Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.18576/amis/190509

MACT: A Novel Framework for Automated Mobile Application Testing Using Machine Learning

Moheb R. Girgis, Alaa M. Zaki, Enas Elgeldawi, Mohamed M. Abdallah, and Ali A. Ahmed[*]

Computer Science Department, Faculty of Science, Minia University, Minia, Egypt

Received: 22 Feb. 2025, Revised: 12 May 2025, Accepted: 27 Jun. 2025.
Published online: 1 Sep. 2025

Abstract: The rapid expansion of mobile applications has amplified challenges in software testing, with inadequate testing responsible for 42% of application failures. Traditional testing methods are often time-consuming, resource-intensive, and hindered by issues such as GUI element identification and cross-device compatibility. This paper introduces MACT (Mobile Application Classification and Testing), a novel framework for automated mobile application testing that integrates machine learning-based activity classification with predefined, reusable test case execution. The framework automatically identifies and categorizes Android application screens (e.g., login, settings) and applies tailored test cases for each screen type, eliminating the need for custom test scripts and reducing maintenance overhead. Leveraging the MASC dataset, developed in our previous work, which comprises 7,065 screens from over 3,400 apps, the framework employs the Gradient Boosting model in activity classification that achieves 93.48% accuracy. Empirical evaluations demonstrate that MACT detects all planted bugs with 100% accuracy, significantly outperforming traditional tools like the Android Application Monkey. The framework reduces test script size by 87% and testing duration by 89% compared to manual methods like Android Espresso, providing a scalable, efficient, and resilient approach to mobile application testing. The modular design of MACT ensures seamless adaptation to various applications, addressing key limitations in GUI modeling and automated testing. This research makes a significant contribution to mobile application testing by reducing manual effort, increasing test coverage, and boosting reliability and efficiency.

Keywords: Automated Testing; Mobile applications; MACT; Android applications testing; Machine Learning Algorithms.

1 Introduction

The continuous expansion of mobile applications has revolutionized how users engage with digital platforms, such as Android hosting over 2.3 million applications [1]. This expansion has intensified application testing challenges, as studies indicate that inadequate testing accounts for approximately 42% of application failures [2]. Moreover, software testing consumes nearly 50% of development costs, a percentage potentially higher for mobile applications due to their inherent complexities [3].

Automation allows developers to easily conduct extensive test suites, saving time and resources compared to manual approaches [4]. Current frameworks allow developers to record predetermined actions, run test scenarios, and compare the results to expected behaviors [5]. Reusable test cases allow developers to validate functionality across different device setups and scenarios, which would otherwise take a lot of time with manual testing [6]. Despite these benefits, mobile application testing presents significant challenges. According to studies, 32% of testing efforts are dedicated to maintaining and upgrading test suites in response to frequent application changes [7]. Additional challenges include GUI element identification (31.2%), test environment setup (27.8%), and cross-device compatibility (24.5%) [8] .The results of empirical research on 1,000 Android apps showed that traditional testing methods require substantial manual effort, with teams dedicating an average of 41 hours per month to test script maintenance [9]. Moreover, only 41% of mobile development teams effectively adopt automation practices due to technical difficulties, such as managing activity states, dynamic content, and asynchronous events [3].

To solve these difficulties, we propose an innovative Mobile Application Classification and Testing (MACT) framework that combines machine learning-driven activity classification with predefined test case execution. This framework automatically identifies and categorizes Android application screens such as login, settings, and list views activities and applies reusable test cases tailored to each screen type, significantly reducing the need for custom scripts.

This approach has several advantages over traditional testing methods, which include eliminating the need to write custom test scripts for common UI patterns, lowering maintenance overhead, and ensuring consistent testing across similar screens in different applications. Furthermore, the framework includes a repository of predefined test scripts that can be reused for similar activities. These scripts use dynamic widget detection technique to ensure robust test case execution across multiple application contexts. This approach reduces the

[*]Corresponding author e-mail: ali.ahmed@mu.edu.eg

effort needed to maintain tests while also ensuring consistent testing across similar app components. The proposed approach provides several important contributions:

1. A framework for automatically classifying activities based on Machine Learning (ML).

2. A library of reusable test cases for common Android UI activities.

3. An integrated framework linking activity classification with automated testing.

This research aims to reduce the cost and effort of mobile app testing while enhancing test coverage and reliability. By making automated testing more accessible and maintainable, it encourages adoption among mobile development teams, leading to better software quality and shorter development time.

The remainder of this paper is organized as follows: Section 2 reviews related work in mobile app testing automation. Section 3 describes the proposed mobile testing methodology and framework architecture. Section 4 presents implementation specifics. Section 5 presents a case study to illustrate the working of MACT. Section 6 provides empirical evaluation results. Section 7 discusses findings and limitations, and Section 8 provides the research conclusion and future directions.

2. Related work

Testing mobile applications has been the focus of several studies, with researchers exploring various methods and tools. Linares-Vasquez et al. [10] classified Android testing tools and methodologies into three main categories, laying the groundwork for further exploration. These categories are:

- **Random Exploration Strategy:** This approach involves generating random, independent UI events to test applications. It is especially useful for stress testing because it can create many test sequences with minimal effort. However, it lacks precision in delivering specific inputs, doesn't consider coverage levels, and often results in repetitive or redundant sequences. Tools such as Monkey [11], and Dynodroid [12] employ this strategy.

- **Systematic Exploration Strategy:** This strategy uses advanced techniques, such as genetic algorithms, to guide testing toward untested sections of code. Its goal is to identify hidden faults, enhance coverage, and focus on areas more likely to contain defects. While effective, it faces challenges in scaling up. Tools like CrashScope [13], and EvoDroid [14] implement this strategy.

- **Model-Based Exploration Strategy**: This strategy involves creating a symbolic model of the application, which is treated as a finite-state machine. In this model, each state represents a specific screen, and transitions between states are based on user interactions. The exploration process dynamically generates new states and transitions until all possible paths loop back to previously explored states. Although this method ensures thorough and non-redundant GUI coverage, it might not detect internal changes that don't visibly impact the graphical interface. Tools such as MobiGUItar [15], and GuiRipper [16] implement this strategy.

While previous studies emphasize the importance of GUI modeling and classification, they lack comprehensive frameworks that integrate all facets of these aspects. For instance, Yang et al. [17] proposed LACTA, which classifies Android applications using domain-specific knowledge and topic extraction via LDA. Despite its promising results, LACTA relies heavily on Android expertise and a limited dataset of 42 applications.

Li et al. [18] introduced ClassifyDroid, an approach that analyzes API usage frequencies from decompiled app code and applies a semi-supervised Multinomial Naive Bayes classifier. This method is particularly effective with limited labeled data. Similarly, Hamedani et al. [19] developed AndroClass, which extracts and refines features like APIs and intents for application classification using machine learning techniques. AndroClass achieved high accuracy with curated datasets but lacks focus on automated testing.

Rosenfeld et al. [20], proposed a ML-based approach that reuses popular test scenarios to automate testing. While effective, this method relies on pre-existing scripts, limiting its adaptability to specialized applications. Hu et al. [21] introduced AppFlow, which uses ML to identify common screens and widgets, enabling reusable UI test synthesis. Despite reducing testing effort by up to 90%, its dependence on predefined libraries restricts its flexibility. Ardito et al. [22] presented a framework for Android app testing that automates script generation. This modular framework combines GUI modeling, app classification, and activity classification, enabling tailored test scripts. Empirical results demonstrated its adaptability and efficiency, outperforming existing approaches. To better illustrate the specific limitations in prior work, Table 1 summarizes the key approaches and their respective constraints, highlighting the gaps that our proposed framework aims to address.

Table 1: Summary of limitations in existing approaches

Study	Proposed Approach	Limitations
Yang et al. [17]	LACTA: Classifies apps using domain-specific knowledge and topic extraction via LDA.	- Heavily relies on Android expertise. - Limited dataset with only 42 applications.
Li et al. [18]	ClassifyDroid: Analyzes API usage frequencies using decompiled app code with a semi-supervised Multinomial Naive Bayes classifier.	- Effective with limited labeled data but may struggle with large, unstructured datasets. - Lacks focus on automated testing.
Hamedani et al. [19]	AndroClass: Extracts and refined features like APIs and intents for app classification using ML techniques.	- High accuracy only with curated datasets. - Does not focus on automated testing.
Rosenfeld et al. [20]	ML-based approach that reuses popular test scenarios for automation.	- Relies on pre-existing test scripts, limiting adaptability to 2 specialized applications. - Limited dataset with only 80 activity samples.
Hu et al. [21]	AppFlow: Uses ML to identify common screens and widgets, enabling reusable UI test synthesis.	- Depends on predefined libraries, limiting flexibility. - Reduces testing effort by up to 90% but struggles with unique interfaces.
Ardito et al. [22]	A framework combining GUI modeling, app classification, and activity classification to generate tailored test scripts.	- Highly efficient but relies on APK files instead of source code. - Limiting deeper analysis of structure and behavior. - Limited dataset with only 100 activity samples.

Compared to prior work, the proposed framework introduces a more advanced solution for automated Android app testing. Unlike Rosenfeld et al. [20], who focused on reusing predefined test scenarios, and Hu et al. [21], who introduced modular test generation, our approach leverages ML in activity classification to create pre-defined test cases tailored to each activity category. These test cases are dynamically applied based on the recognized category, ensuring efficient and adaptive testing.

The proposed framework also builds upon the advancements presented by Ardito et al. [22], who developed a modular framework combining GUI modeling, app classification, and activity classification to generate tailored test scripts. While their framework demonstrated adaptability and efficiency, our approach extends this by integrating ML for more accurate activity classification and leveraging the application source code instead of APK files. This enables a deeper analysis of application structure and behavior. Additionally, we employ a larger and more diverse dataset compared to previous studies, further enhancing classification and testing accuracy. This combination of techniques results in a more robust, scalable, and effective testing framework that addresses limitations in existing methodologies and significantly improves Android apps performance.

3. The Proposed Testing Framework

The proposed framework, referred to as **MACT (Mobile Application Classification and Testing)**, implements a comprehensive automated testing approach for mobile applications, building upon the ML-based screen classification system. Figure 1 illustrates the testing process as structured within the MACT framework. It consists of the following modules:

- **Application Analyzer:** This module analyzes the mobile application's source code to extract all activities and their associated components. Key elements such as buttons, text fields, images, and layouts are identified to gather relevant features. These features are then processed and transformed into normalized vector representations, ensuring seamless compatibility with the employed ML model for activity classification.
- **Activity Classification:** The activity classification process is triggered for each identified activity extracted from the source code. Using the ML model trained on our MASC dataset, which comprises 7,065 screens from over 3,400 apps, the features of each activity are analyzed to determine its category. This classification step ensures that the activity type is accurately identified, enabling the framework to map it to the appropriate test cases and execute relevant tests effectively.
- **Test Case Management :**This module incorporates a repository of generic test scripts broadly applicable across various application types and activity categories, such as Login, Settings, and Profile. During the testing process, the module dynamically selects and executes the appropriate test scripts based on the classified activity, ensuring that only relevant tests are applied to enhance activity testing precision and effectiveness. Leveraging Android Espresso [23], a robust tool for UI automation, the module validates the application's

behavior and functionality with a high degree of automation. The module is designed to adapt dynamically to the unique attributes and requirements of each identified activity, ensuring full and context-specific test coverage. Additionally, the module collects detailed execution results, including pass/fail status, execution time, and error logs, and generates comprehensive test reports summarizing the testing outcomes.

Detailed description of each module of MACT and its implementation will be provided in the subsequent section.

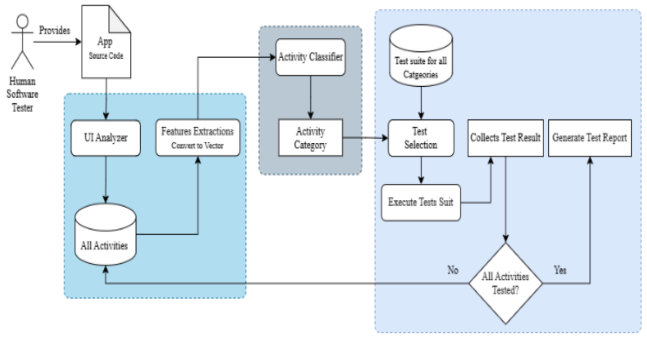


Fig. 1: Workflow of the complete testing process.

4. The MACT Framework Implementation

4.1 Application Analysis

The first module of the proposed framework is the ***Application Analyzer***, which focuses on application source code analysis. This module extracts all activities and their associated features from the mobile application's source code, instead of using an APK file. For each activity, the module extracts the associated layout files and relevant metadata. Following this, key UI components such as buttons, text fields, images, and layouts are identified, and their specific attributes are captured.

4.2 Activity Classification

The second module of the proposed framework is the *Activity Classifier*. This module categorizes each identified activity within a given Android application by using the Gradient Boosting algorithm. This classification is grounded on the observation that numerous activities, even across different applications, exhibit shared characteristics due to their adherence to common structural and design patterns. These recurring patterns can be leveraged for functional testing purposes. For example, many applications include a Setting activity, which typically follows a consistent design structure, often incorporating a ListView element containing clickable items and toggle switches. This structural similarity suggests that such activities may require similar test cases. As illustrated in Figure 2, the workflow of the activity classification process involves identifying these patterns. By categorizing activities based on their shared characteristics, the framework enhances both efficiency and consistency in the testing process. This approach aligns with established practices in automated testing, where commonalities in application design are utilized to streamline test case development and execution.

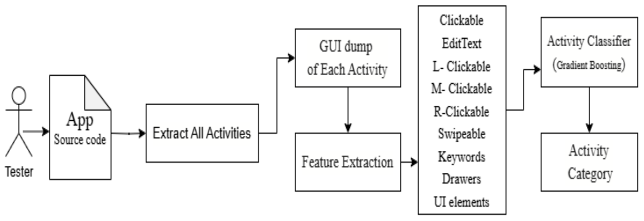


Fig. 2: Workflow of the Activity Classification Process.

Building The Dataset

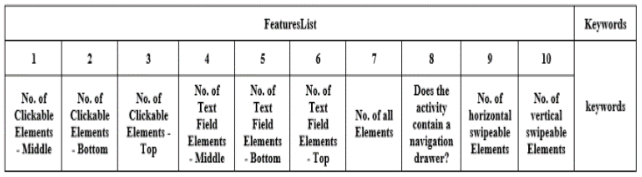
In our previous research (currently under publication [24]) we introduced a novel dataset called MASC (Mobile App Screens Classification) to facilitate the application of ML in mobile UI classification. It was created by collecting 30,000 mobile UI screenshots from three open-source datasets: Rico [25], Screen2Words [26], and Enrico [27]. Following a rigorous selection process and manual classification by three annotators, only those with agreement from at least two annotators were included to ensure consistency and accuracy. This rigorous process is aimed at building a high-quality dataset suitable for ML applications in mobile UI classification. Of over 3,400 apps, 7,065 screenshots were retained. These screens were categorized into ten distinct design classes. Table 2 presents the number of UI designs belonging to each class along with a brief description.

Table 2: Composition of the MASC Dataset.

Class	No. of labeled Screens	Description
Chat	329	Chat communication
Home	866	App navigation
List	960	Data visualization in a column
Login	889	User authentication
Map	500	Geographic display
Menu	557	Item selection
Profile	526	User profile information
Search	725	Content discovery
Setting	629	App configuration
Welcome	1084	First-run experience

4.2.2 Feature Extraction

Careful selection of feature improves classification accuracy and reduced model training time. The feature extraction process focused on identifying the most informative aspects of each activity screen, which vary according to the activity type. Drawing from Wang et al.'s findings [26], it was assumed that interactive elements, such as buttons and text boxes, along with their spatial distribution, effectively represented the activity. Each screen was divided into three parts—top (15%), middle (70%), and bottom (15%)—based on Android Studio's activity design guidelines. This division helped capture the spatial distribution of UI elements, which was crucial for distinguishing between different screen types. The feature selection process was guided by common UI design patterns, quantifying clickable elements, text fields, and swipeable components to capture screen interactivity. Login screens typically contained more text fields, while list screens had more vertically swipeable elements. Keyword analysis further refined classification by identifying frequent words for each screen type, such as 'username' and 'password' for login screens and 'settings' and 'preferences' for settings screens. The final feature set included 11 features, grouped into three categories. The first set comprised six features, representing the number of clickable and text field elements in each of the three screen sections. The second set included four features: the total number of elements on the screen, the presence of a navigation drawer, and the number of horizontal and vertical swipeable elements in the center. The final feature was based on keywords found in the screen's text. These 11 features formed the feature vector for each screen, as illustrated in Figure3, enabling the model to effectively classify mobile app screens.



FeaturesList										Keywords
1	2	3	4	5	6	7	8	9	10	keywords
No. of Clickable Elements - Middle	No. of Clickable Elements - Bottom	No. of Clickable Elements - Top	No. of Text Field Elements - Middle	No. of Text Field Elements - Bottom	No. of Text Field Elements - Top	No. of all Elements	Does the activity contain a navigation drawer?	No. of horizontal swipeable Elements	No. of vertical swipeable Elements	

Fig. 3: The feature vector structure for any activity screen.

ML Model

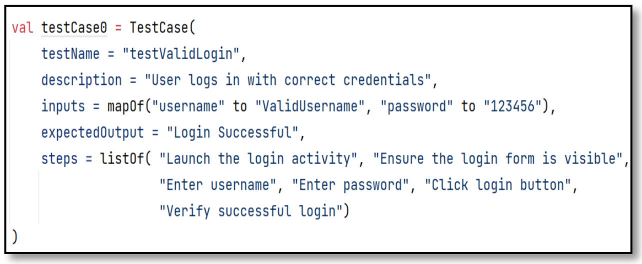
In our previous study, we evaluated ten machine learning algorithms for mobile app screen classification, including Gradient Boosting, XGBoost, Random Forest, and SVM. Performance was assessed using accuracy, precision, recall, ROC-AUC, and F1-score. Gradient Boosting achieved the highest accuracy (93.48%), proving its effectiveness in handling non-linear data and resisting overfitting. Based on these results, it was selected as the most suitable algorithm for this task in the proposed framework.

4.3 Test Case Management

The third module of the MACT framework is ***Test Case Management***, which is responsible for automating the execution of test cases based on the classified UI categories. It ensures that the test cases are dynamically adapted to the application's context and executed efficiently. Below is a detailed explanation the components of this module.

4.3.1 Test Case Repository

The Test Case Repository serves as the backbone of the framework, housing a collection of pre-defined test cases tailored to specific UI categories. Each test case is encapsulated in a TestCase object, which includes metadata such as the test name, description, inputs, expected outputs, and execution steps. Figure 4 shows the structure of a sample test suite metadata, highlighting key fields such as test name, description, and execution steps. This repository ensures that the framework can handle a wide range of scenarios, from simple login validations to complex payment workflows. By organizing test cases into categories, the framework can quickly retrieve and execute relevant tests based on the classified UI



```
val testCase0 = TestCase(
    testName = "testValidLogin",
    description = "User logs in with correct credentials",
    inputs = mapOf("username" to "ValidUsername", "password" to "123456"),
    expectedOutput = "Login Successful",
    steps = listOf( "Launch the login activity", "Ensure the login form is visible",
                "Enter username", "Enter password", "Click login button",
                "Verify successful login")
)
```

Fig. 4: A sample test suite metadata.

4.3.2 Test Case Execution

The Test Case Execution component is responsible for automating the interaction with the application's UI and verifying the expected outcomes. It consists of three main sub-components, each playing a vital role in ensuring the accuracy and efficiency of the testing process. The module was developed using Kotlin [28], a statically-typed language preferred for Android development, due to its concise syntax, Java interoperability, and features like null safety and coroutines. Combined with the Android Espresso framework [23], it enables efficient UI testing and reduces boilerplate code, making it ideal for scalable and maintainable frameworks. Below is a detailed explanation of each sub-component:

4.3.2.1 UI Element identifier

The UI Element Identifier is responsible for locating and interacting with the actual UI elements during test execution. This step ensures that the test cases can adapt to dynamic changes in the application's UI, such as variations in element IDs or layouts. The identification process involves parsing the complete GUI structure of the screen, considering three key factors:

- **Textual Hints:** Textual information, such as text content, content descriptions, and resource IDs, is used to determine the compatibility of a widget with specific interactions.
- **Class Type:** The class type of a widget determines its compatibility with specific interactions. For instance, toggle action is compatible with spinners but not with buttons. ensuring that only suitable elements are

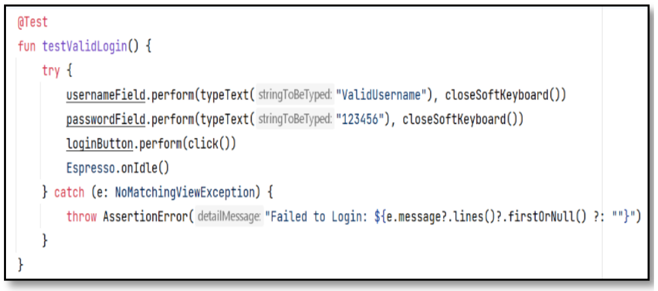
selected for operations, thereby reducing errors during test execution.

- **Attribute Values:** Attribute values help verify a widget's compatibility with interactions. For instance, the **isChecked** attribute determines a Switch's state (on/off), while attributes like **clickable** or **isDisplayed** ensure the widget is ready for interaction.

We used the heuristic of selecting the widget with the most compatible factors. We use each detected widget's unique ID to communicate with it within the Kotlin test case. After identifying the UI elements, the framework proceeds to execute the test cases using the Test Executor module.

4.3.2.2 Test Executor

The Test Executor is responsible for performing the actual execution of test cases. Once the current screen category is identified, the module retrieves the relevant test cases from the repository and executes the current screen with them. Actions such as typing text, clicking buttons, and verifying results are performed programmatically using Android Espresso, ensuring accurate and efficient UI testing across the application. Figure 5 shows, a sample test case script demonstrating how the framework automates interactions and validations, ensuring consistency in UI testing.

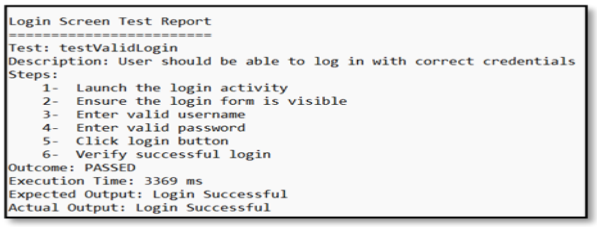


```
@Test
fun testValidLogin() {
    try {
        usernameField.perform(typeText( stringToBeTyped: "ValidUsername"), closeSoftKeyboard())
        passwordField.perform(typeText( stringToBeTyped: "123456"), closeSoftKeyboard())
        loginButton.perform(click())
        Espresso.onIdle()
    } catch (e: NoMatchingViewException) {
        throw AssertionError( detailMessage: "Failed to Login: ${e.message?.lines()?.firstOrNull() ?: ""}")
    }
}
```

Fig. 5: A sample test case script

4.3.2.3 Test Watcher

The Test Watcher is a monitoring module designed to track the status and performance of each test case. It records whether a test case passed or failed, measures the execution time for each test, captures error messages and logs for failed cases, and, at the end of each test case, generates a detailed report summarizing the results, as shown in Figure 6. This test report includes pass/fail status, execution time, and error details, providing developers with actionable insights for debugging and optimization. At the end of a test case execution, the results are assessed to determine the success or failure of the test. Following this, the application is reset to its initial state to ensure a consistent environment for subsequent tests. If additional test cases are associated with the current activity type, they are executed sequentially. Otherwise, the exploration advances to the next application state, continuing the systematic testing process.

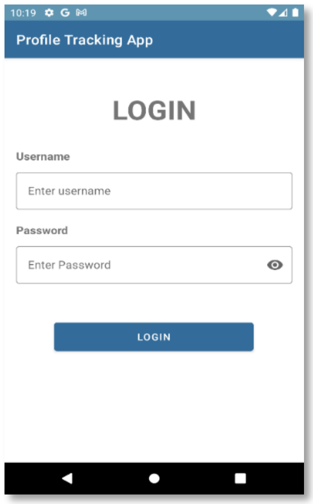


```
Login Screen Test Report
========================
Test: testValidLogin
Description: User should be able to log in with correct credentials
Steps:
    1-  Launch the login activity
    2-  Ensure the login form is visible
    3-  Enter valid username
    4-  Enter valid password
    5-  Click login button
    6-  Verify successful login
Outcome: PASSED
Execution Time: 3369 ms
Expected Output: Login Successful
Actual Output: Login Successful
```

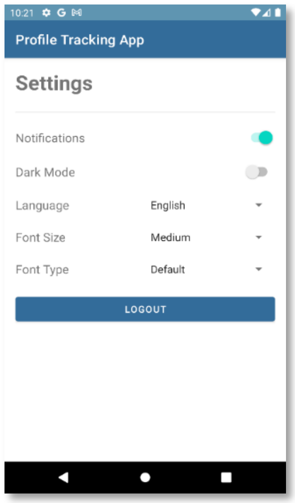
Fig. 6: A sample test case result report

5. Case Study

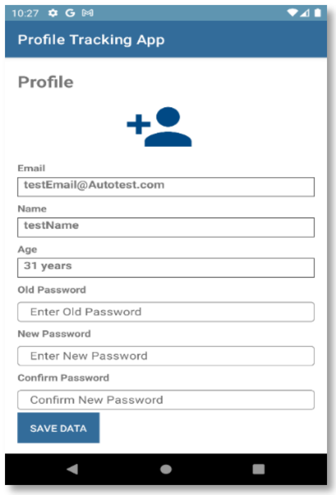
To illustrate the working of MACT, we have applied it to an example app, called **ProfileTrackingAPP** [29], which includes three activities, **activity_main, activity_profile2**, and **activity_settings**, as shown in Figure 7



(a) UI of activity_main



(b) UI of activity_Settings

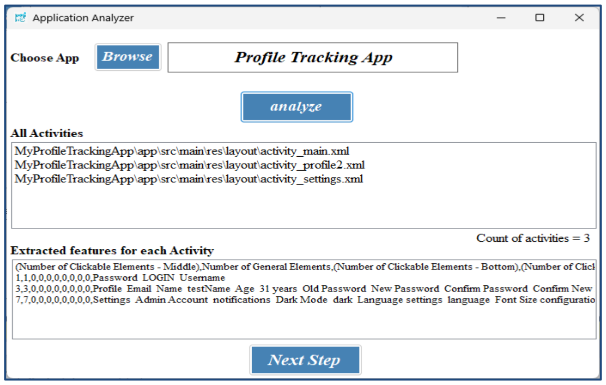


Email: testEmail@Autotest.com
Name: testName
Age: 31 years

(c) UI of activity_profile2

Fig. 7: The three activities of the example app ProfileTrackingAPP

In the first step, MACT analyzes the target mobile application to identify its structural components and extract relevant features from its UIs. As shown in Figure 8, this step begins with selecting the application to be tested through the "**Browse**" button. Then, upon clicking the "**Analyze**" button, MACT accesses and scans the application's XML layout files to detect and list all activities within the app. In this case, MACT identified three primary activities: **activity_main.xml, activity_profile2.xml**, and **activity_settings.xml**, which are listed in the section "All activities", as shown in Figure 8. Following activity identification, MACT extracted key UI components and their associated textual content, which are listed in the section "Extracted features for each activity" in Figure 8.

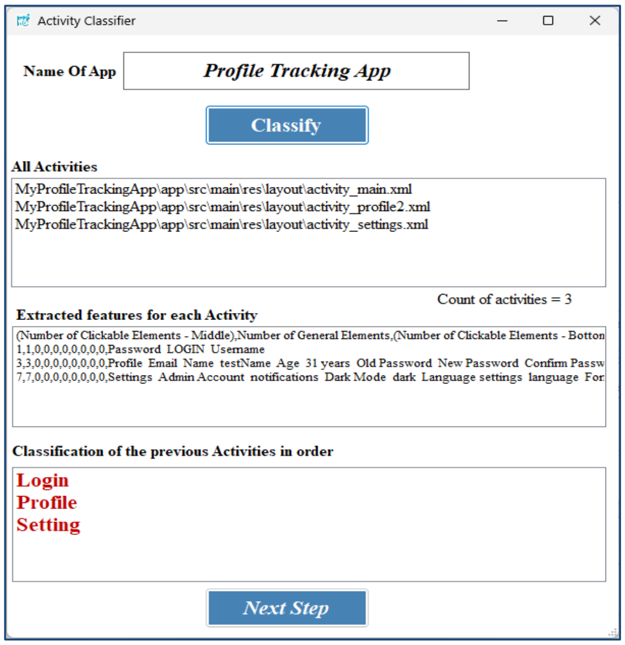


```
MyProfileTrackingApp\app\src\main\res\layout\activity_main.xml
MyProfileTrackingApp\app\src\main\res\layout\activity_profile2.xml
MyProfileTrackingApp\app\src\main\res\layout\activity_settings.xml
```

```
(Number of Clickable Elements - Middle),Number of General Elements,(Number of Clickable Elements - Bottom),(Number of Click
1,1,0,0,0,0,0,0,0,0,Password  LOGIN  Username
3,3,0,0,0,0,0,0,0,0,Profile  Email  Name  testName  Age  31 years  Old Password  New Password  Confirm Password  Confirm New
7,7,0,0,0,0,0,0,0,0,Settings  Admin Account  notifications  Dark Mode  dark  Language settings  language  Font Size configuratio
```

Fig. 8: The first step in MACT

In the second step, upon clicking the "**Next Step**" button, MACT classifies the extracted activities based on their functional characteristics and identified key UI components. The input to this step consists of the extracted activities and their associated features, which were identified in the first step. Based on these features, MACT classifies each activity into one of the predefined types. For instance, **activity_main** is classified as a ***Login*** screen as it includes username and password fields, **activity_profile2** is classified as a ***Profile*** screen as it includes user profile attributes such as email, name, and age, and **activity_settings** is classified as a ***Setting*** screen as it includes configuration options like dark mode and font settings. Figure 9. displays the classification results in an ordered list. Upon completing this step, the user proceeds to the next step, where the extracted data is used to select and execute automated test scripts.



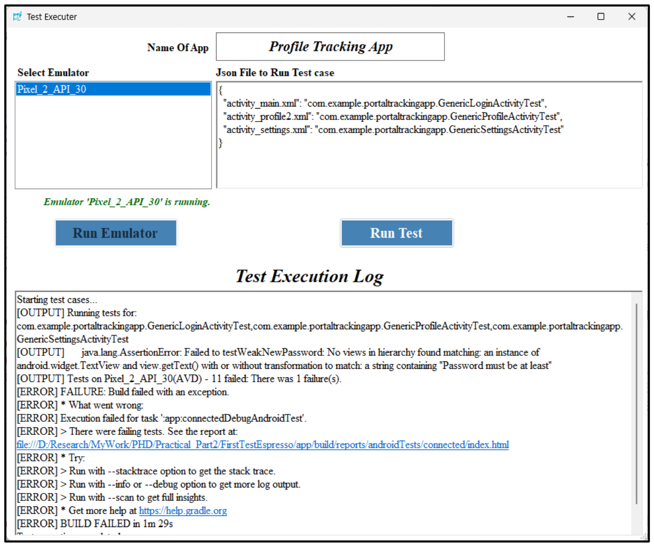
```
MyProfileTrackingApp\app\src\main\res\layout\activity_main.xml
MyProfileTrackingApp\app\src\main\res\layout\activity_profile2.xml
MyProfileTrackingApp\app\src\main\res\layout\activity_settings.xml
```

```
(Number of Clickable Elements - Middle),Number of General Elements,(Number of Clickable Elements - Botton
1,1,0,0,0,0,0,0,0,0,Password  LOGIN  Username
3,3,0,0,0,0,0,0,0,0,Profile  Email  Name  testName  Age  31 years  Old Password  New Password  Confirm Passw
7,7,0,0,0,0,0,0,0,0,Settings  Admin Account  notifications  Dark Mode  dark  Language settings  language  For
```

```
Login
Profile
Setting
```

Fig. 9: The result of second step in MACT

In the third step, upon clicking the "**Next Step**" button, MACT executes the predefined test cases on the target application using an emulator. Figure 10 shows the test execution process, which begins by loading the JSON file extracted in the second step. This file contains the classified activities and their corresponding test cases. An emulator, such as Pixel 2 API 30, is selected from the available options, and its status is verified. If the emulator is not running, it is manually launched. Upon clicking the "Run Test" button, MACT automatically executes the test cases for each activity, including **GenericLoginActivityTest**, **GenericProfileActivityTest**, and **GenericSettingsActivityTest**. The execution details, such as the activities tested, pass/fail status, and execution time, are recorded in the Test Execution Log. In the case of errors, they are displayed in the log to facilitate debugging. For example, the log indicates a successful build (BUILD SUCCESSFUL) in 1 minute and 6 seconds. Detailed test reports are then generated, providing developers with actionable insights to identify and resolve issues effectively.



```
{
 "activity_main.xml": "com.example.portaltrackingapp.GenericLoginActivityTest",
 "activity_profile2.xml": "com.example.portaltrackingapp.GenericProfileActivityTest",
 "activity_settings.xml": "com.example.portaltrackingapp.GenericSettingsActivityTest"
}
```

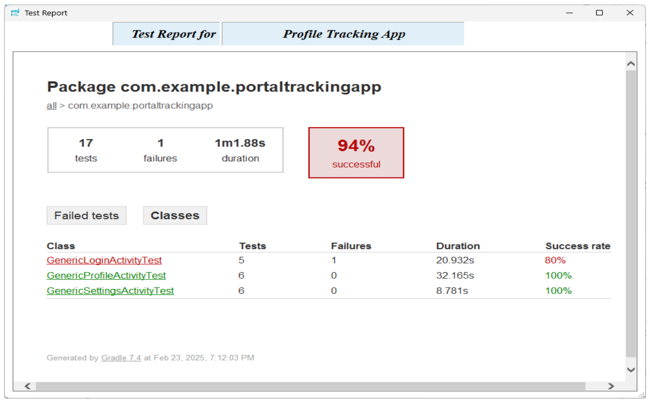
```
Starting test cases...
[OUTPUT] Running tests for:
com.example.portaltrackingapp.GenericLoginActivityTest,com.example.portaltrackingapp.GenericProfileActivityTest,com.example.portaltrackingapp.
GenericSettingsActivityTest
[OUTPUT]      java.lang.AssertionError: Failed to testWeakNewPassword: No views in hierarchy found matching: an instance of
android.widget.TextView and view.getText() with or without transformation to match: a string containing "Password must be at least"
[OUTPUT] Tests on Pixel_2_API_30(AVD) - 11 failed: There was 1 failure(s).
[ERROR] FAILURE: Build failed with an exception.
[ERROR] * What went wrong:
[ERROR] Execution failed for task ':app:connectedDebugAndroidTest'.
[ERROR] > There were failing tests. See the report at:
file:///D:/Research/MyWork/PHD/Practical_Part2/FirstTestEspresso/app/build/reports/androidTests/connected/index.html
[ERROR] * Try:
[ERROR] > Run with --stacktrace option to get the stack trace.
[ERROR] > Run with --info or --debug option to get more log output.
[ERROR] > Run with --scan to get full insights.
[ERROR] * Get more help at https://help.gradle.org
[ERROR] BUILD FAILED in 1m 29s
```

Fig. 10: The third step in MACT. Part of the test execution report, which shows the test case of weak password.

In the final step, a test report summarizing the execution results of all predefined test cases is produced, as shown in Figure 11. The report includes the total number of tests executed, successful and failed cases, and the overall execution time. As shown in the results, 17 test cases were executed, with one failure detected, resulting in a 94% success rate. In this report, by clicking the test name of an activity in the ***Class*** column, a detailed test report for this activity is displayed, which shows the number of tests, execution duration, success rate, and a list of the test cases executed on it. For example, by clicking the test name of ***activity_main*** in the ***Class*** column in Figure 11, a detailed test report for this activity is displayed, as shown in Figure 12.

Additionally, MACT produces a text file containing a detailed test report, as shown in Figure 13. By viewing this report, developers can further explore detailed test case results for each activity to analyze the performance of the app under test and fix any potential issues.

This final step ensures that the application meets the expected quality standards before deployment, providing a clear assessment of test effectiveness and system stability.


Test Report

Test Report for *Profile Tracking App*

Package com.example.portaltrackingapp

all > com.example.portaltrackingapp

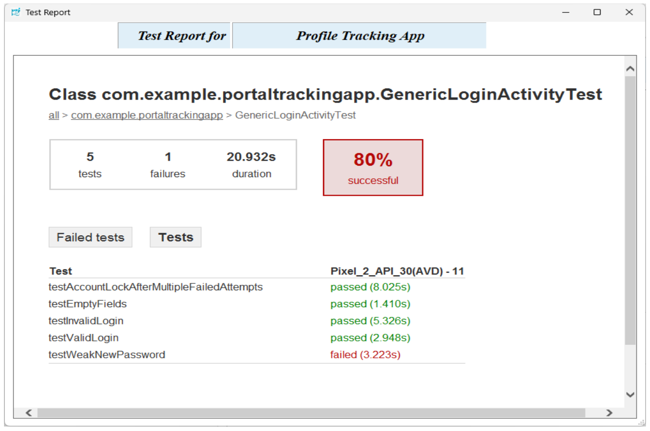
17 tests | 1 failures | 1m1.88s duration | 94% successful

Failed tests | Classes

Class	Tests	Failures	Duration	Success rate
GenericLoginActivityTest	5	1	20.932s	80%
GenericProfileActivityTest	6	0	32.165s	100%
GenericSettingsActivityTest	6	0	8.781s	100%

Generated by Gradle 7.4 at Feb 23, 2025, 7:12:03 PM

Fig. 11: Report on all test cases


Test Report

Test Report for *Profile Tracking App*

Class com.example.portaltrackingapp.GenericLoginActivityTest

all > com.example.portaltrackingapp > GenericLoginActivityTest

5 tests | 1 failures | 20.932s duration | 80% successful

Failed tests | Tests

Test	Pixel_2_API_30(AVD) - 11
testAccountLockAfterMultipleFailedAttempts	passed (8.025s)
testEmptyFields	passed (1.410s)
testInvalidLogin	passed (5.326s)
testValidLogin	passed (2.948s)
testWeakNewPassword	failed (3.223s)

Fig. 12: Report on test cases of activity_main

activity_main_Test Report

==========================

Test: testInvalidLogin
Description: User should not be able to log in with incorrect credentials
Steps:
entering a valid username with an invalid password, clicking the login button, verifying the error message, clearing the fields, then entering an invalid username with a valid password, clicking the login button again, verifying the error message, and ensuring the second activity is not launched.
Outcome: Passed
Execution Time: 5326 ms
Expected Output: Invalid credentials.
Actual Output: Invalid credentials.

Test: testWeakNewPassword
Description: System should reject passwords shorter than 6 characters
Steps:
entering a valid username and a short password (less than 6 characters), clicking the login button, and verifying the error message about the password length requirement.
Outcome: Failed
Execution Time: 3223 ms
Expected Output: Password must be at least 6 characters long
Actual Output: Failed to **testWeakNewPassword**: No views in hierarchy found matching: an instance of android widget TextView and view.getText() with or without transformation to match: a string containing "Password must be at least"

Fig. 13: Part of the report text file for activity_main

6. The MACT Framework Evaluation

To evaluate the proposed MACT framework, we focused on five critical research questions:

- **RQ1:** Can the framework accurately identify UI elements, and how does it achieve this?
- **RQ2:** Can machine learning models accurately classify Android activities using UI features?
- **RQ3:** Do auto-generated test cases reliably detect bugs in real-world applications?
- **RQ4:** How much time and effort does the framework save compared to traditional methods?

RQ1: Accurate Identification of UI Elements

The motivation behind **RQ1** is to determine whether textual hints (e.g., labels, resource IDs) and layout attributes (e.g., clickability, bounds) are sufficient to robustly identify UI elements across diverse applications and contexts. This is critical for ensuring that test scripts remain adaptable to varying app implementations and UI designs. To answer this question, the framework was applied to 10 different applications [29], Successfully identifying and extracting elements from 93 different UI screens with an accuracy exceeding 90%. This high success rate is attributed to the framework's ability to work directly with XML layout files, which provide precise and detailed UI metadata. Unlike approaches that rely on post-publication APKs, working with source code ensures accurate parsing of UI elements, avoiding errors caused by dynamic content in compiled applications. The framework achieves high accuracy by utilizing three key factors: **Textual Hints** (e.g., text content, resource IDs, and descriptions) to locate elements relevant to specific interactions, **Class Types** (e.g., buttons, spinners) to verify widget compatibility, and **Attribute Values** (e.g., isChecked, clickable, isDisplayed) to confirm element state and readiness. By combining these factors and processing the raw XML files, the framework effectively handles dynamic changes in UI structure, providing reliable and consistent identification of elements across different applications.

RQ2: Classification of Android Activities

The UI classification module was rigorously tested using the MASC dataset that includes a collection of 7,065 screens across 10 activity types. We have chosen Gradient Boosting as the most suitable algorithm for mobile app screen classification among the algorithms we have tested, as shown in Table 3 and Figure 14. It has the highest scores in accuracy.

A comparison with other research projects demonstrates that our classifier outperforms existing baselines in this domain. For instance, previous studies, such as Rosenfeld et al.[20] achieved 86.25% accuracy using 80 training samples, while our framework, with a larger dataset, achieved 93.48% accuracy. Additionally, we compare our results with the work of Hu et al.[21], who classified Android activities by extracting text from screenshots using Optical Character Recognition (OCR). Their best accuracy of 87.3% is achieved for a binary classification task, which is less complex compared to our multi-category classification. Furthermore, Ardito et al. [22] classified 100 samples into 8 categories, achieving a top accuracy of 91%. Although their results are noteworthy, our classifier excels in handling more complex and diverse activity categories. These comparisons underscore the robustness and scalability of our approach, as it not only achieves higher accuracy but also operates effectively with a significantly larger dataset.

Table 3: Comparison of ML algorithms for activity classification and other research studies.

	Accuracy	Dataset size
Gradient Boosting	93.48 %	7,065 UI is classified into ten categories.
XGBoost	93.20 %	
SVM Linear Kernel	93.20 %	
Ardito et al. [22]	91.01 %	100 UI is classified into eight categories.
Rosenfeld et al. [20]	86.25 %	80 UI is classified into seven categories.
AppFlow [21]	87.30 %	1785 UI is classified into two categories.

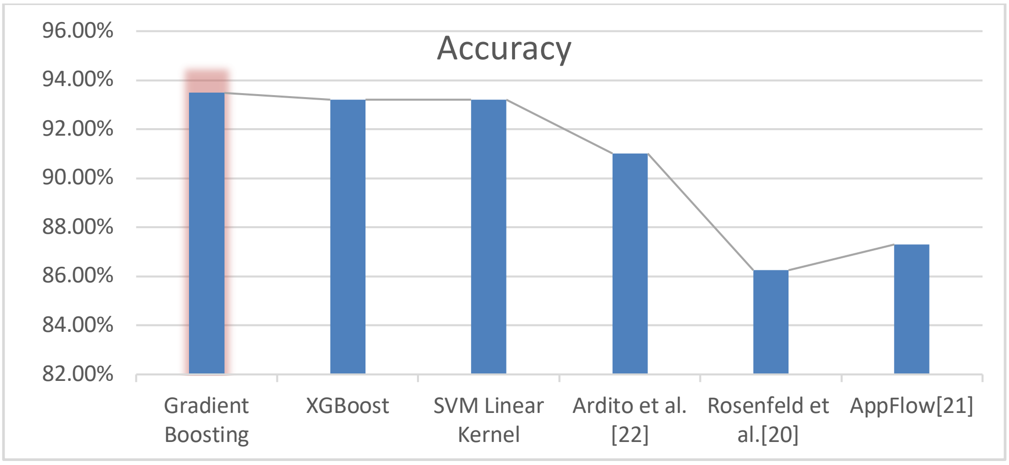


Fig. 14: Comparison of ML algorithms for activity classification, including results from other research studies.

RQ3: Effectiveness of Auto-Generated Test Cases

To address RQ3, we have conducted some experiments to evaluate the effectiveness of auto-generated test cases in detecting bugs in real-world applications. In these experiments, we have used four open-source Android applications [29]. Since applications typically undergo extensive testing before being released on app stores, we have introduced a variety of realistic logical bugs into these applications by modifying their source code, in addition to any pre-existing issues.

We have defined 14 test scenarios to test the core functionalities of different activity categories. To simulate diverse testing conditions, some scenarios were deliberately set to fail by inserting incorrect explicit assertions, in addition to fault injection into the applications. Table 4 provides detailed information about the test scenarios, including the applications tested, descriptions of test cases, expected and actual outcomes. The comparison of expected and actual outcomes allows us to assess the effectiveness of the auto-generated test cases in detecting bugs.

Table 4: Quality Evaluation of the 14 test cases of the selected four applications

Activity Name	App name	Script	Test Description	Expected Outcome	Actual Outcome
Setting	**MyPortal App**	Script1	write large font size	Fail	Fail
		Script2	Switch to Dark Mode	Pass	Pass
		Script3	Click on Logout to return to the Main Activity	Fail	Fail
		Script4	Set the Notification Switch state to ON	Pass	Pass
Login	**Smart App**	Script5	Login without username and password	Pass	Pass
		Script6	Login with wrong username and password	Pass	Pass
		Script7	Login with valid username and password	Pass	Pass
		Script8	Login with valid username and week password	Fail	Fail
Profile	**Profile Tracking App**	Script9	Change the password with valid and matching inputs	Pass	Pass
		Script10	Change the password with mismatched new and confirm passwords	Pass	Pass
		Script11	Setting a new password that matches the old password	Fail	Fail
		Script12	Change the password using an incorrect old password	Pass	Pass
Map	**Maps App**	Script13	Search for a valid city, and then enter	Pass	Pass
		Script14	Search for an invalid city, and then enter	Pass	Fail

The scripts successfully identified failures when incorrect assertions were added or when bugs were injected into the tested applications. For example, scripts 1 and 11 failed as expected due to bugs being injected into the respective applications (MyPortal and Profile Tracking). Script 1 failed because it could not apply the typed font size, while Script 11 failed due to a mismatch between the old and new passwords. However, one script (Script 14 for Maps App) produced an unexpected result caused by the application's unusual behavior when entering a city name containing symbols like "@."

In order to highlight the limitations of existing tools and to demonstrate how our approach can overcome these limitations, we have compared its performance with the Android Application Monkey tool. This tool was selected as a representative of current random-testing tools due to its widespread use and its integration into the Android developer toolkit. Table 5 summarizes the differences in identifying logical bugs, execution efficiency, and overall reliability.

Table 5: A comparison between bugs detection ability of MACT and Android Application Monkey

Test Scenario	Activity Type	Tool	No. of Bugs Planted	No. of Real-Time Crashes Detected	Logical Bugs Detected	Detection Rate (%)
Original	Settings Activity	MACT	0	0	0	N/A
		Monkey	0	0	0	N/A
Faulted		MACT	3	1– (null cannot be cast to non-null type android.widget.Switch)	2 Bugs: • Select font type from the list -failed • Change the font size -failed	100%
		Monkey	3	1 crash	0	33%
Original	Login Activity	MACT	0	0	0	N/A
		Monkey	0	0	0	N/A
Faulted		MACT	4	0	4 bugs have been found: • Login without username and password - Failed • Login with wrong username and password - Failed • Login with valid username and password – Failed • Login with valid username and week password – Failed	100%
		Monkey	4	0	0	0%
Original	Profile Activity	MACT	0	0	0	N/A
		Monkey	0	0	0	N/A
Faulted		MACT	3	0	3 bugs have been found: • Change the password with mismatched	100%

Test Scenario	Activity Type	Tool	No. of Bugs Planted	No. of Real-Time Crashes Detected	Logical Bugs Detected	Detection Rate (%)
					new and confirm passwords - Failed • Setting a new password that matches the old password – Failed • Change the password using an incorrect old password – Failed	
		Monkey	3	0	0	0%

The results of the experiment highlight the performance of the MACT framework compared to the Android Application Monkey tool in detecting bugs. As shown in Table 5, the MACT framework detected all injected bugs with 100% accuracy, outperforming the Monkey tool, which failed to identify any logical bugs.

RQ4: Time and Effort Savings

To answer **RQ4**, we have evaluated the ability of MACT to save time and effort compared to traditional manual testing methods. This evaluation showed that, in MACT, 11 test scripts, totaling 159 lines of code, were written only and executed across 8 different applications without modification, taking only 2.5 hours for both script writing and execution. In contrast, the manual method using tools like Espresso would require 159 lines of code per app, leading to a total of 1,272 lines of code for the 8 applications. Additionally, manual testing would take approximately 3 hours per app, totaling 24 hours for the 8 applications. This results in 87% reduction in code size and 89% saving on time with MACT. Moreover, the MACT framework's ability to reuse scripts across different applications eliminates the need to rewrite test scripts for each app, further improving efficiency. For instance, a Login script (15 lines) can be executed on 8 apps with a "Login" screen without rewriting, whereas the manual method would require 120 lines for the same task. This demonstrates that MACT is eight times more efficient than traditional methods, significantly reducing both the development and maintenance efforts involved in mobile app testing. Table 6 shows the results of this experiment.

Table 6: Efficiency comparison between MACT and manual testing

Criterion	Proposed Framework (MACT)	Manual Approach (Espresso)	Relative Savings
Total Lines of Code (LOC)	159	1272	87%
Total Time (Hours)	2.5	24	89%
Maintenance	Modify one script	Modify eight scripts	87%

7. Discussion

The results, as detailed in Section 6, reveal the superiority of the MACT framework in bugs detection. While the Android Application Monkey failed to detect any logical bugs, MACT successfully identified all the injected bugs in the applications' source code. This success can be attributed to our activity classification approach, which empowers the framework to generate activity-based tests tailored to the expected behavior of each activity. These tests were executed in the appropriate context, enabled by the proposed ML model.

Our experiment highlights a key principle: treating each activity as an independent entity, rather than the application, enables a more granular and targeted testing process. This approach transforms the test suite into a collection of scenario-based tests tailored to individual activities, improving both efficiency and accuracy.

It is worth noting that our work is not the first to explore the use of ML in testing mobile applications. As shown in Table 3, previous studies such as Ardito et al. [22], Rosenfeld et al. [20], and AppFlow [21] have also investigated activity classification and testing. However, our approach outperforms these existing methods in terms of accuracy, particularly when handling a larger and more diverse dataset. For instance, while Ardito et al. achieved an accuracy of 91.01% with 100 UI samples classified into eight categories, our framework achieved an accuracy of 93.48% with 7,065 UI samples classified into ten categories. Similarly, Rosenfeld et al. and AppFlow achieved lower accuracy rates of 86.25% and 87.30%, respectively, with smaller datasets and fewer activity categories. These results demonstrate the superiority of our approach while highlighting its robustness, scalability, and suitability for diverse and dynamic UI applications.

In addition to comparing our framework with prior research, we have evaluated its efficiency against widely used traditional testing tools such as Espresso. As shown in Table 4, our framework significantly reduces the time and effort required for testing compared to manual approaches using Espresso. The framework significantly reduces development and maintenance efforts by reusing scripts across multiple applications. For instance, a 15-line Login script can be executed on 8 different applications without modification, compared to 120 lines required manually. This results in an 87% reduction in code and 89% time saving, making the framework eight times more efficient than traditional methods.

8. Conclusion and Future Work

This paper introduced a ML-based framework, called MACT (Mobile Application Classification and Testing), that automates mobile application testing by classifying Android application activities and executing predefined, reusable test cases. MACT framework addresses major challenges in mobile app testing, such as GUI element identification, cross-device compatibility, and the resource-intensive nature of traditional testing methods. By leveraging the MASC dataset, developed in our previous work, which contains 7,065 screens from over 3,400 applications, MACT utilizes a Gradient Boosting classifier achieving 93.48% accuracy in activity classification. Empirical evaluations demonstrated that MACT detects all injected bugs with 100% accuracy, significantly reducing code size by 87% and testing time by 89% compared to traditional tools like Android Espresso and Android Application Monkey. These results highlight the MACT's efficiency, scalability, and robustness in automated testing.

MACT's high-level approach to testing, which abstracts the testing process from specific implementation details, reduces maintenance overhead and facilitates the creation of generalizable, low-weight test scripts. By associating predefined test cases with activity types rather than individual widgets, MACT enables reusability of test cases across multiple applications with minimal adjustments.

Despite its contributions, MACT has several limitations. It currently supports only predefined activity types and lacks the ability to handle highly dynamic or unpredictable application behaviors. The 10 activity types identified in this work were selected as a proof of concept, and the full implementation of the framework will include a broader range of activity types and tests. Additionally, the reliance on textual attributes and predefined activity layouts limits its adaptability to more complex UI scenarios or applications with custom widgets.

Additionally, it is important to consider the scope of our experiment, which involved testing 12 open-source Android applications with pre- injected bugs. While this setup provided a controlled environment for evaluation, real-world applications may present more complex and unpredictable scenarios. Future work will focus on expanding the framework's capabilities to handle a wider variety of activity types and testing conditions, as well as evaluating its performance on a larger and more diverse set of applications.

In conclusion, our framework demonstrates significant potential in improving the efficiency and accuracy of mobile app testing by leveraging machine learning for activity classification and automated test generation. While there are limitations to address, the results highlight the advantages of treating activities as independent entities and the value of integrating machine learning into the testing process. The comparisons with prior studies and traditional tools like Espresso show the significant contribution that our approach makes to the field of mobile application testing.

Building on these findings, we aim to enhance MACT through several key research directions:

- **Expanding Activity Classification**: Expand MACT to classify more specialized activity types, such as dashboards or to-do list, by incorporating more diverse datasets and advanced feature extraction techniques.
- **Integration of Visual Features**: Utilizing CNN-based image analysis to improve classification accuracy for visually similar or highly dynamic screens, enhancing the robustness of the classifier.
- **Cross-Platform Support:** We will expand MACT to include iOS applications, enabling comprehensive testing across multiple platforms.
- **Dynamic Content and State Transitions**: Improving MACT's ability to handle dynamic content, state transitions, and interactive components, ensuring broader applicability across real-world applications.

Empirical Evaluation and Dataset Expansion: Conducting large-scale empirical studies on a diverse set of applications to validate the framework's scalability, fault detection capabilities, and GUI coverage. Additionally, expanding the MASC dataset to include more labeled examples and varied activity types will strengthen MACT's adaptability and accuracy.

- Finally, we envision integrating conversational AI to assist testers by identifying flaws in test procedures, recommending corrections, and facilitating a collaborative testing environment.

Acknowledgments

Authors sincerely acknowledge the Computer Science Department in Faculty of Science, Minia University for the facilities and support.

Authors' Contributions

This study was conducted through the collaborative efforts of all authors. Author A.A designed the study, conducted the statistical analysis, and developed the research protocol. Authors A.Z, E.E, M.A, and M.G contributed data analysis, managed literature reviews, and drafted the initial version of the manuscript. All authors reviewed and approved the final manuscript.

Disclosure of potential Conflict of Interest:

The authors declare no conflicts of interest.

Ethical Statement:

The study utilized publicly available datasets and did not involve direct human participation or experimentation.

Data availability:

The data that support the findings of this study are publicly available. The MASC dataset can be accessed at https://doi.org/10.5281/zenodo.14783065 , and the complete source code is available on https://github.com/Ali-Aahmed/MASC-Dataset.

References

[1] Statista. *Number of apps available in leading app stores as of August 2024*. 2024; Available from: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[2] Ahmad A., Li K., Feng C., Asim S.M., Yousif A., and Ge S., *An Empirical Study of Investigating Mobile Applications Development Challenges.* IEEE Access, 2018. **6**: p. 17711-17728.

[3] Choudhary S.R., Gorla A., and Orso A. *Automated Test Input Generation for Android: Are We There Yet? (E)*. in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015.

[4] Tramontana P., Amalfitano D., Amatucci N., and Fasolino A.R., *Automated functional testing of mobile applications: a systematic mapping study.* Software Quality Journal, 2019. **27**(1): p. 149-201.

[5] Hanna M., El-Haggar N., and Mostafa M.-S., *A Review of Scripting Techniques Used in Automated Software Testing.* International Journal of Advanced Computer Science and Applications, 2014. **5**.

[6] Berihun N., Dongmo C., and Poll J., *The Applicability of Automated Testing Frameworks for Mobile Application Testing: A Systematic Literature Review.* Computers, 2023. **12**: p. 97.

[7] Mustofa K. and Fajar S., *Selenium-Based Multithreading Functional Testing.* IJCCS (Indonesian Journal of Computing and Cybernetics Systems), 2018. **12**: p. 63.

[8] Kong P., Li L., Gao J., Liu K., Bissyandé T.F., and Klein J., *Automated Testing of Android Apps: A Systematic Literature Review.* IEEE Transactions on Reliability, 2019. **68**(1): p. 45-66.

[9] Li L., Bissyand T.F., Papadakis M., Rasthofer S., Bartel A., Octeau D., Klein J., and Traon L., *Static analysis of android apps.* 2017. **88**(C %J Inf. Softw. Technol.): p. 67–95.

[10] Linares-Vásquez M., Moran K., and Poshyvanyk D., *Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing*. 2017. 399-410.

[11] *Ui Application Exerciser Monkey*. 2025 [cited 2025; Available from: https://developer.android.com/studio/test/other-testing-tools/monkey.

[12] Machiry A., Tahiliani R., and Naik M., *Dynodroid: An input generation system for Android apps*. 2013. 224-234.

[13] Moran K., Linares-Vásquez M., Bernal-Cardenas C., Vendome C., and Poshyvanyk D., *CrashScope: A Practical Tool for Automated Testing of Android Applications*. 2018.

[14] Mahmood R., Mirzaei N., and Malek S., *Evodroid: Segmented evolutionary testing of Android apps.* Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014: p. 599-609.

[15] Amalfitano D., Fasolino A., Tramontana P., Ta B., and Memon A., *MobiGUITAR -- A Tool for Automated Model-Based Testing of Mobile Apps.* IEEE Software, 2014. **32**: p. 1-1.

[16] Memon A., Banerjee I., Nguyen B., and Robbins B., *The first decade of GUI ripping: Extensions, applications, and broader impacts*. 2013. 11-20.

[17] 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, 2012. **2195**: p. 769-773.

[18] Dong F., Guo Y., Li C., Xu G., Wei F.J.t.I.C.o.C.C., and Systems I., *ClassifyDroid: Large scale Android applications classification using semi-supervised Multinomial Naive Bayes.* 2016: p. 77-81.

[19] Reyhani Hamedani M., Shin D., Lee M., Cho S.-J., and Hwang C., *AndroClass: An Effective Method to ClassifyAndroid Applications by Applying Deep Neural Networks to Comprehensive Features.* 2018. **2018**(1): p. 1250359.

[20] Rosenfeld A., Kardashov O., and Zang O., *Automation of Android applications functional testing using machine learning activities classification*, in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 2018, Association for Computing Machinery: Gothenburg, Sweden. p. 122–132.

[21] 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*. 2018, Association for Computing Machinery: Lake Buena Vista, FL, USA. p. 269–282.

[22] Ardito L., Coppola R., Leonardi S., Morisio M., and Buy U., *Automated Test Selection for Android Apps Based on APK and Activity Classification.* IEEE Access, 2020. **8**: p. 187648-187670.

[23] Google. *Espresso*. 2025 [cited 2025 10-10-2024]; an open-source framework created by Google for Android]. Available from: https://developer. android.com/training/testing/espresso.

[24] Ahmed A., Zaki A., Elgeldawi E., and et al., *MASC: A Dataset for the Development and Classification of Mobile Applications Screens*. 2024.

[25] Deka B., Huang Z., Franzen C., Hibschman J., Afergan D., Li Y., Nichols J., and Kumar R., *Rico: A Mobile App Dataset for Building Data-Driven Design Applications*, in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 2017, Association for Computing Machinery: Québec City, QC, Canada. p. 845–854.

[26] Wang B., Li G., Zhou X., Chen Z., Grossman T., and Li Y., *Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning*. 2021. 498-510.

[27] Leiva L.A., Hota A., and Oulasvirta A., *Enrico: A Dataset for Topic Modeling of Mobile UI Designs*, in *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*. 2021, Association for Computing Machinery: Oldenburg, Germany. p. Article 9.

[28] Google. *Latest Version kotlin*. 22 August 2024 [cited 2025; Available from: https://kotlinlang.org/.

[29] Moheb G. and Ali A. *MACT Framework Classification Evaluation*. 2025 [cited 2025; Available from: https://github.com/Ali-Aahmed/MACT-framework/tree/master/Classification%20Evaluation

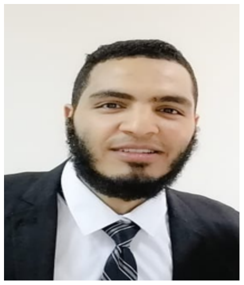
Biography:

Moheb R. Girgis received his B.Sc. Degree from Mansoura University, Egypt, in 1974, M.Sc. degree from Assiut University, Egypt, in 1980, and Ph.D. degree from the University of Liverpool, England, in 1986. He is a professor of computer science at Minia University, Egypt. His research interests include software engineering, software testing, information retrieval, evolutionary algorithms, image processing, computer networks, and bioinformatics.

Alaa M. Zaki received his B.Sc. Degree from Minia University, Egypt, in 1999, M.Sc. degree from Mina University, Egypt, in 2009, and Ph.D. degree from the University of Minia, Egypt, in 2015. He is a professor of computer science at Minia University, Egypt. His research interests include Software Engineering, Data Mining, evolutionary algorithms, and computer networks.

Enas Elgeldawi received here B.Sc. Degree from Minia University, Egypt, in 1999, M.Sc. degree from Mina University, Egypt, in 2005, and Ph.D. degree from the University of Minia, Egypt, in 2015. she is a professor of computer science at Minia University, Egypt. Here research interests include Data Mining, evolutionary algorithms, computer networks, and bioinformatics.

Mohamed M. Abdallah received his B.Sc. Degree from Minia University, Egypt, in 2003, M.Sc. degree from Mina University, Egypt, in 2008, and Ph.D. degree from the University of Minia, Egypt, in 2015. He is a professor of computer science at Minia University, Egypt. His research interests include information retrieval, Data Mining, evolutionary algorithms.



Ali A. Ahmed received his B.Sc. Degree from Minia University, Egypt, in 2015, M.Sc. degree from Minia University, Egypt, in 2021. He is an Assistant lecturer of computer science at Minia University, Egypt. His research interests include Software Engineering, machine learning, deep learning, big data analytics, and mobile applications development testing.