An Approach to GUI Test Scenario Generation Using Machine Learning

Conference Paper · June 2022

CITATIONS
0

READS
548

8 authors, including:

Jerry Gao
San Jose State University
334 PUBLICATIONS **5,498** CITATIONS
SEE PROFILE

Chuanqi Tao
Nanjing University of Aeronautics & Astronautics
78 PUBLICATIONS **519** CITATIONS
SEE PROFILE

Amrutha Pavani Anumalasetty
San Jose State University
2 PUBLICATIONS **1** CITATION
SEE PROFILE

Akshata Hatwar
San Jose State University
1 PUBLICATION **0** CITATIONS
SEE PROFILE

An Approach to GUI Test Scenario Generation Using Machine Learning

Jerry Gao①, Chuanqi Tao②③, Yejun He②③, Amrutha Pavani Anumalasetty①, Erica Wilson Joseph①,
Akshata Hatwar Kumbashi Sripathi①, Himabindu Nayani①
① San Jose State University, San Jose, USA
② College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, P.R. China
③ Ministry Key Laboratory for Safety-Critical Software Development
and Verification, Nanjing University of Aeronautics and Astronautics, Nanjing, China
Correspondence to: taochuanqi@nuaa.edu.cn

***Abstract*—With the fast advance of artificial intelligence technology and data-driven machine learning techniques, more and more AI approaches are applied in software engineering activities, such as coding, testing and etc.. Conventionally, test engineers use manual testing tools to test mobile apps and deliver products. Object detection technology like YOLO is widely used in image processing these days. Inspired from this, on the basis of detecting GUI elements using machine learning models, we propose an automated approach to GUI test scenario generation based on mockup diagrams. The list of possible scenarios can be visualized using NetworkX which can indicate the feasibility and effectiveness of the proposed approach.**

Keywords—test scenario generation, GUI test, mobile app test automation, machine learning, deep learning, mockup diagram.

I. Introduction

Artificial intelligence and machine learning are remodeling multiple sectors of the economy and reshaping various aspects of our everyday lives. The testing powered by AI allows mobile app developers to check an app for defects and loopholes by carrying out effective test cycles without missing any test scenarios that otherwise would be missed by human error [1][2]. The conventional manual testing approach fails to suffice good test scenarios for complicated application's comprehensive performance testing. While in contrast, automated testing by AI can capture the complex test scenarios for mobile applications, especially intelligent mobile apps. The AI-powered machine learning models will self-learn to create every possible scenario and test scenarios intelligently faster than a human tester. Unlike manual testing, automated testing has already helped several professional testing engineers save time to redirect the effort on different tasks. Integrating AI with automated testing will make the job furthermore time and cost-effective. AI has simplified the conventional testing procedure by providing improved test coverage, more reliable control and transparency of test activities, stable defect detection, beneficial reuse of test scenarios, and reduced test time and costs [3].

The paper aims to utilize deep learning to overcome testing intelligent apps' limitations by building a data-driven machine learning solution. Our solution will be comprehensive and incorporate various functional and non-functional requirements. We have identified some applicable requirements that would directly affect our data-driven test generation AI model.

Overall, the paper has two contributions:

1. Using machine learning to detect all the GUI elements for intelligent mobile apps like input/output buttons, camera buttons, back buttons. Based on three deep learning models, this paper then automatically classifies these elements as input elements, output elements, AI input, or AI output.

2. Aside from GUI element detection, unlike other papers [4][5], the paper proposes an automated approach to GUI test scenario generation using mockup diagrams.

The paper is organized as follows. Section 2 reviews the related work about AI tools for software testing. Section 3 discusses approaches on GUI element detection and relationship classification, as well as the way to generate GUI test scenario generation. Section 4 evaluates the models and displays process of generating test scenarios. At last, conclusion remarks are presented in Section 5.

II. Related Work

Recently Artificial intelligence has emerged as cutting-edge technology, especially while building and testing mobile apps. To understand the evolution of the AI technology applications, especially on the mobile apps – on android and IOS platforms and to help us achieve our goal of building a machine learning model that would generate test scenarios based on mockup diagrams as input. We categorized our survey of the related AI testing work done in the past and compared various related papers in the same field. The related work about GUI element detection is as follows.

We came across a study conducted combining the traditional UI element detection like methods from the computer vision (CV) domain, including old fashioned ones that rely on traditional image processing features like canny edge, contours, and deep learning models that learn to detect from large-scale GUI data. Thus, the paper conducts a test on over 50k GUI images to understand these methods' capabilities, limitations, and influential designs [6]. Furthermore, with the increasing number of apps developed and available, it becomes very cumbersome for developers to implement conceptual diagrams [7]. The research paper explains working of Reverse Engineer Mobile Application user interface (REMAUI), which detects the user interface elements like images, texts, containers, buttons using Optical Character Recognition. The papers use 488 screen shots from both android and IOS applications where the REMAUI' s for detecting UI was compared with the default desktop computers interface [8] shows GUI code generation framework – component identification, mapping, and GUI

code generation. It utilizes image processing and deep learning techniques and can be used for large scale platforms, this framework does not require any other inputs, which makes it possible for large-scale and platform-independent code generation.

Research showed implementation of an android testing tool based on the neural network. This [9] main focus was to use deep neural network models that can be trained to understand human actions based on an app's GUI from human interaction traces. The idea was that the trained model could then guide test input generation to achieve higher coverage. This application has been made public by the papers' authors and adds value to the AI testing community. Thus, it will not be over-the-top to mention that AI has found its new application in the testing world and vice-versa. Testing also needs AI for a practical and automated experience. As the test scenarios are increasing in volume, chances of overseeing a test scenario are highly likely [10] used this drawback of the testing cycle and proposed a mechanism to train a model based on the history of the user actions for which test scenarios fell in the cracks and are not being accounted for by human testers. Thus, learning through the previous mistakes, the authors of the papers' experiment use a neural network for pruning a test scenario set while preserving its effectiveness.

Unlike other research papers [11][12], this paper proposes a model that can not only detect GUI elements of intelligent mobile apps , but also propose an approach to generate GUI test scenarios for various apps based on machine learning.

TABLE I. A COMPARISON OF RESEARCH PAPERS ON GUI ELEMENTS DETECTION AND PROTOTYPING

Purpose	Model Used	Dataset	Input	Accuracy	Output
Components Recognition	CNN	RICO	Screenshots	96.97%	Located UI component
GUI Code Generation	CNN	Android & IOS apps	UI pages	85%	Large-scale GUI code
GUI Prototyping	KNN, CNN	Screens, GUI elements	Mockup artifact	90%	Prototype application
GUI Object Detection	RCNN, CenterNet	COCO	Screenshots	F1: UI 0.449	Detected UI elements
GUI Test Scenario Generation	YOLOv5,Detectron2, G-Net, EfficientDet-	Single screenshot, UI elements, Mockup	Mockup diagrams, Single screenshots	75%	GUI test scenarios

III. METHODOLOGY

As mentioned earlier, the goal is to reduce the manual intervention to generate test scenarios by AI solutions. As shown in the Figure 1, the diagram demonstrates overall method for scenario generation (Fig. 1).

A. GUI Element Detection

YOLO is an innovative CNN [13] that does real-time object detection. So far, five different versions of the YOLO algorithm are in use with some pros and cons. However, YOLOv3, v4, and v5 are still the most widely used Real-time object detection algorithms globally. It is a branch of computer vision that's exploding and doing even better than it did a few years ago [14].

EfficientDet model addresses many of the challenges present in typical object detection tasks done using Deep Learning and is trained and evaluated on the COCO data set. This data set consists of about 170 image classes, and almost 100,000 images are annotated [15].

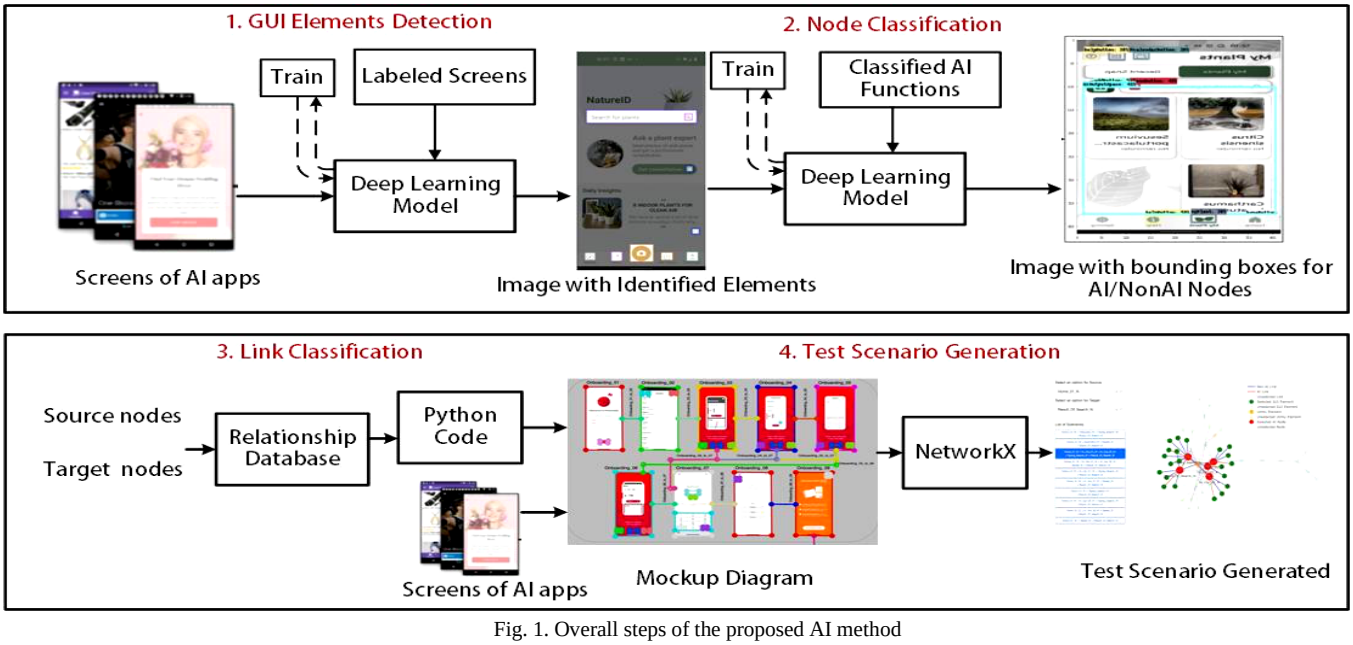


Fig. 1. Overall steps of the proposed AI method

B. Relationship Classification

1) Node Classification

Aside from YOLOv5 and EfficientDet, Detectron2 [16] contains a lot of pre-trained models. This is a group of models that have been pre-trained on the COCO data set. For our model, we will utilize the three models. Detectron2's pre-trained weights of Base (Faster) R-CNN with Feature Pyramid Network (Base-RCNN-FPN) for node classification, which is a primary bounding box detector extendable to Mask R-CNN. The faster R-CNN detector with an FPN backbone is a multi-scale detector that achieves great accuracy for detecting small to large objects, making it the practical standard. This cuts training time and improves performance considerably.

2) Link Classification

Graphs are advanced and robust data structures that represent a set of objects and their relationships. These objects represent the nodes, and the edges are represented by the relationships [17].

We assume a graph, G(Fig. 2). Then, G = (V,E). This graph describes V as the vertex set/nodes and E as the edges.

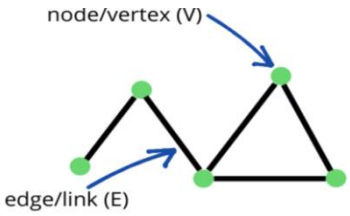


Fig. 2. Graph and graph components

C. Test Scenario Generation

1) Depth First Search Algorithm

In DFS, we start from a particular node or vertex and explore as feasible along each branch before backtracking. We additionally keep tracking the visited vertices. To enable reversal, we utilize a stack data structure while implementing DFS [18].

2) NetworkX Python Package

NetworkX is a package for the Python programming language that's mainly used to create, manipulate, and study the structure, dynamics, and functions of complex graph networks [19]. NetworkX was the optimum choice for us to understand the flow of a particular AI app and come up with possible screen-by-screen scenarios.

IV. Empirical Study

Here, we use Confusion Matrix to evaluate the ability of the three methods for GUI elements detection and relationship classification.

A. Metric

- Precision: In our case, we want to see how often the GUI elements and the relationships are correctly predicted. It is calculated as:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

- Recall: The recall is the measure of how accurately we detect all the objects/items in the data. When it predicts the GUI element or the relation, how often it is predicting the correct GUI element or relationship. It is calculated as:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

TensorBoard: TensorBoard [20] is a visualization toolkit from TensorFlow that allows you to show various metrics, parameters, and other visualizations to help you debug, monitor, fine-tune, optimize, and share the results of your deep learning experiments.

B. Model Evaluation

1) GUI Element Detection and Relationship Classification

a) YOLOv5: We chose YOLOv5 again to compare with other models as annotations changed for this task with seven object detection and classification classes. In this case, the images were preprocessed and resized with 448*448*3, and augmentation techniques to obtain optimized classification results. The dataset was split into a 60% training set, 20% validation set, and 20% test set. As seen in Fig. 3, the train box loss decreased significantly by the end of the 150 epochs. The loss started with 0.10 was the first epoch and significantly decreased over the epochs. While the validation box loss also improved from 0.06 to 0.01 by the end of the iterations. Another important metric for evaluation is the classification loss, as it is essential to understand how well the model performs for relationship classification. The validation classification loss demonstrated significant loss from 0.040 to 0.010.

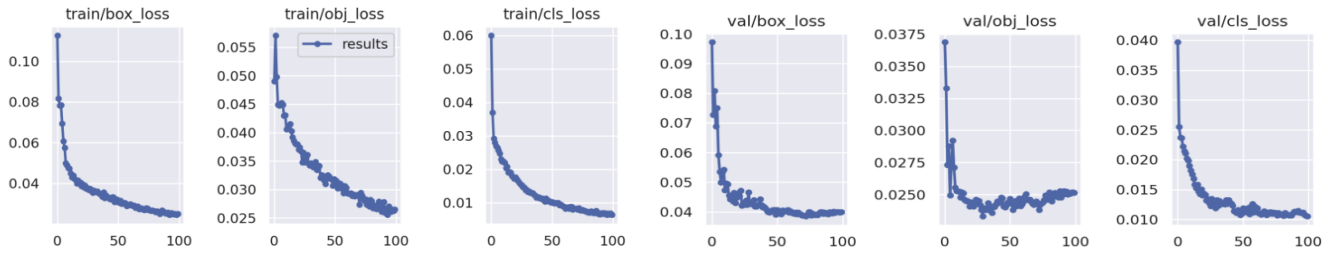


Fig. 3. Train and Validation loss graphs - AI/non-AI classification

b) EfficientDet-D0: For training EffecientDet-D0 on our custom data to perform node prediction and detect AI/Non-AI elements, the dataset was split into 60% training set, 20% validation set, and 20% test set fashion. We resized all the screenshots in the dataset to 416*416, and contrast was auto adjusted using Adaptive Equalization as a pre-processing step. We then augmented the images to get a larger training data size.

The resulting dataset consisted of 1.1k images in the training set, 123 in the validation set, and 123 in the testing set. We trained the model for 5000 steps, and the below graph shows various losses vs. steps trained. The four losses considered to evaluate the performance of EffecientDet-D0 are classification loss, localization loss, regularization loss, and total loss. Classification loss started reducing after the first checkpoint at epoch = 500, and the loss continued to decline until epoch = 4k. And the classification loss did not vary as much from epoch = 4k to epoch = 5k. In contrast, the regularization loss shows a minute increase throughout the checkpoints from epoch = 500 to epoch = 5k. Total loss stabilized from epoch = 4k and stayed pretty much constant until the end. (Fig. 4).

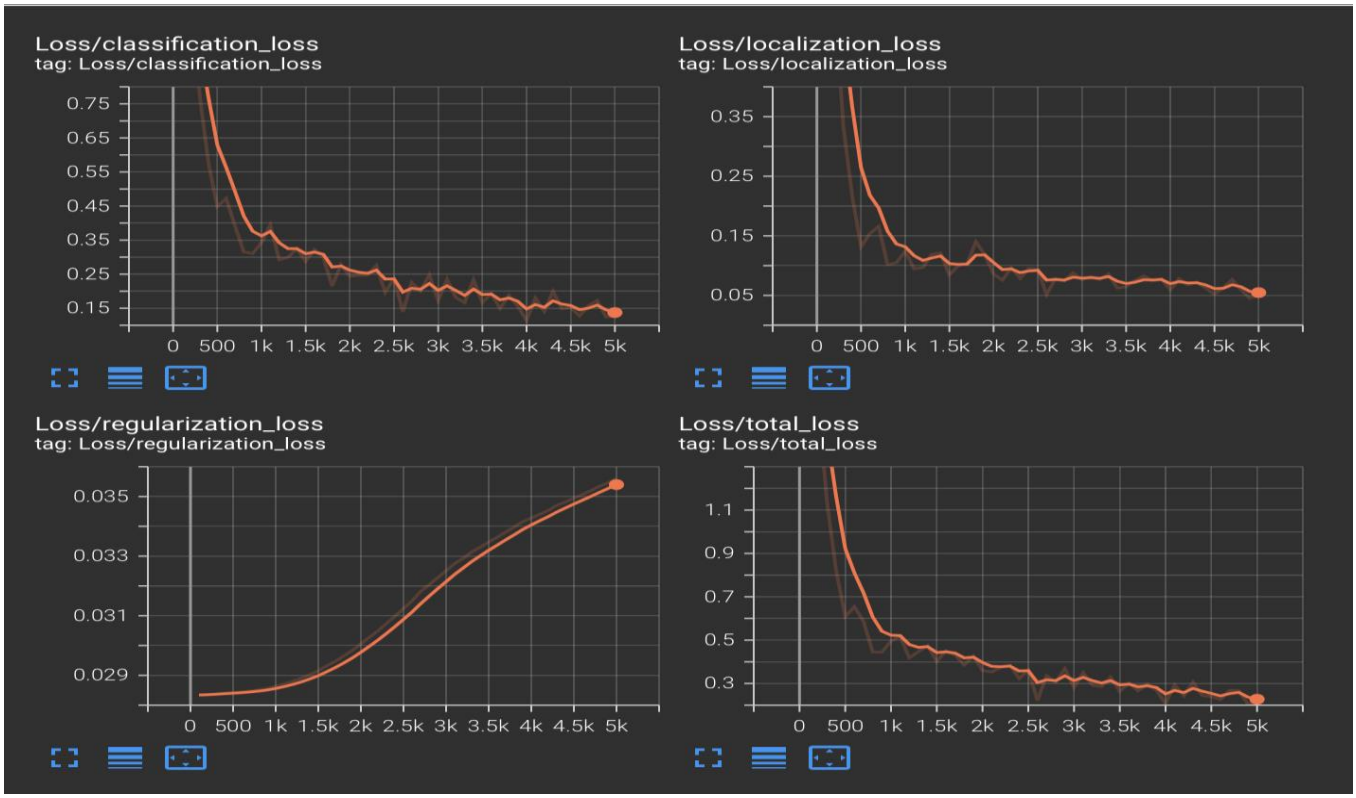


Fig. 4. Various Loss vs epochs for model EfficientDet-D0 trained for Node Classification

TABLE II. MODEL IMPROVEMENT COMPARISONS FOR DIFFERENT CLASSIFICATION TASK

Model Improvement Comparison							
Task	**Model**	**Previous Accuracy**			**Improved Accuracy**		
		Learning rate	**Classification loss**	**Total loss**	**Learning rate**	**Classification loss**	**Total loss**
GUI element detection	YOLOv5	NA	0.021	NA	NA	0.0112	NA
	EfficientDet-	0.0796	0.2509	0.4565	0.0796	0.2070	0.4009
Node Classification	YOLOv5	NA	0.01368	NA	NA	0.011	NA
	EfficientDet-	0.0799	0.1511	0.2505	0.0798	0.1092	0.1768
	Detectron2	0.000050	0.096	0.403	0.00005	0.087	0.340

c) Detectron2: To train Detectron2 model on our custom data, all the images were preprocessed, and augmentation techniques were added to help the model perform better in unprecedented situations. The dataset was split into 60% training set, 20% validation set, and 20% test set. The resulting dataset consisted of 1.1k images in the training set, 123 in the validation set, and 123 in the testing set. The model was trained for 5000 steps, and the below graphs show various losses vs. steps trained. The four losses considered to evaluate the performance of Detectron2 are classification loss, localization loss, regularization loss, learning rate, and total loss.

Classification loss started reducing after the first checkpoint at epoch = 123 and the loss continued to reduce until epoch = 5k. Furthermore, the classification

loss did not vary as much from epoch = 4k to epoch = 5k. In contrast, the regularization loss shows a minute decrease throughout the checkpoints from epoch = 1k to epoch = 5k. Total loss stabilized from epoch = 4k.

Apart from the loss rate, we explored the classification accuracy of the fast_rcnn and how the false negatives rates dropped from epoch = 500, and there was a minute decrease after epoch = 4k till epoch = 5k. The classification accuracy increased after epoch = 500 till epoch = 4.5k, however there was a consistency from epoch = 4.5k to epoch = 5k.

3) Overall Model Improvement

To fine-tune the YOLOv5, EfficientDet-D0, and Detectron2, we added 249 images to existing data for object detection to enhance accuracy compared to prior accuracy. We deployed each model with flask, and outputs are displayed with label and corresponding probability. From table 2, we can see that for GUI detection, the classification loss has decreased from 0.021 to 0.0112. For EfficientDet-D0, the classification loss decreased from 0.26 to 0.20, the learning rate decreased from 0.08 to 0.07, and the total loss also saw a minute decrease.

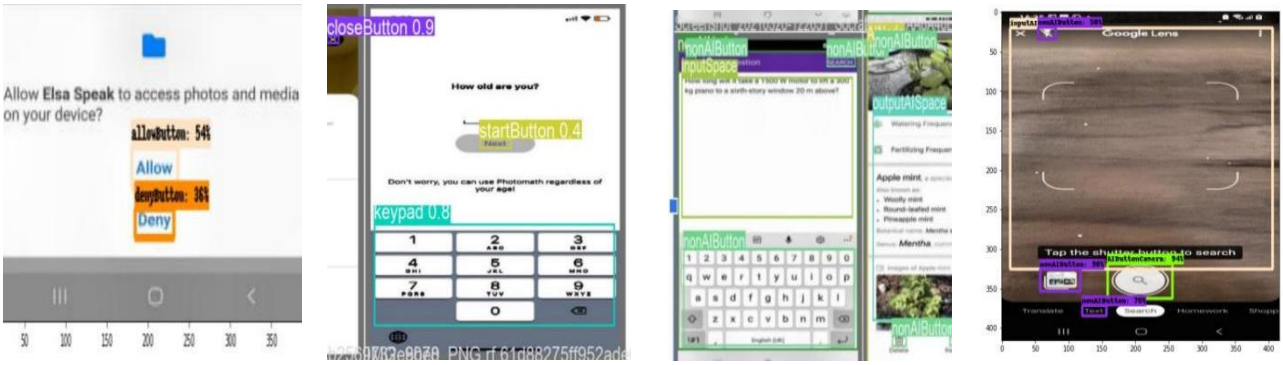
We fine-tuned three models for node classification. YOLOv5 showed a small decrease in classification loss. EfficientDet-D0 also showed improvement in both classification loss and total loss. Detectron2 has seen a significant change in classification loss from 0.096 to 0.087 and total loss from 0.403 to 0.340. So fine-tuning the model has changed detection accuracy.

C. Model Validation

1) GUI Detection

a) YOLOv5: Below is the visualization, Fig. 5 (a), of the YOLOv5 model's test results that were trained on our custom dataset. The model could reasonably detect that most of the GUI elements on the test data were not part of the training or validation. The accuracy of detecting all the GUI elements is a 72% precision score. Thus, to increase the accuracy, we plan to train the model with more epochs and feed in more data.

b) EfficientDet-D0: The below image shows the model validation of EffecientDet-D0 to detect GUI elements given a test image. Here the model detected the camera space as AIInputSpace with 30% accuracy, Home icon as HomeButton with 52% accuracy, and Help icon as HelpButton with 81%. This is the expected result, and the model performed well for the test image (Fig. 5 (b)). There are several other examples where the model successfully detected all the GUI elements with appropriate labels. However, there are cases where the model failed to detect some GUI elements in the test images and these cases are very rare.



(a) YOLOv5 model (b) EfficientDet-D0 model (c) YOLOv5 model (d) EfficientDet-D0 model

Fig. 5. Validation Results, (a)(b): Elements detection; (c)(d): Node Classification

2) Relationship Classification

a) YOLOv5: The model accuracy for relationship classification was 79%, thus as seen in the Fig. 5 (c), the model reasonably classified and detected elements on the given input, classifying the node as AI or non-AI node.

b) EfficientDet-D0: The Fig. 5 (d) shows the model validation of Efficient Det-D0 to perform Node Classification. Here the model detects if the node is AI or Non-AI, and also the model detects if all the GUI elements present in the given test image is an AI button or Non-AI button. In the first test image, the model detected the entire node as nonAINode with 20% accuracy, the Camera icon as AIButtonCamera with 47% accuracy, and several other Non-AI icons with respective accuracies. This is the expected result, and the model performed well for the test image.

c) Detectron2: Here the model detected the camera space as InputAISpace with 93% accuracy, camera icon as HomeButton with 99% accuracy, gallery button with 100%, and the entire node as AI node with 75% accuracy. This is the expected result, and the model performed well for the test images.

3) System Visualization

The first task was GUI element detection and classification, where we incorporated two models, namely YOLOv5 and Efficient-Det0, to detect and classify 128

labeled GUI elements. The accuracy of both the models for this task was ~67% after improving the model by adding training data to the original dataset. In this task, the model was able to detect most of the GUI elements on the given single screen of the AI application.

For the second task, classes were established to classify the nodes and the GUI elements relationship type - namely if the element was AI element or Non-AI element and if the screen, which we referred to as a node, was AI node or Non-AI node. We incorporated three models to perform the task where the validation loss of YOLOv5 was 0.011, Efficient-Det0 was 0.1511, and Detectron2 validation loss was 0.087. In this case, all the models that were used were able to classify the nodes (single screens) if they were “AI nodes” or “Non-AI nodes”. Also, we noticed successful classification and detection of the “Non-AI Button” or “AI function buttons” like “AI Camera button”, “AI Speak Button”, or “AI chat button” based on the presence of GUI elements on the screen.

Finally, to generate test scenarios of the mockup diagrams (Fig. 6), we subdivided the graph generation of test scenarios into three stages.

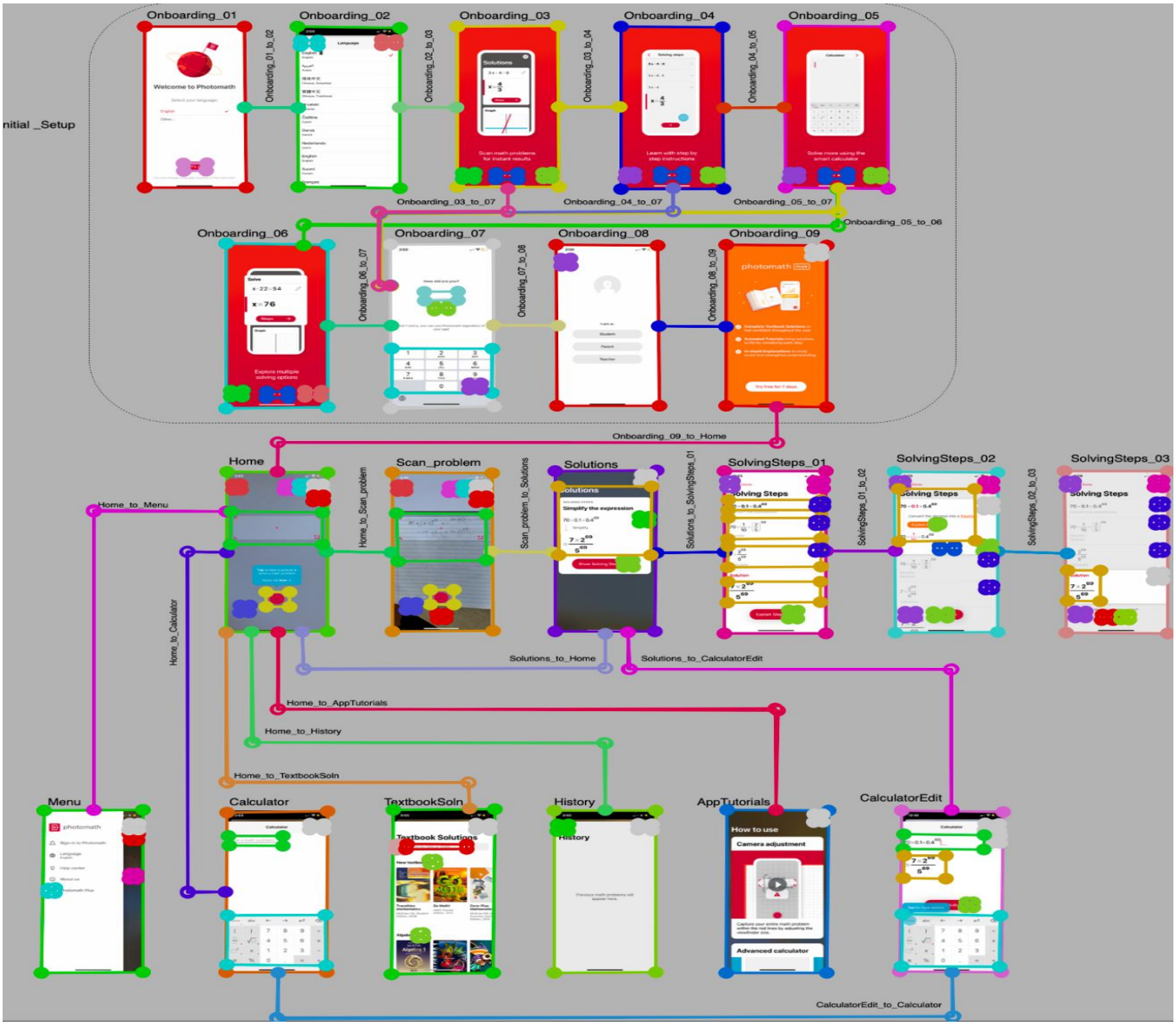


Fig. 6. Mockup diagram with annotations

At the first stage, the whole mockup is represented in the graph format for the end-users to understand the whole mockup visually. To improve the end user's experience, we extracted the GUI elements from GUI detection models and added an extra layer of intel that would help users understand the elements present within a screen.

Along with the first stage, we color-coded GUI elements in green and nodes (pages/screens) in blue based on the functionality of the nodes- if it is AI node then its color coded as red and if it's not AI node then its color coded as blue. Along with that we added a shared/utility button, buttons that have same functionality throughout the app to the network graph so that it can help the end users with regression testing as the functionality of these elements remain the same throughout the app (Fig. 7).

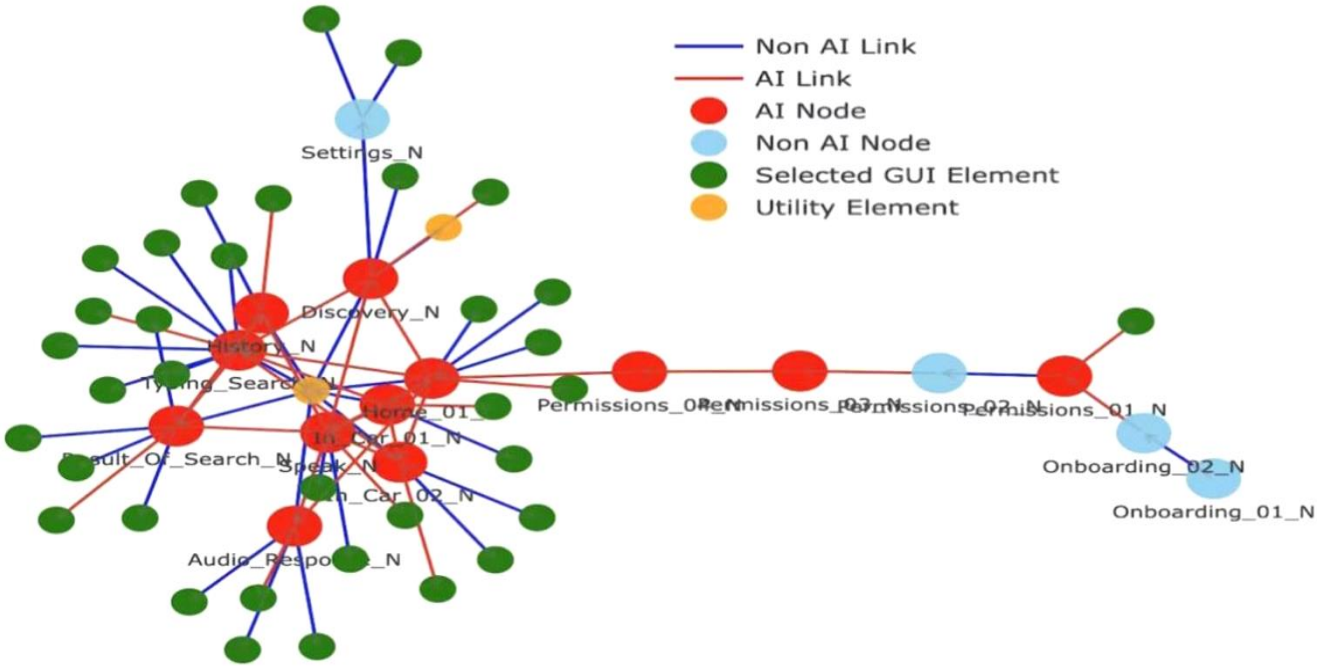


Fig. 7. Graphical representation of an AI app

The second stage was enhancing the user experience, where the user would have the ability to search for possible test scenarios based on the source and target. Thus, we perform this function on the website; on the backend, we incorporate a graph algorithm - Depth First Search (DFS) to traverse through the graph and give the possible scenarios based on the given source and target.

Finally, to add an additional layer of network graph of an application, we provide the visualization of a selected scenario from the list of possible scenarios. The idea here is to provide users an insight of the relationship type of the nodes and edges on the network of the selected path. Along with the nodes, we also highlight the GUI elements that were detected for the selected node. In Fig. 8 illustrated some of the examples of selected scenarios.

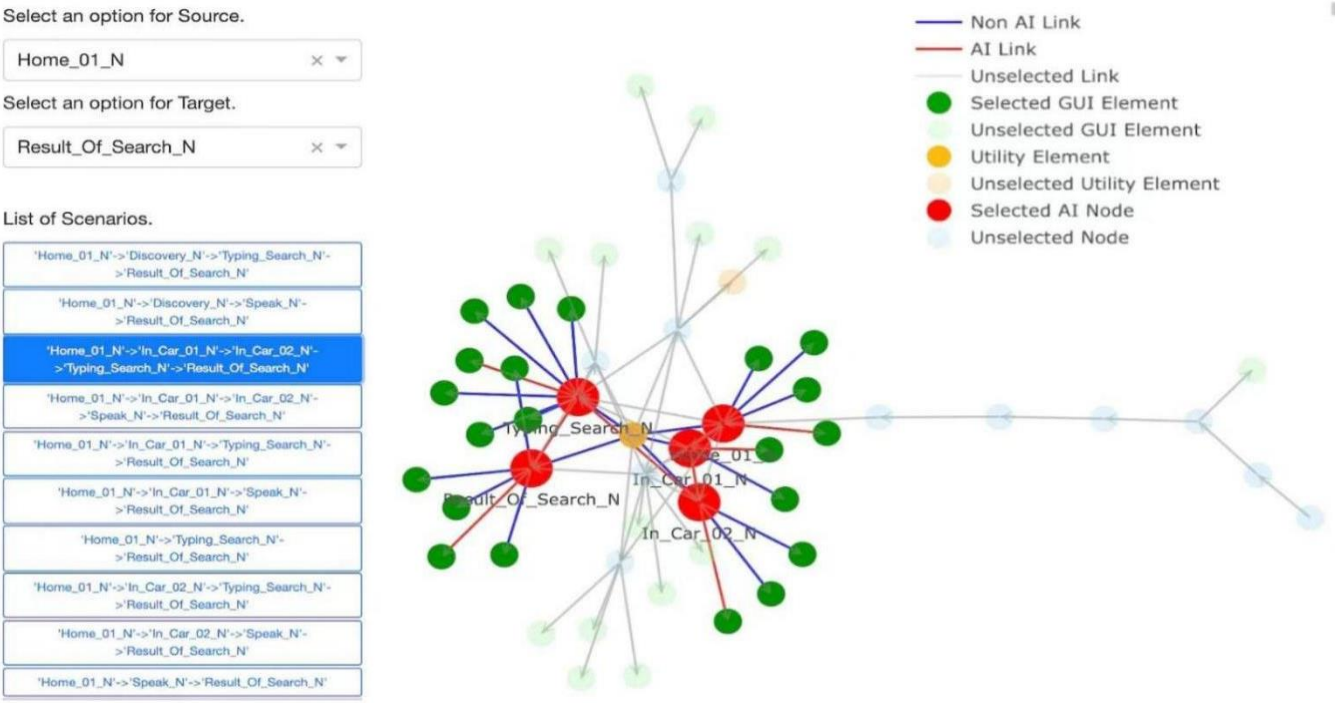


Fig. 8. Example of scenario selected - Hound app

V. SUMMARY

The main idea of the paper is to represent different pages and its embedded GUI elements and form test scenarios based on the traverse path in graphical representation. So that testers can do excessive testing on the concerned AI page if the need be. To achieve the graph representation of the test scenarios as we mentioned in earlier sections as well, we broke down the paper into three major components.

Module 1: GUI detection on the single screen——Once data engineering was complete, we trained two different deep learning object detection algorithms such as YOLOv5 and Efficient-Det0. It was essential to get as accurate results as possible to build reliable test scenarios for a given app.The major finding in this task was that models were able to predict the GUI elements with 60% accuracy but when we increased the training data the accuracy bumped to 75%.

Module 2a: Relationship identification of GUI elements and screens which we refer as nodes——This task involved annotation of the same screenshot dataset but in a different way where we annotated the GUI elements based on the relationship or AI functions. For example: screen with camera button were annotated as AICameraButton whereas other utility functions like settingsButton were annotated as nonAIButton and each screen is annotated as an AI Node or a nonAI Node.

Module 2b: Link classification——Link classification part was mainly to provide a clear understanding of what kind of AI relationship node the link is leading to, so that graphs are intuitive and help the end user understand the scenarios better. To implement a dynamic link classification algorithm, we created a separate relationship database which had the input from the GUI detection models that was appended to the relationship database based on the source and target column based on which the link classification was implemented using Python code.

Module 3: Test scenario generation——The test scenarios are given in a graphical representation using plotly library and NetworkX library. In the module, we were successfully able to project the overall test scenarios for a whole AI application, which consists of nodes(screen) and GUI elements detected by the models on a particular node.

Finally, with each module above, the paper realized test scenario generation driven by AI with minimum user input. The web portal is designed to provides the list of the possible test scenarios for the whole screen and highlight key elements for each pages (nodes/screens) to help end-users to deliver efficient testing results.

In the future, to raise the intuitiveness of the web application, a filter can be added to provide the end user the flexibility to see the possible scenarios when only the source is given. And the approach can be converted into an automated testing too which will be flexible for end-users to implement GUI test scenario generation.

REFERENCES

[1] University of Cambridge. “ “The best or worst thing to happen to humanity ” - Stephen Hawking launches Centre for the Future of Intelligence.”

[2] I. Banerjee, B. Nguyen, V. Garousi, et al. "Graphical user interface (GUI) testing: Systematic mapping and repository." Information and Software Technology 55.10 (2013): 1679-1694.

[3] SmartBear Software. “Artificial Intelligence for Faster and Smarter UI Testing.”

[4] T. Zhang, Y. Liu, J. Gao, et al. "Deep learning-based mobile application isomorphic gui identification for automated robotic testing." IEEE Software 37, no. 4 (2020): 67-74.

[5] T. Zhang, Y. Liu, J. Gao, et al. "FOCUS: THE AI EFFECT." THE AI EFFECT (2020): 67.

[6] J. Chen, M. Xie, Z. Xing, et al. "Object detection for graphical user interface: old fashioned or deep learning or a combination?." In proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1202-1214. 2020.

[7] Nguyen, T. Anh, C. Csallner. "Reverse engineering mobile application user interfaces with remaui (t)." In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 248-259. IEEE, 2015.

[8] S. Chen, L. Fan, T. Su, et al. "Automated cross-platform GUI code generation for mobile apps." In 2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile), pp. 13-16. IEEE, 2019.

[9] Y. Li, Z. Yang, Y. Guo, X. Chen. "Humanoid: A deep learning-based approach to automated black-box android app testing." In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1070-1073. IEEE, 2019.

[10] C. Anderson, A. Von Mayrhauser, et al. "On the use of neural networks to guide software testing activities." In ITC, pp. 720-729. 1995.

[11] X. Sun, T. Li, J. Xu. "Ui components recognition system based on image understanding." 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2020.

[12] K. Moran, C. Bernal-Cárdenas, et al. "Machine learning-based prototyping of graphical user interfaces for mobile apps." IEEE Transactions on Software Engineering 46.2 (2018): 196-221.

[13] GitHub. “ Overview of model structure about YOLOv5. ” https://github.com/ultralytics/yolov5/issues/280

[14] ODSC - Open Data Science. “Overview of the YOLO Object Detection Algorithm.”

[15] M. Tan, R. Pang, et al. "Efficientdet: Scalable and efficient object detection." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2020.

[16] Y. Wu, A. Kirillov, F. Massa, et al. “Detectron2: A PyTorch-based modular object detection library.” accessed September 14, 2021.

[17] W. Ogola. “An Introduction to Graph Neural Networks.” Section.io. accessed April 20, 2021.

[18] A. Parmar. “4 Types of Tree Traversal Algorithms.” Towards Data Science. accessed September 19, 2021.

[19] NetworkX. https://networkx.org/.

[20] TensorBoard. https://www.tensorflow.org/tensorboard.