Evaluating the Impact of GUI Similarity between Android Applications to Measure their Functional Similarity

Sondus Almrayat
CIS Department
The University of Jordan
Amman, Jordan

Rana Yousef
CIS Department
The University of Jordan
Amman, Jordan

Ahmad Sharieh
CS Department
The University of Jordan
Amman, Jordan

ABSTRACT
Finding similar or related Android applications is a feature in popular search engines. An app's appearance is usually the first indicator of similarity. In this paper, the impact of GUI similarity for Android applications in measuring their functional similarity is evaluated. Accordingly, a number of Android applications will be analyzed to identify their resources and extract the most commonly used appearance features from each app's package kit (APK) and its xml layouts. An algorithm that automatically extracts these features is designed and developed. A sample of 50 Android apps from Google play store was chosen, and two separate experiments were performed: one using the presented method to measure appearance similarity, the second using one of the available methods to measure functional similarity, then the results were compared. Results show that there is a relationship between appearance and functional similarities, where a strong relationship exists between appearance similarity and most of the functional similarity anchors.

General Terms
Android Application, Similarity, GUI

Keywords
Appearance Similarity, Functional Similarity, Adaptive Programmable Interface Unit, Android Package Kit (APK). Search engine

1. INTRODUCTION
Smart phones are becoming more integrated and important part of people's daily lives due to their highly powerful computational capabilities, such as email applications, online banking and online shopping…etc. The use of mobile devices has increased in our lives offering almost the same functionality as personal computers. Android devices have appeared lately and, since then, the number of applications available for this operating system has increased exponentially. Finding similar or related Android applications is a feature in popular search engines (e.g., Play store, Galaxy apps). For example, after users submit search queries, Google play displays the search results together with a group of relevant applications labeled as similar applications. Market-specific search engines identify similar apps by relying on textual descriptions only [1]. However, a match between words in a search query with words in the descriptions or in the source code of applications doesn't guarantee that these applications are relevant. In addition, many application repositories are polluted with poorly functioning projects.

In this paper, the aim is to compare the similarity between Android applications' graphical interfaces and their functions to figure out if there is any association between them. This can evolve a new direction in different researches concerned with finding relevant apps in search engines, understanding main features of successful apps, discovering code theft and plagiarism [2, 3], identifying reusable components that can help new android developers to use APIs, and improving understandability of source code and rapid prototyping

Android app's features will be examined both from text elements and image elements. Then, different distance calculating formulas will be used to compute the similarity scores based on different similarity metrics.

Section 2 presents background knowledge and some related work, section 3 describes the algorithms developed to measure appearance similarities and functional similarities, section 4 presents the experiments and results, finally, section 5 concludes the paper.

2. BACKGROUND AND RELATED WORK
In this section, an overview about Android platforms and Android applications' architecture (i.e. the main components of an android application) is resented. Then, the main features of the GUI of an Android application will be discussed. Finally, a review of the literature to addresses current methods of detecting similar Android applications is given.

2.1 Android Platform
Android is a mobile operating system programmed by Google and designed mostly for the purpose to be use for the sophisticated mobile devices with touch-screen capabilities which are known commercially as Smartphones [4]. Android application development depends on four major components, each plays an important role to build the structure of the application. Figure 1 illustrates the structure of Android platform.

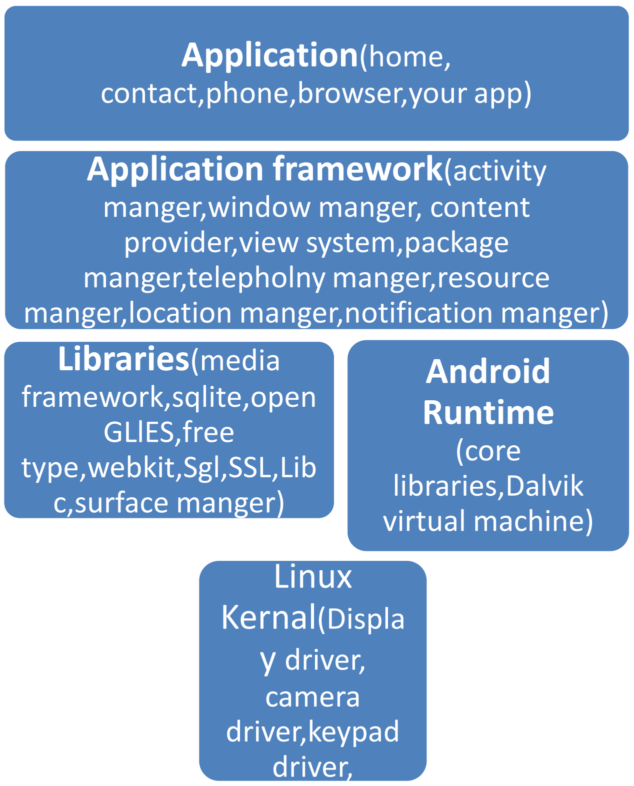
Application(home, contact,phone,browser,your app)
Application framework(activity manger,window manger, content provider,view system,package manger,telepholny manger,resource manger,location manger,notification manger)
Libraries(media framework,sqlite,open GLIES,free type,webkit,Sgl,SSL,Lib c,surface manger)
Android Runtime (core libraries,Dalvik virtual machine)
Linux Kernal(Display driver, camera driver,keypad driver,

Fig 1: Android Platform Architecture [5]

2.2 Android Application Components

In the following subsections, a brief overview of the main components of Android appliances as was depicted in [6] is provided.

2.2.1 Activity

An activity is the most essential part of an Android application that represents every single screen. Each application should at least have one activity to let the user interact with the mobile device. An activity starts running when the application is opened. Each application has a number of activities and each has a lifecycle.

2.2.2 Content provider

A content provider is used to supply and store data in an application. It manages access to data store by the application itself or by other apps and provides means to share data and define security issues related to accessing and sharing the data.

2.2.3 Service:

A service is a process that runs in the background without user interactions (e.g. wifi status is running through a background process by calling the Service class related to this service). The service performs its function by starting to run with an intent to describe the service and to carry any necessary data. Services don't provide user interfaces so other components can start a service, such as an activity or another service in the application. There are two types of services; bounded and unbounded services.

2.2.4 Broadcast receiver:

It is a mechanism to define how Android platform forwards its events to applications. There are two types of Broadcast Receivers: ordered and normal, and the main usage of these receivers are inter-process communication and tracking of specific events (e.g. arrival of an SMS). Applications declare statically or dynamically their interest in receiving a certain event and accordingly the operating system (OS) will try to deliver this event when it happens.

2.3 Android Project Structure

Most of Android applications are developed using the Android Studio environment [6]. There are other environments for creating Android projects such as eclipse IDE and NetBeans IDE. In this section, the project structure of an Android project is presented, the reader should be aware of a few directories and files in the app. For every single Android screen, there are at least two files; one is a Java source code file and the other is an xml layout file. Google now supports Kotlin as a language for mobile development on Android, it is designed to fully interoperate with Java [7]. In this paper, only Java source code files will be examined.

Src: contains source code files for the application project. It is represented by MainActivity.java which is a Java file that represents the app project activities and it is the most important file to be converted to a Dalvik executable and to run activities.

Gen: It contains the R (resource) file, a compiler-generated file that references all the resources found in the application project, and the user should not modify this file because it is generated automatically when the app is created. This file is like the glue between the activity Java files like MainActivity.java and the resources like strings.xml.

Bin: This file contains the Android package files. apk, which is built by the ADT during the build process, and everything else needed to run an Android application.

Res: This folder contains many files such as: drawable, layout, values:

Res/drawable: This directory consists of image components that are designed for screens' interfaces of apps.

***Res/layout*:** This is a directory for the files that define the graphical user interfaces.

Res/values: This directory has other various XML files that contain a collection of resources, such as strings and colors definitions.

AndroidManifest.xml: This file provides a description of the fundamental characteristics of the app and defines each of its components and application permissions.

2.4 Android UI Views

The user interface (UI) for each component of the Android app is defined using a hierarchy of View and View Group objects. A view is an object that draws a component on the screen that the user can interact with, and the view group is an object that holds other view objects in order to define the layout of the user interface.

To declare the app's layout, you must instantiate a view object in code and start building a tree, but the easiest and most effective way to define the app layout is with an XML file. XML offers a human-readable structure for the layout.

Android provides several views which allow the user to build the graphical user interfaces (GUIs) for the app; such as TextView which is used to display text to the user, EditText: a pre-defined subclass of TextView that includes rich editing capabilities, AutoCompleteTextView: a view that is like EditText, except that it shows a list of completion suggestions automatically while the user is typing, Button: can be pressed, or clicked, by the user to perform an action, ImageButton, AbsoluteLayout: enables users to specify the exact location of its children, Checkbox: an on/off switch that can be toggled by the user. The user should use checkboxes when presenting them with a group of selectable options that are not mutually exclusive, Toggle Button: an on/off button with a light indicator. Radio Button: has two states: either checked or unchecked. Radio Group: used to group together one or more Radio Buttons, and Progress Bar view: provides visual feedback about some ongoing tasks, such as when users are performing a task in the background.

2.5 The Strings File

The strings.xml file is in the res/values folder and it contains all the text that the application uses. For example, the names of buttons, labels and default Android text. This file is responsible for the textual content of an app.

2.6 Related Work

There are existing approaches for measuring similarities between Android applications. The similarity approach of Linares and Holtzhauer [8] is based on detecting closely related applications in Android (CLANdroid). The authors relayed on advanced Information Retrieval techniques and five semantic anchors. They evaluated CLANdroid by creating a benchmark consisting of 14,450 apps along with information on similar apps provided by Google Play.

The work of Linares and Holtzhauer was based on a previous work on source code engines, and approaches for detecting similarity. There are also several studies that proposed various code search engines for returning similar code pieces, functions, components, applications, etc, [9]. However, many studies also aim to detect similar code fragments (a.k.a. clone detection) based on text matching, syntax trees, program dependence graphs, etc.[10].

Moreover, Crussell [11] presented a scalable approach to detect similar Android apps based on their semantic information. He implemented his approach in a tool called AnDarwin and evaluated it on 265,359 apps collected from 17 markets including Google Play and numerous third-party markets, such as the app’s market, signature. AnDarwin extract semantic vectors from source code methods in the apps. The main idea is that the methods can be combined in semantic blocks, therefore, if two semantic blocks are code clones, then the semantic vectors representing these blocks are considered similar.

The directory structure in mobile apps has been also used to detect similar apps. For instance, the authors in [12] decompiles an APK and walks through the directories and files of the app to construct a tree, which represents the directory structure. Destruct computes the percent difference between two trees to represent the similarity between two applications. Thus, the smaller the percent difference the more similar the apps are based on their directory structures.

Other approaches have proposed the usage of centroids, topics, and method signatures to detect similar apps. Chen et al [13] has detected the similar apps by comparing centroids created from dependency graphs at method level. However, these similarity measures are used to draw a Boolean value conclusion on the app’s core functionality cloning. That is, either two apps are marked as clones or not, which prevents partial similarity detection. Chen et al. evaluated their approach across multiple different Android markets, yet did not use Google Play. Gorla, et al. [14] applies Latent Dirichlet Allocation on the descriptions of over 32K applications. The k-means algorithm is then used to cluster the apps (by using the topics generated with LDA) and, thus, provides the ability to identify groups of apps with similar descriptions.

Similarly, Desnos [15] used method signatures to detect similar Android apps, where the Signatures were composed of string literals, API calls, control flow structures, and exceptions. Wang et al [16] proposed an approach to detect and identify app clones in two phases, first filtering the code of the application from third-party libraries, and then uses API calls to detect cloned apps across different applications. Another work on detecting repackaged apps in two phases is the one by Shao et al [17], which clusters the apps using resources (e.g., strings and images) and statistical features initially, and then performs a second clustering stage using structural features.

The work by Thung et al [18] is also similar to CLANdroid, because they used an approach based on CLAN for detecting similar software systems, but instead of using API calls, the authors used the tags for the systems in source forge website.

Zhu et. al. [19] proposed a method to design a system to compare the GUI similarity among Android apps and pick up some apps with high similarity on their appearance. In detail, they extracted some features of apps and compute their similarity by their feature vectors. They evaluated their design with 2,000 apps in both official and alternative Android marketplaces to find out such appearance-similar apps in their dataset.

Jadhav et. al. [20] proposed a system to detect malware and plagiarisms by using GUI similarity method. Their approach consists of three steps: pre-processing, dish fit for a king extraction and similarity comparison.

Reviewing the literature, it is realized that most of exiting similarity approaches are based on similarity measures that depend on elements of the code (clone code) to detect malware but few of them handle this problem using graphical user interfaces.

In this research, Android application's GUI features will be analyzed to identify the mi important features that can give an indication of functional similarity between two applications. Then, the extent to which a similarity in those GUI features can indicate a similarity in the applications' functionalities will be measured.

3. METHODOLOGY

The main goal of this research is to evaluate the impact of GUI similarity for Android applications in measuring their functional similarity. In this section, the different phases of the research methodology are descried.

3.1 Phase 1: Identify the most well-known theories, techniques and tools to measure similarity

3.1.1 String Similarity

In many applications of detecting similar apps, it is necessary to algorithmically quantify the similarity of Android applications depending on special features. String similarity can be defined as finding the similarity of two strings that are composed of symbols from a finite alphabet. There are many string similarity measures but the most well-known measures are based on edit distance [21] and the length of the longest common subsequence [22].

Eidt distance: also known as levenshtein distance, is defined as the minimum number of edit operations such as insertion and deletion needed to transform one string into another. Figure 2 shows the edit distance algorithm.

The Longest common subsequence algorithm (LCS): is a well-known algorithm, defined as finding the length of the longest common subsequence (LCS) of two strings. Let two sequences be defined as: $X = (x_1, x_2...x_m)$ and $Y = (y_1, y_2...y_n)$. The prefixes of X are $X_{1, 2,...m}$; the prefixes of Y are $Y_{1, 2,...n}$. Let $LCS(X_i, Y_j)$ represent the set of longest common subsequence of prefixes X_i and Y_j. To find the longest subsequences common to X_i and Y_j, compare the elements x_i and y_j. If they are equal, then the sequence $LCS(X_{i-1}, Y_{j-1})$ is extended by that element, x_i. If they are not equal, then the longer of the two sequences, $LCS(X_i, Y_{j-1})$, and $LCS(X_{i-1}, Y_j)$, is retained. If they are both the same length, but not identical, then both are retained. Notice that the subscripts are reduced by 1 in these formulas. That can result in a subscript of 0. Since the sequence elements are defined to start at 1, it was necessary to add the requirement that the LCS is empty when a subscript is zero [22].

```
Function Levenshtein_Distance(S1,S2)
Begin
for i:=0 to m do
   for j:=0 to n doupper=upperleft=left:=maxint;
      if i>0
        then upper:=dist[i-1,j]+weight(S1[i],ø);
     if i>0 and j>0
      then upperleft:=dist[i-1,j-1]+weight(S1[i],S2[j]);
     if j>0
      then left:=dist[i,j-1]+weight(ø,S2[j]);
       dist[i,j]:=min(upper,upperleft,left);
     if dist[i,j]=maxint then dist[i,j]:=0;
    end
   end
Levenshtein_distance:=dist[m,n];
end
```

Fig 2: Edit distance (Levenshtein) Algorthim

3.1.2 N-Gram Similarity and Distance Algorithm

One of the efficient algorithms for computing string similarity is the n-gram similarity and distance algorithm. Kondrak [23] developed this algorithm to measure similarity between two strings. He showed that edit distance and the length of the LCS are special cases of n-gram. He proved that the main idea of n-gram and distance similarity is generalizing the concept of the longest common subsequence by reporting the results of his experiments. The results suggested that this algorithm outperform the other algorithms.

The affixing method in this algorithm is aimed to emphasize the initial segments, which tends to be much more important than final segments in determining word similarity. The number of n-grams is thus increased from $K+L-2(n-1)$ to $K+L$, where K and L are the lengths of the two compared texts. The normalization is achieved by simply dividing the total similarity score by max (K, L), the original length of the longer text. This procedure guarantees that the new measures return 1 if and only if the texts are identical and 0 if and only if the texts have no letters in common. Figure 3 shows the algorithms for computing the similarity and distance of strings X and Y.

This algorithm is used for the following reasons; first, it is an enhanced version of the most common algorithms edit distance [21] and the length of the longest common subsequence [22]. It is intended to combine the advantages of the unigram (one string or word) and the n-gram (sequence of words or strings) measures. The n-gram similarity and distance algorithm is also applied on three different areas of string science: the word-comparison tasks work, the identification of genetic cognates, and confusable drug names which is very similar to the data types used in this research because most of the elements' contents are kind of confusable labels. In this research and for the purpose of evaluating the effect of appearance similarity on functional similarity, only the appearance similarity between Android applications is measured based on the contents of views (i.e. text elements) in the graphical interfaces.

```
Algorithm N-SIM (X,Y )

K ← length(X)
L ← length(Y )
for u ← 1 to N − 1 do
  X ← x′ 1 + X
  Y ← y′ 1 + Y
for i ← 0 to K do
  S[i,0] ← 0
for j ← 1 to L do
  S[0,j] ← 0
for i ← 1 to K do
  for j ← 1 to L do
    S[i,j] ← max(
    S[i − 1,j],
    S[i,j − 1],
    S[i−1,j−1]+$s_N(\Gamma^N_{i-1,j-1})$)
return S[K,L]/max(K,L)
```

```
Algorithm N-DIST (X,Y )

K ← length(X)
L ← length(Y )
for u ← 1 to N − 1 do
  X ← x′ 1+X
  Y ← y′ 1 + Y
for i ← 0 to K do
  D[i,0] ← i
for j ← 1 to L do
D[0,j] ← j
for i ← 1 to K do
  for j ← 1 to L do
```

Fig 3: The algorithms for computing N-SIM and N-DIST of strings X and Y [23]

3.1.3 *CLANdroid Search Engine*

CLANdroid (Closely Related Android Applications) is a search engine proposed by [8] for detecting similar Android applications. The search engine works by extracting different types of features such as: (1) API calls (Application Programming Interface) which is the set of classes included with the Java Development Environment. These classes are written using the Java language and run on it. The Java API includes everything from collection classes to GUI classes, (2) Intents which are used within applications, (3) User permissions declared in the application's manifest files, and (4) Sensors declared in the source code.

There is an online version of CLANdroid search engine that can be found at http://www.semeru.info/clandroid. It only returns the top 20 ranked similar applications of application’s query.

CLANdroid search engine is used by writing the id (which is the unique name of your application) in a query space, then choosing the search attributes such as same category or all categories in addition to some other properties.

3.2 Phase 2: Analyze an Android app

In this phase, the main features which play an important role in building graphical user interfaces in Android applications are analyzed. As was mentioned in the previous section, the GUI related files are available in the resource files.

In order to analyze an Android's layout (GUI), the xml layout files rather than the source code file is considered. This is because developers use xml layout files to design interfaces through inserting the views (GUI elements) and setting up their properties, where the source code file is used to implement the functionality and the behavior of the application.

However, xml files in the resource directory are compiled into binary format when packaging to apk file. To extract information from this part, Java library apk tool is used to restore the original xml resource files.

In order to measure the GUI similarity between two apps, the focus is on the text contents of the GUI elements. Reviewing literature, it was noted that researchers used only the main views (GUI elements) in their algorithms to measure appearance similarity [24]. The main views used by researchers are text_view, edit_text, image_view, image_button, single_button, radio_button and check_box. The text contents of these views were used in this research to measure similarity.

3.3 Phase 3: Choose the Android Apps Dataset

The apps dataset consists of 50 pairs of APK (Android Package Kit) files of android applications from different markets. The apps are gathered through official Android Markets such as google play and some alternative sources such as CLANdroid search engine. The data (apps) was collected in pairs to apply the proposed method to perform the comparisons. The android apps were chosen from different categories and at the same time they were checked to be available in CLANdroid dataset for comparison purposes.

3.4 Phase 4: Extract Features

The process of extracting features from android apps dataset is explained here and illustrated using an example:

This step starts by fetching android applications from an official market. Figure 4 shows an example of a pair of android applications: dropbox.apk and Microsoft one drive .apk. These two applications’ files were inputted into a program which was developed for preprocessing purposes. Then the files were decompiled to their source codes in order to extract the xml layout from the recourses file. Here, an apk tool in java library is used.

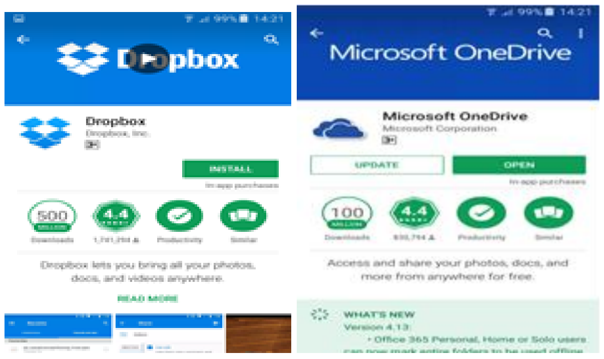

Fig 4: An Example of two Android Applications from the Dataset

The extracted xml layout file contains all the application’s views, i.e. all GUI elements. However, in this research only the main view used by researchers are considered, which are text_view, edit_text, image_view, image_button, single_button, radio_button and check_box. Accordingly, these views will be extracted together with their textual contents, such as their captions, as can be seen in the given example in Figure 5.

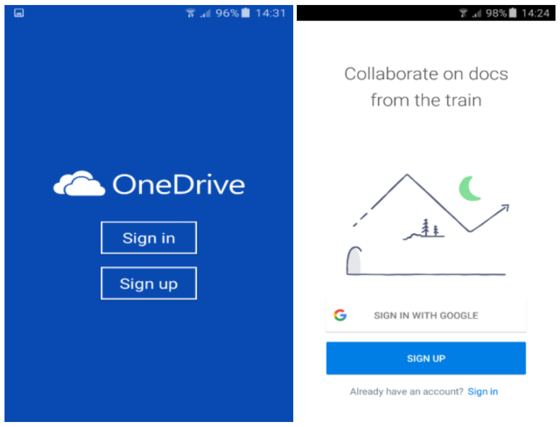
OneDrive
Sign in
Sign up
Collaborate on docs from the train
SIGN IN WITH GOOGLE
SIGN UP
Already have an account? Sign in

Fig 5: Example on views' textual contents

3.5 Phase 5: Design and Develop an Appearance Similarity Algorithm Based on Available Techniques

An appearance similarity algorithm is developed based on the n-gram string similarity and distance algorithm. The input to this algorithm is the pair of applications which degree of appearance similarity needs to be measured. The output is a value based on the similarity calculation. This value is in the range [0,1], 0 means there is no similarity between the two apps and the value 1 indicates that they are identical; otherwise, the apps are similar in a certain degree. The algorithm was implemented and applied to the collected dataset obtained in step 3 which consists of 50 android apps. For each pair, the views text_view, edit_text, image_view, image_button, single_button, radio_button and check_box are extracted from the xml layout file. The first application in each pair of the dataset is represented as a small letter *a* and the application's file is denoted as the capital letter A. So, in this way, the first application can be represented as (a1.1, a1.2, ... , a1.n) and the second app as (a2.1, a2.2…… a2.n). The following steps summarizes the appearance similarity algorithm:

Step 1: Get all the resulting textual contents from the Extraction Phase of the two apps and generate lists of the views' textual contents for the corresponding app.

Step 2: Apply n-Gram Similarity and Distance Algorithm for all element, then save the results for each element and tag the highest similarity value. Figure 6 shows part of the code implementation concerned with text views similarity.

Step 3: Compute the similarity between the two applications as a whole. All elements' similarities are compared to obtain a highest similarity score of each element for every application. The final GUI similarity score between the two apps is calculated. Figure 7 shows part of the code implementation concerned with measuring string similarities for all elements in the two apps.

4. EXPERIMENTS AND RESULTS

This section presents the experiments conducted in this research. The first experiment is to measure appearance similarity using the developed method and the second is to measure functional similarity using CLANdroid. The results of both were compared.

```
List text_Similarity;

variable max_similarity_value=-1;

for element_a1.1value in A1_Text_element{

 for(element a2.1 in A2_Text_element){

  variable temp_value=
getSimilarity(element.getTextContent(),element
2.getTextContent());

  if max_similarity_value < temp_value then {

  max_similarity_value = temp_value}}
```

Fig 6: Text views similarity code snippet

```
var text_similarity_summation=0;

for variable index = 0 , index
<text_Similarity.size(), index++ {

text_similarity_summation+=text_Similarity[index];

}

var image_similarity_summation=0;

for variable index = 0 , index
<image_Similarity.size(), index++ {

image_similarity_summation+=image_Similarity[index]
; }

var radio_similarity_summation=0;

for variable index = 0 , index
<radio_Similarity.size(), index++ {

radio_similarity_summation+=radio_Similarity[index]
; }

variable Total_Similarity_value=
(text_similarity_summation)+
(image_similarity_summation)+
(radio_similarity_summation)
```

Fig 7: String similarities for all elements code snippet

4.1 Experiment 1: Measuring Appearance Similarity

In the first experiment, the proposed algorithm is applied using a sample of 50 Android applications to get their appearance similarities and 25 pairs of applications were selected. Table 1 shows the results of a sample of 10 pair of Android applications. The first two columns show the apps' names and the third column represent their appearance similarity measured using the proposed method. The results range between 0, which means no similarity, and 1 which means an exact similarity. Other values between 0 and 1 represent the degree of similarity of each pairs of apps. For example, measuring the appearance similarity between the Calendar Widget application and the Dropbox application results in 0.28, while between OneDrive app and Dropbox results in 0.41. Measuring the appearance similarity between the application and itself resulted in 1.

Table 1: Appearance similarity for a sample of Apps' pairs

App1	App2	Appearance similarity
Calendar Widget	Dropbox	0.27526176
Calendar Widget	Microsoft One drive	0.27526176
One Drive	Dropbox	0.40730816
Base CRM	Dropbox	0.3880397
Base CRM	Calendar Widget	0.27209589
Base CRM	ONE DRIVE	0.4432375
Messenger	Google Voice	0.3165981
Messenger	AT&T Messages For Tablet	0.124587
Messenger	Yahoo Messenger - Free Chat	0.39381893
Dropbox	Kobo Books - Reading App	0.30613895

4.2 Experiment 2: Measuring Functional Similarity

In the second experiment, an online version of CLANdroid for detecting functional similarity between Android apps using different semantic anchors (i.e., identifiers, API calls, intents, sensors, and user permissions) is used. CLANdroid is available at http://www.semeru.info/clandroid.

The general process in this experiment is as follows. (i) The APK files are chosen directly from Google Play and their id names are fetched, then (ii) the id name of files are written as a query in the search engine of the online version, finally (iii) CLANdroid decompiles the APK file into JAR files and source codes, and extracts semantic anchors from different artifacts: identifiers and intents from source code, APIs and sensors from JAR files, and permissions from the AndroidManifest.xml files. After fetching these data, the search engine retrieves the relevant applications with ranks in descending order using a similarity matrix [8]. Table 2 shows the results of comparing the functional similarities for the same application pairs used in the previous experiment.

Table 2: Clandroid results for measuring functional similarities

Names of applications	CLANdroid (API)	CLANdroid (Identifiers)	CLANdroid (Combined)	CLANdroid (Intent)	Permission	Sensor
Calendar Widget, dropbox	0.97	0.99	0.98	0	0	0
Calendar Widget, Microsoft OneDrive	0.97	0.99	0.98	0	0	0
One drive, dropbox	0.96	0.94	0.95	0	0.53	0
Base CRM, dropbox	0.99	0.99	0.99	0	0.72	0
Base CRM, Calender widget	0.98	0.99	0.99	0	0	0
Base CRM, one drive	0.98	0.96	0.97	0	0	0
Messenger, Google Voice	0.45	0.40	0.42	0	0	0
Messenger, AT&T messenger for tablet	0.08	0.29	0.18	0.28	0	
Messenger, Yahoo messenger-free chat	0.01	0.24	0.12	0.14	0.18	1
Drop, kobo books-reading App	0.98	0	0	0	0.71	0

4.3 Comparing Appearance Similarity with Functional Similarity

The correlation between the appearance similarity and functional similarity measures were calculated for the 25 pairs of Android apps. The results are shown in Table 3.

As can be seen in Table 3, there is a relationship between appearance and functional similarities. A strong relationship exists between appearance similarity and most of the functional similarity anchors. The correlation was weak when compared to the app's identifiers. This is because the appearance similarity measure was based on string similarities of texts extracted from the apps GUI elements.

As intents are mainly used to communicate between Android components such as activities, the textual elements extracted from labels in the communicating activities such as the screens' titles and headers will be similar to intents. Hence, there is strong correlation between the two similarity measures. The same applied to permissions and sensors. However, the use of APIs to perform different functionalities which could be in the background is rarely relevant to texts that appear on the apps screens. The confusing result was the correlation between appearance similarity and identifiers factor of the functional similarity. As programmers usually use identifiers for the views that are relevant to their functionality and hence the captions displayed on the components. This suggests further investigation for this part.

Table 3: correlation between appearance similarity and the different anchors of CLANdroid functional similarity

Anchor used for comparison	CORREL values
API	0.500741077
Identifiers	0.326907624
Combined (API and Id)	0.484172313
Intent	0.942058173
Permission	0.952360282
Sensor	0.966404385

5. CONCLUSION

In this paper, a method to measure appearance similarity in Android applications is developed using N-gram and distance algorithm. The CLANdroid application was used to detect similar application according to some functional factors. Then, the results were compared. The comparison shows that there is a correlation between appearance similarity and functional similarity in terms of intent, permission and sensor usages. Lower impact was found between appearance similarity and both API usage and identifiers of an app. This is due to relaying on string similarity while measuring appearance similarities.

The results of this research suggest the usage of appearance similarity in researches concerned with malware detection and plagiarisms. This research also contributes in providing feature extraction of Android applications into dataset.

6. REFERENCES

[1] Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M. and Kusumoto, S., 2003, May. Component rank: relative significance rank for software component search. In 25th International Conference on Software Engineering, 2003. Proceedings. (pp. 14-24). IEEE.

[2] Liu, C., Chen, C., Han, J. and Yu, P.S., 2006, August. GPLAG: detection of software plagiarism by program dependence graph analysis. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 872-881). ACM.

[3] Sager, T., Bernstein, A., Pinzger, M. and Kiefer, C., 2006, May. Detecting similar Java classes using tree algorithms. In Proceedings of the 2006 international workshop on Mining software repositories (pp. 65-71). ACM.

[4] Gandhewar, N. and Sheikh, R., 2010. Google Android: An emerging software platform for mobile devices. International Journal on Computer Science and Engineering, 1(1), pp.12-17.

[5] Hanna, S., et al., 2012, July. Juxtapp: A scalable system for detecting code reuse among android applications. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 62-81). Springer, Berlin, Heidelberg.

[6] Chang, G., et., 2008, December. Developing mobile applications on the Android platform. In Workshop of Mobile Multmedia Processing (pp. 264-286). Springer, Berlin, Heidelberg.

[7] Shafirov, M., 2017. Kotlin on android. Now official. A: Jetbrains. Kotlin Blog, 17.

[8] Linares-Vásquez, M., Holtzhauer, A. and Poshyvanyk, D., 2016, May. On automatically detecting similar Android apps. In 2016 IEEE 24th International Conference on Program Comprehension (ICPC) (pp. 1-10). IEEE.

[9] Grechanik, et al., 2010, May. A search engine for finding highly relevant applications. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 (pp. 475-484). ACM.

[10] Bajracharya, S.K., Ossher, J. and Lopes, C.V., 2010, November. Leveraging usage similarity for effective retrieval of examples in code repositories. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (pp. 157-166). ACM.

[11] Crussell, J., 2014. Scalable Semantics-Based Detection of Similar Android Apps: Design, Implementation, and Applications. University of California, Davis.

[12] Li, S., et al., 2012. Juxtapp and dstruct: Detection of similarity among android applications. EECS Department, University of California.

[13] Chen, K., et al., 2014, May. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In Proceedings of the 36th International Conference on Software Engineering (pp. 175-186). ACM.

[14] Gorla, A., Tavecchia, I., Gross, F. and Zeller, A., 2014, May. Checking app behavior against app descriptions. In Proceedings of the 36th International Conference on Software Engineering (pp. 1025-1035). ACM.

[15] Desnos, A., 2012, January. Android: Static analysis using similarity distance. In 2012 45th Hawaii International Conference on System Sciences (pp. 5394-5403). IEEE.

[16] Wang, H., et al 2015. Wukong: A scalable and accurate two-phase approach to android app clone detection. In Proceedings of the 2015 International Symposium on Software Testing and Analysis (pp. 71-82). ACM.

[17] Shao, Yet al., 2014, December. Towards a scalable resource-driven approach for detecting repackaged Android applications. In Proceedings of the 30th Annual Computer Security Applications Conference (pp. 56-65). ACM.

[18] Thung, F., Lo, D. and Jiang, L., 2012, September. Detecting similar applications with collaborative tagging. In 2012 28th IEEE International Conference on Software Maintenance (ICSM) (pp. 600-603). IEEE.

[19] Zhu, J., Wu, Z., Guan, Z. and Chen, Z., 2015, March. Appearance similarity evaluation for Android applications. In 2015 Seventh International Conference on Advanced Computational Intelligence (ICACI) (pp. 323-328). IEEE.

[20] Jadhav Anita, et al, 2017. A Survey on Appearance Similarity Evaluation For Android Application. International Journal of Engineering Research and Management, 3 (1), pp. 1-4.

[21] Yujian, L. and Bo, L., 2007. A normalized Levenshtein distance metric. IEEE transactions on pattern analysis and machine intelligence, 29(6), pp.1091-1095.

[22] Bergroth, L., Hakonen, H. and Raita, T., 2000. A survey of longest common subsequence algorithms. In Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000 (pp. 39-48). IEEE.

[23] Kondrak, G., 2005, November. N-gram similarity and distance. In International symposium on string processing and information retrieval (pp. 115-126). Springer, Berlin, Heidelberg.

[24] Meier, R., 2012. Professional Android 4 application development. John Wiley & Sons.