

Metrics	**Accuracy**	**Precision**	**Recall**
	Feature Vector 1		
Uniform	75.667%+/-8.447%	74.167%+/-9.601%	75.625%+/-8.592%
Normal	69.952%+/-9.405%	68.646%+/-9.345%	70.000%+/-8.292%
Glorot Normal	**75.667%+/-8.884%**	**72.500%+/-11.016%**	**76.250% +/-8.292%**
	Feature Vector 2		
Uniform	73.476%+/-12.114%	71.458%+/-12.432%	73.750%+/-11.110%
Normal	73.524%+/-13.001%	71.042%+/-13.158%	73.350%+/-12.119%
Glorot Normal	**73.524%+/-11.653%**	**73.625%+/-14.156%**	**73.350.411%+/-11.110%**
	Feature Vector 3		
Uniform	52.125%+/-10.354%	42.604%+/-11.388%	53.125%+/-10.551%
Normal	53.429%+/-7.161%	44.292%+/-10.244%	53.125%+/-5.039%
Glorot Normal	**59.095%+/-6.634%**	**51.917%+/-7.627%**	**58.750%+/-6.960%**
Parameters	One dense layer	Batch size is 32	Epoch is 100
Optimizer	Adagrad(lr=0.01)	Number of units	200
Activation	softsign		

Table 5.4: Results obtained from tuning weight initialization with using gridSearchCV

Metrics	Accuracy	Precision	Recall
	Feature Vector 1		
SGD	50.619%+/-10.959%	43.180%+/-11.623%	49.375%+/-9.458%
Adam	70.048%+/-10.006%	67.688%+/-13.333%	70.000%+/-10.383%
Adagrad	68.667%+/-12.399%	66.542%+/-14.172%	68.125%+/-12.006%
Adamax	**70.667%+/-10.200%**	**69.042%+/-14.332%**	**70.625%+/-9.703%**
	Feature Vector 2		
SGD	68.048%+/-9.880%	68.625%+/-14.867%	68.125%+/-10.635%
Adam	71.476%+/-11.841%	68.854%+/-16.194%	70.625%+/-11.875%
Adagrad	70.857%+/-13.443%	69.479%+/-15.894%	70.635%+/-13.707%
Adamax	**73.534%+/-13.190%**	**70.208%+/-16.353%**	**72.500%+/-13.463%**
	Feature Vector 3		
SGD	34.143%+/-8.278%	22.603%+/-8.208%	33.125%+/-7.930%
Adam	**55.475%+/-6.762%**	**48.896%+/-13.078%**	**56.250%+/-6.847%**
Adagrad	54.190%+/-6.049%	45.917%+/-8.327%	53.750%+/-5.728%
Adamax	49.381%+/-6.976%	37.729%+/-9.730%	49.375%+/-9.036%
	One dense layer 100 units Activation	Batch size is 32 Kernel initializer ReLu	Epoch is 100 normal

Table 5.5: Results obtained from tuning optimizer with using gridSearchCV

Metrics	Accuracy	Precision	Recall
	Feature Vector 1		
0.001	68.714%+/-9.128%	63.396%+/-6.268%	67.500%+/-8.292%
0.01	**74.905%+/-11.123%**	**73.937%+/-10.917%**	**74.375%+/-9.862%**
0.1	65.810%+/-12.039%	66.354%+/-13.332%	64.375%+/-12.200%
	One dense layer Optimizer Adamax Activation	Batch size is 62 Kernel initializer Relu	Epoch is 200 Glorot_normal

Table 5.6: Results obtained from tuning learning rate with using gridSearchCV

Metrics	Accuracy	Precision	Recall
	Feature Vector 2		
0.001	**72.095%+/-10.984%**	**67.396%+/-11.122%**	**71.875%+/-9.375%**
0.01	69.429%+/-13.720%	67.167%+/-18.052%	68.750%+/-13.975%
0.1	69.381%+/-9.600%	64.875%+/-10.940%	67.500%+/-10.00%
	One dense layer Optimizer Adamax Activation	Batch size is 62 Kernel initializer Relu	Epoch is 200 Glorot_normal

Table 5.7: Results obtained from tuning learning rate with using gridSearchCV

Metrics	Accuracy	Precision	Recall
	Feature Vector 3		
0.001	55.542%+/-7.863%	48.312%+/-7.559%	55.000%+/-6.124%
0.01	**63.190%+/-11.553%**	**60.170%+/-14.228%**	**45.699%+/-13.305%**
0.1	52.762%+/-12.390%	45.699%+/-13.305%	51.875%+/-10.477%
	One dense layer Optimizer Adam Activation	Batch size is 32 Kernel initializer leakyRelu	Epoch is 200 Glorot_normal

Table 5.8: Results obtained from tuning learning rate with using gridSearchCV

After we have set some basic hyper-parameters, we have decided to add one more hidden layer to our models for obtaining better results. In a model with the feature vector1, we have specified 150 units of dense layer with elu activation function as a second hidden layer. Secondly, in a model with feature vector 2, 15 units of dense layer with elu activation function was added as a second hidden layer. Lastly, in the model feature vector 3, 15 units with softsign activation function were added as a second hidden layer.

5.1.4 Classification Results

In conclusion, Hyper-parameters were specified for each different three neural networks for finding their best performance. In the model that used feature vector 1, the following architecture has been implemented : As an input layer 318 nodes have been used and go towards the dense layer of 400 units using the ReLu activation function and glorot_normal is assigned as a kernel initializer. After the dense layer of 400 units goes into a dense layer with 150 units with elu activation function and glorot_normal kernel initializer. Adamax optimizer with 0.01 learning rate has been used. Lastly, the Neural network ends up with an output layer using softmax activation function with 8 nodes. Using accuracy and loss plots, the results of accuracy and loss can be visualized. As we can see in figure 5.1 in the left plot, we have reached 78% accuracy in the test data set with this architecture and also the right plot shows the model loss result. The model with feature vector 1 test loss is 1.33.

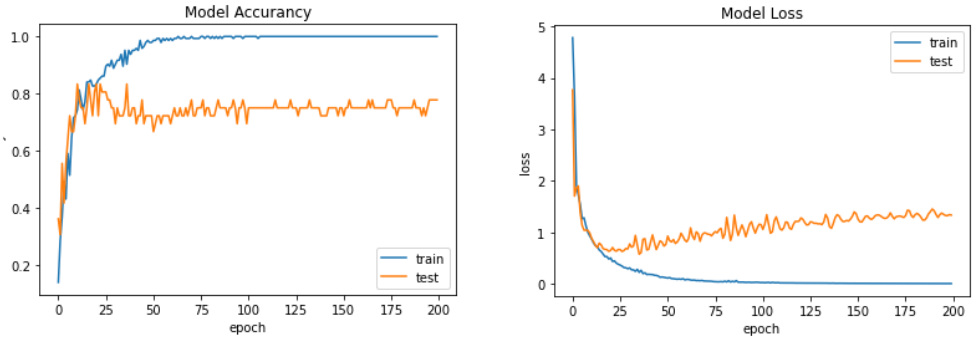

Figure 5.1: Accuracy and loss plots of model with feature vector 1

As the evaluation of activities scores, The highest scores were obtained by search, settings, splash, and login activities. The inference of these results is that activities of the highest results tend to contain similar UI elements and tend to have common resource id UI information. The worst scores were obtained for **main with 33% recall** and **map activities with 33% recall.** Since the main activities have a lot of UI components and are more complex than the other activities the model has difficulty learning. As seen by the following figure which contains the confusion matrix of the model, We have 6 predicted samples for the main activity but only 2 samples are predicted correctly. On the other hand, We have 2 errors out of 3 in the map activity. The model has trouble prediction of map activity.

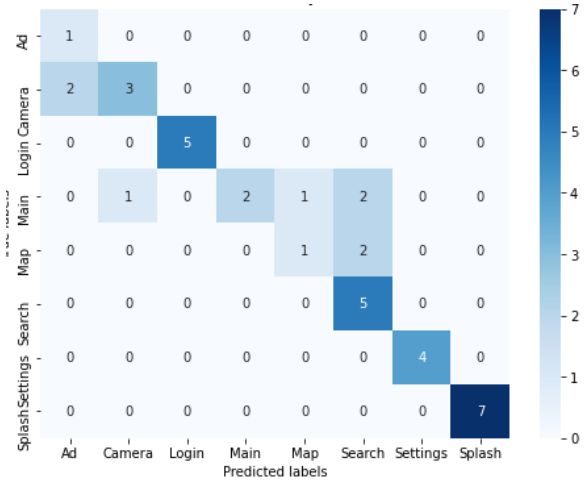

Figure 5.2: Confusion Matrix of model with feature vector 1

In our second model which uses feature vector 2, the following architecture has been implemented : As an input layer 316 nodes has been used and goes towards the dense layer of 200 units using the ReLu activation function and glorot_normal is assigned as a kernel initializer. After the dense layer of 200 units goes into a dense layer with 15 units with elu activation function and glorot_normal kernel initializer. Adamax optimizer with 0.001 learning rate has been used. Lastly, the Neural network ends up with an output layer using softmax activation function with 8 nodes. We are able to reach 83% accuracy using this model. Figure 5.2 shows a model of training and test data set accuracy and loss.

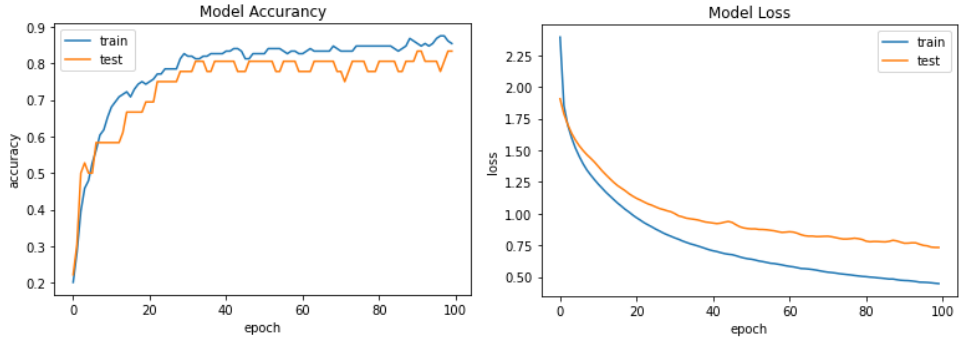

Figure 5.3: Accuracy and loss plots of model with feature vector 2

In the model with feature vector 2, The highest activity scores were obtained by search, settings, splash, and login activities similar to the model with feature vector 1. In addition to this, Map results are quite good compare to feature vector 1. However, model 2 has still problem of learning main activity but we have less errors (3 errors) then model with feature vector 1 and we have 3 errors out of 5 in the camera activity.

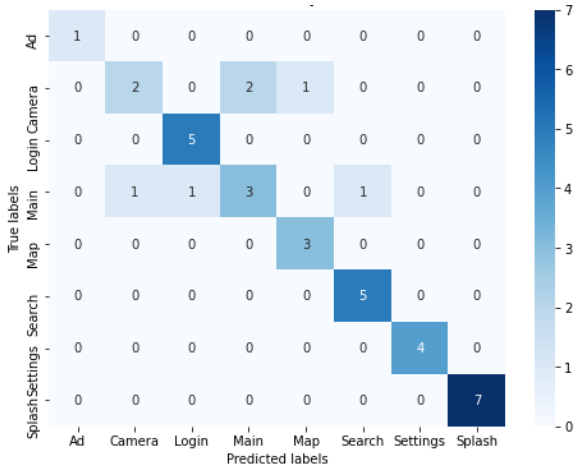

Figure 5.4: Confusion Matrix of model with feature vector 2

In the last model with feature vector 3, the following architecture has been implemented: As an input layer 18 nodes have been used and go towards the dense layer of 300 units using the Leaky ReLu activation function, and glorot_normal is assigned as a kernel initializer. After the dense layer of 300 units goes into a dense layer with 15 units with softsign activation function and glorot_normal kernel initializer. Adam optimizer with 0.001 learning rate has been used. Lastly, the Neural network ends up with an output layer using a softmax activation function with 8 nodes. We are able to reach 67% accuracy using this model. Figure 5.3 shows model accuracy and model loss. Since there is no big gap between training and test data in the accuracy plot, we can say that there is no sign of overfitting.

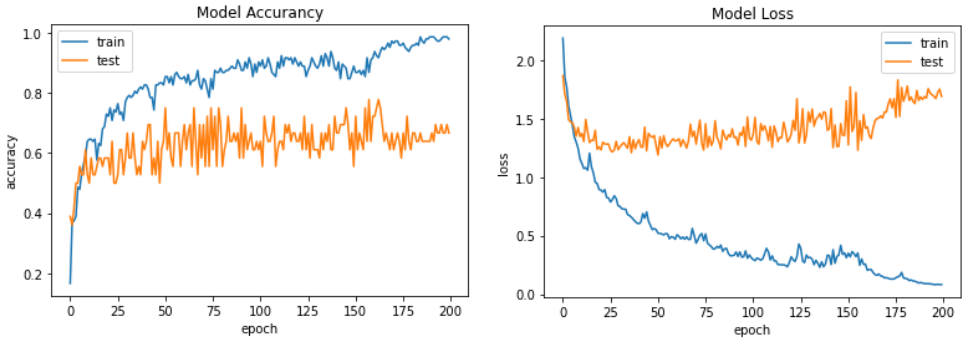

Figure 5.5: Accuracy and loss plots of model with feature vector 3

The highest activity scores were obtained by splash and login activities. Since we didn't include the semantic information to the feature vector 3, model activity learning performance is low according to the other feature vectors.

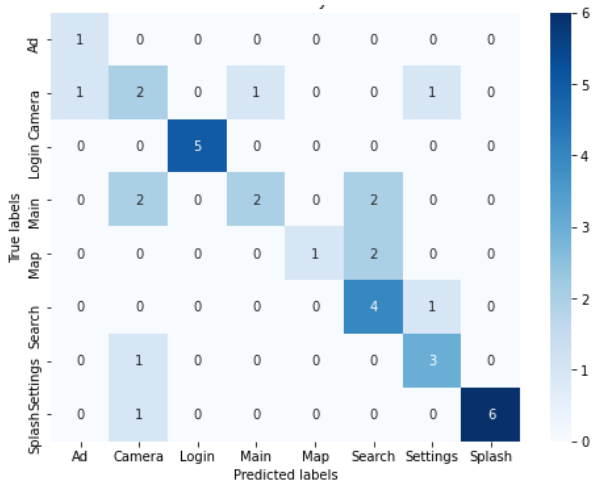

Figure 5.6: Confusion Matrix of model with feature vector 3

5.2 Comparison of Results

As seen in figure 5.7, it has reached %78 accuracy in the model with feature vector 1. The best accuracy(83%) that has been reached with the model using feature vector 2 by eliminating two features. It was decided that the number of recycler views does not have the benefit to categorize the activities since the different types of activities (main, search, setting, and map) use recycler view, and also the number of total UI elements were eliminated.

Since textual information of UI elements was not involved in the model with feature vector 3, the model performance(67%) is lower than the other 2 feature vectors as we can see in figure 5.7. To summarize, it can be said that supporting classification with NLP has a positive effect on the classification result. It can more easily detect which category activities belong to by using the NLP. On the other hand, selecting the right UI elements can improve the performance of the model.

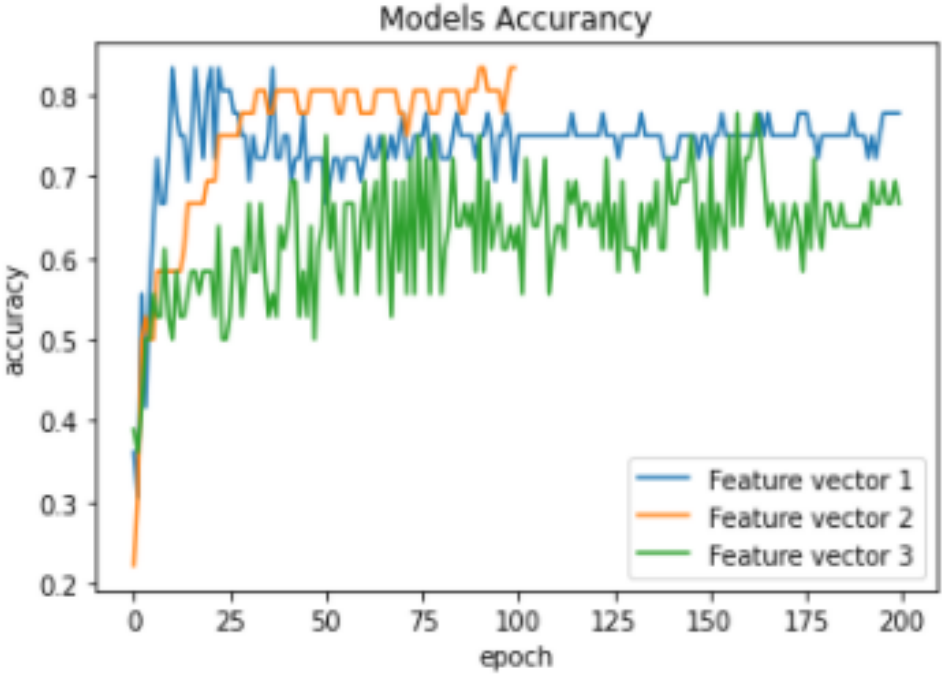

Figure 5.7: Comparison of classification results of three different models

Chapter 6

Android Testing Framework

6.1 Overview

UI testing is one of the significant tasks when an android developer implements apps. In the meantime, the developer has to deal with the staging processes. Firstly, writing a UI testing is often a time-consuming task and also the developer needs to re-write the test scripts for the same functionality in the different apps when he writes more than one app but the difference is the position of UI elements. **Our main aspects are providing a developer to write reusable, flexible, and simpler human-understandable commands.** The developer can write a single test script for a specific activity type and can use it in the different applications at the same activity types. Thanks to the classification process we can make predictions of the app category type and activity layout. Basically, our framework scans the activity layout and can find desirable UI elements then perform the specified action. In addition to this, the developer does not need to care about the graphical form and position of the UI element. As an example, in the login activity the developer just provides information that would like to type an email and password field without any other detail needed. The developer just writes a general and simple command that we want to perform the input task of an email password field and our framework finds the UI elements and performs the task. In this way, since the test scripts are simplified and general developers can re-use this test script for the same types of activities. We kept our command simpler because commands can adapt to many applications and also developers can earn more time writing the simple commands.

6.2 Appium Architecture

Appium[20] is an automated testing tool that supports testing in native apps, mobile web apps, and hybrid applications. One of the essential features is that Appium is a cross-platform testing tool which means that it can support multiple platforms such as Android, iOS, Windows using the same API. Another benefit of using Appium is that it supports multi-programming language and it is possible to write test code in python, JavaScript, Ruby, Java, C#. One of the reasons that we are using Appium in our framework is that it does not require access to reach any application source code.

Appium is based on **client-server** architecture. It is using some components that support connection which are the Appium server, Appium client libraries, and JSON Wire Protocol. To begin with, the Appium client can send a request to the server using JSON Wire protocol for initialization of the session. This request includes desired capabilities that provide control session requests. There are different capabilities such as appActivity, platformName, deviceName, appWaitDuration. As an example, if we want to work with an android device, the platform name must be set to Android ("platformName": "Android"). The Appium server is an HTTP server and one of the essential components of the architecture. Basically, it receives the client request with the JSON wire protocol and checks desired capabilities; after that it triggers the mobile driver to execute the test script by creating the session. Finally, test results return back to the client. In addition, Appium uses the UI Automator to inspect UI elements.

6.3 Specifying Framework Commands

The structure of our script language is simple and easy-readable. Our aim is to develop simple and human-understandable commands and this way test cases can apply to many other apps without the need for any changes. As we mentioned before, a developer does not have to specify location or other detailed information of UI elements.
The pre-conditions for executing the test case:

- Developer must have specified the app category type.
- Current screen associated with the specified activity type.

If these two conditions are not satisfied then the test case is associated as a FAIL. In order to execute our script, the script must be executed using the following instructions.

As a first step, a test suite has to be created which includes one or more than one test case. For creating the new test suite users must declare the type of app category which is one of the 15 categories defined in chapter 2. By specifying the app category we are defining that the test cases only execute in the specified category. The following line shows the sample declaration of the test suite and means that it can apply test cases for apps that are in the TRAVEL category.

- WHEN appcategory == TRAVEL :

After creating the test suite, the second step is to declare the test cases which include commands. Before specifying the commands, the following line must be written which is the header of the test case and includes activity type and post-condition of the test cases.

- LOGIN activity -> check in DIFFERENT condition :

Test cases must execute in the specified activity types and can only use commands included in the specified activity type and generic commands. The activity type can be one of the 7 categories mentioned in chapter 4. We have specified 8 categories in chapter 4 but splash activity is excluded. The reason is that we did not involve the splash activity in our testing framework, the splash activity just appears for at most 10 seconds and the user can not execute some operations.

The post-conditions must be specified to check if the test is passed or failed after the execution. If we specify the post-condition as SAME, we expect to be in the same activity according to the starting one. On the other hand, if the post-condition is specified as DIFFERENT, it is expected to be in a different activity type.

In the following script we can see that we have one test suite and this test suite has two test cases which are going to be executed in exactly the same order. First test cases must be executed in the login activity after the second test case executed in the main activity with the specific commands.

```
WHEN appcategory == TRAVEL :
  LOGIN activity -> check in DIFFERENT condition :
      TYPE INPUT "ayda.tanik@hotmail.com"
      TYPE PASSWORD "sample"
      CLEAR EDITTEXT FIELDS
      CLICK FORGOT PASSWORD
      END
  MAIN activity -> check in SAME condition :
        SCROLL DOWN
        SCROLL UP
        CLICK BOTTOM BAR ELEMENT 2
        END
ENDSUITE
```

After the specified header of the test cases, the commands can be determined. The developer can specify one or more commands We have mainly separated our commands into two different types which are ***activity-specific commands*** and ***generic commands***. Activity-specific commands must be specified associated with the activity type. Different commands exist for each of the 8 activity categories and these commands perform an action for specific cases. In the meantime, generic commands are common commands for all activity types and can be used for all content. In total, we have implemented 37 commands.

6.3.1 Activity Specific Commands

1. ***Login Activity***

   - TYPE INPUT "string" : Type the given string can be username or email into the associated field.
   - TYPE PASSWORD "string" : Type the given string to the password field.
   - CLICK LOGIN : Submit the form for moving the next activity.
   - CLICK FORGOT PASSWORD
   - CLICK CREATE ACCOUNT
   - CLEAR EDITTEXT FIELDS : Delete the text in all fields.
   - CLICK LOGIN WITH FACEBOOK : Proceed with the login with Facebook activity.

2. ***Main Activity***

   - SCROLL DOWN

- SCROLL UP
- SCROLL RIGHT
- SCROLL LEFT
- CLICK DRAWER NAV
- CLICK BOTTOM ACTION BAR
- CLICK BOTTOM BAR ELEMENT number

3. ***Search Activity***
    - TYPE INPUT SEARCH "string": Type the given string into the associated search field.
    - SCROLL DOWN LIST
    - SCROLL UP LIST
    - CLICK LIST INDEX number
    - SCROLL LIST INDEX RANGE number TO number
4. ***Settings Activity***
    - CLICK INDEX number
    - PRESS SWITCH IN INDEX number
    - LONG CLICK INDEX number
    - CLICK POPUP CHECKED ELEMENT INDEX number
5. ***Advertisement Activity***
    - CLICK AD CONTENT
    - CLICK DOWNLOAD
6. ***Map Activity***
    - TYPE INPUT SEARCH MAP "string": Search location on the map by entering the associated string to the field.
    - CLICK PIN INDEX number
    - SCROLL MAP DOWN
    - SCROLL MAP UP
    - SCROLL MAP RIGHT
    - SCROLL MAP LEFT
7. ***Camera Activity***
    - CLICK CAPTURE
    - CLICK GALLERY

6.3.2 Generic Commands

- PRESS ANDROID BACK BUTTON
- PRESS BACK
- PRESS CLOSE
- CLICK ELEMENT "text"

6.4 Implementation

The automated testing framework was developed using the ANTLR[21] tool which automatically produces a parse tree. We have created an interpreter by using ANTLR and also the Python language. In addition to this, the Appium server was used for capturing the layout structure of the activity and executing each command. The interpreter that we implemented consists of a **Lexer (automatedToolLexer.g4)** that takes input strings and transforms them into tokens, **Parser (automatedToolGrammar.g4)** that builds a parse tree (syntactic analysis) and **Semantic Analyzer** consist of all python codes.

6.4.1 Framework Structure

Figure 6.1 shows a general view of our automated testing framework structure. As a first step, the developer must specify the test script which includes a test suite. A test suite can include more than one test case. Each test case has a list of commands. These commands are associated with the python code.

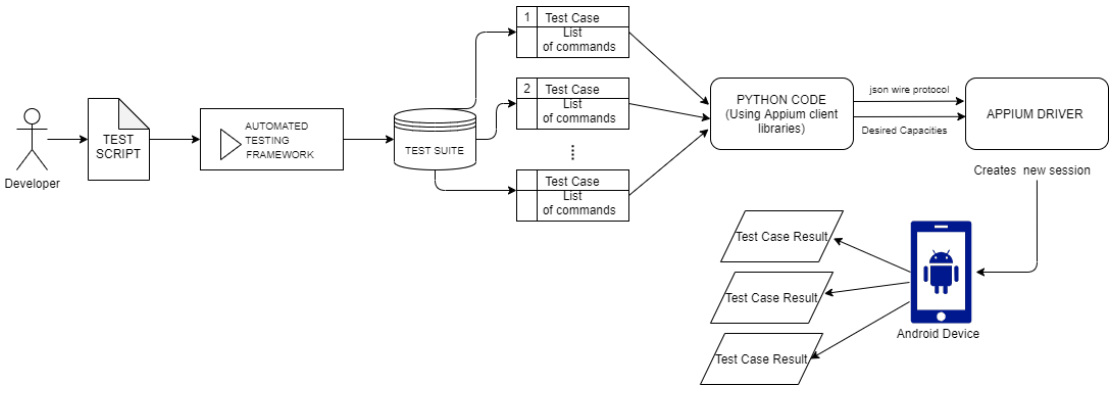

Figure 6.1: Android Automated Testing Framework Structure

While the interpreter is managing the syntax of the script, Python code performs the specified command and for performing commands our code includes appium client libraries and JSON wire protocol. Appium client sends a request to the appium driver. The new appium session starts and performs on an android device for each test case.

6.4.2 Lexical Analysis

The first step is determining the Lexical Analysis which takes the input characters and converts them into tokens. The output of the lexer is the sequence of tokens sent to the second step, which is a parser. In our implementation lexer takes all the strings from the test script and produces outputs which are called tokens. In lexer, we can define letters as fragments for providing to developers writing case-sensitive commands in scripts. After the definition of the fragment, it can be used in the words that we would like to apply to a case-sensitive string.

```
fragment A :('A'|'a');
fragment B :('B'|'e');
fragment G :('G'|'g');
fragment H :('H'|'h');
fragment Z :('Z'|'z');
```

We can use these fragments in the terminal strings for creating case-sensitive strings. In this way our interpreter accepts upper and lower characters for these strings ('WHEN','when').

- WHEN: W H E N;
- CHECK: C H E C K;
- CONDITION: C O N D I T I O N;
- SAME: S A M E;
- DIFFERENT: D I F F E R E N T;

We can use the regular expression for taking the strings ,numbers and for managing white-space. In this way, when the scanner comes across the white-space, it can skip every white-space.

```
NUMBER : [0-9][0-9]*;
QUOTEDSTRING: '"'[a-zA-Z0-9.#$@]*'"';
STRING: [0-9a-zA-Z.#$@]+;
SPACE: ( ' ' | '\t')+-> skip;
NEWLINE: ('\r'? '\n' | '\r')+ -> skip;
```

6.4.3 Syntactic Analysis

After the lexical analysis, the second step is writing the parser which takes the tokens and matches the sequence for creating a proper parse tree. Upper case strings belong to the terminal symbol and on the other hand lower case strings are associated with non-terminal symbols. Our grammar starts with the non-terminal symbol and includes more than one production rule. The production rules were created based on **Context Free Grammar .** The idea of context-free grammar is the left side of the rules which has to be non-terminal symbols replaced by the right side symbols of the rule. It can derivate right side symbols from the left side of the symbols. Terminal symbols can appear on the left side or right side of the production rule and non-terminal symbols can only appear on the right side of the production rule but to remind them that only one terminal symbol can exist on the left side of the production. In the following examples, including our production rules that we have created, we tried to create replicated production rules.

prog: testSuite COL (testcase END (ENDSUITE)*)+ EOF;
testSuite: WHEN APPCATEGORY EQ categoryName;
testcase: activitytype1 ACTIVITY ARROW CHECK IN postcondtype
CONDITION COL (commandlist1)+
commandlist1 : commandtype1str | commandtype1 |genericommand ;

The antlr [22] uses a top-down parsing strategy which is called LL(*) parser. The idea behind the LL(*) parser is to start the build parse tree from the root(top) to the leaves(bottom). The antlr generates a recursive descent parser which is allowed to predict the next production with parsing decisions. After that, it implements look-ahead DFA. The system keeps checking DFA on the input until it finds an accepting state. Figure 6.2 shows the example of the parse tree after the following script is executed.

```
1 WHEN appcategory == SHOPPING :
2   SEARCH activity -> check in DIFFERENT condition :
3       CLICK LIST INDEX 1
4       END
5 ENDSUITE
```

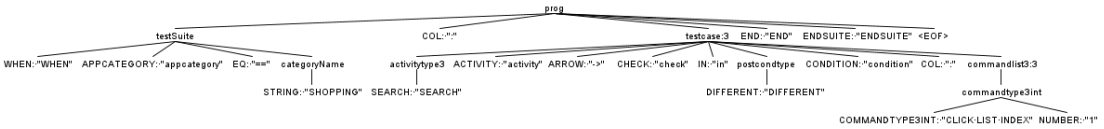

Figure 6.2: The Parse Tree after the sample script executed

6.4.4 Semantic Analysis

The last step is implementing semantic analysis. Type checking, Scope resolution are involved in the semantic analysis step. Since our script language is simple, we will not perform these operations. In this part we will focus on combining the syntax part of the script with our python code. The idea is implementing the python function for each command and this function is supposed to do the actual work. Each command is associated with one python function and this python function performs some testing and visual gestures on specified android devices.

We have proved that the same activity types can include similar design patterns and also similar UI elements. We are able to classify activities supported by Machine learning and NLP. To consider this valuable information, we need to identify UI elements on the screen. We can find desirable UI elements looking for following information:

- **Class Type:** We can eliminate useless elements by using class names. For example, let's consider search activity. In the search activity there is search editText for searching desirable elements so we can only take elements which are editText. In camera activity when we are searching the capturing button, we can decrease the options to imageButton and to imageView.

- **Attribute Values:** When we find the desirable UI elements, in order to constrict our options we can take a look at attribute values. As needed to give an example, In the login page when we would like to perform a TYPE PASSWORD "string" , we can simply search for an element whose *"isPassword"* attribute is equal to True. In the settings activity when we want to execute the PRESS SWITCH IN INDEX command, we can select and take only switch elements in this activity.

- **Textual Informations:** While developers implement the apps, they leave some clues about the apps. It can capture a lot of information using the textual information such as resource-id, contentDescription and text.

  - Resource-id : Developers give ids for the UI elements when they develop an app. The resource-id can be simple and commentator . It can be used to indicate the specified elements. When the developer involves the button for capturing the picture in the camera activity, the button can be named as "captureButton" , "takePicture" or in the login page they can specify "loginButton" or "signInButton". We can detect UI elements with specified keywords. If this keyword occurs in the resource-id of the UI element, we can take this UI element.
  - Content-Description : Another important useful information that we can obtain is content Description. Some elements include context-description. Content description must be unique.
  - Text : UI elements can include textual information and this information can be viewed by the user. In the login activity, the Forgot password button can contain these keywords : "forgot", "remember", "forgotten".

- **Location Information:** We can reach boundaries of UI elements and for this we have X and Y coordinates. For instance, in Main activity when we perform the CLICK DRAWER NAV command, we can use the location information because generally the navigation bar should be at the top of the screen.

According to this information, without knowing the UI interface it can detect elements by combining this information and performing desirable actions. To sum up, The lexer and parser handles the test scripts syntax and on the other hand python code deals with the actual work which is performing the some gestures on the android device.

Chapter 7

Evaluation of Testing Framework

In chapter 6, we have examined our android testing framework architecture in detail. In this chapter, we are going to evaluate our framework with applying some testing procedures.

7.1 Evaluation of Commands Robustness

As a first step, we can try to evaluate the robustness and adaptiveness of our commands. In chapter 6.3, we have listed the activity-specified commands available for 8 different activity categories. Our commands are based on layout attributes and textual information, but **is this information enough to detect desirable UI elements?** In order to answer this question, we have taken complex activity-specified commands and tried to execute them in different kinds of applications.

In table 7.1 we can see the results of evaluated commands. The first command is about the login activity. The 7 test cases have passed out 8, The reason that failure of the one login test cases is application was in different language. The second command is "CLICK BOTTOM BAR ELEMENT 3". It has been tried with 3 different applications and 2 out of 3 tests were passed. The reason the wine-searcher did not pass is that we are looking at the clickable and long-clickable of the elements in the bottom bar when we find the bottom bar element. Since the current fragment is not clickable it is skipping the one fragment in the bottom bar then the framework clicks the next bottom element instead of the desirable one. Our third command which is typing input to the search editText was executed very well and all the test cases have passed. In the "CLICK INDEX 3" command 3 out of 5 commands have passed. Since the 2 apps use unusual structure instead of RecyclerView, ListView and ScrollView, it can not find the list.

To sum up, we have tested ***7 different commands*** for different applications. Most of the commands were perfectly executed and the reason for the failure of commands is unusual or different structural behaviors for the specified activity or because of the different language.

Command Explation	App Name	Pass	Fail	Notes
TYPE INPUT "sample@hotmail.com" TYPE PASSWORD "aaa"	Zalando	X		/
	IMDB		X	Text language
	Trip.com	X		/
	Zara	X		/
	Nespresso	X		/
	Ebay	X		/
	HM	X		/
	Wine-searcher	X		/
CLICK BOTTOM BAR ELEMENT 3	Zara	X		/
	WineSearcher		X	Clicking the wrong bottom bar element
	GoMeetings	X		/
TYPE INPUT SEARCH "element"	Wish	X		/
	IMDB	X		/
	Ebay	X		/
	Takeway	X		/
	PullAndBear	X		/
PRESS SWITCH IN INDEX 4	GoMeetings	X		/
	Vivino	X		/
	Adidas	X		/
CLICK INDEX 3	Zaful		X	Unusual pattern not used RecyclerView
	Ebay	X		/
	Glassdoor		X	Using framelayout
	Wish	X		/
	PrimeVideo	X		/
TYPE INPUT SEARCH MAP "torino"	Lidl plus	X		/
	HM	X		/
	Nespresso	X		/
CLICK POPUP CHECKED ELEMENT INDEX 1	ClearScanner	X		/
	Flixbus	X		/
	Vivino	X		/

Table 7.1: Evaluation of Commands

7.2 Evaluation of Scripts with Functional Testing

Functional testing is based on black box technique which is without knowing the source code of the program. Functional testing aims to check system functionality with the given output and there are two outputs which are expected and actual. In this way, we can verify the actual result with the expected result. Using our testing framework, developers can write test scripts to check if the system functionality is working properly or not. We have created adaptable test scenarios to evaluate our framework. ***15 test scripts*** can be applied to the 8 types of activities. Table 7.1 shows results of test scripts when we applied the functional test. The appendix section includes the corresponding test script. On the other hand, we can observe that the same script can be adapted for the same type of activity. In this way, we can also prove consistency and re-usability of our scripts.

In conclusion, our testing framework benefits, test scripts can be easily implemented with written human-understandable commands by developers and new developers that involve the project can easily understand already written scripts. In UiAutomator and Appium developers need to write more lines of codes according to our framework. The android testing framework reduces the time needed for writing UI testing with simple commands. Lastly, we have proved that our test cases can be used in similar activity types in different applications. Also, developers are able to use the same test suite in the same type of application categories.

Activity	**Script**	**App Name**	**Expected Output**	**Actual Output**	**Commands**
Login Activity	script 1	Nespresso	Passed	Passed	Login with wrong credentials
	script 1	Bershka	Passed	Passed	Login with wrong credentials
	script 1	Winesearcher	Passed	Passed	Login with wrong credentials
	script 2	Tripadvisor	Passed	Passed	Click create account / facebook
	script 2	Zara	Passed	Passed	Click create account / facebook
	script 3	PullBear	Passed	Passed	Click login with facebook
	script 3	Bershka	Passed	Passed	Click login with facebook
Main Activity	script 4	Wish	Passed	Passed	Scroll up-down
	script 4	IMDB	Passed	Passed	Scroll up-down
	script 4	Hotelscom	Passed	Passed	Scroll up-down
	script 5	Zara	Passed	Passed	Scroll left-right
	script 5	Nespresso	Passed	Passed	Scroll left-right
	script 6	Wish	Passed	Passed	Click nav bar
	script 6	HM	Passed	Passed	Click nav bar
	script 7	Lidl	Passed	**Not Passed**	Click bottom bar
	script 7	WineSearcher	Passed	**Not Passed**	Click bottom bar
	script 7	GoMeetings	Passed	Passed	Click bottom bar
Camera Activity	script 8	PdfScanner	Passed	Passed	Scroll up-down
	script 8	WineSearcher	Passed	Passed	Click capture/gallery
	script 8	Wish	Passed	Passed	Click capture/gallery
Map Activity	script 9	TripAdvisor	Passed	Passed	Scroll up-down
	script 9	HomeToGo	Passed	Passed	Scroll up-down
	script 9	KAYAK	Passed	Passed	Scroll up-down
	script 10	HM	Passed	Passed	Click index in map
Search Activity	script 11	IMDB	Passed	Passed	Scroll up-down
	script 11	WineSearcher	Passed	Passed	Search in list
	script 12	WineSearcher	Passed	Passed	Scroll list
Settings Activity	script 11	IMDB	Passed	Passed	Scroll up-down
	script 13	Vivino	Passed	Passed	Press switch
	script 14	Vivino	Passed	Passed	Check Popup element
Ad Activity	script 15	Translator	Passed	Passed	Click ad

Table 7.2: Functional Evaluation of Test Scripts

Chapter 8

Conclusions and Future Work

To summarize, Our work consist of three main part. In the first part we have implemented apk classification with three different feature vectors. Our result is quite well and acceptable, different feature vectors can be created using different sentence embedding vectors for improve the accuracy of the model and also different classifiers can be used. In the second step which is the activity classification step, we have collected the activity samples in different activity types and classified this activities. As a future work, it can be added more samples and more activity types. Also, more feature vector can be generated. Finally, in the last step it has been developed an android automated testing framework providing reusable, flexible, and human-understandable commands. Our framework provides users the ability to write simple test scripts and this test script can use in the same kind of category and activity apps. In this way, our automated tool provides user re-usability and saving time. The framework that we developed is improvable and extendable. More commands can be added to the *automatedToolGrammar.g4* file in future development. Since the appium server is slow to improve the performance of the framework, another supporting tool different than Appium can be used.

Appendix A

Automated Test Framework Scripts

Script 1

```
1  LOGIN activity -> check in SAME condition :
2    TYPE INPUT "atanik@hotmail.com"
3    TYPE PASSWORD "aaa"
4    CLICK LOGIN
5    END
```

Script 2

```
1  LOGIN activity -> check IN SAME condition :
2    CLICK CREATE ACCOUNT
3    PRESS ANDROID BACK BUTTON
4    END
5  LOGIN activity -> check IN SAME condition :
6   CLICK FORGOT PASSWORD
7   END
```

Script 3

```
1  LOGIN activity -> check IN DIFFERENT condition :
2    TYPE INPUT "atanik95@gmail.com"
3    TYPE PASSWORD "bbbb"
4    CLEAR EDITTEXT FIELDS
5    CLICK LOGIN WITH FACEBOOK
6    END
```

Script 4

```
MAIN activity -> check IN SAME condition :
  SCROLL DOWN
  SCROLL UP
  END
```

Script 5

```
MAIN activity -> check IN SAME condition :
  SCROLL RIGHT
  SCROLL LEFT
  END
```

Script 6

```
MAIN activity -> check IN SAME condition :
  CLICK DRAWER NAV
  END
```

Script 7

```
MAIN activity -> check IN SAME condition :
  CLICK BOTTOM BAR ELEMENT 2
  END
```

Script 8

```
CAMERA activity -> check in SAME condition :
   CLICK CAPTURE
   END
CAMERA activity -> check in SAME condition :
   CLICK GALLERY
   END
```

Script 9

```
MAP activity -> check in SAME condition :
      SCROLL MAP DOWN
      SCROLL MAP UP
      END
  MAP activity -> check in SAME condition :
```

```
6       CLICK PIN INDEX 1
7       END
```

Script 10

```
1 MAP activity -> check in SAME condition :
2       TYPE INPUT SEARCH MAP "torino"
3       CLICK PIN INDEX 1
4       END
```

Script 11

```
1   SEARCH activity -> check in DIFFERENT condition :
2       TYPE INPUT SEARCH "chianti"
3       END
4   SEARCH activity -> check in DIFFERENT condition :
5       CLICK LIST INDEX 2
6       END
```

Script 12

```
1   SEARCH activity -> check in SAME condition :
2       SCROLL LIST INDEX RANGE 4 TO 2
3       SCROLL LIST INDEX RANGE 2 TO 4
4       END
```

Script 13

```
1   SETTINGS activity -> check in SAME condition :
2       PRESS SWITCH IN INDEX 4
3       END
```

Script 14

```
1   SETTINGS activity -> check in SAME condition :
2       CLICK INDEX 1
3       CLICK POPUP CHECKED ELEMENT INDEX 2
4       END
```

Script 15

```
1   ADVERTISEMENT activity -> check in DIFFERENT condition :
```

```
2       CLICK AD CONTENT"
3         END
```

Bibliography

[1] Statista. *Mobile operating systems' market share worldwide from December 2009 to December 2020*. 2021. URL: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (cit. on p. 6).

[2] statista. *Number of available applications in the Google Play Store from December 2009 to December 2020*. 2021. URL: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ (cit. on p. 6).

[3] *APKPure*. URL: https://apkpure.com/ (cit. on p. 7).

[4] *APKMirror*. URL: https://www.apkmirror.com/ (cit. on p. 7).

[5] *Apktool*. URL: https://ibotpeaches.github.io/Apktool/ (cit. on p. 7).

[6] *Android Manifest File*. URL: https://developer.android.com/guide/topics/manifest/manifest-intro (cit. on p. 8).

[7] Myeonggeon Lee Seong Je Cho Changha Hwang Masoud Reyhani Hamedani Dongjin Shin. «An Effective Method to Classify Android Applications by Applying Deep Neural Networks to Comprehensive Features». In: (May 2018) (cit. on p. 9).

[8] Christopher D. Manning Jeffrey Pennington Richard Socher. «GloVe: Global Vectors for Word Representation». In: () (cit. on p. 14).

[9] URL: https://developer.android.com/reference/android/Manifest.permission (cit. on p. 14).

[10] URL: https://developer.android.com/reference/classes (cit. on p. 14).

[11] A. Joulin T. Mikolov E. Grave P. Bojanowski C. Puhrsch. «Advances in Pre-Training Distributed Word Representations». In: (Dec. 2017) (cit. on p. 16).

[12] Piotr Bojanowski Edouard Grave Armand Joulin Tomas Mikolov. «Enriching Word Vectors with Subword Information». In: (June 2017) (cit. on p. 16).

[13] Ming-Wei Chang Jacob Devlin. «Open Sourcing BERT: State-of-the-Art Pre-training for Natural Language Processing». In: (Nov. 2018) (cit. on p. 17).

[14] Lee Kristina Toutanova Jacob Devlin Ming-Wei Chang Kenton. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: (May 2019) (cit. on p. 17).

[15] *Neural networks and their application to textile technology*. URL: https://www.sciencedirect.com/topics/engineering/sigmoid-function (cit. on p. 23).

[16] URL: https://www.researchgate.net/figure/Illustration-of-output-of-ELU-vs-ReLU-vs-Leaky-ReLU-function-with-varying-input-values_fig8_334389306/actions#reference (cit. on p. 24).

[17] URL: https://paperswithcode.com/method/adagrad (cit. on p. 27).

[18] Odaya Kardashov Ariel Rosenfeld and Orel Zang. «Automation of Android Applications Testing Using Machine Learning Activities Classification». In: (Sept. 2017) (cit. on pp. 41, 43).

[19] Statista Research Department. *Most popular Google Play app categories 2020*. May 2021. URL: https://www.statista.com/statistics/279286/google-play-android-app-categories/ (cit. on p. 42).

[20] URL: http://appium.io/docs/en/about-appium/intro/?lang=en (cit. on p. 64).

[21] Terence Parr Jean Bovet. «ANTLRWorks: An ANTLR Grammar Development Environment». In: (July 2007) (cit. on p. 68).

[22] Kathleen S. Fisher Terence Parr. «The Foundation of the ANTLR Parser Generator». In: (2011) (cit. on p. 70).