Machine Learning Methods for Adaptive Test Case Generation for Android Activities

BY

ARTURO CARDONE
B.S., Politecnico di Torino, Turin, Italy, September 2017

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

Defense Committee:

Ugo Buy, Chair and Advisor
Elena Zheleva
Maurizio Morisio, Politecnico di Torino

To my Family, who helped me transform into reality what up to a couple of years ago seemed just fantasy.

ACKNOWLEDGMENTS

There are many special people who contributed to the success of my work and experience at UIC. First, I would like to thank Prof. Ugo Buy, who as advisor was always attentive to my needs, and was kind enough to share his valuable experience through proposals and solutions to each of my problems. Next, I am very thankful to Prof. Maurizio Morisio, who introduced me to the world of software engineering through his Politecnico courses, and has always been helpful when needed. Last but not least, I must mention Prof. Elena Zheleva, who never hesitated to show her availability in taking part in the committee.

Then, I am also grateful for having found the Tailor Family, who made Chicago feel like home. My dear friends Gabriele, Riccardo, Tommaso, Alessio and Federico were always on the piece, never missing an opportunity to laugh, help and live adventures. Without them, this journey would've not been shaped in the same way, even though this is just the beginning.

I also want to thank the Rafiki's Squad, which was part of an incredible experience across all the US, from mountain tops down to the Pacific ocean and through ghost towns.

A special thanks goes to my parents, Erica, grandparents and Wooper, who helped me in every imaginable way, and demonstrated that distance is just a number.

AC

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF TABLES (continued)

TABLE **PAGE**

LIST OF FIGURES

LIST OF FIGURES (continued)

FIGURE **PAGE**

SUMMARY

In the following Thesis work, we'll illustrate both the design and implementation of a testing framework for Android applications, which is able to adapt its execution according to the type of app under examination.

The system is very modular, and as such can be divided into four phases. First, a logic model of the whole user interface is automatically built by traversing the screens' layout, while also performing a UI stress test. Then, thanks to machine learning algorithms, we are able to classify the entire application into a category, and the same is done with the app's activities. In this way, we are able to gain some insights about the expected structure and behavior of each screen, allowing us to perform the last phase, which is the execution of test scripts written in a specifically-tailored custom language. The user has to write a suite of test cases, where each one is fired upon encountering a specific category of activity. The suite consists of a succession of adaptive commands which are able to interpret the current screen layout to perform an action with a given semantic value.

In this way, the user has just to write a single test script that can be recycled on many other instances, without needing to rewrite it to conform to new or modified UI layouts.

CHAPTER 1

INTRODUCTION

1.1 Scope and Goals of this Thesis

Performing manual UI and functional testing is a tedious and costly process that might even result into faults determined by human errors in writing scripts. When a full suite has been composed, small changes in the user interface or in the functionalities of the program imply re-designing a great number of test cases just to accomplish regression testing.

Android applications have to follow some standard patterns in both their design and functional structures that might be exploited in order to identify some expected behaviour. Additionally, semantically similar apps can be grouped into app categories, and structurally similar screens can be clustered in activity categories, yielding an even more precise guess of the behavior of the program under examination.

Our work tries to build a testing framework for Android applications capable of allowing the user to write extremely simple test scripts producing robust and adaptive tests that can be executed on a great number of apps without any kind of manual re-adaptation. This is made possible by the adoption of machine learning algorithms used to classify every activity that is met, allowing us to follow the typical patterns of that screen category to identify specific widgets or information. In contrast with other Android testing scripting tools such as *Appium* [1], which require the developer to identify IDs and classes of every UI element that he needs

to interact with, in our framework app-independent commands can be used without having to investigate every screen's layout structure.

1.2 Background

Android applications are event driven, and the purpose of their tests should be to exercise some specific behavior through both system events and user interactions (touch events, scrolls, text inputs and such). We can gather the principal testing methodologies into three big families:

- ***Random Exploration Strategy***: As the name itself suggests, random independent UI events are generated and applied to the application under examination in hope of finding some sort of faults. The advantage of this method is that we can generate a lot of 'test cases' with very little effort, making it suitable for *stress testing*. The main drawbacks are that we cannot generate targeted inputs; the tool is not aware of how much behavior of the app has already been explored leading to redundant tests that are not useful; and the absence of a precise termination criterion makes it necessary to set an exploration timeout. Some examples are Dynodroid [2], [3] and Cadage [4].
- ***Model-Based Exploration Strategy***: In a crawler-fashion, a symbolic model of the app is systematically generated and explored to investigate its behavior. We can think of the model as a finite state machine where states are screens and transitions are events: all of those are typically generated in a dynamic way, stopping when all possible routes lead to already explored states. The strong point of this approach is the possibility of reaching full UI coverage of the app without too much redundancy, but on the other hand, internal

changes outside of the GUI might not be registered. The main approaches following this methodology are MobiGuitar [5] and GuiRipper [6].

- ***Systematic Exploration Strategy***: This category of strategies refers to a family of more sophisticated techniques (such as genetic algorithms) that are used to guide the exploration towards previously uncovered code. The aim is to provide a sequence of inputs which might reveal an undesired behavior. Compared to both random and model-based strategies, this one is able to potentially achieve greater coverage and to target more dangerous areas of code of the application. The limitation lies in the scalability of the chosen algorithm. The best example is EvoDroid [7], adopting an Evolutionary algorithm.

In our case, even though we put the basis of our framework on a model-based strategy, we still follow a custom approach that will be *Black Box* (meaning that won't require any access to the app's original source code, in contrast with *White Box* approaches, which require access to the source code).

1.3 Related Work

Even though no other research work tried to develop a framework that is similar to ours in every step, some others explored the various modular phases of which it is composed.

Regarding the *State Graph Modeling phase*, the already cited MobiGuitar [5] also builds a state-machine model where each state is a specific state of the app's screen, and each transition is a UI event. After having created this graph in 'breadth-first' fashion, test cases are generated in such a way that each test is a path starting from a given initial state, and crossing one

out of every possible pair of in-coming/out-coming edges for every node. Many other similar model-based approaches can be found, but MobiGuitar can be considered the most influential.

The *Application Classification* process has been investigated adopting many different approaches: some of them require access to the full original source code of the app, such as Lacta [8], which uses information retrieval techniques to extract identifiers from the Java code, and uses them to identify the app's category, in the hope they are hiding some semantic meaning. ClassifyDroid [9] decompiles the app's code and counts the number of occurrences for every possible Android API function call that has been made in order to find its category using a Multinomial Naïve Bayes classifier. AndroClass [10] uses again decompiled code in order to retrieve the list of called API methods, plus some information about the app's intents and hard-coded strings in order to perform classification using a set of Machine Learning algorithms.

For what concerns the *Activity Classification* process, which can be considered the core of our approach, it has not been extensively explored yet: we can identify the main attempts in the research by A. Rosenfeld et al. [11], which retrieves the list of swipeable, clickable and edittext elements for every screen and uses it to perform classification through a set of classifiers, and in AppFlow [12], which is a bit more complex system: it converts the entire screen layout into a single string, and uses a screenshot and Optical Character Recognition techniques to identify useful text that might be hidden in banners or ImageButtons. All of this meta-information is then used in the classification process.

Finally, regarding the *Test Scripting* phase, the only direct comparison can be made with the just cited AppFlow, which allows the user to write test scripts for scenarios of just shopping

and news apps, where conditions with some semantic values can be imposed (such as the number of elements in the shopping cart).

1.4 Research Environment

On the mobile side, the majority of our work has been developed using *Android 9 Pie* (API level 28) running on both emulated devices (created via Android Virtual Device Manager) and a real Sony Xperia XZ2. However, some lower level APIs have been tried as well without causing any issues or harm to the presented results. When some app-development work had to be done (as for injecting faults in the programs), the Android Studio 3.5 IDE was adopted.

Concerning the development of the testing framework itself, we used the Python programming language for the majority of the needed tasks, with the help of hundreds of libraries, among which we note Keras[1] and Scikit-Learn[2] for machine learning purposes, and AndroidViewClient[3], which is a very useful Python interface for interacting with the Android Debugging Bridge (ADB).

1.5 Framework Architecture and Thesis Organization

The structure of our system is very modular, and as such be distinguished into four distinct phases:

---

[1]Keras - https://keras.io/

[2]Scikit-Learn - https://scikit-learn.org

[3]AndroidViewClient - https://github.com/dtmilano/AndroidViewClient

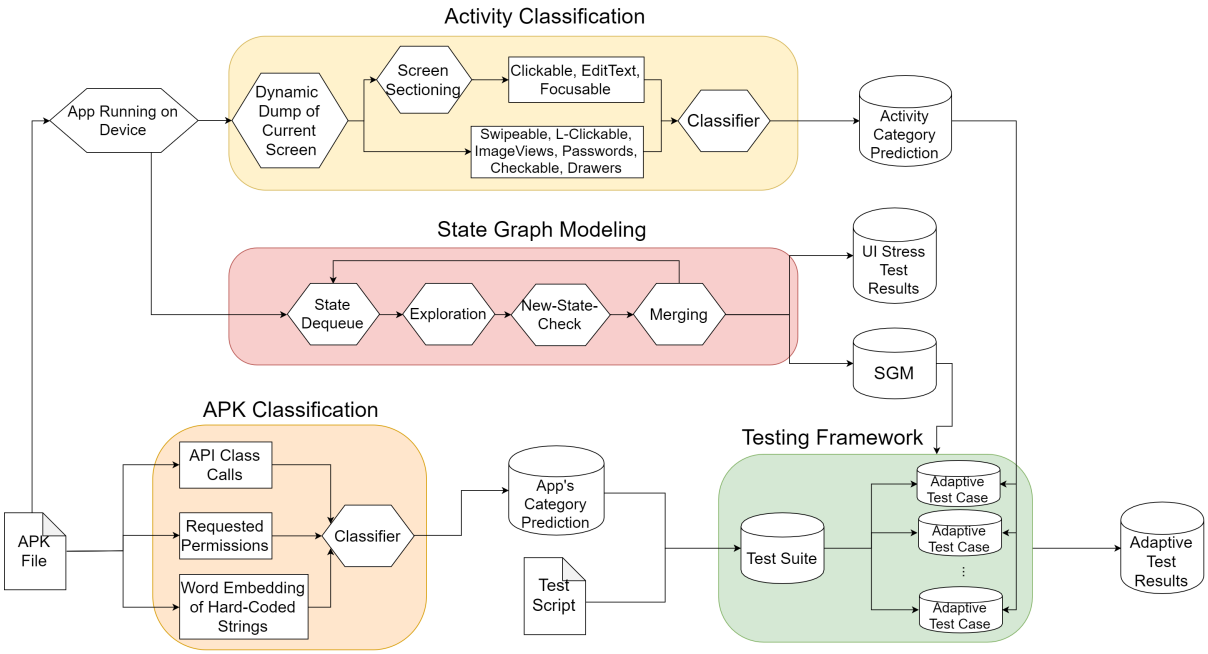


Figure 1: Flowchart summarizing the main steps of our Testing Framework.

- ***State Graph Modeling***: The process of forming a finite state machine logic model of the application, useful both as a UI stress test and as a functional map for the final test execution of the user's scripts. This is explained in Chapter 2.

- ***APK Classification***: This phase analyses the APK file of the application under test, retrieving useful and symbolic information that can be used to classify the app into a specific category. Those results are then used in the testing phase to understand when to execute a given test suite. This is discussed in detail in Chapter 3.

- ***Activity Classification***: The dynamic layout structure of every activity is analyzed to understand if it might be respecting the usual pattern of one of the possible activity categories. This information is used to 'predict' where to find specific UI widgets, and how to achieve specific semantic actions that are typical of that type of screen. Also, we can fire test cases specific for each class of activity. This process is illustrated in Chapter 4.

- ***Testing Framework***: This is where all the previous phases come together, where the user/developer has to write a test script for a suite that will depend on a particular application category, composed of test cases tailored specifically for an activity class (meaning that those tests are fired only when that type of screen category is met in the State Graph Model - SGM). The commands that can be used in those tests are special in that they can adapt to execute on many different apps without modifications to their code; also, the syntax is simple enough so that it's very simple to translate a set of functional or UI Specifications into a list of commands composing a test. More details can be found in Chapter 5.

After having explained the whole system, we will try to evaluate the robustness of the testing framework in Chapter 6. In Figure 1 we built a simple flowchart representing the steps that we've just presented together with the main interactions among them.

CHAPTER 2

STATE GRAPH MODELING

2.1 Motivation

An essential step in building this testing framework consists in being able to build a reliable functional and logical map of the application. This can be used to preventively know which screen widgets take us from one screen to the other, and it is also very useful to be conscious in every moment in which part of the app we're currently located.

Thus, building automatically this graph is equivalent to drawing a ***Formal Model as a Finite State Machine*** (in opposition to a *Semiformal Model*, such as class and object diagrams), as the application can only be in one of a finite set of states at a given time. As illustrated by Pezzè and Young in [13], usually we would have a hand-crafted diagram derived from a set of Specifications, representing only the most important states and considering all the others as '*don't care*' or '*error*' situations. In our case instead, the model is built automatically by exploring all the possible transitions out of a state, meaning exploring all possible UI elements in a given screen state of the application. Thus, potentially all states are explored, not only the most important ones.

The construction of this graph is very important specifically for the functional testing phase, as there we need to find a systematic way to explore all sections of the application in order to understand when to fire the execution of specific test cases. Having a Graph Model to follow, we

can test some screen states against the app's specifications inferring immediately the feasibility of the post-condition by the presence (or absence) of transitions bringing us to the desired state. This means that we can preventively know whether a resulting condition required by a test can be obtained without actually performing the whole test. Also, we can avoid testing the same state multiple times. If we had just tracked activities, we would've lost the concept of screen state, testing just one out tens per activity; and if we didn't have a state model at all, we would end up testing the same states over and over again.

A recurring scenario in which we'll often find ourselves and which proves the value of this graph as a 'logical map' is the following: we have a specific screen that we want to reach but that we have left for some reason, and we cannot be sure that just by reversing the input sequence (through 'back button' presses) that took us out of it will bring us to the desired state, as this is a directed graph. The only solution is to keep track for every node of the list of edges that were traversed from an arbitrary initial node of the graph up to the current one. By following this procedure, we can be certain of the sequence of inputs necessary to reach the desired screen state.

2.2 Graph Representation

Our SGM is actually a ***Directed Graph***, meaning that we have nodes connected by edges which can only be traveled in a specified direction. This is so because we are symbolizing how interacting with the UI on-screen elements of an application can bring us to a new screen, but the same element does not necessarily take us back to the previous state.

Just as in any other graph, the core elements that we need to represent are just two:

- ***Nodes***: each node is representing the *State of an Activity*. There is a big difference between representing an *Activity* or its *State*: in the first case, we are just worried about keeping track of the structure of an application's screen in terms of its layout (e.g. hierarchy of RelativeLayouts, LinearLayouts, and their respective widgets such as Buttons or EditTexts), while in the former we also take into consideration the set of attributes associated to each UI element and their current values: this means that we're tracking whether a certain button is enabled, or a toggle has been switched.

  In the context of our graph, this implies that we're tracking the same screens multiple times, each with at least one different attribute. Thus, we're able not only to preventively infer how to reach a certain screen, but also its specific state flavors. The big advantage is that thanks to this representation, we're able to execute tests on specific states of an activity, making sure that a certain specification or functionality performs correctly in a variety of situations.

  As already stated, *each node contains the full structural hierarchy of layouts for the respective activity, plus the values for all its available attributes.* Other than this, we also save the link to the predecessor state, and the list of all nodes that can be reached from here. Most importantly, we also record what we call the node's *Roadmap*: that is the list of all edges that we need to traverse in order to reach this node starting from the initial node of the graph (which will most likely be the first screen showed to the user when the application is opened). Thus, given a node, we will always be able to reach it on the Android device from the start-up screen.

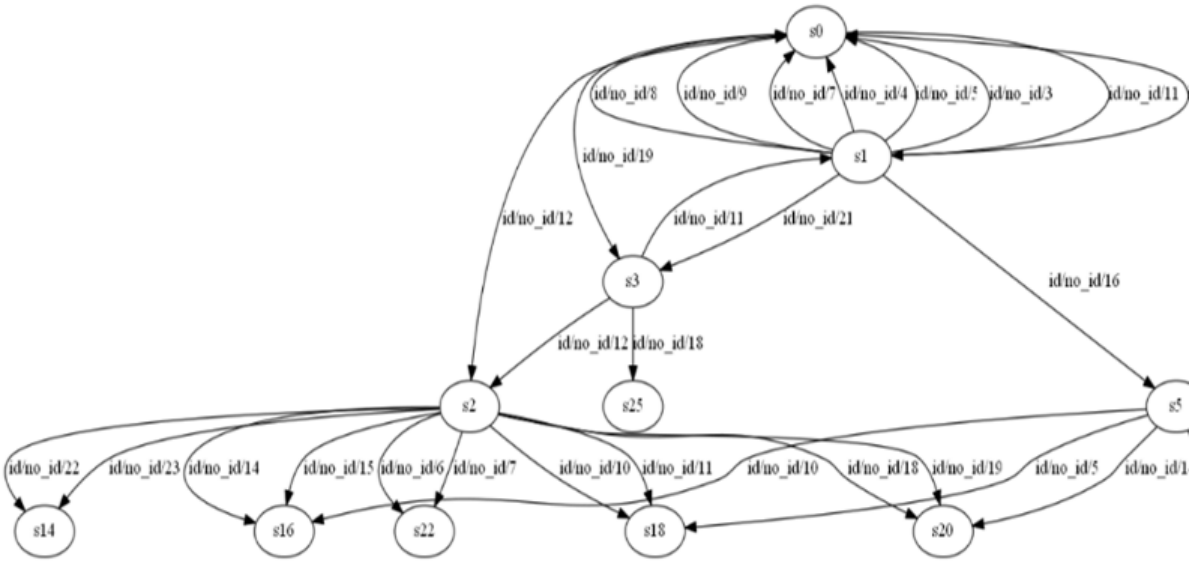


Figure 2: Sample graphical visualization of a section of the State Graph Model built from the Android stock messaging App.

- ***Edges***: in our representation, each Edge corresponds to a specific UI element the user should select in order to reach a new screen state. As every widget is represented by a multitude of attributes, we had to choose one whose value could be unique at least among the same activity: we chose the '*UniqueID*', which is an integer identifying each UI element of an activity. This value is allowed to be repeated in different screens, but we just care for it to be unique among the attributes of the same one, as we use it to identify the specific widget (edge) that will bring us to a new screen state.

In Figure 2 we can see an example of State Graph Model that was produced by crawling the stock Android messaging app. State `s0` indicates the starting state (the first screen showed up at the application's start-up): all the arrows that are going towards it are symbolizing ways that the system found to go back to the home screen from other sections of the app. An example of what has been said about activity states can be seen in state `s2`: that is the same activity as `s0`, but with an opened Spinner (drop-down) selection menu; from there, we can notice transitions going towards 5 new nodes that represent all the possible options screens reachable from that menu. If we're very careful, we can also note that we have two edges per node in this case; that is because the system learnt that it could reach those sections either by pressing the layout composing the corresponding Buttons in the Menu, or simply by tapping the TextView inside of them.

2.3 Graph Building Procedure

The actual SGM is built in breadth first search fashion using a FIFO (First In, First Out) queue to store the set of states that we've discovered, but not yet explored. The entire construction process consists of iterations that can be outlined in 4 steps:

1. ***Dequeue Phase***: The first available un-explored state is dequeued from the FIFO queue and thanks to its *Roadmap* (list of UI elements/edges to traverse to reach it) we reproduce this state on the actual Android device starting from State `s0` (the start-up screen).
2. ***Exploring Phase***: Once we've reproduced the state to explore, we dump its content to access the layout structure: we save the list of its attribute values and we try to press one by one all the available widgets. We interact with both clickable and non-clickable

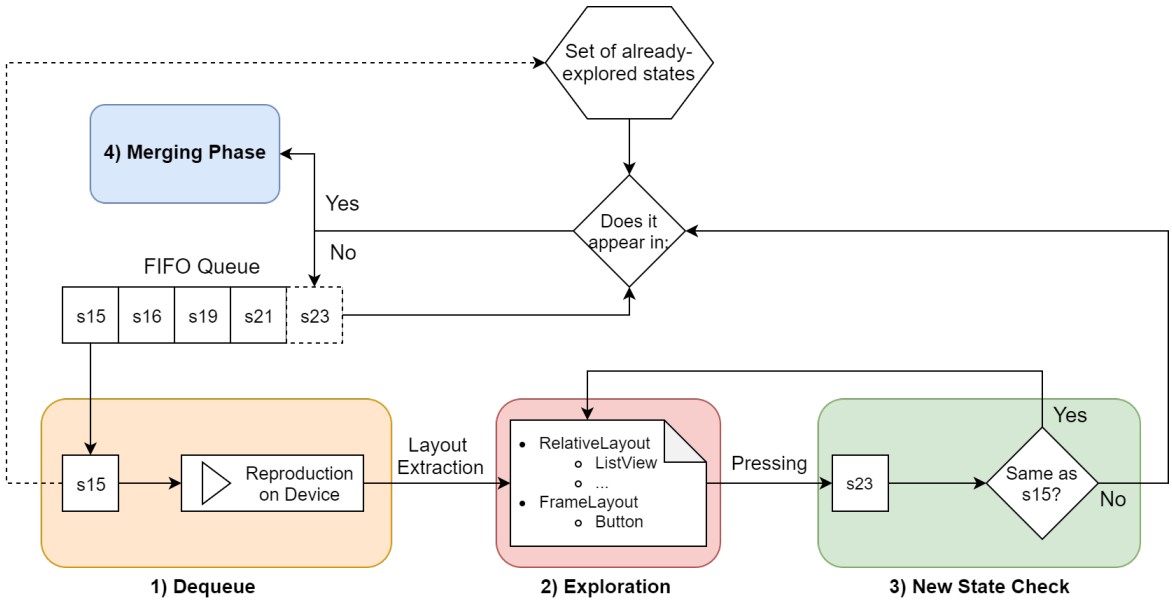


Figure 3: Diagram showing the main phases of the State Graph Model building procedure.

elements, as we might find unwanted behaviours already in this stage of the process (in a *UI Testing fashion*).

3. ***New-State-Check Phase***: After an element has been interacted with, we temporarily store the state of the activity and we check whether the resulting screen coincides with the original one: if so, then we're still in the same state and thus we don't need to do anything. However, if some differences are found, then this can potentially be a new state, so we append it to the end of the FIFO queue.

4. ***Merging Phase***: An iteration of the algorithm consists of trying to interact with all the possible UI elements belonging to a Popped State. Once an iteration has been completed,

we do two types of check. First, we check whether some of the states inserted during this last iteration into the queue are equivalent (this might happen if we end up in the same state starting from two separate activities, or by clicking two different buttons), and if so, we just keep one version of them. Second, we do a similar check with the already-explored states: if a match is found, then we simply remove those nodes from the queue, and we make sure to update all the other nodes that could be affected by this modification (change of parents, or list of linked nodes).

Once this process has been completed, we'll end up with a *complete State Graph Model capable of fully describing the logical structure of the application that we're about to test.*

2.4 Testing User Interface Elements

A great bonus earned with creating this State Graph Model is the possibility of performing some *User Interface Testing* at the same time. This is a form of model-based testing where we are trying whether every single UI element is working correctly or not. In a standard coverage testing procedure for a given model, we would be trying to follow every transition of the graph, measuring the percentage of diagram that was actually covered in respect to some features (in structural testing, we would be measuring branch coverage, statement coverage, and so on).

However, in our case we are actually building the model: this implies that every edge and node of the graph is traveled *while* the model itself is still being built. Thus, even though we cannot say to be performing rigorous coverage testing as we do not have a pre-existent model to compare with and our Node and Transition coverage will always be 100%, we are *performing input on every single UI element of the application.*

While the most simple and direct way to be performing UI testing on Android activities would be to click on all visible widgets (or even just touching random sections of the screen), this approach might still miss on some basic bugs. Our State Graph Model construction process parses every single UI element present in every activity (and thus, on the screen) independently from it being visible, invisible, clickable or disabled, and performs a basic input operation on it. This means that while our logical map of the application is being built, we are also verifying that no *Unhandled Exceptions* or *Application Not Responding* errors are being thrown caused by a bad UI element.

CHAPTER 3

APK CLASSIFICATION

3.1 Overview

The Google Play Store, the main source of Android applications, contains around 2.7 million entries[1] available for download. This is a really impressive number, and in order to manage the entries in an organized way, the apps are divided into categories. Splitting all those applications into classes is useful for both the user and the developer.

For *users*, it is easier to discover applications in case they don't know their exact name, or they are just browsing the store for an application to fulfill a specific task (if the user needs a music player, it is easier to simply browse the 'Music and Audio' category to compare a great number of entries, instead of looking at every possible app available for download).

For *developers*, on the other hand, it is simpler to conform and adapt their applications to some standards given by a category, instead of sketching every app from the ground up. Therefore, it is just a matter of making sure that some functionalities comply with the standards of apps belonging to the same category.

Applications are categorized either manually by the developer, or by the store itself, according to the app's description. This might cause misclassification problems, leading to both

[1] https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

difficulties for the user in finding applications, and security issues if the store is checking for category-specific vulnerabilities.

An *Automatic Application Classification system* might therefore be useful not only to objectively categorize apps once they are uploaded to the store without trusting user preferences or subjective descriptions, but also to execute some category-specific tests that might expose vulnerabilities typical to that category. Thus, we are defining a way to classify applications according to their practical functionalities and features that can be retrieved from their APK file in a black-box fashion (without need of direct-access to the source code). If all apps belonging to the same category have the same outline of general functionality, then this must be reflected in some structural patterns inside the APK file.

3.2 Composing the Feature Vector

Our first aim is to find objective features of the application that can hint towards a specific classification. We also want to be able to obtain those without having to access the full source code of the application: the APK file should be the only source material needed in order to avoid availability and privacy issues.

3.2.1 The APK File

The Android Package file (APK) is a compressed archive (similar to JAR or ZIP files) containing a compiled program for Android and additional assets belonging to the app such as widgets, screen layout specifications and strings in different languages. It is the format in which applications are distributed for Android systems, only if digitally signed with a certificate. If

this archive is decompiled (using ApkTool[2] in our case), the following files and folders will appear:

- **assets -** A directory containing all the asset files needed by the application, such as music, video, fonts and so on.
- **res -** A directory containing all the resources files (such as drawables and layout XML files) except for the *res/values* content, as the XML files are converted into binary XML for performance reasons.
- **lib -** A directory containing a folder for each one of the supported CPU architectures. Those enclose compiled libraries used by the program.
- **META-INF -** A directory containing the app signature and other meta information about the APK file.
- **AndroidManifest.xml -** This file describes some essential information about the app to Google Play, the Android OS and building tools. More specifically, it declares: the package name of the app, activities, services, broadcast receivers, and content providers, as well as permissions and hardware requirements.
- **classes.dex -** The compiled Android application code file.
- **resources.arsc -** File with precompiled strings, colors and styles in order to optimize performance.

---

[2]https://ibotpeaches.github.io/Apktool/

3.2.2 Retrieving Android API Class Calls

One of the first places where we should look to find common aspects between apps belonging to the same category would be the list of Android OS API calls. As an example, a specific call such as `android.widget.EditText.setText(...)` to change the content of a text box might be more common in some categories, and less in others.

Some other approaches to the problem of app classification fully decompile the app to the point of having Java source code and using information retrieval techniques to gather information from semantics of the code identifiers (LACTA [8]), while others referred to the full list of all possible Android OS API methods referred by the app (such as AndroClass [10] and ClassifyDroid [9]), but those are tens of thousands of methods, resulting in very sparse data about the 'services' requested by the app (even though a bit more specific). For this reason, we are considering with a broader granularity only the *Android OS API Class calls*, which are way fewer, but should still bring the same information.

Because of that, we are building a binary feature vector where each cell represents one of the possible Android OS API Classes, and its content is 0 or 1 depending on the absence/presence of a call from the application to one of its methods. Therefore, we need to find a list of all possible Android OS API calls, and also identify which ones were actually called by the application.

3.2.2.1 Collecting OS classes from Android.jar

The *Android.jar* file is a Java Archive that is used by the Java compiler before deployment on the actual Android platform. It does not contain the Android framework code as the name might suggest, but only holds stubs for classes, methods and types that our application might

⋮	
android.widget.EditText	1 (Called)
android.widget.Toolbar	1 (Called)
android.telecom.Call	0 (Not Called)
android.media.AudioTrack	0 (Not Called)
android.bluetooth.BluetoothHealth	1 (Called)
⋮	

Figure 4: A section of the binary vector for Android OS API Class Calls (4339 cells).

refer to. The actual API framework is contained in another Android.jar file present on the device itself.

For our purpose, stubs are enough, as we just need a full list of all possible Android API classes that can be called by an application. Therefore, just by using the `"jar -tf Android.jar"` shell command, we are able to obtain the full list. We end up with 4339 possible classes.

3.2.2.2 Detecting referred classes in Classes.dex

The "dex" extension stands for *Dalvik EXecutable*, which is used to hold class definitions and some other data associated to them. Dalvik is an open-source virtual machine able to execute applications written for Android. It was an integral part of the Android software stack up to version 4.4 (*KitKat*). Generally, Android apps are compiled to bytecode for the JVM

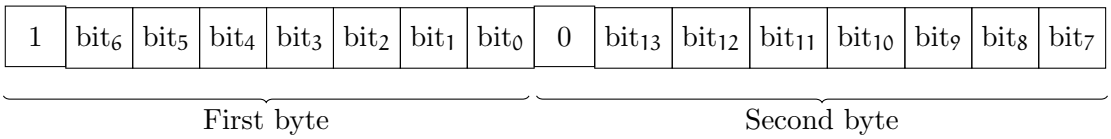


Figure 5: An example of LEB128 encoding with just 2 bytes.

(Java Virtual Machine) and then converted into Dalvik bytecode and stored into those .dex files[3].

Therefore, this is the right place where to look for the methods (and classes) referred by the application. But being this an already-compiled file in `LEB128` encoding (*Little-Endian Base 128*), we have to do a bit of reverse engineering. An encoded value inside this type of file is made of one to five bytes: each one of those bytes has its most significant bit set, except for the last byte of the sequence, which has its MSB (Most Significant Bit) clear. Thus, we have 7 bits per byte carring an actual payload. As an example, we might notice that the unsigned LEB128 encoded hexadecimal sequence "807F" (`10000000 01111111`) is converted to the value 16256 instead of the usual little-endian format 32640 as the MSB are not counted.

Knowing how to decode sequences, it's just a matter of understanding the structure of the dex file itself. This type of file is divided into different sections, but the most notable ones are `header`, `string_ids`, `type_ids`, `proto_ids`, `method_ids`, `class_defs`: `string_ids` contains identifiers for every string (in UTF-16 format) referenced in the file. It is sorted according to

[3]https://source.android.com/devices/tech/dalvik/dex-format.html

the string content and does not contain any duplicates; `type_ids` is a list of identifiers for all primitive types, classes, and arrays; `proto_ids` contains identifiers for all the prototypes that are defined (consisting of argument lists and return types); `method_ids` is a very fundamental section, as it contains identifiers for all API and user-defined methods; in the same way, `class_defs` contains the list of classes referenced by the methods. Finally, the `header` section contains the offset and size of all the other sections, permitting us to understand their location in the file.

Therefore, the procedure should be to look into the header to find the offset and size indicating the start and end (by adding offset and size) of the `method_ids` section: there, each entry is a data structure containing some information about the method itself under the shape of indexes of other sections. The `class_idx` is the index indicating the class to which this method belongs; `proto_idx` is the index indicating the parameters and return type, and `name_idx` refers to the string indicating the name of the method. By adding the index value to the offset of the correct section, we are able to retrieve the desired information: if we keep following this procedure recursively every time we encounter an index, we will end up with some strings from `string_ids`. Therefore, we can concatenate the strings indicating the class owner, method name and prototype to obtain the full method signature. If we do this for every method, we will end up with a list of all method signatures called by the application. Even when obfuscation is performed on the source code of the app in order to reduce the size of the DEX file (by shortening names of classes, methods and fields), this methods should still work as native OS classes are not refactored.

The final step consists of parsing the list of methods (and in our case ignoring the name of the method, keeping just the rest of the signature indicating the full classpath), and comparing them with the full collection of Android API classes (coming from Android.jar) to remove custom made functions, and just keep the called methods that are Android API calls. Eventually, we will be able to compose the final binary feature vector simply by indicating which API classes are used out of the 4339 possible ones. Considering that we do not need to count the actual number of class calls, but simply which ones were referenced, standard decompiling techniques are not necessary in our case, being more suited for a white-box approach.

3.2.3 Retrieving Permissions

Every Android application must contain a file named *AndroidManifest.xml*, which holds essential information about the application itself that need to be shown to various agents, such as Google Play, build tools and even the Android OS. Some of the most important information described in here comprise the hardware requirements needed to run the application on some devices, the application's package name (which is generally the same as the project's namespace, but is later replaced with a unique app ID, provided by the Gradle build tools in order to identify it uniquely on both the system and Google Play) and details about the list of activities that are contained in the application. But, most importantly, the manifest file contains information about the *permissions* that the application is requiring to the Android operating system in order to be executed. This is necessary in order to have a controlled access to some features and areas of the device that might be sensitive or vulnerable. The philosophy behind the system of permissions is that no application should be able to perform operations that might impact

other apps, the operating system, or the privacy of the user; this means that some actions such as writing or reading the user's private data, accessing the internet, keeping the device awake, or modifying files belonging to other apps should not possible in a standard behavior without permissions. For many of those, the user is even prompted to manually consent to their use (this is the case of Contacts, Calendar, Location, Camera and so on).

In our case, permissions might be very helpful in the classification task that we're trying to achieve: communication apps will obviously require the `INTERNET` and `VIBRATE` permissions, which are generally not used by offline video and music players in favor of the `READ_EXTERNAL_STORAGE` and `MODIFY_AUDIO_SETTINGS` permissions. Pretty much every geo-localization application (maps, navigation gps, and so on) will require `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION`, while a calendar will require `READ_CALENDAR` and `GET_ACCOUNTS`. Therefore, a classification algorithm is hopefully able to find a pattern in how those permissions are distributed among all of the categories.

As already previously stated, nearly all of the XML files composing and APK are binaries in order to improve the performance of the whole system. Therefore, the first step should be to get the *AndroidManifest.xml* file in a readable format: to do that, we use *Apktool*[4], which is a reverse engineering tool capable of decoding some resources to their original form. Then, we can simply parse the resulting XML file looking for the `<uses-permission>` tag to retrieve the permission's name: in fact, every android app must specify the permissions it is

---

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

using in the first part of the manifest, right before describing the `<application>` attributes. As an example, a permission for sending SMS would be requested as `<uses-permission android:name="android.permission.SEND_SMS"/>`.

Finally, in order to build the structure of the binary feature array, we still need a complete list with all the possible Android permissions that can be requested, so that every time we read an app's manifest looking for its permissions, we can simply mark them in the full list as "requested". In order to make a satisfactory comparison, we have built two lists: first, a group of the main 60 permissions (both normal and dangerous) available since Android API 23 (Android 6.0)[5], and then a full fledged list of 158 entries, representing all the possible permissions specified in the `android.Manifest.permission` class[6].

Being a binary array, we can simply indicate with a 1 a permission that was requested, and with a 0 a permission that instead was not requested.

3.2.4 Retrieving Hard-coded Strings as Word Vectors

Every Android application features a file named *strings.xml* which is located in `project/res/values/`. This file is the string resource location where the whole application will refer to when a particular string value is needed[7]. It may contain *hard-coded strings* that can be changed during language localization processes, *string arrays* for when a list with multiple im-

---

[5]https://stackoverflow.com/questions/36936914/list-of-android-permissions-normal-permissions-and-dangerous-permissions-in-api

[6]https://developer.android.com/reference/android/Manifest.permission

[7]https://developer.android.com/guide/topics/resources/string-resource

	⋮
INTERNET	0 (Not Requested)
READ_CONTACTS	1 (Requested)
SEND_SMS	0 (Not Requested)
READ_EXTERNAL_STORAGE	1 (Requested)
ACCESS_FINE_LOCATION	0 (Not Requested)
	⋮

Figure 6: A section of the binary vector for Requested Permissions (60 cells).

mutable strings has to be loaded (an example might be a list of countries or languages, which is a constant), or *quantity strings* containing different variants of the same words that can be used according the quantity of objects they are referring to (e.g. "one book" and "many books"). An unique-id is associated with each string using a standard XML encoding: as an example, if we have the following string resource `<string name="string_id">Some text value</string>`, we can refer to it from any part of our application's source code simply by specifying its id as `R.id.string_id`. Therefore, if we need to translate the language of the app, or we just need to change some strings, we can simply modify the values of *strings.xml* instead of changing the source code.

Even though this way of organizing the string structure of the app is not strictly enforced, it is often associated with a good programming style of the app. Therefore, we can expect to see it in most of the publicly available apps. For this specific reason, it is evident how we might have a very important semantic core concentrated in *strings.xml*: being those strings that are

referred multiple times by the app, we are expecting them to echo its main functionalities. With the exception of few common words (such as "options", "account", "confirm" and so on), we are expecting to see a set of words whose semantic value reflect the category of the application. Therefore, in order to collect all strings of our interest, we can simply parse this file and get the value of every <string> field. Then, considering that we might have complex sentences instead of just one word, we split the string on whitespaces obtaining a set of strings from them. If some of those are stop words, we simply ignore them.

Now that we have actually gotten the words that we need, we have the problem of representing them inside the rest of the Feature Vector. One of the most similar approaches (AndroClass [10]) collects all the `N` words appearing among all of APKs used as training set and simply uses a binary vector of size `N` to represent with a 1 the words that are present in the current app instance. The big drawback is that the size and diversity of the vocabulary depends on the number of applications used in the dataset. Also, it is very improbable of finding a very big variety of words, implying that the test set might comprise words that cannot be used if not met before during the training process.

In order to solve the problem of word representation in a more clever way, *Word Embeddings* can be used. Those are a way of representing a word using a numerical vector capable of capturing its context inside of a document and the semantic and syntactic similarity with other words. One of the most popular word embedding representation is **Word2Vec**: this uses an extremely large corpus of text to produce a vector space of hundreds of dimensions where each word is located using its assigned word vector. Therefore, for each word we have its

vectorial representation. Having a numerical representation of a word opens up the possibility of performing arithmetical operations among them: the most classical example given is that `vector("King") - vector("Man") + vector("Woman")` gives us a vector representation that is the closest to `vector("Queen")`.

The next question regards the way in which we can produce those word vectors in the first place. As described on the official Google website[8], it all depends on 5 factors: the *training architecture*, *training algorithm*, *sub-sampling value of frequent words*, *dimensionality of the word vectors* and *context (window) size*. Regarding the architecture, [14] explains very well the two main types of architecture currently available: the first one is **Continuous Bag-of-Words Model (CBOW)**. Here words are represented as binary arrays of size `D`, which is the cardinality of the dictionary (thus, a value of 1 is located in the position corresponding to that word in the dictionary, meaning a one hot encoding), and a set of precedent and subsequent words (`N` in total, representing the context of the word in question) is given as input to the model. This consists of a projection layer which produces as output a distribution of probabilities (softmax of size `D`) indicating the most plausible word to fit in that context. The training criterion is to correctly classify the current (middle) word, given its context. In this case, the order of words inside the context is not taken into account, hence the name "Bag of Words". The second case is **Continuous Skip-gram Model**, as described in [15]: this architecture is in a way similar to CBOW, but specular. The input is the current word, which is represented as an

---

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

array of size `D`, which goes into a projection layer producing as output a prediction of the words appearing before and after the current word as probability distributions. In this case, the order of words is taken into account: the more distant words are usually less related to the current word than those close to it, therefore a lesser weight is given to them. In Figure 7 we can see a simplified representation of those two models, where the lateral lists of rectangles represent one-hot-encoded words, while the central ones are symbolizing numeric matrices used for the projection of the words themselves, and will therefore yield the word embedding values.

In our case, instead of performing the training of a new model from the ground up (which would require extensive time and resources), we are using **Gensim's Keyed Vectors**[9]: those are simple mappings between strings and their correspondent one-dimensional array representing their Word Vector. The difference with a standard model is that a Keyed Vector model cannot longer be trained, and therefore is optimized in terms of space to be just used as a readable source. More specifically, the *Google News Word2Vec model*[10] has been used in its Keyed Vector form. This model has been trained on 100 billion words from a Google News dataset, and contains a dictionary of 3 million distinct words including even misspelled and derivated words[11] (this is why stemming and lemmatization have not been executed, which

---

[9] https://radimrehurek.com/gensim/models/keyedvectors.html

[10] https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit

[11] https://mccormickml.com/2016/04/12/googles-pretrained-word2vec-model-in-python/

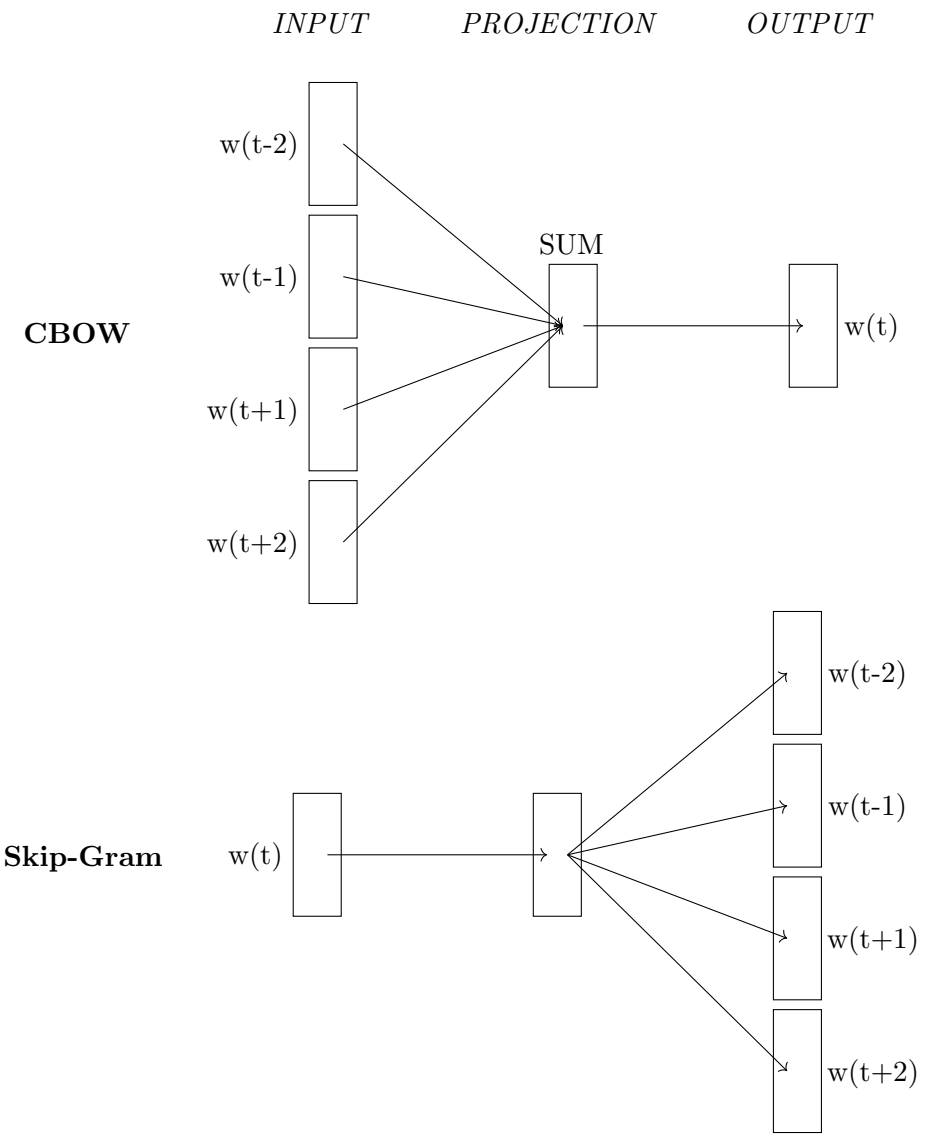


Figure 7: Graphical representation of CBOW and Skip-gram Word2Vec architectures.

would imply a loss of context of the word itself). Each embedding of this model is vector of 300 values (dimensions).

The process is pretty simple, as we simply query the model for every word string value that has been found in *strings.xml* in order to obtain the corresponding Word Vector of 300 values. Then, considering that we might have a variable number of strings, we calculate the average Word Vector value among all the strings: if we think about the spatial abstraction of the word embeddings, this means finding an averaged point that locates this app in a 300-dimensions space. Therefore, we end up with 300 cells of float values for our feature array. The usefulness of this representation stays in the fact that we can use a compact array learned from the typical use of the words, instead of using sparse representation.

3.2.5 Resulting Vector

Summarizing, we have obtained the a Feature Vector composed of the following elements:

- **Referenced Android API Classes -** A binary indication of whether a specific class of the Android operative system API has been referenced by the app under examination. This is composed of 4339 cells.
- **Requested Permissions -** A binary indication of whether one of the Android OS permissions has been requested or not. Corresponding to 60 cells.
- **Embedding of Hard-Coded strings -** A set of 300 float values representing the average of the word embeddings of all the hardcoded strings of the application. Consisting of 300 cells.

Therefore, we have a Feature Vector composed of 4699 elements, which can be seen represented in Figure 8.

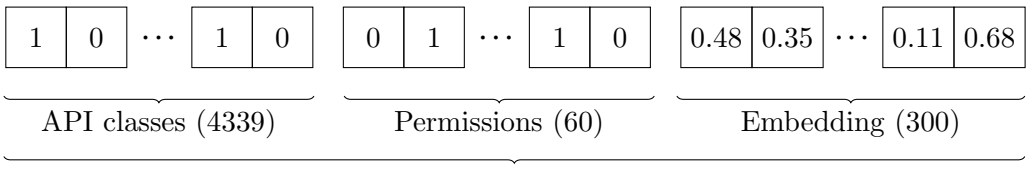


Figure 8: Representation of the final structure for the Features Array.

3.3 Building the Dataset

A key step consists of finding the set of APK files that will be used in the training process for the classification algorithm. One of the positive aspects of using a *black box* approach (meaning that we do not need to access the original source code of the application) to the feature extraction process is that we are not forced to look only for applications which are open source or whose source code is publicly available: we can simply retrieve applications that are normally available in the usual Android app stores.

Therefore, the main question should concern which would be the right source of applications: we have many different possibilities such as the official Google Play Store[12], APK Mirror[13] and other websites. In our case, the **APK Pure store**[14] was chosen, as it provides a very wide selection of app categories on one side, and it provides direct download of all of its apps on the other, without requiring an actual Android device. Thus, we built a very simple crawler in Python programming language that is able to parse every download link of apps belonging to a specific given category on that website. In order to manage the download times, apps with a size greater than 100 MB or belonging to the *games* category were avoided. However, due to unstable networks and traffic bottlenecks on the website, the time necessary to download each one of those applications ranges from 2 to 5 minutes. This means that it takes a large time to gather a very large amount of APKs.

Regarding the actual categories used for the classification process, we selected **10 categories** among all of the ones that the APK Pure website offers. This means that for our training set we are trusting the class subdivision that they are providing as our ground truth.

Table I shows that the categories were chosen as the subset of the APK Pure categories that could have the most diversity among them. Each category can be identified in a very distinctive way, so that we do not have the problem of ambiguity; a *Social* app will always be very distinct

---

[12]https://play.google.com/store?

[13]https://www.apkmirror.com/

[14]https://apkpure.com/

TABLE I: STRUCTURE OF THE APK CLASSIFICATION DATASET

Category	Number of APKs
News and Magazines	293
Communication	342
Shopping	293
Education	295
Social	292
Food and Drink	294
Video Players	307
Medical	330
Weather	409
Music and Audio	341
	Total: 3196

from a *Weather* app. The hope is that this variety can still be maintained and represented by their feature vectors. Also, an average of 300+ apps were downloaded for each category: this number is due to the time it takes to gather that amount of APKs. Thus, we end up with **3196 apps** shared among 10 categories, which is a ratio perfectly comparable with the one used in AndroClass [10]. They use about 270 apps per category.

3.4 Model Architecture

Having determined both the Features Vector and the Dataset of APK files, we just need to find a training algorithm for our classifier. Instead of starting from scratch and trying out many different algorithms, we used as a reference point the paper documenting AndroClass [10], which is an approach very similar to ours: considering that we both use information coming from APIs, Strings, and Manifest to perform APK Classification, we should be able to use some

of their results without losing all their validity. More specifically, its authors have tried to use K-Nearest Neighbors (KNN), Naïve Bayes (NB), Support Vector Machine (SVM) and Deep Neural Networks (DNN) even adopting different datasets (comprising ours, APK Pure[15]). In all cases, DNNs seem to outperform the other algorithms in terms of Accuracy, Precision and Recall.

For this very reason, we adopted neural networks as the mean of classification in our approach. Thus, the next step is understanding how to set up its architecture. But, in order to be able to choose one model over the other, we need a reliable way of comparing their performance.

In a ***Hold-Out*** fashion, the entire dataset was split into a 10% of data points to be the test set, and the remaining 90% to be the Training Set. However, in order to be able to determine the correct set of hyper-parameters, we also had to introduce a validation split. For this exact purpose, we have adopted the ***Stratified 10-Fold Cross Validation***, splitting our Training Set (90% of whole data) into *10 folds*: in this way, the measurements obtained by choosing a set of hyper-parameters will not be relative to a given validation split; we have 10 iterations where each one of the 10 sections is adopted as validation set at some point, while the remaining 9 folds act as a training set. During the first iteration, the first fold is used as validation set, while the remaining 9 as training set for training the model. In the second iteration, the second fold becomes the validation set. This process keeps going up to the point where all 10 iterations have been completed and all folds have been used as validation set exactly once. The final

---

[15]https://apkpure.com/

resulting measure will be the average of the scores obtained in every one of the K iterations. After the best performing set of hyper-parameters has been chosen, it can be used to execute the final measurement on the Test Set.

A small yet important detail to notice about our problem, is that the feature vector that we've built in the previous sections is composed of *Mixed Data*, meaning that we have both binary and continuous features. If one of those shows a variance that is way greater than the others, it might dominate the objective function overshadowing some other features, which wont be used by the model to learn. For this reason, we have to perform ***Data Scaling***, and more precisely ***Data Standardization*** for continuous features: to each sample is assigned a new value $z = \frac{(\text{old\_value} - \text{mean})}{\text{std}}$. Performing standardization means trying to reorganize our data in such a way that the new mean is 0, and the new standard deviation is 1, resembling a Gaussian distribution. In this way, every feature should be able to equally influence the model.

In order to build our architecture, we have to take care of **tuning the Hyper-parameters**:

- **Number of Units per Layer -** This parameter depends on the dimensionality of the training data, and can be used to find a balance between bias and variance (meaning reaching high accuracy just on the training set vs being able to generalize very well on new test sets).
- **Number of Layers -** Also depending on the size of the training data: a high number of layers might lead to over-fitting and vanishing gradient.
- **Activation Function -** Those include ReLu, Sigmoid and Tanh.

- **Optimizer -** Consists of the algorithm used by the model to update the weights of each layer after every iteration during back propagation. The most common are Adam and Stochastic Gradient Descent (often abbreviated SGD).
- **Learning Rate -** How the name might suggest, it determines the the speed of convergence: a value too high might prevent convergence to the global minimum, while a value too low might slow down the learning process too much to the point of reaching no convergence at all.
- **Batch Size -** Indicates the number of data points used in each training iteration before updating the weights. Might influence the possibility of convergence.
- **Number of Epochs -** It consists of the number of times the training data is shown to the model. Higher values favor over-fitting, while lower ones might be an obstacle to convergence.
- **Dropout -** It is a regularization technique that prevents some weights to be updated with certain probabilities. It should avoid over-fitting.

The quest for the perfect values can be performed in many different ways[16]: we might just go with a *Hand Tuning (Trial and Error)* approach where we simply try to execute some experiments and according to their results we try other values, a *Grid Search* where we try all possible combinations of parameters, a *Random Search* among which we choose the parameters that yielded the best value, and *Bayesian Optimization*, which uses the accuracy of a previously

---

[16]http://neupy.com/2016/12/17/hyperparameter_optimization_for_neural_networks.html

evaluated set of parameters to make assumptions about unobserved combinations of parameters. In our case, a flavor of the Hand-Picked approach was adopted: we manually choose a subset of parameter combinations to try out, guided by the performance that the architecture yields every time.

Many different experiments have been performed with the architecture of the Neural Network: the most significant of those can be be seen in both Table II and Table III with their respective measures. As already stated, those measures are coming from 10-Fold Cross Validation processes.

The first hyper-parameter that we have been tuning was the **number of units**: just by keeping one single Relu layer, the number of Units has been incremented from 500 to 4000. The highest value of accuracy (51.37%) can be obtained with around 1000 Units, but fluctuates of just 2 percentage points over an increment of 3000 more Units. Precision changes a bit more, but is highest for 1000 Units or less. All of this suggests us that *we should stick with around 1000 Units on the first layer* for now. Also, during those initial experiments, we kept a Stochastic Gradient Descent optimizator and a batch size of 32, which is generally considered fine for most of the cases.

Then, we've spent some time looking at the **Dropout layer**: as [16] explains very clearly, the training set often presents some noise that will be learned by the network even if this does not really exists in other datasets or real scenarios. This might obviously lead to over-fitting. Some solutions are to stop training as soon as the validation performance starts to worsen, use weight penalization techniques such as L1 and L2 regularizations, or adopt Dropout: that is a

TABLE II: RESULTS OBTAINED WHILE EXPERIMENTING WITH DIFFERENT ARCHITECTURES FOR THE APK CLASSIFICATION NEURAL NETWORK - PART 1

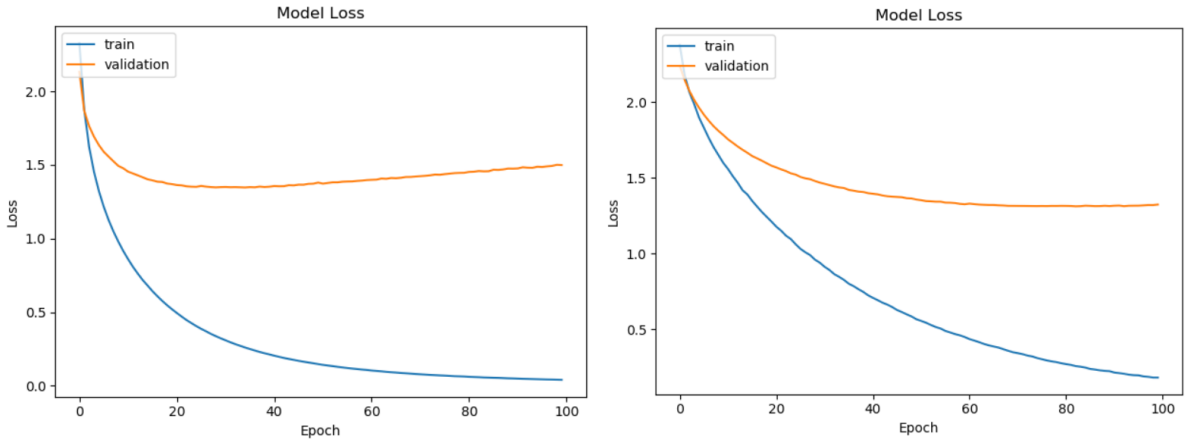
Architecture	Accuracy	Precision	Recall
500 Units	50.16 % ± 1.87 %	60.91 % ± 10.70 %	45.96 % ± 10.67 %
1000 Units	51.37 % ± 5.52 %	60.81 % ± 11.22 %	54.71 % ± 10.95 %
1500 Units	50.64 % ± 2.62 %	59.87 % ± 14.15 %	49.47 % ± 11.07 %
2000 Units	50.75 % ± 2.78 %	57.18 % ± 12.99 %	53.20 % ± 13.05 %
4000 Units	49.48 % ± 2.94 %	55.46 % ± 12.12 %	54.38 % ± 8.45 %
One dense Relu layer. Batch is 32, optimizer is SGD (LR=0.01).			
Dropout 10%	49.48 % ± 2.59 %	53.35 % ± 9.81 %	54.96 % ± 7.11 %
Dropout 30%	49.05 % ± 2.86 %	62.67 % ± 10.57 %	52.03 % ± 9.62 %
Dropout 50%	48.36 % ± 1.74 %	58.64 % ± 13.46 %	49.94 % ± 11.05 %
One dense Relu (1000 Units) layer with dropout regularization. Batch is 32, optimizer is SGD (LR=0.01).			
Batch 64	51.38 % ± 2.40 %	56.27 % ± 12.89 %	58.55 % ± 9.87 %
Batch 128	52.96 % ± 4.91 %	60.49 % ± 6.25 %	56.72 % ± 12.77 %
Batch 256	52.03 % ± 2.56 %	64.59 % ± 10.70 %	53.18 % ± 11.49 %
One dense Relu (1000 Units) layer. Optimizer is SGD (LR=0.01).			
tanh	52.06 % ± 1.87 %	59.88 % ± 11.84 %	56.42 % ± 7.29 %
sigmoid	53.03 % ± 2.46 %	65.54 % ± 13.64 %	53.79 % ± 7.69 %
swish	52.23 % ± 2.93 %	57.94 % ± 11.74 %	55.55 % ± 10.25 %
One dense layer (1000 Units). Optimizer is SGD (LR=0.01). Batch is 128.			
SGD LR=0.001	46.46 % ± 2.35 %	83.93 % ± 12.33 %	23.12 % ± 8.29 %
SGD LR=0.01	53.03 % ± 2.46 %	65.54 % ± 13.64 %	53.79 % ± 7.69 %
SGD LR=0.1	10.66 % ± 1.18 %	19.99 % ± 39.99 %	0.58 % ± 1.17 %
ADAM LR=0.01	11.15 % ± 1.51 %	14.99 % ± 32.01 %	0.57 % ± 1.15 %
ADAM LR=0.001	55.98 % ± 2.73 %	63.40 % ± 8.05 %	59.09 % ± 11.78 %
ADAM LR=0.0001	57.24 % ± 1.80 %	70.09 % ± 5.54 %	57.29 % ± 7.46 %
ADAM LR=0.00001	57.27 % ± 2.16 %	74.49 % ± 6.23 %	57.63 % ± 8.77 %
One dense Sigmoid (1000 Units) layer. Batch is 128.			

TABLE III: RESULTS OBTAINED WHILE EXPERIMENTING WITH DIFFERENT ARCHITECTURES FOR THE APK CLASSIFICATION NEURAL NETWORK - PART 2

Architecture	Accuracy	Precision	Recall
Dense (1000) + Dense (500)	$58.28\,\% \pm 2.22\,\%$	$75.69\,\% \pm 8.55\,\%$	$53.44\,\% \pm 9.21\,\%$
Dense (1000) + Dropout 50% + Dense (500)	$58.62\,\% \pm 3.21\,\%$	$78.25\,\% \pm 7.70\,\%$	$54.38\,\% \pm 6.93\,\%$
Dense (2000) + Dense (100)	$56.29\,\% \pm 1.81\,\%$	$87.48\,\% \pm 10.16\,\%$	$44.78\,\% \pm 9.71\,\%$
Dense (1000) + Dense (500) + Dense(100)	$57.79\,\% \pm 1.47\,\%$	$91.57\,\% \pm 6.53\,\%$	$46.18\,\% \pm 5.39\,\%$
Dense (1000) + Dense (100)	$57.19\,\% \pm 2.47\,\%$	$83.79\,\% \pm 5.92\,\%$	$50.90\,\% \pm 9.86\,\%$
Dense (1500) + Dropout 50% + Dense (300)	$56.94\,\% \pm 2.93\,\%$	$80.98\,\% \pm 7.83\,\%$	$51.78\,\% \pm 11.62\,\%$

regularization technique that allows the network to drop each node with a specified probability during every iteration. If a unit is dropped during an iteration, then the unit itself and every edge coming in and out of it are ignored. This is supposed to reduce the risk of over-fitting. Baldi and Sadowski [17] try to demonstrate that a dropout of 50% is fine for most of the cases, thus we started with that one and decremented it down to 10%. The measurements indicate that having just one layer, every single unit is fundamental for the best functioning of the classifier. However, if we look at the loss chart for our model in Figure 10, it is very notable

how without regularization the model tends to over-fit after 30 epochs. Instead, if we add a Dropout with 50% probability, the overall loss decreases slower than before, but also starts to grow back at a much later time, indicating that over-fitting is being fought.



No Dropout. Dropout at 50%.

Figure 10: Effect of Dropout Regularization on model loss.

We can also now discuss the right **number of Epochs** for our training process: in Figure 10 we can notice that if if we add some dropout regularization, we still have stable and acceptable loss values up to around 90 epochs. After that they start to increase a bit. If instead we don't do any kind of regularization at all, over-fitting takes control of the validation curve as soon as 40 epochs into the training process. Therefore, considering that few epochs later (around 90)

both the training accuracy and loss start to saturate, we could say that *90 epochs is a great compromise* between lack of over-fitting and enough training for the model. Referring to the accuracy chart shown in Figure 11, this number of epochs is even enough for the model to reach around 98% training accuracy (while the validation accuracy is already stable at 70 epochs). Both figures are referred to an architecture of Dense Sigmoid (1000) + Dropout (50%) with an ADAM optimizer (LR=0.00001) and Kullback Divergence loss.

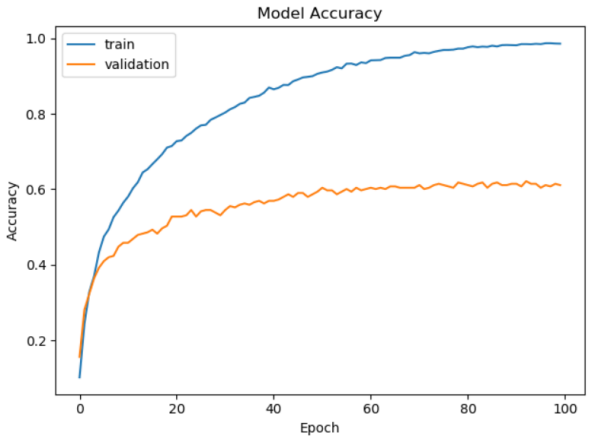


Figure 11: Accuracy behavior in terms of both Training and Validation sets.

Next step is deciding the right **Batch size**: as already stated, this influences the number of data points that can be trained together in the network before updating the weights: increasing it should make the system converge faster, but maybe not to the global maximum. Values of 32, 64, 128 and 256 have been tried using the same architecture for the rest of the system, and

the value of 128 seems to give the best overall validation accuracy. However, even values of the batch size that are slightly smaller or bigger could still do fine, as the change in performance is negligible.

The **Activation function** is also very important, as it defines the output of a node given set of its inputs. We are mainly interested in non-linear activation functions, as they allow for a more differentiated output. The *ReLU (Rectified Linear Unit)* function has been tried first with an accuracy of 52.96%, *Tanh (Hyperbolic Tangent)* yields an accuracy of 52.06%, while *Sigmoid (Logistic Activation Function)* reaches the highest value, 53.03%. Also, we have tried a pretty "new" activation function presented by Google engineers in [18]: this function is called *Swish* and it is supposed to resemble the broad behavior of a Relu function, even permitting to recycle some parameters tuned for it. Anyhow, it is composed by a sigmoid function, it is able to achieve negative values for $x < 0$, and it is supposed to achieve slightly better performance:

$$f(x) = x * \mathrm{sigmoid}(x)$$

However, in our case this function produces performance values that are extremely similar to the Relu one (both 52%). Anyways, it does not match the Sigmoid function, which still remains the one with the highest accuracy.

We have also played a bit with the **Optimizer**. First, we have made some experiments with the *SGD (Stochastic Gradient Descent)*, which is a gradient descent technique (meaning that parameters are changed in order to move the value of the objective function in a way such

that the error rate is reduced up to convergence) where instead of making calculations on the whole dataset, we just perform them on a small subset of data examples. Different Learning rates (LR) have been tried to understand which is the right speed of convergence, and in SGD 0.01 largely shines over the others two values reaching an accuracy of 53.03%. As a term of comparison, we have also implemented the *Adam optimizer*, which is an algorithm for gradient-based optimization of stochastic objective functions. Even in this case, we have tried different Learning Rates, and we've discovered that a slower rate of 0.00001 fits this algorithm the best, yielding an Accuracy of 57.27%. Therefore, the Adam optimizer has to be kept.

We've also tried to compare a couple of **Loss Functions** for multi-class classification problems: *Kullback Leibler Divergence* and *Categorical Crossentropy*. They both perform in a very similar way (57.27% for Kullback vs 56.50% for Categorical), but Kullback performs about half percentage point better than the other.

Finally, we have also made some experiments with **different numbers of Layers** which can be seen in Table III: in all of those we kept the same ranges of hyper-parameters that were discussed before. Evidently, keeping a first layer of 1000 units together with an additional second layer increases the performance of the system a little bit. More specifically, the highest accuracy is yielded by an architecture of two dense layers (1000 and 500 units respectively), plus a dropout layer between the two to avoid overfitting. This means that adding a non-linearity factor helps the model to adapt more carefully to the data of our problem. Eventually, we reach an accuracy of 58.62%, few percentage points higher than simply using one single layer.

As few final notes on the architecture, being this a Multi-class classification problem where only one label at a time can be given to the input data, the activation function of the output layer is *Softmax*; in this way, the output probabilities of belonging to a specific class add up to 1. Also, the dataset has been divided into 90% training and 10% testing: not having a very massive number of APKs to be used in training the system, we had every interest of keeping the training set the largest possible. In this way the performance of the system cannot be compromised too much while still having a test set capable of producing some meaningful performance indexes.

3.5 Classification Results

Summarizing, we have opted for the following architecture (which can be seen in Figure 12): an input layer with a dimensionality of 4399 nodes goes towards a Dense layer of 1000 Units with a Sigmoid activation function that flows into a second Sigmoid Dense layer of 500 units through a Dropout layer with a dropping probability of 50% in order to reduce the effect of over-fitting. All of this end up in an output layer of 10 nodes (because such are the possible categories) with a Softmax activation function. The loss function is the Kullback Leibler Divergence, while the optimizer is the Adam optimizer with a Learning Rate of 0.00001.

Using this architecture, we are able to **reach about 58% accuracy**. More specifically, accuracy is 58.46% with a standard deviation (std) of 2.88%, while **precision is 81.36%** (std is 9.31%). Finally, **recall is 54.05%** with a std of 7.75%. Those results are referred to using the group of only 60 permissions. If instead we use the full set of 158 permissions, accuracy is 58.18%, indicating that the advantage of having the possibility to represent every single

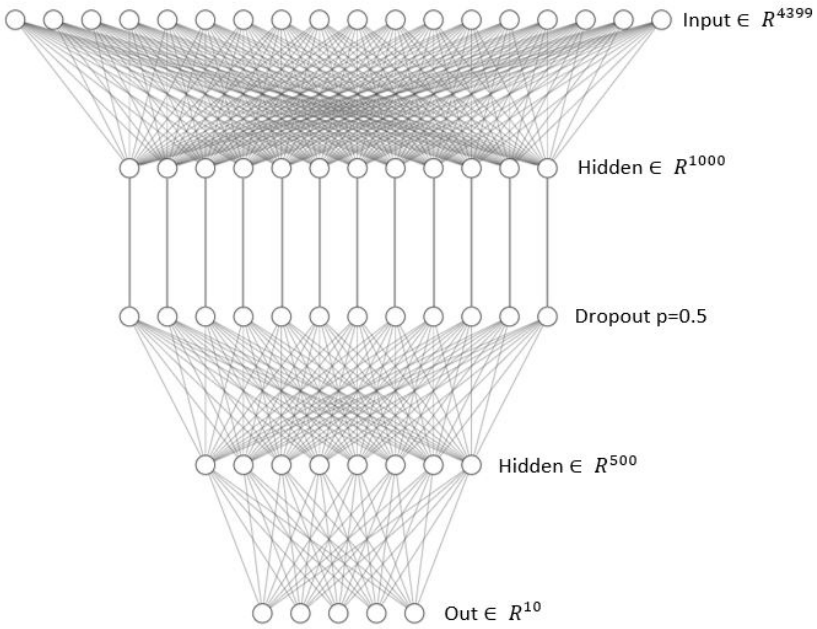


Figure 12: APK Classification Neural Network architecture.

requested permission is compensated by a wider feature array that makes the classification process harder. Thus, there is not much difference in using one group of permissions or the other.

We can now compare our performance with some other Application Classification approaches. ClassifyDroid [9] tries to classify apps into 10 categories as well using a modified version of Multinomial Naive Bayes, but they use a way bigger dataset composed of 15590 samples coming from the Chinese MM App aarket[17]. However, they also present the accuracy

[17]http://mm.10086.cn/

TABLE IV: COMPARING FINAL APK CLASSIFICATION RESULTS WITH DIFFERENT FEATURE VECTORS

Feature Vector	Accuracy	Precision	Recall
Standard (API+Perm.+WV)	$58.46\,\% \pm 2.88\,\%$	$81.36\,\% \pm 9.31\,\%$	$54.05\,\% \pm 7.75\,\%$
No WordVector	$49.29\,\% \pm 2.91\,\%$	$61.02\,\% \pm 15.11\,\%$	$45.37\,\% \pm 11.41\,\%$
No Permissions	$45.49\,\% \pm 3.95\,\%$	$54.82\,\% \pm 11.75\,\%$	$45.96\,\% \pm 10.44\,\%$

trend for different percentages of labeled samples used, and for 3118 apps (20% of whole dataset, perfectly comparable with our dataset of 3196 data points) they reach an accuracy of 55%. The Lacta approach [8] classifies apps into 8 categories using a dataset composed of only 42 apps (probably chosen to be suited for the purpose) and they report precision measures, omitting accuracy. If we use the same number of categories we reach 88% precision, which is perfectly comparable with their 89%. However, using tailored apps for constructing the dataset might bias the performances. *AndroClass* [10], which is a very similar approach to ours, uses a NN to classify apps retrieved from our same source (APK Pure), using about 277 apps per category (similar to our ratio). In this case they reach an Accuracy of 48%, a Precision of 45% and a Recall of 41%.

As an experiment, we've tried to *modify the Features Vector* by removing some of its sections in order to prove that all of its elements are necessary in a way or the other. In Table IV it is possible to see that removing the Word Vector representation of the hard-coded strings inside the application damages the model about 9% in terms of Accuracy, while removing Permissions

reduces the performance even more, up to 13%. This is just a small comparison to highlight that none of elements constituting our Feature Vector are superfluous, but instead all of them have a semantic meaning that is helpful in the classification process.

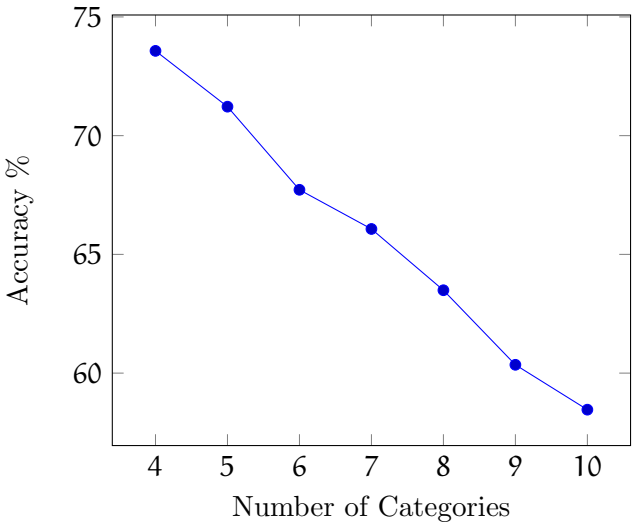


Figure 13: Effect of Number of APK classes on model Accuracy.

Finally, we've also tried to understand which is the *effect that the number of APK categories has on the overall accuracy* of the model. In Figure 13 it is possible to see the results plotted on a chart: using only 4 categories accuracy is at 73.57%; from there, it starts to decline linearly. At 5 classes, which is the half of the number of categories used in our other measurements, we are still at 71.22% of Accuracy. This means that even though the decline is linear in the

number of categories, the loss of accuracy is still very little overall. Thanks to this information, we would even be able to project the plausible values of accuracy for any higher number of classes (speculating that its behavior remains close to a linear one).

In *conclusion*, this architecture seems strong enough to solve the APK classification problem, but, as can be clearly seen in the conclusive section of the AndroClass paper [10], results can change drastically simply by switching datasets and keeping the same architecture: in AndroClass, the authors reach an accuracy of 48% using the APK Pure[18] dataset (same source as ours), 57% using the Google dataset[19] and even 85% using a hand-made dataset. This implies that the way in which training applications are categorized in certain classes instead of others can influence massively the performance of the system. Having some applications in the training dataset that are mis-classified or ambiguous (meaning that they could belong to more than one category) can disrupt the entire training process. On the other hand, APKs that are clearly belonging to their category without any uncertainty could improve it. Therefore, *ideally we would have a manually-labeled dataset that has a reduced yet distinct set of categories* (as AndroClass demonstrates that by doing just that, performances in terms of accuracy could increase up to 37%). All of this highlights once again how the manual categorization of Android applications performed on some app stores can lead to ambiguities. In our case, due to time and resource constraints, it was possible to just crawl the APK Pure source, but it would be

---

[18] https://apkpure.com/

[19] https://play.google.com/

interesting to see what happens with a dataset composed of thousands of manually-labeled applications: if the same increase applies to us, we could potentially look at around 90% of accuracy.

CHAPTER 4

ACTIVITY CLASSIFICATION

4.1 Overview

When a Developer has to build a Desktop application, he's generally very free in deciding the structure of his program, not having to respect any enforced design pattern. Desktop applications can therefore appear in many different shapes, to the point of being extremely different one from the other even if they are for the same purpose or belonging to the same topic.

Android applications on the other hand have to respect *very specific patterns* in both their structure and design:

1. Every Android application is made up of **activities**[1], which are essentially the basic element composing the screen that is showed to the user. An app is composed of one or more activities, and every activity can host a set of UI elements organized in a hierarchical structure and/or fragments[2]. Thus, an Android application can be thought as a series of activities connected by the input of different "buttons" present on the user interface. The activity is ultimately responsible of creating, controlling and destroying the user interface assigned to it.

---

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

[2] https://developer.android.com/guide/components/fragments

2. One of the strong points of Android applications stays in the very high user friendliness given by the use of graphical design patterns that repeat themselves in pretty much every application. Having developers follow precise **design guidelines**[3], users do not need to learn how to design every new app from the ground up and the learning curve becomes very shallow. For this specific reason, users will always know where to find the button for opening the side drawer, the screen section to tap to open the app options spinner, or simply the location of the arrow to press to go back to the previous screen. As Nagel and Fulchen explain during the 2013 Google I/O conference[4], by following those repeating patterns the user doesn't need to learn any new gestures to use the app, and the developer does not need to worry about designing user-friendly layouts, as they are all pre-defined by the affirmed Android guidelines.

Therefore, an Android application follows some structural and design patterns by definition that we can exploit in a functional testing process: if, for example, many applications appear to have a "Settings" section that is always structured in the same way as a ListView and composed of the same clickable elements and toggles, that suggests that those activities might require a very similar treatment in testing terms. Instead of manually writing hand-crafted testing scripts that need to be re-adapted to every new instance of the same type of screen, we might just have a tool to execute some adaptive functional test scripts for a given specific category of activity.

---

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

[4]Structure in Android App Design - https://www.youtube.com/watch?v=XpqyiBR0lJ4

This could save hours of work, and could also open up the way to a new kind of functional testing on mobile applications, where we try to execute the same set of commands on the same type of screen on hundreds of different applications, measuring which are the apps that conform the least to a given standard behavior.

Thus, the first step should be to find a way to classify the Android activities that we encounter, in order to know which test scripts to fire off.

4.2 Determining the Set of Features

We have already stated that our objective is to be able to classify every Activity that we meet in an Android application in order to execute some adaptive test scripts. Therefore, we have a Multi-class classification problem in our hands.

As stated just above, an Activity can be considered as a container and manager of UI elements. Now, we can have two main kinds of those elements: on one hand we have *visible elements* that can be actually seen by the user on the app's screen and they allow him to communicate with the system though some gestures. Some examples might be clickable buttons, switches and text fields. On the other hand, we have *invisible elements* that are fundamental to control the structure and behavior of other UI elements (including the visible ones); those might simply be structures arranging elements vertically or horizontally, or layout types indicating how the relative placement of UI elements on the screen should be managed.

We should keep all of this into consideration while trying to classify an activity, as some structural patterns or specific UI elements could be appear often in a specific type of screen. Let's take as an example a *login activity*: usually, it features at least two `EditText` boxes, of

which one is for passwords; therefore, the `isPassword` attribute would be set to `true`. The presence of this field suggests it might be a Login Activity. Thus, not only the user interface elements are important, but *also their attributes.*

4.2.1 Dumping Activities

Considering that we care about the layout structure of the screen and that we need it for our classification purposes, we could simply retrieve it from the layout XML file to which the activity in question is associated. However, we are not doing that as we are also interested in some other information:

- The **attributes** of some UI elements can have a value, and this value can change over time giving some important indication about the status of the Activity in question. Let's take a `CheckBox`[5] as an example: that is a type of button with two states that can be either in a checked or unchecked state. Also, many different UI elements have a `isFocusable` or `isClickable` attribute that indicates whether a specific element (e.g. an `EditText`[6]) is available to the user input. The presence of some of those attributes can be vital in the classification process, giving us hints towards a specific category of activity (e.g. a `isPassword=True` field might indicate a login activity).

---

[5]https://developer.android.com/reference/android/widget/CheckBox

[6]https://developer.android.com/reference/android/widget/EditText

- Some UI elements are **dynamically generated at run-time**: if for instance we think about a `ListView`[7] for displaying a list of user-generated content (such as notes, images or files), it is surely generated at run-time by the application, by producing the layout of each line inserted and composing the complete list. Therefore, the full layout can only be retrieved during the execution of the program and not before that, not knowing which elements are missing.

The objective is now to retrieve all information about an activity's layout at run-time. A tool that comes up as very useful is the **UiAutomator framework**[8]: that is a set of API calls part of the Android Software Development Kit (SDK) built to ease the execution of some black box tests. This tool provides two very important functionalities: first, it allows to *inspect the layout hierarchy* currently loaded on the screen of the device; secondly, it is able to *perform basic input operations* on the device itself. More specifically, it provides access to a `UiDevice` class that represents the Android device upon which the app is running. By calling methods of this class, we can access the status of the device itself and send input events (such as "`UiDevice.pressHome()`").

However, in order to be able to easily access those methods through a script instead of an *Android Debug Bridge* (adb) shell, we used the **AndroidViewClient**[9] Python library: that

---

[7] https://developer.android.com/reference/android/widget/ListView

[8] https://developer.android.com/training/testing/ui-automator

[9] https://github.com/dtmilano/AndroidViewClient

```
View[ class=android.widget.FrameLayout ... ]
    View[ lass=android.view.ViewGroup ... ]
        View[ class=android.widget.TextView ... ]
View[ class=android.widget.FrameLayout ... ]
   View[ class=android.widget.RelativeLayout ... ]
        View[ class=android.widget.TextView ... ]
        View[ class=android.widget.Button ... ]
```

Figure 14: An example of a View Hierarchy Dump using UI Automator.

```
    View[ class=android.widget.Button index=1 selected=false checked=false
clickable=true package=com.example.artur.myapplication text=BUTTON long-
clickable=false enabled=true bounds=((0, 297), (1080, 441)) content-desc=
focusable=true focused=false uniqueId=id/no_id/11 checkable=false resource-
id=com.example.artur.myapplication:id/button password=false class=android.
widget.Button scrollable=false ]   parent=android.widget.RelativeLayout
```

Figure 15: The complete set of Attributes for a dumped UI element.

provides a set of methods that are interfaced with some other UI Automator functions in order to allow the user to provide commands and retrieve responses in Python format (instead of manually going through a shell each time). In few words, AndroidViewClient allows to call UI Automator methods via Python code.

The most important command that we can use is `dump`[10]: as the name suggests, it performs a dump of the current screen that is displayed on the device. The result is a textual representation of the logical hierarchy of the UI elements currently present on screen. An example of this can

[10]https://github.com/dtmilano/AndroidViewClient/wiki/dump

be seen in Figure 14, where it is possible to notice how even information about invisible layout structures (e.g. RelativeLayout, FrameLayout) and their parent/child relationships are shown. Each line represents the full list of attributes about a specific UI element; in the example of Figure 15 we can read the full set of specific Attributes that is retrieved: we go from the spatial location of the *element's bounds*, *class*, *id* and *package* to the *scrollable*, *focusable* and *clickable* attributes.

4.2.2 Screen Sectioning

We already stated many times that we are trying to exploit some common Android app design patterns in order to ease the classification process of an Activity. As stressed by Rosenfeld, Kardashov and Zang [11], many applications tend to place the same kind of UI elements in the same areas of the screen. Therefore, a good idea might be to *split the device screen in some areas* where we know we are supposed to find elements specific for a particular category of Activity.

As a starting point, we tried the screen subdivision proposed in [11], splitting the device's screen into *three areas*: top 20%, mid 60% and bottom 20%.

- In the **top 20%** we usually find the *App Bar*[11] with Drawer and Options buttons, or, in less structured apps, simply the header of the app with some sort of top menu. This section does not change a lot among apps of any category.

---

[11] https://developer.android.com/training/appbar

- In the **mid 60%** section we can find the core of the app's content, which might different a lot from category to category.
- In the **bottom 20%** we can either find a navigation bar[12] with a selection of Image buttons, or simply a Floating Action Button (FAB) to perform some kind of operation. In any case, this bottom part tends to be very similar among every category of application.

Therefore, this sectioning operation is helping classification by introducing some **localization information** in the Feature Vector: the presence of a button in the top part of the screen rather than the bottom one will head us towards different Activity categories. Also, knowing that the middle section is the one that hosts the most of an app's content, we can try to count the number of elements with a specific attribute's value only in this area, in order to avoid losing focus on the top and bottom parts, which might hold redundant or not significant information.

We can simply look at Figure 16 to see a graphical representation of the three areas on a 1920x1080 pixels screen: it is easy to notice how the bottom part comprises two different navigation bars in this case, and the upper section includes the action bar plus a "Shuffle Button", while the main content of the application is reserved for the middle section.

4.2.3 Selecting UI elements

We now need to understand which elements to split among the three sections of the screen, and which ones to count just once. An interesting approach is the one followed by AppFlow [12], which simply traverses the UI hierarchy in pre-order transforming the whole set of UI

---

[12]https://developer.android.com/guide/navigation/navigation-ui

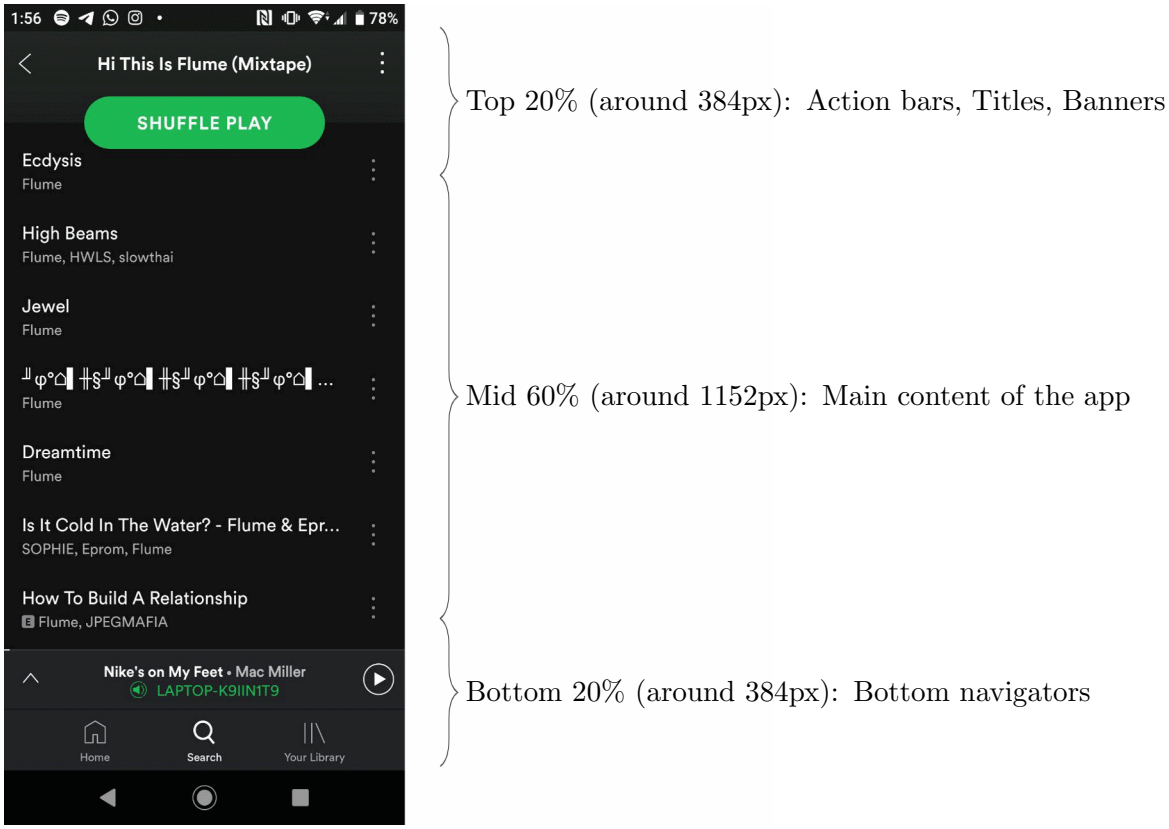


Figure 16: Sectioning of a 1920 x 1080 pixels screen into Three Areas.

elements plus their attributes in just one string. This text string together with some class and Optical Character Recognition (OCR) information is used to train a Neural Network. Rosenfeld, Kardashov and Zang [11] decided to use mainly counters for Clickable, Swipeable and Text field elements. We decided to extend this second approach.

In Table V it is possible to have a look at the first set of features that was used in our classification process:

TABLE V: FIRST SET OF FEATURES USED FOR ACTIVITY CLASSIFICATION

Feature Name	3 Screen Sections Separation	Number of Features
Number of Clickable elements	YES	3
Number of Swipeable elements	YES	3
Number of EditText elements	YES	3
Number of Long-Clickable elements	YES	3
Number of Password elements	NO	1
Number of Checkable elements	NO	1
Presence of a side Drawer	NO	1
Number of UI elements on screen	NO	1
		Total Features: 16

- The counter for **Clickable elements** is a great indication of the number of UI elements on the screen that can interact with the user: buttons, text boxes, list members and so on. For this very reason, it seems wise to separate the counter for the three sections of the screen so we can identify activities that require inputs in specific areas of the screen more than another. A great number of this type of element might indicate a settings or list screen, while a very small counter value may refer to map or login screen (where the user is prompted for fewer, more focused inputs).
- The number of **Swipeable elements** will obviously hint towards a very long (or wide) screen that the user needs to scroll. This might mean that we are in front of a list screen, a picture gallery, or even a messages activity and will cross out the possibility of being in an advertisement or login screen (which the user generally can't swipe).

- The number of **Edit Text boxes** is an essential element, as it might indicate a search bar for a browser or map (if located in the top part of the screen), or a login prompt (if located in the central section). Also, the value of this counter is very useful, as very few activities will appear with more than one Edit text element on the same screen, and when that happens it is generally a login or signup (having one box for username and one for password).
- The counter for **Long-Clickable elements** is used to identify the few elements on screen that allow the user to have a secondary form of input over an UI widget. This generally happens in a list or messages activity, where by long clicking a composing element, a spinner selection menu can pop up (to save, modify or delete something).
- The counter for **Password elements** is a very specific yet fundamental indicator: it is an Attribute that simply hides the already inserted text from the user for safety purposes so that no one can see it; if the value of this counter is different from zero, we can be pretty sure that we are currently in front of a login activity, as normally no other activities will set this attribute to `True`.
- A very similar consideration can also be made for the counter of **Checkable elements**: these are pretty rare, and when they appear they are mostly belonging to check-boxes in settings and to-do lists.
- The **Side Drawer** value is more of a binary indicator than an actual counter, as no more than one lateral drawer can be present in the same activity. This drawer generally

contains additional selection menus for the app, and they can appear in a wide variety of situations and screens.

- The **Total Number of UI elements on screen** takes into consideration even UI elements that are not directly visible to the user. Therefore, a very high value of this counter might indicate the presence of many UI widgets, or simply a very complex layout structure behind the current activity. A low value (and thus, a simple layout) can be found in browsers, portal screens and advertisements; on the other hand, a settings or list screen might contain hundreds of elements depending one from the other (we might think of the ListView structures as an example).

Also, in the first tests we have kept a *snapshot* of the user interface for each data point in order to do some experiments with a convolutional network to see whether image recognition could help us in the activity classification process. The screenshot can easily be captured through ADB[13] using UI Automator with its function `UiDevice.takeScreenshot(File)`.

We've also tried to *modify the features* used in the classification process in order to check whether this could yield a performance improvement or not. To do so, we have retrieved the features shown in Table VI to do some other experiments: while many features remained untouched, we have started to use one only counter for both the number of swipeable elements and number of clickable elements; this is because many times the top and bottom sections of the screen had none of those, meaning that the only counter with values different from zero was

[13]Android Debug Bridge - https://developer.android.com/studio/command-line/adb

TABLE VI: SECOND SET OF FEATURES USED FOR ACTIVITY CLASSIFICATION

Feature Name	3 Screen Sections Separation	Number of Features
Number of Clickable elements	YES	3
Number of Swipeable elements	NO	1
Number of EditText elements	YES	3
Number of Long-Clickable elements	NO	1
Number of Focusable elements	YES	3
Number of ImageViews	NO	1
Number of Password elements	NO	1
Number of Checkable elements	NO	1
Presence of a side Drawer	NO	1
Number of UI elements on screen	NO	1
		Total Features: 16

often the one belonging to the middle area of the screen. Therefore, we have tried to merge all three counters in just one to see what might happen. Also, some new features were inserted:

- First of all, we have tried to include separately for the three sections of the screen the **Number of Focusable elements**. The "focusable" attribute[14] in Android is a property that allows some specific UI elements to be selected while using a controller or keyboard instead of the usual touch screen input. This implies that every element on the screen that the user is supposed to interact with, is a focusable element. A map, browser or login activity will have not so many of those (requiring the user for very simple input actions that are generally on rails), while on the other hand a portal or settings screen will have

[14]Support keyboard navigation - https://developer.android.com/training/keyboard-input/navigation

lots of different icons or buttons that the user might select. It also made sense to split the counter into three for the different screen sections, as it is common to find interactive elements in both the top and bottom navigation drawers of many apps.

- Another new counter that has been introduced is the one for the **Number of ImageViews** that are present in the whole screen. In Android apps, ImageViews host image files, and therefore does not make sense to count their number in the top and bottom sections of the screen (as they are very small areas generally not including any image besides logos). However, they might be a good indication of being in a chat or in a portal activity, as those might often include images.

In both cases we end up with a features vector composed of *16 integer positive values*. This array is fed to a classifier that is supposed to find some kind of pattern in those numbers and determine the category of Activity in which we are in.

4.3 Composing the Dataset

We now need to compose our Dataset. Ideally, we would have a set of Activities' structures, each labeled with its category name. Unfortunately, it was not possible to find such a dataset on the internet at the time of this writing. Thus, it was necessary to *create a set of labeled Android Activities on our own*. The first step concerns deciding which are the possible categories (or labels) that we may assign to our activities. As a starting point, we have adopted some of the classes proposed by Rosenfeld, Kardashov and Zang [11] in their Activity classification work, while integrating them with some others that are new. Our aim is to find a set of categories that are different enough one from each other, so that the classifier doesn't get confused: letting

list and email categories coexist might not be wise, as even a human might be confused in classifying them. Each category must be a common type of activity that can be found in a big number of applications and that, at the same time, must have some kind of easily identifiable characteristic that makes it unique. This is equivalent of having a 16-dimensional space (as 16 is the number of features) where we are trying to have clusters of points being the most separate from each other, so that they do not overlap excessively. Trying to respect those conditions, we ended up with the following *categories of Activity*:

- **Advertisement activity**: As shown in a brief study by M. E. Gordon, PhD [15] during the years the number of applications available for free on the major App Stores has risen up to the point of covering over 90% of all apps available to download. Considering that fewer and fewer users are willing to spend money on mobile applications, developers are shifting to an ad-revenue model by placing banners and full-screen advertisements on their programs. This implies that a vast majority of this free applications will have ads popping up at anytime in sections of the app that are difficult to foresee. Also, touching those banners will take the user out of the app, making the testing process even more difficult. For this reason, we have decided to consider in this category advertisements that are full-screen and that need to be closed by the user by pressing a specific area.
- **Login activity**: Pretty much every application that is able to connect to a server or some sort of online service will prompt the user for some credentials (User-name or e-

---

[15] History of App Pricing, And Why Most Apps Are Free - https://www.flurry.com/post/115189750715/the-history-of-app-pricing-and-why-most-apps-are

mail and password) in a login form. The nice aspect about those forms is that they are implemented in the same way in nearly all cases: using the Android layout framework, they are composed of a standard EditText box for the user-name/e-mail, and another EditText for the password field that will most likely have the `isPassword` attribute field set to `True`, and a button to proceed. Therefore, this category of activity will often be characterized by the presence of 2 EditText elements with one of them having the password attribute set to true. Being very distinct and easily identifiable, this is the ideal class of activity.

- **Portal activity**: Most modern apps have a "*hub*" screen as the main section of the app, in which the main news, info or elements are presented in a user-friendly way. Many times this coincides with the home screen of the application (the first one that is shown after the app is fired). This type of activity is extremely common in news, music and audio apps, but often appears in a big variety of contexts. The main characteristic of this activity class is that it generally features at least one swipeable element (if not two, for vertical and horizontal scrolling) and many ImageViews that are composing icons shown to the user. Therefore, it seems to be easily identifiable as well.

- **List activity**: With this category we refer to a set of activities that are composed by a list of elements that the user can select. Those might be settings, menus or even selection screens. Generally, developers implement those using the `ListView` layout, as it is the most easy, convenient and elegant way to implement a list. Therefore, just by looking at the layout's class name we might try to infer something. Also, in the case of settings, we

often find *Switches*[16] or *Toggle Buttons*[17] that might be used to activate or deactivate a certain option: they are rarely found elsewhere, so they might help in distinguishing this activity class as well.

- **To-Do activity**: Some applications allow the user to create and modify a list of tasks in specific areas. In those activities, we might find many checkable elements that the user can tick in order to show that a specific task has been completed, or insert a new check box indicating something new has yet to be done. This category of activity is easily identifiable by the big number of `CheckBox` elements, which often do not appear in big chunks among other screens.
- **Browser activity**: As the name suggests, with this category we are comprising many apps where there is a specific screen used for browsing the internet. The structure of this activity is pretty much always the same, being composed of a web-view (whose duty is to render the web page) in its central section, and a search bar in the upper part (together with an "options" button in some cases). Therefore, even the layout of this activity class is very distinct.
- **Map activity**: In a similar fashion to what happens with the browser screen, many apps also have a map screen that is used to show the location of a specific point of interest. This activity often features a very complex layout due to the high number of icons, names

---

[16] https://developer.android.com/reference/android/widget/Switch

[17] https://developer.android.com/reference/android/widget/ToggleButton

and figures that have to be shown on the screen at any time. Not many types of activities in Android show this level of complexity in their layouts, so we might be pushed towards this kind of activity when we see that a screen is not featuring a simple structure.

- **Messages activity**: This category of Activity is very common among *Communication apps* such as Messaging and Social-Network apps. But often we find it even in shopping, transport and commercial apps due to the always greater presence of chat-bots[18]. Anyways, independently from the specific purpose of a chat screen, they are all structured in very similar ways: we can find a list of TextView elements in the central part of the screen representing the set of messages that was exchanged previously by the users, some sort of title or contact name on the top of the screen, and an EditText bar on the bottom that is used by the user to input the message that has to be sent by pressing a "*send*" button often located on its right side. Therefore, even this category of activity seems pretty unique in its structure.

Once the categories are decided, it's just a matter of creating a hand-crafted dataset. In order to do so, we have crawled for **70 applications** on the Apk Pure Store[19] and we have gone through most of the activities contained therein. For each of those activities, we have stored a dump of the screen, a screenshot and we have manually given a label according to the most plausible category among the ones just described. From the dump, we have extracted

---

[18]6 Critical Chatbot Statistics for 2018 - https://www.convinceandconvert.com/digital-marketing/6-critical-chatbot-statistics-for-2018/

[19]APK Pure Store - https://apkpure.com/

TABLE VII: COMPOSITION OF THE ACTIVITY CLASSIFICATION DATASET

Activity Category	Number of Labeled Samples
To-Do Activity	12
Advertisement Activity	13
Login Activity	12
List Activity	13
Portal Activity	13
Browser Activity	12
Map Activity	12
Messages Activity	13
	Total Samples: 100

a features vector composed of the 16 values previously shown to be used by the classificator. In the end, we've come to have **100 Labeled Activities**, equal to 100 Labeled Vectors of 16 values each split among **8 Activity Categories**.

In Table VII we can see the complete composition of the dataset: on average, we have around 12-13 sample Vectors per Activity Category. This might seem a very small number, but as it will be shown in the Results section of this chapter, it is enough to yield good classification performances.

4.4 Analyzing Effectiveness of ML Algorithms

Now that we have the structure of our features vector and a dataset with its activity categories, we just miss finding the right classifier. In order to do so, we have gone through many of the most popular machine learning algorithms, and by doing some measurements we have determined which of those are best suited for our activity classification problem.

4.4.1 Model Evaluation Techniques

For every model that is considered, some fundamental steps consist in the validation procedure in order to determine the best set of hyper-parameters, and the final evaluation of the model itself.

In each case, we have decided to use **10% of the whole data as testing set**: choosing a greater percentage such as 20% might yield a more solid performance index of our model, but considering that our dataset is very small (composed of only 100 samples), increasing the size of the test set will determine a notable decrease in performance due to the smaller size of the training set. Thus, 10% seems a good compromise between a reliable performance index and an acceptable Training process.

Regarding the evaluation techniques, we have decided to use 3 approaches that provide *gradually more conservative results*:

- **Hold-Out**: after having separated the Test Set, we perform a Stratified 5-Fold Cross Validation (such that each fold is a good representation of the proportions of the whole dataset) on the remaining 90% of the data in order to determine the right set of hyper-parameter that we will use in the final evaluation on the test set. Considering that Hold-Out is a technique that yields results very relative to the set of samples used as test set, their values might vary a lot from one execution to the other. Therefore, in order to have a bit more reliable results, we have decided to use their *Average over 20 executions of the Hold-Out procedure*, where each time the 10% of data adopted as test set is randomly chosen.

- **Nested Cross Validation**: in this case, two nested loops are performed on the model. The *outer loop* can be considered a 10-Fold Cross Test procedure, where in each one of the 10 iterations, we choose one of the 10 folds as a test set, and the remaining 9 as Training Set. The *inner loop* is a Stratified 5-Fold Cross Validation performed on the 9 training folds of the outer loop. The best set of hyper-parameters found during the inner validation procedure is used to in the testing procedure in the outer loop. However, having 10 outer loop iterations, we will end up with 10 distinct models, each with different hyper-parameters. Therefore, this procedure does not yield the 'ideal' set of hyper-parameters to use as it might happen in Hold-Out, but simply gives us an indicative reliable index of performance and stability of the model when trained from zero. Also, what makes this technique dependable is that the each element of the dataset is used for Testing purposes exactly once.

- **Leave One Out**: this can be considered as a Nested Cross Validation taken to the extreme, where in the outer loop we use as testing set one data sample at a time (in a *100-Fold fashion* in our case), while the inner loop remains the usual 5-Fold Cross Validation. The strong point of this approach is that we cross out the variable of how we decided to split the dataset into K folds, as this time we have a data point granularity. Therefore, this measurement technique has a lower variance than the other methods, even though it requires much more time to execute due to the way higher number of iterations.

As already stated, we decided to use those three measurement techniques together in order to have *gradually more conservative and comprehensive performance indexes.*

We have adopted a **Grid Search** approach for the hyper-parameters optimization. This consists of an exhaustive search through every possible combination of manually-selected values belonging to a set of hyper-parameters specific to the model that is being analyzed. This procedure is executed with the aid of the already-stated 5-Fold Cross Validation. Also, the Grid Search technique is guided by a specified metric that we try to optimize: in our case we try to optimize performances in terms of accuracy.

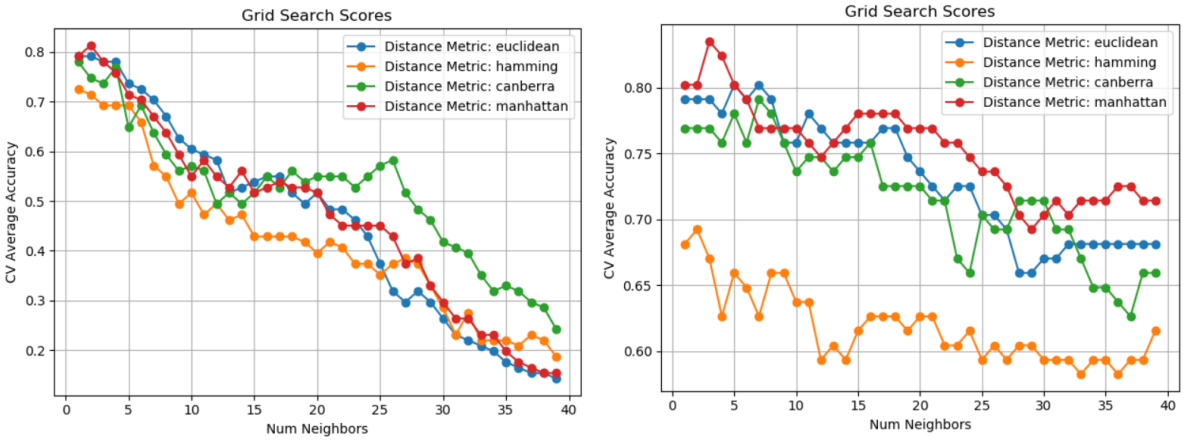
4.4.2 K-Nearest Neighbors

K-Nearest Neighbors (KNN) is one of the most common and simple Machine Learning algorithms; it is used for a very wide range of domains, and can be used for either classification and regression problems.

Given a space of labeled data points and a new element yet to be classified, this algorithm simply calculates the distance between this new point and all the others according to some *distance metric*. Then, the K closest data points according to this metric are kept in consideration, and in case of regression problems we will use the *mean* of the K closest labels as the label for the new data point; instead, in classification problems such as ours, the assigned label will be the *mode* of the K closest labels.

Batista and Silva in [19] explain in details how the hyper-parameters that influence KNN the most are the *Number of Neighbors K*, the *Distance Function* and the *Weighting Function*, which is used among the K neighbors to penalize in the voting procedure the data points that are the furthest for the point yet to be labeled.

The first hyper-parameter we can focus on is the **number of neighbors K**: having ordered all the data samples closest to furthest from our new unknown data point, this parameter is referred to the number of elements to pick from the top of that list that will vote to decide on the final label. A very small value of K means that noise will have a great impact on our results, as only few of the closest data points will have a say, while a great value of K will determine a slow execution of the algorithm and a more confused classification (if uniform weighting is adopted, meaning that the furthest point's opinion will have the same value as the closest). In Figure 18 we have plotted the average accuracy value of a 5-Fold Cross Validation procedure, according to both the number of neighbors (shown on the X axis), and the distance metrics (shown via different colors). In the first picture to the left, those measurements have been executed with a Uniform Weighting of the neighbors. For this reason, independently of the metric chosen, the validation accuracy tends to drop in a nearly linear way with the increase of the number of neighbors. Therefore, in this case a small value of K is what would make more sense to preserve accuracy. As suggested by S. A. Dudani [20], we can try to give less importance to further votes by weighting them by the inverse of the distance from the unlabeled data point (**Inverse Weighting**), and the results are shown in the right hand picture of Figure 18: it is clear how the Accuracy decreases way less with an higher value of K (we now just have a drop of 20% accuracy going from 1 to 40 neighbors, against the previous drop of 60%), but still, a small K value of around 5 yields the best performance. Therefore, a K value of 5 and inverse weighting might seem the best set of hyper-parameters to proceed.



Uniform Weight. Inverse Weight.

Figure 18: Effect of number of neighbors and distance metrics on KNN's validation accuracy.

The next step is to identify the best **Distance Metric** for our purposes, which can still be seen in Figure 18 with the different colors. Using the *Euclidean distance*, which is really the Minkowski metric ($\sum_{i=1}^{N}(|x_i - y_i|^p)^{(1/p)}$, where $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$ are two feature vectors) with the p parameter equal to 2, we have in both charts (uniform and inverse weighting) really good results, mainly with lower values of K. The *Hamming distance* ($N\_unequal(X, Y)/N\_tot$), which is specific for integer-valued vector spaces, does not yield the best performance, as its plots show in both cases the lowest accuracies for the whole range of K values. The *Canberra metric* ($\sum_{i=1}^{N}(|x_i - y_i|/(|x_i| + |y_i|))$), also specific for integers, gives average performance, but with uniform weighting we can see a spike of Accuracy for big numbers of neighbors. Finally, using the *Manhattan distance* ($\sum_{i=1}^{N}(|x_i - y_i|)$) we have again

TABLE VIII: RESULTS USING THE K-NEAREST NEIGHBORS ALGORITHM

Feature Set	Metric	Hold-Out Avg	10-Fold	Leave One Out
First set of Features	Accuracy	81.10%	76.06%	73.00%
	Precision	91.60%	87.70%	73.00%
	Recall	87.60%	82.00%	73.00%
Second set of Features	Accuracy	85.00%	82.90%	79.00%
	Precision	94.29%	93.70%	79.00%
	Recall	91.25%	89.37%	79.00%

great results, comparable with the Euclidean one, being towards the top of both charts and mainly with inverse weighting. Therefore, after all those considerations, it seems wise to keep either the Euclidean or Manhattan distance metrics.

Interestingly enough, we ended up with a selection of hyper-parameters that was suggested by Batista and Silva in [19]: we have a **K value of around 5**, an **Euclidean distance metric**, and an **inverse weighting** as one of the best performing hyper-parameters set. In Table VIII it's clear that the values of accuracy for both the first and second set of features seem to be high enough, reaching 76.06% and 82.90% respectively with K-Fold Nested Cross Validation. We have reported measures for an average of 20 Hold-out measures, 10 Fold and Leave One Out techniques, so that we have gradually stricter measurements that are always less dependent on the split of the dataset in test and training partitions, arriving to the point of having tried to evaluate one data point at a time in the Leave One Out. Even in this case the performance

is notable, reaching 79% of accuracy. It is evident then how the second type of feature vector really helps this classifier.

4.4.3 Decision Trees

The **Decision Tree Learning** algorithm is a very intuitive method that can be used for either classification or regression problems. In our case (classification), the tree tries to classify an unlabeled data point by making some decisions governed by the value of some of the features constituting that point. More specifically, each *interior node* of the tree is referred to a specific feature of our Features Vector, and its duty is to split data according to its value. Therefore, each *edge* coming out of it represents a set of values for a feature that respect a specific condition (for example, values that are grater, lesser or equal to a specific number). Obviously, a node has to manage all possible values that might come to it from its specific assigned feature. A *leaf* represents a label to be given to the data point: therefore, if we follow the tree from the root down to a leaf, its Features' values will guide a path towards a specific label.

Building a Decision Tree is a *recursive process*: we start from the root, where we can choose to divide our data according to the whole set of 16 features (in our case). We have to choose the feature that grants the least loss of accuracy: ideally we would choose a feature that could split data into two uniformly labeled groups; but in reality, to do this, we have to choose one feature at the time, split the data, and calculate the **Gini index** ($\sum_{i=1}^{NumLabels} p_i(1-p_i)$ where p_i is the fraction of items with label i in the set), which gives us an index of impurity for the just generated set. This index reaches 0 when all cases in the node fall into a single category. Having chosen the feature to split on using the Gini index, we keep splitting recursively on the

children nodes. We also need to know when to stop generating our tree. The options could be to set a maximum depth for the tree, or a minimum number of data points that must fall in each leaf.

Therefore, one of the first hyper-parameters that we have to tune[20] is the **Maximum Depth** allowed to the tree: a tree that is too complex can lead to overfitting, while a tree that is too shallow could not let the data express its articulation. In Figure 19 we have plotted the effect that this parameter has on the validation accuracy: we notice that with small values of depth, accuracy is extremely low; however, it is able to increase together with the number of levels of the tree, up to about a depth 8, where validation accuracy is maxed out. Thus it is clear that smaller values of maximum depth can harm the ability of the model to adapt to the complexity of the data.

In the same chart of Figure 19, we can see some variations of another hyper-parameter that we have the possibility to tune: the **Minimum Number of samples required to split an Internal Node**, which indicates how many data points need to be available to allow the splitting of a specific node (thus, becoming an internal node). This parameter controls the rigidity of the model, as a very big value will cause us to consider many samples at each node, and a small one will determine too specific decision processes. We have tried changing its value, showing the effect on the chart: it is immediately evident how in our case, having a value as small as 2 helps maxing out the accuracy, while if we increase the number of Minimum Samples,

---

[20] In depth parameter tuning for Decision Tree - https://medium.com/@mohtedibf/indepth-parameter-tuning-for-decision-tree-6753118a03c3

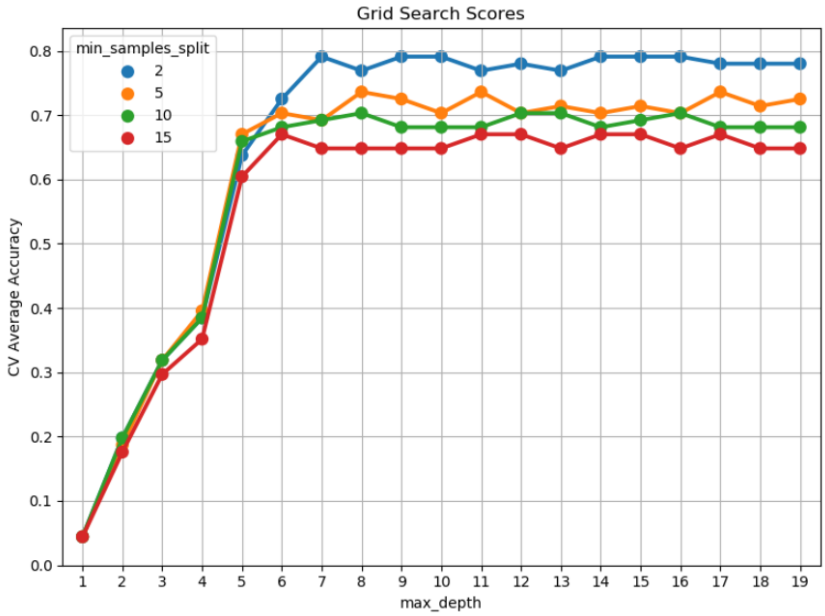


Figure 19: Effect of Maximum Depth and Minimum Samples per Internal Node on Decision Tree's Validation Accuracy.

we can see the plot starting to fall in terms of accuracy performance. The chart shows that increasing the threshold of samples of a value of 10, determines a 10% reduction in terms of accuracy. Therefore, *small values are ideal for our problem.* However, independent from the Minimum Number of Samples required to split an internal node, all curves stabilize at less than 10 levels of tree depth.

Next step is determining the **Minimum Number of samples required for a Leaf Node**. Similar to what we have already discussed for the previous parameter, a split point will only be considered if it leaves at least this minimum value of samples in both the left and right branches. In other words, this value indicates how many data points are necessary at minimum

in a node for it to be considered and external node (a leaf). The chart is not reported as we have pretty much the same identical situation as what happened in Figure 19 for the internal nodes. Even in this case, incrementing the samples at leaves from 2 up to 15 shows a decline in terms of validation accuracy, indicating that a small value of samples is ideal here too. All those observations tell us that our problem is so articulated that the decision tree model tries to adapt to it the most by using very sensible splittings and specific feature conditions at every node.

However, if we are not careful enough, we could end up overfitting by using too small values; in order to avoid this effect, there is another hyper-parameter that we can tune: the **Maximum Number of Features to be considered at each node**. If we suppose to set this parameter to 8, we have to choose at each node 8 random features out of the 16 (which is for us the total number of features available), and we select the best one among those according to the Gini index. We've tried to move the value of this parameter from 2 to 16, but it was not possible to notice an impactful effect on the accuracy of the system, unless its value is as small as 2 or 3: in those cases, accuracy is reduced due to the small selection of features that our model has at every node. For this reason, a great compromise determined by the grid search tuning procedure is to set it at around 10-12 features in order to still maintain the deterministic element of actively choosing a good feature to split on.

Having modeled this classifier after the observation that we've just made, we obtain the results shown in Table IX: values of accuracy, precision and recall seem to be quite high, and they do not change too much between the two sets of features. This similarity can be seen

TABLE IX: RESULTS USING THE DECISION TREE LEARNING ALGORITHM

Feature Set	Metric	Hold-Out Avg	10-Fold	Leave One Out
First set of Features	Accuracy	83.00%	82.02%	81.00%
	Precision	88.00%	92.30%	81.00%
	Recall	93.40%	87.50%	81.00%
Second set of Features	Accuracy	87.22%	83.03%	80.00%
	Precision	91.88%	93.77%	80.00%
	Recall	95.40%	91.02%	80.00%

particularly while measuring with the Nested K-Fold and Leave One Out methods, which are more conservative measurements in comparison with the simple average of 20 Hold-out runs. Reaching an **Accuracy of 80-81% with LOO**, this method has a performance comparable with the one of KNN with the second feature set. Also, it is notable the peak of accuracy of 87.22% using a simple Hold-out average.

Also, in Figure 20 it is possible to see a graphical representation of an instance of a Decision Tree with the hyper-parameters set as previously discussed. It is possible to see the feature name and Value Condition for each intermediate node, together with the Gini index score and the number of samples that have come down to a specific node. We can also see the presence of 8 tuples (which correspond to the 8 possible labels that we have): each one of those has on the left the number of samples not belonging to it, and on the right the number of data points actually belonging to its class. Thus, we can notice how some leaves have a Gini index value of 0, having all samples classified uniformly into only one category.

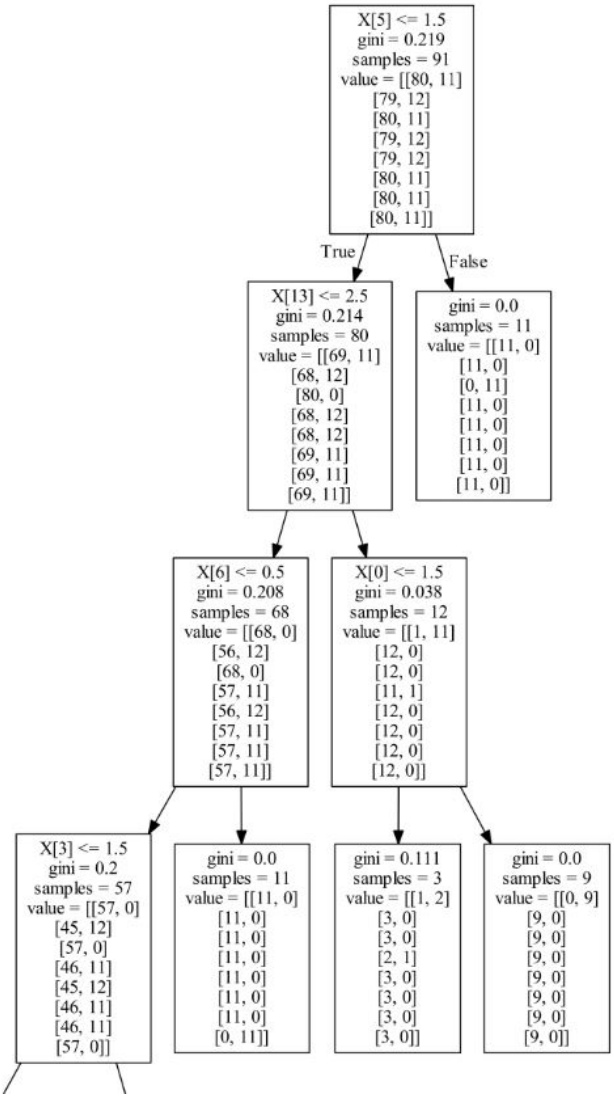


Figure 20: Sample section visualization of a Decision Tree resulting from a training process.

4.4.4 Random Forest

In 2001 Breiman introduced the concept of Random Forests [21]: as the name suggests, they are simply an *ensemble* of Decision Trees. Each one of those predicts a label for our unknown data point, and in broad terms we determine the definitive label as the one that has the most votes. As high-lined by a "Towards Data Science" article[21], the key point of this classifier is that having a crowd of trees that are not correlated among them, their vote as a group can be more reliable than the vote given by the best of those models alone. This is because it is plausible for a tree to make a mistake, but this will be compensated by the set of remaining trees, which will probably make correct predictions (or if they make mistakes, they probably will not bring to the same wrong label predicted by the original tree).

The element that characterizes those trees and makes them less correlated is the fact that at every node, instead of choosing the best feature to split on among the N total features, the model *randomly* chooses a subset of K features ($K<N$), and only then the best feature is chosen among the K available.

The parameters that we can play with are mainly the *Maximum Number of features to consider at a node*, the *Number of Estimators* and the *Minimum Samples at leaves*: regarding the **Maximum Number of Features to Consider at a Node**, the points made for the Decision Tree are still valid; thus, this parameter mainly aims at reducing overfitting and similarities among the forest of trees (by making the feature selection process a bit less deterministic) and

---

[21] Understanding Random forest - https://towardsdatascience.com/understanding-random-forest-58381e0602d2

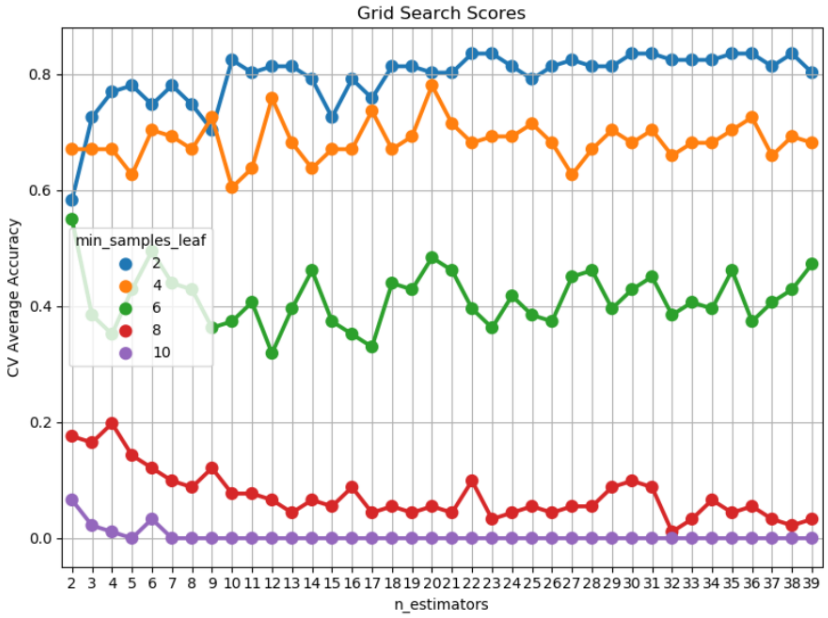


Figure 21: Validation accuracy of Random Forest according to number of trees and minimum samples at leaves.

does not have an effect over the accuracy of the model. By making some experiments, the best results are given when this parameter is in the middle of its possible range.

On the other hand, a parameter that is exclusive to the Random Forest classifier is the **Number of Estimators (Trees)**: this value indicates the number of Decision Trees that the model has to build before taking the number of votes for each label and therefore deciding the final classification. Having a small number of estimators will hurt the performance of the system, as the vote of a mistaken tree might be valued more; on the other hand, having a very wide selection of trees will make the classification process more reliable but will obviously slow down the execution time. We can see this behavior in Figure 21, where it is possible to notice

TABLE X: RESULTS USING THE RANDOM FOREST LEARNING ALGORITHM

Feature Set	Metric	Hold-Out Avg	10-Fold	Leave One Out
First set of Features	Accuracy	88.88%	86.96%	85.00%
	Precision	94.40%	93.40%	85.00%
	Recall	89.20%	89.50%	85.00%
Second set of Features	Accuracy	88.89%	87.98%	87.00%
	Precision	95.25%	95.67%	87.00%
	Recall	95.49%	92.26%	87.00%

how the performance in terms of accuracy become a little bit more stable with the increase of the number of estimators.

In the same chart, we can also notice five different colored curves that correspond to that many values of **Minimum samples at leaves**, which is a hyper-parameter that we've already encountered and analyzed for Decision Trees: it indicates the minimum number of data points that need to present at a node for it to be considered a leaf. Just as expected, we can notice a similar behavior as in the Decision Trees, where the highest performance is obtained with just 2 minimum samples per leaf. A curious observation is that using a small number of samples at leaves such as 2, 4 or 6, the accuracy tends to increase with greater numbers of estimators. On the opposite side, with more samples required at leaves the accuracy starts to drop while augmenting the number of trees. This is because trees have to respect stricter rules at their leaves, and thus they tend to be more similar one to the other and are less flexible to the variability of our data. Also, a smaller value lets the trees to be more articulated and adaptive.

In Table X we have reported the results obtained with this model: we've reached an improvement of 2% in terms of Accuracy (but the same goes also for Precision and Recall) if we make a comparison with the standard Decision Tree classifier. This confirms the idea that having multiple estimators can stabilize the system, and errors of one single tree can be covered by the others. Also, the second set of Features performs again better than the first one: this can be mainly seen in the Leave One Out measurements, which can be compared to a 100-fold cross validation, where we always obtain the same results independently from the number of executions; here we reach an extremely high value of **87% of Accuracy**, which is the *highest obtained up to now* (the first set of features is 2 percentage points below, at 85%).

4.4.5 Support Vector Machine

Another classifier that we've tried is the **Support Vector Machine (SVM)**: this is a Supervised Learning method that can be used for both Classification and Regression problems. The basic idea behind it is to find the best hyperplane that separates the data points into classes in their features-dimensional space. We might have many possible hyper-planes, thus the best one is considered to be the one which maximizes the distance from all the data points. What makes this classifier so powerful is that even in the case in which it seems we are not able to split our data, it performs some transformations to add new dimensions, in which a possible definite data separation might be found: as explained in a good example by S. Patel[22], if we have a two-dimensional space with X and Y axis populated with data points that seem

[22]Theory of SVMs - https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72

inseparable, we might add a third dimension with the Z axis where a possible hyperplane can be found.

Therefore, the first and most essential parameter is the **kernel function**: this determines whether we are using a linear or non-linear separation (hyperplane) of our data. Starting from the simplest case, we have tested the *linear kernel*: as the name suggests, the classifier tries to find the ideal linear hyperplane, but this also depends on the **regularization parameter C**, which is a kind of penalty coefficient that indicates how much we are willing to avoid mis-classification of data points. On one hand, for small values of C, the model will look for larger margin hyperplanes, meaning that it will prefer to find a smooth separation of data over correctly classifying all the data points. On the other hand, a big value of this parameter will move the model towards hyperplanes with a smaller margin, preferring to avoid mis-classification totally; however, this will lead to obvious overfitting. In Figure 22 we have plotted the effect of this regularization parameter on all the kernel types that we've tested. The values of C have been presented on a logarithmic scale in order to show a wide a comprehensive behaviour. In the case of linear kernel, we can notice the just described behavior: very small values hurt the correct classification of data points, while at a value of around 1, the validation accuracy of the model is maxed out, and after that it starts to drop a bit as a possible sign of overfitting.

In order to also try some *non-linear kernels*: we started with the *Radial Basis Function (RBF) Kernel*. A kernel is a function $k(x, y)$ that corresponds to a dot product between the x and y vectors: $k(x, y) = \varphi(x)^T \varphi(y)$, where φ is a mapping $\mathbb{R}^n \to \mathbb{R}^m$ that brings our vectors to some other feature space. Therefore, we are able to compute dot products in a space

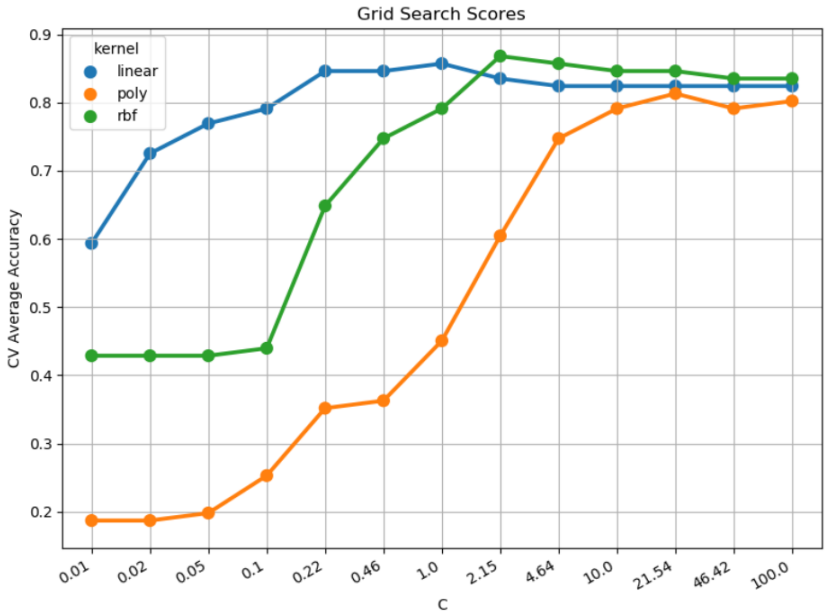


Figure 22: Modifying the Normalization Parameter (C) for Linear, RBF and Polynomial kernels of SVM.

without even knowing it (nor the mapping φ). The Support Vectors are the data points that define the hyperplane, and are referred as x_i, while x is the input vector. Having made all the necessary considerations, the RBF kernel is defined by the equation $k(x, x_i) = (-\frac{\|x-x_i\|^2}{2\sigma^2})$, where $\| x - x_i \|^2$ is the square of the Euclidean distance between the two feature vectors, and σ is a free parameter.

The first hyper-parameter that we can tune is the *normalization C value*, which has already been explained, and whose behavior can be seen again in Figure 22: in this case, values of C under 0.1 seem to not even modify the performance in terms of accuracy, remaining constant for few measurements; however its growth peaks at a C value of 2, only to then diminish indicating

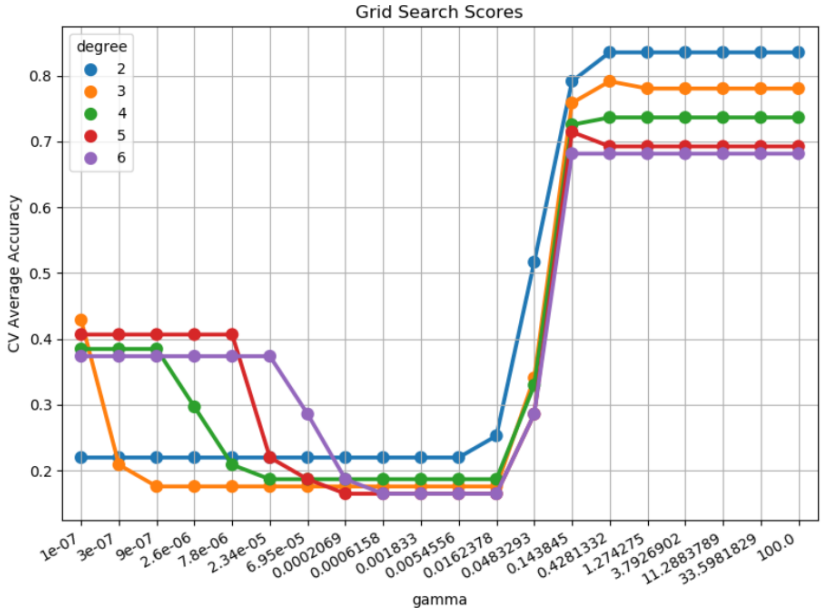


Figure 23: Validation Accuracy of SVM with Polynomial Kernel with different Polynomial Degrees and Gamma values.

again probable overfitting. Another parameter which is exclusive to non-linear kernels is the **Gamma value** ($\gamma = \frac{1}{2\sigma^2}$), which indicates the influence that the data points have on the calculation of the split; with a low value, points that are far away are still considered in the calculation, while with high values, only closer points are considered, leading to a separation that tries to perfectly fit the training data set.

Finally, we've tested the *Polynomial Kernel function* as well: it is characterized by the equation $k(x, x_i) = 1 + \sum_{i=1}^{N}(x * x_i)^d$, where d is the degree of the polynomial. Therefore, the most important parameter that we can tune in this case is the **degree of the polynomial**. In Figure 23 we can see how changing this parameter influences the accuracy of the SVM. For very

TABLE XI: RESULTS USING THE SUPPORT VECTOR MACHINE LEARNING ALGORITHM AND ITS KERNELS

Kernel Type	Feature Set	Metric	Hold-Out Avg	10-Fold	Leave One Out
Linear Kernel	Features Set 1	Accuracy	87.22%	83.33%	81.00%
		Precision	92.40%	91.10%	81.00%
		Recall	97.60%	92.40%	81.00%
	Features Set 2	Accuracy	90.00%	88.08%	84.00%
		Precision	93.92%	95.40%	84.00%
		Recall	98.75%	94.57%	84.00%
RBF Kernel	Features Set 1	Accuracy	86.20%	84.84%	84.00%
		Precision	91.90%	93.20%	84.00%
		Recall	96.90%	92.60%	84.00%
	Features Set 2	Accuracy	88.83%	87.17%	83.00%
		Precision	92.83%	94.82%	83.00%
		Recall	98.66%	95.67%	83.00%
Poly Kernel	Features Set 1	Accuracy	86.80%	85.01%	82.00%
		Precision	91.30%	93.80%	82.00%
		Recall	97.50%	91.50%	82.00%
	Features Set 2	Accuracy	90.01%	83.03%	81.00%
		Precision	93.78%	95.02%	81.00%
		Recall	98.12%	88.94%	81.00%

small gammas in the order of 10^{-5} or lower, performance in terms of accuracy is bad only to get worse up to when we set gamma to around 0.01: after that, all polynomial degrees see a big increment in accuracy, reaching its top value. This indicates that avoiding the consideration of very far data points is healthy for our model, which then is able to adapt better to the complexity of our problem. Also, it is evident how the lower is the degree of our polynomial

kernel, the better the performance is. Therefore, moving to a space of high dimensionality does not seem to help us, and a degree of just 2 is best.

However, as noted in Figure 22, the best overall performances in terms of accuracy are yielded by the linear and RBF kernels, leaving the polynomial one (even with a degree of just 2) as the least performing kernel.

In Table XI we can see the performance of the model in terms of accuracy, precision and recall for both the available feature sets, and for all three kernel types. On one hand, for the second features set the most suited kernel is the linear one, reaching an accuracy of 84% with Leave One Out and 88.08% with 10-Fold. On the other hand, the best performance for the first features set is given by the RBF kernel, also at 84% LOO. The polynomial kernel seems to have an average performance, without shining in both sets.

4.4.6 Naïve Bayes

A simple probabilistic classifier that we may use is Naïve Bayes: this uses the Bayes theorem together with Naïve Independence assumptions in order to solve classification problems. This classifier generally comes in three forms: *Multinomial Naïve Bayes*, which is used when the frequency count of some kind of feature (such as word counts, considering that it is mainly used in text classification problems) is to be considered; the *Bernoulli Naïve Bayes* is used when do not care about the occurrences, but simply about the presence/absence of a specific feature; finally, we have *Gaussian Naïve Bayes*, used when we have continuous data. In our case, having features to count, we will be using **Multinomial Naïve Bayes**. The model works in a pretty simple way, by calculating the prior probability of each class as the relative frequency of data

TABLE XII: RESULTS USING THE MULTINOMIAL NAÏVE BAYES ALGORITHM

Feature Set	Metric	Hold-Out Avg	10-Fold	Leave One Out
First set of Features	Accuracy	75.00%	72.92%	71.00%
	Precision	79.70%	89.10%	71.00%
	Recall	90.90%	78.40%	71.00%
Second set of Features	Accuracy	75.55%	67.87%	62.00%
	Precision	82.52%	87.34%	62.00%
	Recall	91.36%	82.83%	62.00%

points per class $\mathtt{prior}(\mathtt{c}) = \frac{\mathtt{N_c}}{\mathtt{N}}$ and also the conditional probability of a feature, given a class: $\mathtt{P}(\mathtt{i}|\mathtt{c}) = \frac{\mathtt{x_{ic}}}{\sum_{\mathtt{j}\in\mathtt{F}} \mathtt{x_{jc}}}$, where $\mathtt{x_{ic}}$ is the counter for the feature $\mathtt{i}$ in class $\mathtt{c}$, and $\mathtt{F}$ is the number of features. An important note to make is that a specific feature might not appear in a class, determining a counter of 0: this would wipe the entire calculation of the product necessary for the conditioned probability; to solve this problem we apply the so called "*smoothing*", consisting of the addition of α to the numerator, and $\alpha\mathtt{F}$ to the denominator, where $\alpha \geq 0$ (if set to 1, we have Laplacian smoothing). Finally, we can infer the label of our unknown data point as $\mathtt{y} = \mathtt{argmax}_{\mathtt{c_k}} \mathtt{P}(\mathtt{c_k}) \prod_{\mathtt{j}=1}^{\mathtt{F}} \mathtt{P}(\mathtt{i_j}|\mathtt{c_k})$.

In Table XII we can observe the results yielded by this algorithm: unfortunately, the *performance seems to be worse* than what we've achieved with other classifiers by tens of percentage points. We even drop to the 60% Accuracy area using the Leave One Out procedure. Also, another notable observation is that in this instance the first set of features performs better

than the second one, going against what seems a trend for our other classifiers to have better performances with the second one.

4.4.7 Logistic Regression

The **Logistic Regression** classifier is originally used in order to solve binary classification problems, where we just have two classes (such as succeeding/failing a test): the model is able to output a real value comprised between 0 and 1 indicating the certainty of belonging to the positive class (therefore, a value of 0 indicates the certainty of belonging to the negative class).

Now we can suppose to have a model with just two predictors[23] (x_1 and x_2) that are linearly dependent (log-odds $l = \beta_0 + \beta_1 x_1 + \beta_2 x_2$, where β_i are the parameters of the model): the log-odds formula is also $l = \log_e \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$, therefore we can remove the logarithm obtaining the odds $\frac{p}{1-p} = e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2}$. Finally, we can remodel this equation to obtain the Sigmoid function, which also indicates the probability of the positive label: $p_{Y=1}(X) = \frac{1}{1+e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$.

Once the model has been trained to find the most suited set of β_i parameters, we can simply feed the value of our feature observations and obtain as output the probability of the unknown data point of belonging to the positive class.

But obviously, in our case we are trying to solve a multi-class classification problem: we can adopt the **Multinomial Logistic Regression**. This classifier can be implemented in a One

---

[23] Logistic Regression - https://en.wikipedia.org/wiki/Logistic_regression

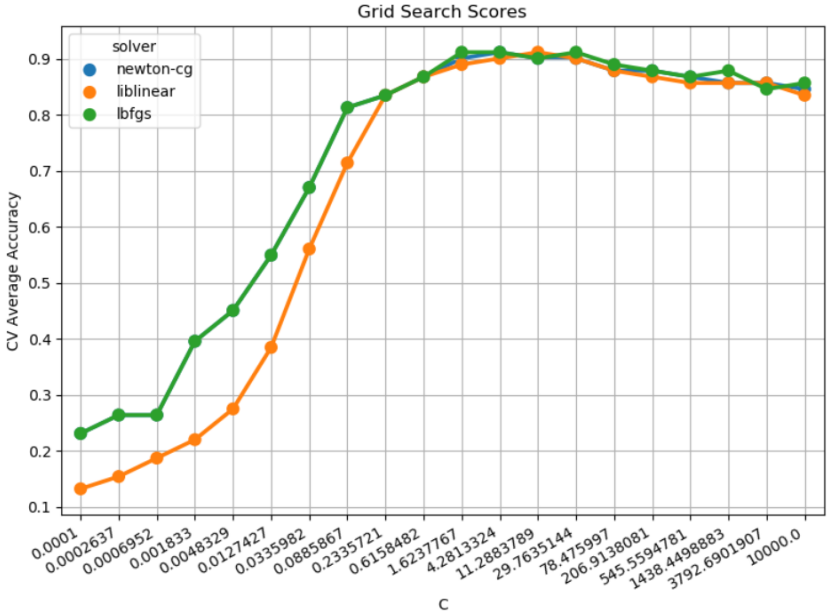


Figure 24: Effect of C Normalization parameter on Logistic Regression.

vs All fashion[24], by hypothesizing that all classes are independent of one another. Having 8 categories, we produce 8 binary classification problems where each of the 8 classes at a time is considered as the positive label, and all the others are considered as the negative label at once. Training those 8 problems separately, we will end up with that many scores corresponding to the probability of the input data of belonging to each category. We simply pick the class for which the Logistic Regression classifier has given the highest score.

---

[24]One Vs. Rest Logistic Regression - https://chrisalbon.com/machine_learning/logistic_regression/one-vs-rest_logistic_regression/

TABLE XIII: RESULTS USING THE LOGISTIC REGRESSION ALGORITHM

Feature Set	Metric	Hold-Out Avg	10-Fold	Leave One Out
First set of Features	Accuracy	88.13%	88.00%	85.00%
	Precision	91.90%	92.60%	85.00%
	Recall	96.90%	90.80%	85.00%
Second set of Features	Accuracy	94.44%	91.01%	89.00%
	Precision	96.35%	96.54%	89.00%
	Recall	98.66%	95.25%	89.00%

The most important parameter that we can tune in this case is the **C parameter**, which is the inverse of the regularization strength: for small values, the regularization will be strong, the model will be simpler, and we'll have risk of underfitting the data; high values will determine low regularization, meaning more complex models and risk of overfitting.

In Figure 24 we have plotted the effects on validation accuracy of moving the value of this parameter on a logarithmic scale. In the same chart we've also drawn the curves corresponding to three solvers used in the optimization problem consisting of finding the set of parameters (not hyper-parameters) of the model that yield the best performances. We can notice they differ only with C values smaller than 1, but after that they all perform in the same way. Speaking of the C hyper-parameter, we see that it yields the best accuracy for values comprised between 1 and 30: as described in the previous chapter, for smaller values, the model is too simple to fit the problem, but for bigger values, the curves start to drop as an effect of overfitting.

In Table XIII we report the results obtained with the Multinomial Logistic Regression classifier: the first thing that we can notice is how high the accuracy scores are in using this algorithm. The model even goes above 90% Accuracy, which is higher than all the other models tested up to this moment. Also, once again the second Set of Features outperforms the first one by many percentage points. The most notable measurements is the **91.01% accuracy using the 10-Fold Nested Cross Validation procedure**. Also, using a simple average of 20 Hold-Out Runs with a 10% Testing split randomly chosen each time, we get to 94.44%, even though this value could be a bit less reliable, being able to change substantially among one run and the other. Notable are also the precision and recall measurements, which are the *highest reached up to this moment.*

4.4.8 Convolutional Neural Network

One last experiment that has been made is using **Convolutional Neural Networks (CNNs)**: those are a class of Deep Neural Networks[25] that are regularized through the use of the convolution operation (in place of general matrix multiplication), that avoids the risk of overfitting typical of standard NNs due to gradually smaller patterns instead of the usual fully-connectedness. For that reason, this algorithm is popular in computer vision problems.

In our case, the idea was to use the a screenshot of the activity together with the usual features vector of 16 integer values. Therefore, the input is not only an image, and we need to make the architecture of the CNN a bit more complex. We created *two parallel networks*:

---

[25] Convolutional Neural Network - https://en.wikipedia.org/wiki/Convolutional_neural_network

on one side we have a **standard CNN** taking as input our screenshot, while on the other we have a **1-layer NN** taking as input the features vector; then, the **output of those two is concatenated** and used as input of a **third Dense Network**, which will eventually yield 8 outputs corresponding to the probability of belonging to our 8 categories.

The first Network is a Convolutional Neural Network and the inputs are 1920x1080 pixels screenshots. First of all we crop some pixels in order to remove the bottom Android OS action bar, and the top OS status bar, which are both graphical elements that will be the same in every Activity, therefore it is useless to consider those areas in the classification process. Then, the screenshot is converted to gray-scale in order to reduce the dimensionality of the image by removing the color channels. After a preliminary pooling operation, the input has a shape of (778, 486, 1). Finally, the image is actually fed to the network. The architecture that was chosen is a slightly simplified version of the so called *AlexNet*[22]. We have in order: a Convolutional layer (kernel 11x11, stride 4), Max Pooling (kernel 3x3, stride 2), Convolutional (kernel 5x5, stride 1), Max Pooling (kernel 3x3, stride 2), 3 Convolutional Layers (kernel 3x3, stride 1), Max Pooling (kernel 3x3, stride 2), and finally a Dense Relu Layer of 100 units. This output is then sent to the concatenation layer of the third network.

The second Network is extremely simple, as it is simply composed by one Relu layer that feeds the 16 inputs of the Features Vector to the concatenation layer.

The third Network has the duty of joining together the two previous models: their tensors are concatenated in just one input for our last model. After this, just one dense layer has been added (Relu Dense of 50 units), ending up in an output layer with a Softmax activation layer

that gives us the conditioned probability of each one of the 8 classes. This is just one of the many architectures that have been tried.

As loss function, the *Categorical Crossentropy* has been chosen, together with a *Stochastic gradient descent optimizer* with learning rate of 0.001.

Unfortunately, when training our model the system never converges, remaining stuck at around 16% training accuracy for even up to 100 epochs. Having about 988.000 parameters determining the behavior of our classifier, maybe we have too many variables with too little data points (we have to remember that we just have 100 labeled samples). Therefore, using the activity's screenshot to aid us in the classification process *did not have any positive effect* probably due to the lack of training data.

4.5 Results Analysis

Now that we've presented in details our approach to the problem of activity classification and its basic results using different machine learning algorithms, it is time to analyze some of their implications.

4.5.1 Comparing Feature Vectors

In Section 4.2.3 we have presented our approach at selecting an activity's set of features, and we've build two features vectors presented respectively in Table V and Table VI. The first one is composed by a less diverse set of UI features, but can rely on having more counters dedicated specifically to the three areas in which we've divided the screen, and therefore it should bring more information about the spatial location of the elements. The second features vector on the other hand loses spatial information regarding Swipeable and Long-Clickable UI elements, but

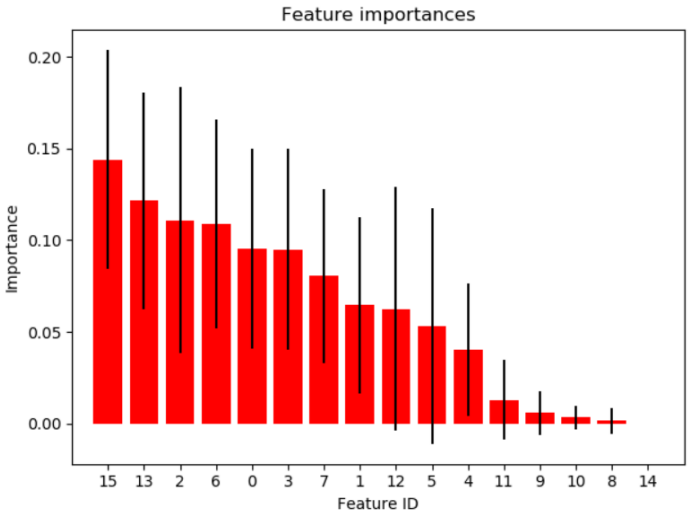
TABLE XIV: COMPARING THE PERFORMANCE OF THE SECOND FEATURES VECTOR IN RESPECT TO THE FIRST ONE

Algorithm	Metric	10-Fold	10-Fold Variation	LOO	LOO Variation
Logistic Regression	Accuracy	91.01%	+3.01	89.00%	+4.00
	Precision	96.54%	+3.94	89.00%	+4.00
	Recall	95.25%	+4.45	89.00%	+4.00
Random Forest	Accuracy	87.98%	+1.02	87.00%	+2.00
	Precision	95.67%	+2.27	87.00%	+2.00
	Recall	92.26%	+2.76	87.00%	+2.00
SVM - Linear Kernel	Accuracy	88.08%	+4.75	84.00%	+3.00
	Precision	95.40%	+4.30	84.00%	+3.00
	Recall	94.57%	+2.17	84.00%	+3.00
SVM - RBF Kernel	Accuracy	87.17%	+2.33	83.00%	-1.00
	Precision	94.82%	+1.62	83.00%	-1.00
	Recall	95.67%	+3.07	83.00%	-1.00

hosts new counters for Focusable elements and ImageViews. Thus, the first vector symbolizes a more focused and spatially-aware set of information about UI elements, while the second vector is more about gathering basic information about various on-screen elements.

In Table XIV we present the performance in terms of accuracy, precision and recall of the top 4 algorithms (according to our previous experiments) for the second features vector, and we focus on the variations in respect to using the same procedures but on the first features vector. If we just look at the accuracy for the the Nested 10-Fold Cross Validation technique, we have an average increment of 2.78% among all the presented algorithms: this demonstrates that in no one of those cases it resulted convenient to adopt the first vector. The same story goes

TABLE XV: FEATURE IMPORTANCES FOR THE SECOND FEATURES VECTOR



(0) Clickable el. Top: 0.0954
(1) Clickable el. Mid: 0.0646
(2) Clickable el. Bot: 0.1109
(3) Swipeable el.: 0.0949
(4) EditText boxes Top: 0.0402
(5) EditText boxes Mid: 0.0530
(6) EditText boxes Bot: 0.1088
(7) Long-Clickable el.: 0.0804
(8) Focusable el. Top: 0.0015
(9) Focusable el. Mid: 0.0057
(10) Focusable el. Bot: 0.0032
(11) Imageviews el.: 0.0129
(12) Password el: 0.0623
(13) Checkable el.: 0.1214
(14) Side Drawer: 0.0000
(15) Num Total UI el: 0.1439

for the Leave One Out technique, were the average increment is of 2.00%. For what concerns precision, the average increment is 3.03%, and for recall is even 3.11%. All those results show how the more general combination of UI elements of the second vector is better performing in all metrics and situations (with the only exception of LOO for the RBF SVM, where we had a small decrement of one percentage point).

Random Forests are also a great tool to estimate the **Feature Importance**: that is defined as the sum (averaged over all the trees of the forest) among every node splitting on a specific feature of the error reduction (Gini impurity of set of samples getting at the node, minus the sum of Gini impurity of the two split sets) and the number of samples routed to this node.

In Table XV we have ranked all the features composing the second features vector according to this metric. For every column we even show the standard deviation of the importance among the trees of the forest using vertical bars. It is interesting to notice how the top four features are the *total number of UI elements*, the *number of Checkable elements*, the *number of Clickable elements* and the *number of EditText boxes* both in the bottom part of the screen. This means that the key difference among the various types of activities stays in the cardinality of the elements that they are composed of; also, it is peculiar to notice how the bottom part of the screen is so important in the decision process. Besides this, the conclusion is that both the Clickable and EditText elements are essential in the classification process, and therefore it makes sense to count them separately for the three areas of the screen in order to gather the highest number of positional information possible. On the other hand, the presence of the Side drawer has an extremely small value in terms of importance, thus it is not useful.

4.5.2 Best Model and Performance Comparisons

We've already stated that the second features vector yields better performances, but independently from the Vector that has been chosen, we can see the same pattern when comparing the classifiers: **Logistic Regression** is in both cases the most accurate algorithm, **yielding a 10-Fold Nested Cross Validation accuracy of 91.01**%.

As a comparison we can consider the work of ***Rosenfeld, Kardashov and Zang published in 2018*** [11]: they classify Android activities into 7 categories (we're using one extra classification category, having 8 of them, making our problem a bit harder), and they evaluate their models on 10% of testing data (same percentage as ours), using a 10-Fold validation pro-

cedure (same number of folds as ours). They use 80 training activities versus our 100 samples. Thus, this is the perfect term of comparison for our work. They reported their results for some models that we've also tried to measured:

- *Logistic Regression*: they reach 77.5% accuracy vs. our 91.01% **(+13.51)**.
- *Random Forest*: they reach 82.5% accuracy vs. our 87.98% **(+5.48)**.
- *Decision Tree*: they reach 63.75% accuracy vs our 83.03% **(+19.28)**.
- *K-Nearest Neighbors*: they reach 77.5% accuracy vs 82.90% **(+5.40)**.

It is evident how we outperformed their results with each one of those algorithms, with an **average increase of Accuracy of 10.92%**. As a definitive comparison, **their top accuracy is 86.25% (obtained usign KStar): we reached 91.01% using Logistic Regression.** This is an increment of 4.76%, demonstrating how our model is very well suited for solving the activity classification problem.

As a second comparison, we can compare ourselves with another paper from 2018: ***AppFlow by Hu, Zhu and Yang*** [12]. This approach tries to classify Android apps starting from their screens: AppFlow retrieves all the info about the screen's layout and converts them in one only string, and uses Optical Character Recognition technologies to retrieve text from the activity's screenshot. The authors have used only two categories (Shopping and News) using a dataset of 1620 and 396 screens respectively. Thus, the size of their dataset is way greater than ours. Their measurements are performed using Leave-One-Out Cross Validation. Using only 2 categories and an unspecified Machine Learning model, they reach a top accuracy of 87.3%. Ours method,

TABLE XVI: PER-CLASS EVALUATION IN TERMS OF PRECISION AND RECALL

Activity Class	Precision	Recall
Todo	76.97%	87.20%
Advertisement	83.88%	92.46%
Login	96.82%	94.03%
List Selection	75.37%	73.33%
Portal	93.08%	80.49%
Browser	89.62%	83.25%
Map	81.54%	71.53%
Messages	95.94%	95.04%

using 8 categories, and a Logistic Regression model measured with the same Leave One Out metric, reaches an accuracy of 89.00%, meaning an **improvement of 1.70%**: this might not seem an impressive increase of performances, but we have to note our disadvantage in terms of number of samples (2016 vs 100), and number of categories (8 vs 2).

4.5.3 Evaluating Classes

Some last words can be spent on the performance specific to each one of our 8 categories: the results in terms of precision and recall can be seen in Table XVI. Those measurements were achieved in a 10-Fold Cross Validation fashion using our best model (Logistic Regression), retrieving a confusion matrix from each iteration, and calculating from there both Precision and Recall. Those were then averaged among the 10 iterations of the algorithm.

The *highest scores are yielded by Login, Messages and Portal activities*, showing that those are the most "standardized" type of screens (meaning they change the least from one app to

the other), and thus they have the most predictable layout structure, making our testing efforts easier. This implies that those Activities are always appearing with a similar set of widgets and UI elements, and many of those must be located in the same areas of the screen (let's think of EditText boxes for login screens).

The *worst results are obtained in the Todo and List Selection screens* (even though they are still pretty high): this indicates that those two types of activities vary the most among applications. This is a pretty intuitive conclusion, considering that we can have many different combinations of toggles an widgets in selection and settings screens. However, the ListView layout is always present, granting a still recognizable structure.

CHAPTER 5

TESTING FRAMEWORK

5.1 Overview

Writing UI and functional tests is often a very laborious process, as the developer needs to invest a great amount of time in creating the test script initially, but then he also needs to check for possible failures due faulty test scripts, and adapt them to any change that might be applied to the app's layout or functional structure. Another issue with manually-developed Test scripts is the very low reusability factor: if a developer needs to verify the same functionality (a Login procedure, as example) on two apps that differ just for the position of some of their UI widgets (as the position of the EditText boxes), he'd be forced to write two completely different scripts from scratches.

The objective of our framework is to ***allow the developer to write highly adaptive, reusable and modular test scripts aimed at accomplishing UI, functional and regression testing***:

- **Adaptiveness**: what makes our system original is the capability of classifying both the entire Application, and the specific Activity under test. This allows us to make reasonable assumptions on the expected structure and behavior of the screen we want to test. As an example, we can expect the Portal activity of a News App to be different from the Portal activity of a Social app, but very similar to another Portal of yet another News

app. Therefore, we can exploit the similarities and patterns among the same activity types in applications classified in the same category to *fire a test script that is suited to a specific app/activity combination.*

- **Reusability**: as a already stated, we can make suppositions on the expected layout structure and functionalities of an app's screen thanks to the classification process. Additionally, thanks to our dumping tools (Section 4.2.1), we are able to exploit both *visible UI elements* (such as TextViews or text prompts showed as hints in EditText boxes), and *invisible meta-data* (such as IDs used by developers to refer to UI elements, layout structures and other attribute values) to find specific widgets simply by having a broad semantic knowledge of their functionality. As an example, in a Login activity we'd just need to indicate that we want to input a specific email/password combination, and without any further provided details, the system would be able to identify the required UI elements, and perform the required actions: we don't care about their positions or graphical form. Therefore, we simply need to indicate in a broad and generic way the actions that we want to perform, and our framework will take care of identifying and locating the specific UI elements that are required. In this way, *the developer just needs to write one single test script that can be re-used on any flavour of the same app, or same category of apps.*

- **Modularity**: while we know that some actions can be performed in the same way in the identical activity/app combination, we cannot guarantee the same for very long and complex sequences of gestures. The basic idea is that "the more complex the command

becomes, the harder it will be to adapt it to a great number of applications". For this reason, *our framework provides a wide number of simpler and modular commands that can adapt very well to a great number of applications, but can still be composed into a complex sequence when the developer wishes so.*

5.2 The Test Scripting Language

We shall now proceed to explain our testing scripting framework in more detail, talking about the structure of our language, its general implementation, and the possible commands that were integrated.

5.2.1 Logical Structure of the Language

We've built a Test Scripting Language with the idea of making it very easily-readable on one side, and intuitive to write on the other. Considering that one of our objectives is to write generic tests (so that they can be applied to many apps at once without modifications), such should also be our language: commands have to be talkative to briefly indicate the gesture (or set of gestures) that they are representing, and the user should not worry of specifying screen coordinates, layout structures, or any other detail that might be app-specific.

The execution of a test case is fired when ***Two Pre-conditions*** are met:

- The app in use is part of the specified application category.
- The current screen coincides with the specified activity category.

The outcome of a test case (Passed/Failed) depends on ***Three Post-conditions***:

- All the commands composing the test case were executed without problems.

- The conditions of "Assertion commands" (if any are present) must be successfully satisfied.
- The screen in which we end up after all the commands have been executed has to match the activity category specified by the user.

If one or more of those conditions are not met, then the test case is considered *Failed*. On the other hand, if all of the three are satisfied, that means our app is behaving exactly as expected and thus the test case is considered *Passed*. A test suite is considered *Passed* only if each one of its constituting test cases were successfull.

The following is an explanation of the logical structure that our scripting language should follow in order to be recognized by the interpreter:

1. **Declaration of the Test Suite**: a detail to keep in mind is that our user might not just want to write one single test case, but a collection of them in order to constitute a Test Suite. Other than this, the execution of one or more tests must be confined inside of the same application: thus, we made the test suite declaration coincide with the specification of the application category used as pre-condition. Thus, the user must specify one of the 10 application categories (Section 3.3) to declare a new test suite:

   `When` *`NEWS`* `app:`

   Indicates that the list of test cases that follows must be executed only in an app whose APK file was classified in the 'NEWS' category.

2. **Declaration of Test Cases**: after having declared a test suite, the user must specify a list of one or more test cases that will be executed in the same order they are entered. A test case consists of a series of commands, but must be first defined by a header

```
When COMMUNICATION app:

    In LOGIN check for SAME state:
        Input name "arturo@gmail.com";
        Input password "abcd123";
        Click next;

    In TODO check for DIFFERENT state TODO:
        Add task "title";
        Assert linecount equals 1;
```

Figure 25: A sample Test Suite Script, containing two Test Cases.

which specifies the category of activity (Section 4.3) in which this test must be executed. In the same header, the user must also specify the expected post-condition of success:

`In` *`LOGIN`* `check for` *`DIFFERENT`* `state` *`PORTAL`*`:`

`In` *`MESSAGES`* `check for` *`SAME`* `state:`

The post-condition must also specify whether the final activity should be the same as the starting one, or be a completely different one. This feature is useful in case we are checking for a resulting activity which, even though is of the same category as the starting one, is a different instance.

3. **Test Case Content**: for every one of the declared test cases, the user has to specify one or more *Commands* that are executed in the same order they are inserted. The syntax of a command is pre-defined, and it needs to be followed by a semicolon. The full list of commands is presented in the next section.

In Figure 25 we present a sample script of a test suite consisting of two simple test cases: the first one verifies that our app does not allow the user to proceed to the next screen after having entered wrong login information; the second one checks whether after having added a new task, the task actually appears on the screen.

5.2.2 Test Commands

Each Test Case consists of a list of commands which were created with in mind the idea of making them easily comprehensible and readable. Each command tries to be the most adaptable possible, meaning that it can work on the greatest number of activities belonging to the same category; thus, a direct consequence is that the learning curve required from the user to use this language is very shallow, as the syntax is very simple, and the number of commands is not too high.

Our commands were divided into two big families: ***Activity-Specific Commands***, which are developed specifically for each of the 8 Activity categories, and they perform actions that are focused on the context given by a particular screen and thus, they try to adapt to every Activity of the same class and they require minimum input from the user. On the other hand, we have ***Generic Commands***, which can be used across every activity category as they perform actions that can usually be executed in every context (such as clicking or dragging the screen); this last family of commands was implemented to aid the user to fill possible little holes in the functionalities of a test script.

The following is a comprehensive explanation of the set of Activity-Specific Commands:

- ***Todo Activity***:

  - `ADD TASK "String"`: the system adds a new entry with the given title.
  - `TICK LINE number`: mark a specific entry as completed.
  - `TICK ALL`: mark all entries.
  - `CLICK LINE number`: click one of the entries to access it.
  - `SWIPE UP`
  - `SWIPE DOWN`
  - `ASSERT LINECOUNT EQUALS number`: count the number of entries, and check that they are equal to a given number.

- ***Advertisement Activity***:
  - `CLICK CLOSE`: the advertisement is closed by clicking the 'close' button.
  - `CLICK AD`: the system clicks the banner.
  - `PRESS BACK`: the advertisement is closed by pressing the Android 'back' button.

- ***Login Activity***:
  - `INPUT NAME "String"`: enter the given string as user-name/email in the correspondent field.
  - `INPUT PASSWORD "String"`: enter the given string as password.
  - `CLEAR FIELDS`: delete all the text in the editable fields.
  - `CLICK NEXT`: submit the credentials and proceed to the next screen.

- ***List Selection Activity***:

  - `CLICK LINE number`: select the indicated line of the list.
  - `TOGGLE LINE number`: toggle the switch if present on the indicated line of the list.
  - `LONG CLICK LINE number`: long-click a line of the list to pop-up a selection spinner menu.
  - `LONG CLICK ALL`
  - `PRESS BACK`
  - `SWIPE UP`
  - `SWIPE DOWN`

- ***Portal Activity***:
  - `SWIPE UP`
  - `SWIPE DOWN`
  - `SWIPE LEFT`
  - `SWIPE RIGHT`

- ***Browser Activity***:
  - `INPUT URL "String"`: write the given string in the URL bar section of the browser.
  - `PRESS ENTER`: confirm the URL and load the page.
  - `PRESS BACK`: go to the previous page.

- ***Map Activity***:

  - `INPUT SEARCH "String"`: search on the map for a location with a name equal to the String given in input.
  - `SWIPE UP`
  - `SWIPE DOWN`
  - `SWIPE LEFT`
  - `SWIPE RIGHT`
  - `CLICK CENTER`: press the center of the map.
  - `LONG CLICK CENTER`

- ***Messages Activity***:

  - `INPUT MESSAGE "String"`: write a new message with the given string as its text content.
  - `PRESS ENTER`: send the message.
  - `PRESS BACK`: come back from a particular chat.
  - `SWIPE UP`
  - `SWIPE DOWN`

We now present the list of Generic Commands, which can be executed during any category of activity, being very common actions. This list, however, is not to be considered as exhaustive and complete, as it just tries to cover the most symbolic functionalities of each activity category.

- ***Custom Commands***:

- `CUSTOM CLICK x y`: press the given set of screen coordinates.
- `CUSTOM LONG CLICK x y`: long-press the given set of screen coordinates.
- `CUSTOM DRAG FROM x y TO x y DURATION number`: this command allows for custom swiping gestures; the user just needs to provide starting and ending coordinates, together with the duration of the gesture.
- `CUSTOM TYPE "String"`: input the given string in the first EditText box that is found on the screen.
- `CUSTOM PRESS DEVICE BACK`: press the Android back button.
- `CUSTOM SLEEP number`: stop the execution of the script for a given number of ms.
- `CUSTOM ASSERT TEXT EQUALS "String"`: check whether it is possible to find a UI element on-screen presenting the given string.
- `CUSTOM CLICK TEXT "String"`: click a UI element containing the given string,

5.3 Implementation

Building a framework capable of understanding our language can be a very laborious process if not approached in the right way. We've built an interpreter that eventually executes Python code in order to perform the actual testing actions. An interpreter is usually composed of 4 blocks: first, a *Lexer*, which performs Lexical Analysis, then a *Parser* which analyzes the Syntax of the language, a *Semantic Analyzer* capable of giving a meaning to constructs coming from the Parse Tree, and finally the *Intermediate code Generator*, which in our case simply consists of Python code.

The process that we've followed was to first design a grammar, then use the ***ANTLR (Another Tool For Language Recognition)*** [23] tool to help us generate a valid parser for it, and eventually build the actual testing procedures in Python.

5.3.1 Lexical Analysis

A fundamental step is defining the lexical analysis, performed by the so-called *Lexer* or *Tokenizer*. Its duty is to convert a sequence of characters (thus, a string) into a *Token*, which can then be used in the Parse Tree. In other words, we are determining the terminal symbols of our grammar.

In order let our user to write his scripts comfortably, we must allow the interpretation of strings composed of any combination of uppercase and lowercase letters (e.g. "state", "STATE", and "StAte" should yield the same results). For this reason, we've declared fragments for every letter of the alphabet:

```
fragment A : [aA];

fragment E : [eE];

fragment S : [sS];

fragment T : [tT];
```

so that we should only worry about letters and not their form. Then, the actual terminal tokens were indicated as a combination of those fragments:

```
STATE : S T A T E;
```

Other than this, we've managed variable sequences of characters through regular expressions, and we've imposed the Scanner to skip every white-space, tab or newline that is found:

```
QUOTEDSTRING : '"'[-a-zA-Z0-9!@#$&()`.+, ]*'"';

WHITESPACE : (' ' | '\t')+ -> skip;

NEWLINE : ('\r'?  '\n' | '\r')+ -> skip;
```

In order to recognize those Tokens, the system generates a *Finite State Machine (FSM)*, which has a finite number of states, a starting state, and a set of accepted ending states. Our Lexer will simply consume one by one the characters found in the input to transition among those states, up to when a terminal state is reached, eventually yielding the corresponding terminal Token.

5.3.2 Syntactical Analysis

We can then proceed to design the production rules of the grammar that will be used to generate the Parse Tree. We created a *Context Free Grammar (CFG or type-2 grammar)*, indicating that our production rules are in the form $S \rightarrow \gamma$, where S is always a non-terminal symbol, and γ is a string of both terminals and non-terminals in any order. Thus, the only thing we need to be careful about is to just have one single non-terminal symbol in the left side of the rule.

Keeping those details in mind, we developed the set of rules allowing us to replicate the language shown in Section 5.2, of which we can have a glimpse by looking at the following example (consisting of a selection of few among all the production rules):

```
suite :  WHEN apptype APP COL testlist;

testlist :  (testtype1 | testtype2 |...| testtype8)+;

testtype1 :  (testsame1 | testdiff1);
```

```
testsame1 :  IN ACTTYPE1 CHECK FOR SAME STATE COL commandlist1;
```

As a small note on the format used, uppercase strings are used to indicate terminal symbols, while non-terminals are indicated by lowercase words.

The complete grammar (meaning both the Lexer and Parser rules) is then given as input to the *ANTLR* tool, which is capable of producing as output a working Parser written in the programming language of our choice (Python in this case), which will automatically build a Parse Tree (a structure representing how a grammar matches the user's input).

This tool produces an ***LL(*) Parser*** [24]: an *LL Parser* is a type of Top-Down Parser (meaning that it first analyzes the highest production rules of the Tree's hierarchy, and then proceeds down trying to match more specific rules) that parses the input provided by the user from left to right, deriving first the leftmost non-terminal symbol. An *LL(k) Parser* retrieves k Look-ahead symbols to preventively solve conflicts (when we can follow more than one rule) and to reduce the number of duplicate states: being the number of look-ahead symbols a bounded value, we can still represent the language using an acyclic DFA (Deterministic Finite Automaton). *LL(*) Parsers* are an extension of this last case, because the system can scan arbitrarily far ahead, thus allowing cycles in the language's DFA. The advantage is that instead of running the usual recursive-descent parsing process for each possible alternative, the LL(*) parser keeps looking at lookahead symbols stopping as soon as it is possible to decide among alternatives.

In Figure 26 we can see the Abstract Syntax Tree (AST) that is produced by our parser after having encountered the the following script as input:

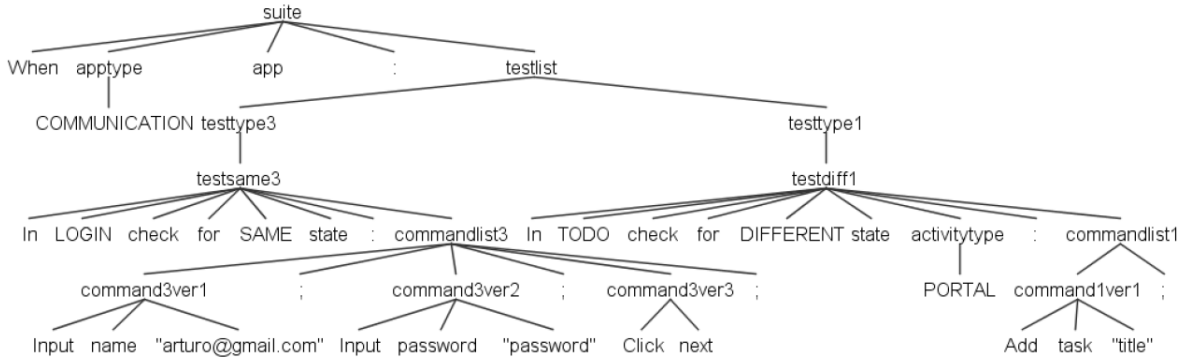


Figure 26: The Abstract Syntax Tree resulting from a sample input.

```
When COMMUNICATION app:
    In LOGIN check for SAME state:
        Input name "arturo@gmail.com";
        Input password "password";
        Click next;
    In TODO check for DIFFERENT state PORTAL:
        Add task "title";
```

Therefore, our Lexer is able to convert all of the strings present in the input script into the correspondent Tokens, which are then used by the parser to identify the correct set of grammar rules that is able to fully cover and explain

5.3.3 Semantical Analysis and Code Generation

In our work, the steps of Semantical Analysis and Code Generation are very closely related: usually, the Semantic Analyzer would perform operations such as *Type Checking* (e.g. avoiding

that a Float value is assigned to a String variable) and *Symbol Management* (using a Symbol Table to remember which symbols were already used or declared in the program); however, our language is very simple, and does not use explicit variables or need a Symbol Table. On the contrary, our scripts are mainly just a succession of commands with at most 2-3 user-passed parameters that need to be translated into practical Python code.

For that reason, our main concern was to translate the logical structure of the test suite from the Script format into pure Python code that could perform actual Tests on Android devices. Thus, we use *Listener Functions* associated to every production rule of the grammar which are called once their rule has been successfully parsed in the AST in order to make the transition to the Python environment. We have modeled the test suite as an object containing a queue of test cases, and test cases as objects containing a queue of functions to be executed and the conditions for test success. Each listener simply adds a specific element to the corresponding queue using the parameters given by the user.

Having such a structure simplifies very much the work pipeline necessary for a developer to add new commands to the framework: *adding a new Command simply means creating the corresponding Python function*, whose duty is to actually perform gestures on the Android device.

Thus, our final task is to write some Python code to perform the commands that were described in Section 5.2.2, and the interpreter will take care of managing the Scripts given as input by the user, telling us which commands were actually invoked.

5.3.4 Performing Adaptive Commands

As already stated multiple times, thanks to the machine learning methods used to classify Android activities, our commands can rely a lot on the typical structure of a given activity type. Knowing the usual structural pattern of an application's screen opens the doors for expecting certain UI elements to perform specific actions. Thus, the first question is "*how can we identify the UI elements that we're looking for, if we don't know their exact on-screen location?*" The answer is that we parse a dump of all the screen's interface, *looking at 4 factors*:

- ***Class Type***: using just this information we can already reduce the number of candidate elements; if we're looking for a Button, we won't be considering all the elements belonging to other classes. On the other hand, if we're trying to input a password through the `INPUT PASSWORD "String"` command, we'll be taking into consideration only EditText elements.
- ***Attribute Values***: in order to keep reducing the set of elements taken into consideration in our search, we can even look at the value of the attributes yielded by the dump. Following the same example of before, to perform the `INPUT PASSWORD "String"` we'll consider only EditTexts with the "*isPassword*" attribute set to `True`. In other instances, if we were in a Todo screen performing the `TICK LINE number` command, we would probably be interested in the "*isCheckable*" attribute. Three attributes that result very important in every case are "*isClickable*", "*isFocusable*" and "*isEnabled*" ones: those usually represent a great indication of whether the user is allowed to interact with their respective UI elements or not

- ***Locational Hints***: among the other attributes, the screen dump also provides the coordinates of the *boundaries for each UI element*; we have the X and Y coordinates for the bottom-left and upper-right corners of the box containing the element in question. From this we can infer if the widget under consideration is in a screen-area that makes sense for the type of activity we're in or not. Continuing the same example as before, the EditTexts of a Login procedure are expected to be in the central part of the screen, thus we would ignore EditText boxes that are in the upper or lower sections of the screen.
- ***Textual Hints***: this is the key element of our approach, as we can retrieve lots of information by looking at the meaning of some strings:
  - *Text Content*: certain UI elements provide a textual String that can be viewed by the user. This might consist of the words composing a TextView, or the textual hints that are shown in an EditText box before the user writes something in it. Being shown to the user in a way to shortly describe a concept or rule, those strings usually contain keywords that are semantically indicative, such as "Insert Password here", "Menu", "Confirm" or "Accept".
  - *Content-Description*: some Android UI focusable elements, such as ImageViews, can host a textual content Description that is used to describe themselves to aiding tools (such as TalkBack[1]) in case the screen cannot be seen. By definition, those strings try to explain a widget in the most clear and concise way possible, and thus we can

[1]Android TalkBack - https://support.google.com/accessibility/android/answer/6283677?hl=en

exploit them to find important keywords that can aid us in identifying the element we're looking for.

- *Resource-ID*: developers use this ID to easily refer to widgets via code when they are developing their application. It's in their best interests to use symbolic and explicative names in order to easily remember the use of a particular element. As an example, it is pretty common to see indicated as "continueButton" or "nextButton" the Resource-ID for a button used to proceed to the next screen. Again, we see present in those some Keywords that we could easily infer from the nature of a particular widget.

Therefore, we can reduce the problem of finding textual hints to simply looking for a specific set of keywords in those three types of strings. As an example, if we were executing the `CLICK NEXT` command, we'd be looking for words such as "Next", "Confirm" or "Submit", and we'd be able to identify even an ImageButton (meaning a Button without text, but just with a drawing, such as an arrow) simply by using its Content-Description and Resource-ID.

Considering that not every screen perfectly resembles the average layout structure of its category, the most complex scripting commands try to follow 2 or more "intuition paths", looking from zero at different sets of hints in order to maximize the probabilities of finding the desired widget.

Once we are sure enough of having found the correct UI element, we simply retrieve its Unique-ID, and we use that to interact with it via some sort of input (touch gestures or textual input).

5.4 The Testing Process Workflow

Now that we have explained all the parts constituting the framework, we can proceed to illustrate how the entire workflow for creating and executing a new test suite would look like. The entire process can be divided into 4 phases:

1. ***Writing the Test Script***: as a first step, the user has to write the Script for the Test Suite and for all the Test Cases contained in it using the language described in Section 5.2. As already stated in this chapter, the test suite declaration statement indicates the type of application for which the entire script is supposed to be for. In the same way, every test cases is declared specifically for a category of activity. This implies that the user has to write tests that will be fired when specific types of screens are met. We can even have more than one test case written for the same activity category, which will be executed in the same succession they are declared.

   Few words can be spent about the best practises of writing new Tests: keeping in mind that mobile applications are built with a minimalistic design in order to provide functionalities to users requiring the smallest amount of focus and effort from them, then the *test cases that we provide should be fairly simple as well.* Good app design practises also impose that their functionalities should be grouped into completely different sections of the program (for instance, we should not find settings and login forms in the same place). Thus, our

test cases should mainly focus on a single screen, without trying to navigate multiple sections of the app; this is also because the longer is our script, the most likely is that some issues might pop up with the recognition of the UI widgets (higher test complexity translates into a lower level of generalization among apps of the same category).

Therefore, as long as our tests are each focused on testing small functionalities of a specific screen and respect a modular structure, we should be able to build a comprehensive test suite, capable of running without issues.

2. ***APK Classification***: in Chapter 3 we have extensively explained how the classification process of the Android APK file is executed. Therefore, we simply provide the APK file of the application that we're willing to test on, and the framework will be able to extract the usual features vector and determine the app's category. The Test Suite written for that specific class of applications will be executed.

3. ***State Graph Modeling***: this is one of the core aspects our framework, as the system starts to crawl across the screens of the application by performing input actions on all the available UI elements of the screen. The result will be a graph with screen states as *nodes*, and identifiers of UI elements as *edges*. Let's note how nodes are not simply activities, but they are instead states, meaning that they differ in the position of a widget, or of a value of some attributes: this implies that we are able to execute our test cases on multiple instances of the same screen, with UI elements being in different conditions. This process alone could add a new level of depth to the way in which UI and functional testing is performed on Android programs, as there is a chance that a specific condition

(such as a Button that is not enabled) could impact its correct functioning. Also, this step performs some basic UI testing, finding broken elements and widgets if any.

4. ***Test Execution***: the final phase consists of the actual execution of the tests described by their respective scripts. The process is fairly simple: the system walks through the State Graph Model, and at each node (screen state) it runs the activity classifier to understand in which category of activity we're currently in; then, each test case that in the test script was written specifically for this category of activity is fired in the same order they are presented. For each test case, if all its conditions are successfully respected and the resulting activity is the expected one, we note the test was passed and then we return to the original state to prepare for the next one: this is a pretty complex procedure, as in order to be sure to come back to the correct state, we reset our position in the State Graph, and we follow the 'roadmap' (succession of UI elements to interact with on-screen) from the initial state up to the state in question.

CHAPTER 6

EVALUATION OF TESTING METHODS

6.1 Evaluation of User Interface Testing

In Chapter 2 we have explained how during the building process of the State Graph Model, we're also able to extensively test the basic correct functioning of the app's User Interface. *UI Testing* aims at checking whether the screen's elements work as assumed in the specifications, not to be confused with GUI testing (Graphical User Interface testing), which also focuses on verifying aesthetical and design properties (such as font types and sizes, alignment of images and positioning of text and widgets).

In our case, we will assume that the specification for UI elements are successfully satisfied up to when either one of those two conditions are met:

- An *Unhandled Exception or Crash* stops the execution of the app.
- The app *Freezes*, not responding to input anymore

Thus, we can compare our performances with ***Monkey***[1], an official Android Testing tool which *stress-tests* the User Interface of the application by sending pseudo-randomic input events (such as touch events and gestures) up to when one of the two previous conditions are met.

---

[1] Android Monkey - https://developer.android.com/studio/test/monkey

TABLE XVII: TIME TO FIND AN INJECTED BUG IN BOTH STATE GRAPH MODEL BUILDING PROCEDURE, AND ANDROID MONKEY TOOL

Application Name	Injected Bug Description	Time To Find - SGM	Time To Find - Monkey
Omni Notes (it.feio.android.omninotes.alpha)	Opening side drawer causes crash	36.21s	61.60s
	Opening the 'Camera' option inside a FAB menu causes crash	78.01s	191.24s
	Opening FAB, selecting 'New Note' and changing Reminder Date causes crash	173.32s	204.64s
MovieGuide (com.esoxjem.movieguide)	Pressing the 'Like Button' in the details page of a movie causes a crash	64.11s	29.59s
	Changing the the movies Sorting Order causes a crash	93.15s	47.93s
Flym (net.frju.flym)	Clicking on the GitHub Link of the 'About' page of the app causes a crash	597.96s	747.45s
	Opening the Search Bar causes a crash	49.74s	19.51s
Timber Music Player (naman14.timber.dev)	Opening the side drawer, then settings, and clicking on a specific line will cause a crash	1726.57s	2194.07s
	Selecting the 'Artists' tab causes a crash	65.38s	6.44s

In order to be able to compare the performances of the two, we have decided to go for a *Fault Injection* technique, where we manually modified the code of the app in order to throw an *Unhandled Exception* when a specific widget is activated. Therefore, it is just a matter of timing how long it takes to the two tools to find those bugs.

In Table XVII we've presented our results after having performed tests on some applications, which were necessarily Open Source, as we needed to access their source code in order to fit in the new bug. Also, for every one of those measurements, we've always started the execution

from the start-up screen of the app, and the Monkey tool was instructed to always remain inside the same package.

After a quick glance, we can notice how in every case both tools were able to find all injected faults. However, the most interesting observation is that the time to find a fault for the Android Monkey is extremely low when the bug is located in the app's home screen (or in a screen immediately reachable from there), as this tool will swipe and touch randomly on every part of the screen with an high chance of hitting the 'right' widget very rapidly. Our SGM Building process instead takes a bit more time to identify the bug, as it explores systematically every possible widget and screen one by one. On the contrary, when we plant our fault in a deeper level of the app (meaning that a longer input sequence is required to reach it), both tools take more time to discover it, but our SGM Building process starts to become faster as it is very hard for the randomic approach of the Android Monkey to reach this specific point of the app with a non-deterministic sequence of input events.

Therefore, with our State Graph Model Building Process, the *time required to find a UI fault seems to grow in a linear way in respect to the current size of state graph* (particularly, in respect to the number of transitions). Instead, Android Monkey will always cover too much the first few nodes of the graph (main screens of the app), missing a lot of edges that are further from them. Thus, our *SGM is a balanced and reasonable way do a basic round of UI testing on every User Interface element of our application.*

6.2 Evaluation of Command Adaptiveness

In Section 5.2.2 we have extensively presented the list of available adaptive commands, highlighting how they are divided into 8 groups, corresponding to the 8 possible categories of activity. Being able to exploit structural and semantical similarities among activities of the same class (thus, even among different versions of the same one), we are able to execute the same command on tens of screens belonging to different apps. In Section 5.3.4 it is possible to have a deeper insight on how this adaptiveness was achieved.

An essential step consists in trying to have an index of robustness for our test commands; we have to answer the question "***are textual hints and layout attribute Values enough to identify widgets with the same semantic value, but in different contexts?***". In order to find a symbolic response, we have gathered some of the most complex adaptive commands (thus, the most likely to fail when executed on many different instances) and we've tried to execute them on a number of distinct applications, detailing every unexpected behaviour.

In Table XVIII we can see the results of our experiments. The `ADD TASK "NoteTitle"` is without any doubt the most difficult command to adapt, as every app tries to implement the 'adding new task' functionality in its own way: some will simply require the user to press a button, while others will show a form to complete with additional information (and even those information could differ from one case to the other). Also, the shape, position and labels of those widgets will vary a lot. For this reason, our implementation of the command tries a wide selection of Textual Hints to find the button to add a new note (such as 'new', 'create' or 'write'), then uses structural hints to identify the right EditText box to insert the desired

TABLE XVIII: EVALUATION OF THE ROBUSTNESS OF THE MAIN ADAPTIVE COMMANDS

Command	Application Name	Success	Failure	Cause of Failure	Notes
`ADD TASK "NoteTitle";`	Todo List Easy	X		/	/
	ToDo List	X		/	Sometimes, an Advertisement pops up after having entered the note's Title, forcing us to press the confirm button twice.
	Tasks	X		/	/
	Cute ToDo List		X	Could not proceed after entering the note's Title, as an ImageButton with no hints had to be pressed.	/
	Reminder	X		/	/
	ToDo		X	After having pressed the "Add New Note" button, an unexpected list selector of note types appears.	/
	Inception List	X		/	/
	Checklist	X		/	The "Add New Note" button is uncommonly placed inside a drawer.
`INPUT NAME "testemail@gmail.com"; INPUT PASSWORD "password"; CLICK NEXT;`	Turo	X		/	/
	KAYAK	X		/	/
	Polito App		X	Language of the app was different from English, therefore could not detect the "Next" button.	/
	TEDx App	X		/	/
	TripAdvisor App	X		/	/
	JUST EAT	X		/	/
	Postmates	X		/	/
	GRUBHUB	X		/	/
`CLICK LINE 3;`	YouTube	X		/	/
	Google Maps	X		/	/
	Google Chrome	X		/	The first element of the ListView is a Login button, so the line indexes are incremented by 1.
	Android Settings	X		/	/
	Expedia App		X	Settings are wrapped in an unconventional way, not using any ListView structure.	/
	BBC News App	X		/	/
	Booking.com App	X		/	/
	LinkedIn App	X		/	/
`INPUT MESSAGE "Test String";`	Stock Android Messaging App	X		/	/
	Message Classic	X		/	/
	Pulse Messaging	X		/	/
	QKSMS	X		/	/
	Apple Message	X		/	/
	Messages GO	X		/	/
	Messages	X		/	Unusual structure of Message Composition screen, having multiple EditText boxes.
	Messages App	X		/	/

task title in, and finally the same kind of hints are adopted to confirm the creation of the entry. Therefore, the system is very flexible, and tries to find a great variety of hints before giving up. We can note how the command *worked flawlessly in 6 cases out of 8*. Focusing on those two failures, in the first case a confirmation Button had no hints of being such, while in the second an unexpected menu selector appeared, allowing the user to choose the type of task to add before entering its details. Therefore, the first problem was mainly caused by the programming style on behalf of the developer, which was not giving meaningfull names or descriptions to UI widgets, while the second was due to the logical structure of the activity that was different by the standard of the 'Todo list' category.

We've also tried to perform a full login sequence, which was executed *without any problems in 7 cases out of 8*. The only failure was due to an application being in a language different from English: all of the hints used to identify the widget of interest are in English, thus the system would not currently work in other languages for that reason. Other than that, this sequence of commands seems to execute in a pretty solid way due to the high similarity of login screens among them in therms of layout structures.

The same can be said about the `CLICK LINE 3` command, which was *executed correctly in all but one cases* of List activities: this fault was due to an application having built a settings list with a very uncommon layout instead of the usual ListView structure.

Finally, the `INPUT MESSAGE "Test string"` command was *executed perfectly in all 8 instances* of activities belonging to the 'Messages' category. In the same way of what had been observed about the Login procedure, this command is capable of adapting so well because the

layout structures of this category of activity do not differ so much from one instance to the other.

Drawing some conclusions from those observations, the system was able to ***perfectly adapt commands in 28 out of the 32 tested instances***. Even though we've just tried 4 commands, those can be considered functionally more complex than many others of Section 5.2.2, thus we can expect the most of the remaining to behave in a similar way. The only possible causes of harm can be either languages different from English, or layout structures that differ a lot from the baseline of their category.

6.3 Evaluation of the Functional Testing Procedure

During functional testing a developer tries to verify the correct functioning of his program in respect to the list of functional requirements/specifications. This is a form of *Black Box* testing, meaning that no access to the source code of the software is required: the developer simply provides inputs, and checks whether the output of the program corresponds to the desired one.

Using our Testing Scripting Framework described in Chapter 5, a developer can simply translate his functional requirements (and even UI specifications) into a set of tests; this process, thanks to the easily-readable/writable and human-like form of our scripts, can be achieved with very little efforts.

Thus, we just need a way of measuring the quality of tests that can be produced thanks to our framework. Pfaller, Wagner et al. observe in [25] that the challenge is to ensure that a test method is able to find faults in any program it is applied to, due to the fact that if a method reveals a great number of faults in an application, we might infer two conclusions: either

some classes of bugs might occur more often in the tested system, or the testing method might produce tests that are *well suited to detect faults of this class.* Thus, it is close to impossible to show how our framework is generalizable to a specific percentage of cases, as we cannot test them all one by one (every single case might concern a particular class of faults). What we can limit ourselves to do instead, is to find a meaningfull subset of testing scenarios that can be translated to many common cases. In this way, we can at least demonstrate how this subset can be correctly testable by our scripts. As shown in the Literature Review of Android Automated Testing Approaches by P. Kong et al. [26], most of research papers dynamically try their testing approaches on 8 applications, never going beyond 100, meaning that they also adopt a similar evaluation technique avoiding the testing procedure on thousands of apps.

For the reasons just described, *we've decided to write about ten Test Scripts to be applied in various cases spread across the 8 Categories of Activity that we have at our disposal.* In some of those, we have adopted the *Fault Injection* technique, meaning that we've accessed the source code of Open Source applications in order to add some simple functional bugs (such as broken buttons, or modifications to strings inputted by the user): this is because published apps generally already underwent an extensive testing process, and thus it would be pretty difficult to find undiscovered bugs. ***Those 11 testing scenarios were specifically built to test core functionalities of the activity category they belong to, so that their results can be symbolically translated to a very wide subset of applications***. In the meanwhile, we also keep proving the adaptability of our commands to a wide range of scenarios, and the overall reliability of our testing framework.

TABLE XIX: QUALITY EVALUATION OF THE FUNCTIONAL TESTING PROCEDURE

Starting Activity Category	Script	App Name	Test Explanation	Expected Outcome	Actual Outcome	Notes
ToDo	Script 1	ToDo List (Open Source)	Adding a single note note and then removing it	Test Failed	Test Failed	Test fails because the final state is the same as the initial one: the script was voluntarily wrong, expecting a different state
	Script 2	ToDo List (Open Source)	Same as previous case, but a bug preventing new note insertion was injected	Test Failed	Test Failed	Test fails because cannot execute the "tick" command due to the absence of the new note
	Script 3	ToDo List (Open Source)	In a list with 2 notes, add a new one, and check that we end up with 3 notes	Test Passed	Test Passed	/
Ad	Script 4	Ads Toolbox	Opening an Advertisement, then closing it	Test Passed	Test Passed	The resulting state is still an Ad Activity because actual Ads are just an overlay to the standard Activity. Also, layout changes a lot from one Ad provider to the other.
Login	Script 5	Patreon App	Suite of 3 Tests: trying to login without credentials, with wrong credentials and correct credentials	Test Passed	Test Passed	/
List Selection	Script 6	Android Stock Messaging App	Sending a message to the most recent contact	Test Passed	Test Passed	/
	Script 7	Amaze File Manager (Open Source)	Creating a new folder, but a bug making all names lowercase was introduced	Test Failed	Test Failed	Test fails because text inserted by user does not match new folder's name
Portal	Script 8	Expedia App	Suite of 2 Tests: try to visit all tabs of the app. then try to change app's country	Test Passed	Test Passed	/
Browser	Script 9	DuckDuckGo	Check whether the correct website is loaded after inputting its URL	Test Passed	Test Failed	This specific browser deletes the URL when the keyboard is closed. Thus, the query is always empty when pressing enter.
Map	Script 10	Google Maps	Searching for a specific city, and then tryingits Street View	Test Passed	Test Passed	/
Messages	Script 11	QKSMS	Test to check if message's string content is displayed correctly after being sent, but a bug changing random characters was injected	Test Failed	Test Failed	/

In Table XIX we have reported the set of results obtained with our tests; the corresponding test scripts can be found in the Appendix Chapter of this work. We can immediately notice how our scripts were able to handle pretty much every situation, finding all the UI widgets that they needed, and correctly identifying the inserted bugs (if any). The only case where our commands failed was in *Script 9*, when the app under examination presented a very uncommon behavior in deleting the user's input after the keyboard is closed (even though if the same script was applied to other apps of the same category, this problem would not happen). In the other cases, when tests failed it was because either the injected fault was found, or the script was voluntarily formulated in a wrong way (e.g. *Script 1*, where the test expects a different resulting screen, but due to inserting and then immediately removing a task, the final screen is the same as the starting one). Our activity classifier reveals itself to be a very powerful tool, allowing a new level of depth for those scripts: instead of simply relying on assertions (which are used as well to identify specific strings on-screen, or count elements), we can also verify the screen in which we end up after the script execution, inferring for example if a Login procedure had been successfull or not.

6.4 Labor Savings in Creating Tests

Few last words can be spent talking about the costs and savings in terms of developer's time and labor. In our framework, the user will *lose the majority of its time in two specific phases*: first, he will have to wait for the creation of the State Graph Model, a process that is linear in the number of possible states of the app (thus, proportional to the complexity of the app's screens) and might take tens of minutes, up to an hour; however, during this phase the system

also executes an extensive stress test of the User Interface. The second cause of time loss for the developer is the process of waiting for the execution of the test Scripts.

However, *our framework is capable of a great saving in terms of time and labor during the test scripts Writing process.* This is because of two factors:

- Thanks to a syntactically-simple scripting language, the learning curve necessary to understand how to write tests is very shallow. Mainly, ***the translation of the functional specifications document into a set of functional test scripts becomes very fast and direct***: it is no longer necessary to tediously transform a humanly-readable sentence into a succession of lines of codes referring to the IDs of elements of the User Interface. This means that those tests could be already prepared during the decisional process for the app's requirements, or even replace the document itself (if few more commands were implemented). This implies that our test cases are easily readable, and a new developer could be able to fully understand their logic and semantics without necessarily having to learn how to program tests and backtrack the identifiers of every widget and layout structure of the screen.
- ***The time necessary to the developer for writing a test script is reduced to the minimum***. If we consider as an example the command to add a new task in a 'ToDo Activity', it consists of 1 line of code in our script, but requires 89 lines of Python code (with UiAutomator) to be implemented, or 31 lines of Appium code [1] in Java. *On average, each one of our adaptive commands is actually representing 27.1 lines of Python*

code. This means that in an average-length test case script of 6 lines, we are actually saving to the developer the burden of writing 156 lines of Python code.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

Summarizing, we have built a testing framework composed of four modular functional units, each contributing to an all-rounded and adaptive testing process targeted at a great variety of Android applications with little efforts required on the part of the developer, who has simply to convert the list of functional specifications into a user-friendly scripting language.

The State Graph Model building procedure might be a little time consuming, but provides a useful logical map of the app, and performs an extensive stress test of the User Interface at the same time. The APK classification procedure yields with an acceptable level of accuracy the most plausible category for our application, allowing the developer to prepare Test Suites that will be fired once a specific type of app is found. On the other hand, the activity classification phase is at the core of our system, producing a very high level of accuracy thanks to its Logistic Regression model, allowing for the execution of adaptive commands that rely on common patterns among activities of the same category. Finally, our test scripting framework lets users write sets of robust and reliable test cases composed of commands which can be performed upon a great number of apps belonging to the same screen category, saving the developer lots of efforts and time.

7.2 Future Works

The system that we've created is a great base on which many additional improvements could be developed. Starting with the State Graph Model, its building process is slow due to the time necessary to capture a dump of the current screen of the device. Also, sometimes a wrong layout structure is yielded in a non-deterministic way, damaging the overall accuracy of the logic map that is being built. A possible solution might be trying to use another tool different from UiAutomator to retrieve layout structures, which could be the cause. Also, even though the APK classification process is giving acceptable results, some other different Feature Vectors might be extracted from the file and new architectures could be tried in order to increase the accuracy of the model. Using a more precisely-label dataset might increase performances too, as shown in the results section. Finally, the adaptive power of the scripting commands might be increased by fine-tuning and trying them on an even greater number of apps, and the selection of commands itself could be expanded with new ones. The entire testing framework has been built with in mind the idea of making it easily expandable by developers, in fact new commands can be added by simply modifying the grammar file (*scriptingLanguage.g4*) and inserting the corresponding Python code in the respective listeners of the Parse Tree .

APPENDIX

TEST SCRIPTS USED IN MEASUREMENTS

The following is the set of Test Scripts used in the measurements shown in Table XIX.

- Script 1:

```
IN TODO check for DIFFERENT state TODO:
    ADD TASK "Title";
    CUSTOM SLEEP 2000;
    TICK LINE 1;
```

- Script 2:

```
IN TODO check for SAME state:
    ADD TASK "Title";
    CUSTOM SLEEP 2000;
    TICK LINE 1;
```

- Script 3:

```
IN TODO check for DIFFERENT state TODO:
    ADD TASK "Title";
    CUSTOM SLEEP 2000;
    ASSERT LINECOUNT EQUALS 3;
```

- Script 4:

```
IN AD check for DIFFERENT state AD:
```

APPENDIX (continued)

```
                        CLICK AD;
                        PRESS BACK;
                        CUSTOM SLEEP 3000;
                        CLICK CLOSE;
```

- Script 5:

```
                    IN LOGIN check for SAME state:
                        CLICK NEXT;

                    IN LOGIN check for SAME state:
                        INPUT PASSWORD "wrongPassword";
                        INPUT NAME "correctEmail@gmail.com";
                        CLICK NEXT;
                        CLEAR FIELDS;

                    IN LOGIN check for DIFFERENT state PORTAL:
                        INPUT PASSWORD "correctPassword";
                        INPUT NAME "correctEmail@gmail.com";
                        CLICK NEXT;
```

- Script 6:

```
                    IN LIST check for DIFFERENT state MESSAGES:
                        CLICK LINE 0;   // Selecting latest contact
                        CUSTOM SLEEP 1000 ;
                        CUSTOM CLICK TEXT "Text message";   // Opening the
    text input prompt
```

APPENDIX (continued)

```
                        CUSTOM SLEEP 1000 ;
                        CUSTOM TYPE "HelloWorld";   // Typing the message
                        CUSTOM SLEEP 1000 ;
                        CUSTOM CLICK TEXT "send";   // Using context to
    send the message
                        CUSTOM SLEEP 1000 ;
                        CUSTOM ASSERT TEXT EQUALS "HelloWorld";   //
    Checking if the correct message was sent
```

- Script 7:

```
                    IN LIST check for DIFFERENT state LIST:
                        CUSTOM CLICK TEXT "Download";
                        CUSTOM SLEEP 2000;
                        CUSTOM CLICK 940 1640;  // Clicking "add" button
    without context
                        CUSTOM SLEEP 1000;
                        CUSTOM CLICK TEXT "Folder";   // Selecting type of
     content to add
                        CUSTOM SLEEP 1000;
                        CUSTOM CLICK TEXT "Enter Name";   // Opening
    edittext prompt
                        CUSTOM SLEEP 1000;
                        CUSTOM TYPE "ThisIsAName";  // Entering the folder
     name
                        CUSTOM SLEEP 2000;
```

APPENDIX (continued)

```
12                          CUSTOM CLICK TEXT "CREATE";   // Creating the
    folder
13                          CUSTOM SLEEP 1000;
14                          CUSTOM ASSERT TEXT EQUALS "ThisIsAName";   //
    Checking if the folder exists
```

- Script 8:

```
1                       In PORTAL check for SAME state:
2                           SWIPE RIGHT;
3                           CUSTOM SLEEP 1000;
4                           SWIPE RIGHT;
5                           CUSTOM SLEEP 1000;
6                           SWIPE LEFT;
7                           CUSTOM SLEEP 1000;
8                           SWIPE LEFT;
9
10                      In PORTAL check for DIFFERENT state PORTAL:
11                          SWIPE RIGHT;
12                          CUSTOM SLEEP 1000;
13                          SWIPE RIGHT;
14                          CUSTOM SLEEP 2000;
15                          CUSTOM CLICK TEXT "Country";
16                          CUSTOM SLEEP 3000;
17                          CUSTOM CLICK TEXT "Canada";
```

- Script 9:

APPENDIX (continued)

```
In BROWSER check for SAME state:
    INPUT URL "bbc.co.uk";
    PRESS ENTER;
    CUSTOM ASSERT TEXT EQUALS "bbc";
```

- Script 10:

```
IN MAP check for DIFFERENT state MAP:
    INPUT SEARCH "San Francisco";
    CUSTOM SLEEP 1000 ;
    CUSTOM LONG CLICK 226 1220;   // Clicking the
street view icon
    CUSTOM SLEEP 1000 ;
    SWIPE UP;
    SWIPE DOWN;
    SWIPE LEFT;
    SWIPE RIGHT;
    CUSTOM PRESS DEVICE BACK;
```

- Script 11:

```
In MESSAGES check for DIFFERENT state MESSAGES:
    INPUT MESSAGE "A sentence";
    INPUT MESSAGE "Another sentence";
    PRESS ENTER;
    CUSTOM ASSERT TEXT EQUALS "sentence";
```

CITED LITERATURE

1. Shah, G., Shah, P., and Muchhala, R.: Software testing automation using appium. International Journal of Current Engineering and Technology, 4(5):3528–3531, 2014.

2. Machiry, A., Tahiliani, R., and Naik, M.: Dynodroid: An input generation system for android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

3. Liu, Z., Gao, X., and Long, X.: Adaptive random testing of mobile application. 2010 2nd International Conference on Computer Engineering and Technology, 2:V2–297–V2–301, 2010.

4. Zhu, H., Ye, X., Zhang, X., and Shen, K.: A context-aware approach for dynamic gui testing of android applications. 2015 IEEE 39th Annual Computer Software and Applications Conference, 2:248–253, 2015.

5. Amalfitano, D., Fasolino, A., Tramontana, P., Ta, B., and Memon, A.: Mobiguitar – a tool for automated model-based testing of mobile apps. IEEE Software, 32:1–1, 04 2014.

6. Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M.: Using gui ripping for automated testing of android applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

7. Mahmood, R., Mirzaei, N., and Malek, S.: Evodroid: Segmented evolutionary testing of android apps. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

8. Yang, C.-Z. and Tu, M.-H.: Lacta: An enhanced automatic software categorization on the native code of android applications. Lecture Notes in Engineering and Computer Science, 2195:769–773, 03 2012.

CITED LITERATURE (continued)

9. Dong, F., Guo, Y., Li, C., Xu, G., and Wei, F.: Classifydroid: Large scale android applications classification using semi-supervised multinomial naive bayes. 2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS), 2016.

10. Hamedani, M. R., Shin, D., Lee, M., Cho, S.-J., and Hwang, C.: Androclass: An effective method to classify android applications by applying deep neural networks to comprehensive features. Wireless Communications and Mobile Computing, 2018:1–21, 2018.

11. Rosenfeld, A., Kardashov, O., and Zang, O.: Automation of android applications functional testing using machine learning activities classification. Proceedings of the 5th International Conference on Mobile Software Engineering and Systems - MOBILESoft 18, 2018.

12. Hu, G., Zhu, L., and Yang, J.: Appflow: Using machine learning to synthesize robust, reusable ui tests. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pages 269–282, New York, NY, USA, 2018. ACM.

13. Young, M. and Pezze, M.: Software Testing and Analysis: Process, Principles and Techniques. USA, John Wiley & Sons, Inc., 2005.

14. Mikolov, T., Chen, K., Corrado, G. S., and Dean, J.: Efficient estimation of word representations in vector space, 2013.

15. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J.: Distributed representations of words and phrases and their compositionality. In Advances in Neural Information Processing Systems 26, eds. C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, pages 3111–3119. Curran Associates, Inc., 2013.

16. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. J. Mach. Learn. Res., 15(1):1929–1958, 2014.

17. Baldi, P. and Sadowski, P. J.: Understanding dropout. In Advances in Neural Information Processing Systems 26, eds. C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, pages 2814–2822. Curran Associates, Inc., 2013.

CITED LITERATURE (continued)

18. Ramachandran, P., Zoph, B., and Le, Q. V.: Searching for activation functions. CoRR, abs/1710.05941, 2017.

19. Batista, G., Silva, D. F., et al.: How k-nearest neighbor parameters affect its performance. In Argentine symposium on artificial intelligence, pages 1–12. sn, 2009.

20. Dudani, S. A.: The distance-weighted k-nearest-neighbor rule. IEEE Transactions on Systems, Man, and Cybernetics, SMC-6(4):325–327, 1976.

21. Breiman, L.: Random forests. Machine Learning, 45(1):5–32, 2001.

22. Krizhevsky, A., Sutskever, I., and Hinton, G. E.: Imagenet classification with deep convolutional neural networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

23. Bovet, J. and Parr, T.: Antlrworks: An antlr grammar development environment. Softw. Pract. Exper., 38(12):1305–1332, October 2008.

24. Parr, T. and Fisher, K.: Ll(*): The foundation of the antlr parser generator. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

25. Pfaller, C., Wagner, S., Gericke, J., and Wiemann, M.: Multi-dimensional measures for test case quality. 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, pages 364–368, 2008.

26. Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T., and Klein, J.: Automated testing of android apps: A systematic literature review. IEEE Transactions on Reliability, PP:1–22, 2018.

VITA

NAME ARTURO CARDONE

EDUCATION

B.S., Computer Engineering
Politecnico di Torino
2014 - 2017

M.S., Software Engineering
Politecnico di Torino
2017 - 2019 (*Current*)

M.S., Computer Science
University of Illinois at Chicago
2018 - 2019 (*Current*)

EXPERIENCE

Software Engineering Intern (2017 - *Grugliasco, Turin, Italy*)
Comau S.p.A (Fiat Chrysler Automobiles Group)
Three months as a Software Engineer Intern in Comau's Innovation Team spent developing an Android Dashboard and its Backend for predictive maintenance of robotic arms used in FCA industrial assembly lines.

HONORS

Winner of "2017 Droidcon Italy - AAL theme" Hackathon
2017 - Turin, Italy

Winner of "2016 Make It Wearable" Hackathon
2016 - Turin, Italy

Expositor at "WTT Wearable Tech Turin 2016" Event
2016 - Turin, Italy