




# GUI Element Detection from Mobile UI Images Using YOLOv5

Mehmet Dogan Altinbas<sup>(✉)</sup>  and Tacha Serif<sup>(✉)</sup> 

Yeditepe University, 34755 Atasehir Istanbul, Turkey  
{daltinbas, tserif}@cse.yeditepe.edu.tr

**Abstract.** In mobile application development, building a consistent user interface (UI) might be a costly and time-consuming process. This is especially the case if an organization has a separate team for each mobile platform such as iOS and Android. In this regard, the companies that choose the native mobile app development path end up going through do-overs as the UI work done on one platform needs to be repeated for other platforms too. One of the tedious parts of UI design tasks is creating a graphical user interface (GUI). There are numerous tools and prototypes in the literature that aim to create feasible GUI automation solutions to speed up this process and reduce the labor workload. However, as the technologies evolve and improve new versions of existing algorithms are created and offered. Accordingly, this study aims to employ the latest version of YOLO, which is YOLOv5, to create a custom object detection model that recognizes GUI elements in a given UI image. In order to benchmark the newly trained YOLOv5 GUI element detection model, existing work from the literature and their data set is considered and used for comparison purposes. Therefore, this study makes use of 450 UI samples of the VINS dataset for testing, a similar amount for validation and the rest for model training. Then the findings of this work are compared with another study that has used the SSD algorithm and VINS dataset to train, validate and test its model, which showed that proposed algorithm outperformed SSD's mean average precision (mAP) by 15.69%.

**Keywords:** Object detection · Graphical user interface · Deep learning

## 1 Introduction

Nowadays smartphones are much more capable than the computer that guided the first man to the moon [1]. Indeed, smartphones these days have powerful processors, advanced cameras, precise sensors, and user-friendly operating systems. Currently, there are more than 6 million mobile apps available in leading app markets for smartphones, tablets, and other devices running on different operating systems [2]. The revenue generated by these mobile apps is expected to be more than \$935 billion by 2023. That is almost double the revenue generated in 2020 [3].

One of the most cumbersome parts of mobile application development is the design of its UI. There are multiple steps that need to be undertaken to build and design a user-centered and user-friendly interface that would be easy to use by its target audience.

Hence, depending on the target group - e.g. young children, teenagers, professionals, elderly etc. - of the user base, the UI is created with different GUI element sizes, color schemes and haptics. Developing a GUI is one of those repetitive tasks since every mobile platform has its own UI design editor, programming, or markup language. There are several techniques - such as native development and cross-platform development - to implement mobile applications and their UI. Cross-platform applications are the ones that can run on multiple platforms with little or no modification. On the other hand, native apps are built specifically for a particular platform or operating system. Essentially, when these kinds of apps are built, they are designed to run on one specific platform. So, if an app is made for Android, it will not work on iOS or vice versa. Therefore, work done for one platform is required to be duplicated for another. There are a considerable number of studies in the literature that have proposed tools and frameworks to improve GUI development processes by addressing GUI element detection and automated GUI generation. These studies utilize current or newly emerged algorithms or models to achieve this automation. Bearing in mind that, GUI element detection in mobile UI images is a domain-specific object detection task, there is still room for improvement based on the cutting-edge improvements in the area of object detection. To the best of our knowledge, no work has utilized the YOLOv5 object detection algorithm for mobile GUI element detection. Therefore, this study aims to create and train a model that identifies a set of known GUI elements and detects their location within a UI image.

Accordingly, this paper is structured as follows; Sect. 2 provides a brief introduction to GUI element detection and elaborates on its common use cases. Section 3 details the tools, algorithms, UI image datasets, and performance evaluation metrics for object detection tasks. Section 4 describes the requirements and system design. Section 5 depicts the implementation of the prototype, and Sect. 6 provides the results in detail and elaborates on the findings. Finally, Sect. 7 summarizes the outcomes and discusses future development areas.

## 2 Background

By making use of 50000 GUI images, Chen et al. [4] attempted to undertake an empirical comparative study using the seven most popular GUI detection methods to evaluate their capabilities, limitations, and effectiveness. The findings of this study point out that existing computer vision (CV) techniques are inadequate in extracting the unique characteristics of GUI elements and detecting their location with high accuracy. So, the authors propose and implement a hybrid prototype, where they combine both the CV methods and deep learning models to test the hybrid method's extraction and location accuracy. The newly proposed prototype uses CV methods to detect non-text GUI elements and deep learning models for text-based GUI elements. As a result, their findings indicated that the proposed system performed 86% better in accuracy of region classification for non-textual GUI elements; whereas 91% accuracy in region classification of all elements.

Similarly, Xie et al. [5] suggested a tool that utilizes CV and object detection methods to provide accurate positioning of GUI elements along with their types. This tool is designed to assist GUI reverse engineering or GUI testing operations. Accordingly,

in their work, they take the advantage of CV algorithms to detect the position and frame of non-text GUI elements. Algorithms such as flood-filling, Sklansky's, and shape recognition are used to obtain layout blocks and as a follow up; the connected component labeling technique is applied to identify the region of the GUI components. Following that, the ResNet50 classifier, trained with 90000 GUI elements, is then used to recognize and group each GUI element in predefined regions into 15 categories. On the other hand, in order to detect and expose text-based GUI elements, they utilize the EAST text detector. Consequently, it has been determined that the suggested tool produces more accurate identification results than the similar studies in the literature.

Bunian et al. [6] proposed a visual search framework that takes a UI design as an input and returns design samples that resemble visually. A part of this study includes creating a custom GUI element detection model. This model is generated by using a custom dataset of 4543 UI images and adopting the SSD (Single Shot MultiBox Detector) high-accuracy object detection algorithm. It is used to find types of GUI components and their positions in the input image. The GUI element detection model is tested using 450 images from their custom dataset and it has achieved a mean average precision (mAP) of 76.39%.

On the other hand, Nguyen et al. [7] proposed a system that automatically identifies UI components from iOS or Android app screenshots and generates user interfaces that closely resemble the original screenshots. The authors of this study intend to automate the UI code generation based on existing app screenshots. Accordingly, the proposed system applies CV and optical character recognition (OCR) algorithms on a given screenshot image. After that, overlapping areas and text blocks are rearranged by adhering a set of rules that were defined by the authors. This rearrangement step is followed by UI element classification and app generation. Their evaluations showed that the proposed system produces very similar UIs to the original input image. Also, despite its high precision, the prototype system was able to generate UI code for each given screenshot in 9 s on average, which is an admirable performance.

Last but not least, Chen et al. [8] proposed a framework that takes a UI image as input and generates GUI code for the requested target platform. This study aimed at lowering the development and maintenance cost of GUI code generation for both Android and iOS platforms. The implemented prototype has three sub-processes. The first process entails the component detection task, which is performed using CV algorithms such as Canny edge and Edge-dilation. Additionally, a convolutional neural network (CNN) is used to train two classification models based on iOS and Android platforms to identify the types of the detected GUI components. The second process involves the type mapping of the detected GUI components for iOS and Android platforms. And lastly, the final process encompasses the GUI file generation, which is accomplished by using individual GUI code templates for Android and iOS apps. The test results of the prototype system showed that it achieved 85% accuracy in GUI element classification and managed to create UI designs that are 60%–70% compatible and similar to the original UI image.

### 3 Methodology

In line with the literature detailed above, this section discusses the tools, algorithms, datasets, and evaluation metrics that are used for GUI element detection.

#### 3.1 Tools

There are numerous approaches to implementing a GUI element detection model. Accordingly, there are many development environments and an extensive set of libraries that are built upon various programming languages. For example, OpenCV (Open-Source Computer Vision Library) [9] is a free and open-source software library for computer vision and machine learning. OpenCV provides a common infrastructure for computer vision applications. It supports several programming languages and platforms. Additionally, Apple has recently launched its CV framework called Vision [10], which enables developers to utilize computer vision algorithms. Also, Google Colab [11] has been developed as an open-source project to provide researchers with a fully functional computing environment, where they can write and execute arbitrary Python code for machine learning tasks without needing to set up a whole computing infrastructure.

#### 3.2 Algorithms

It is undoubtedly true that object detection tasks can take advantage of image processing algorithms. These algorithms can be used to filter out redundant or less relevant information and highlight the desired parts and important features of the image, which can be an invaluable asset in GUI element detection. Accordingly, an image in hand can be sharpened or its contrast can be increased using these algorithms, which as a result end up in a much clearer form with highlighted edges, finer details, and more explicit regions. Also, image processing algorithms can be used to flip, rotate, scale, or crop images to achieve data augmentation and increase the amount of data that can be used while an object detection model is trained. Image processing techniques can be considered as an intermediary step in the preparation and enhancement of a raw input image before it is processed with object detection algorithms to derive some meaningful information.

Even though numerous studies have combined and utilized the aforementioned algorithms successfully, deep learning techniques also produce promising results when used for GUI element detection. Deep learning, which is a subdomain of machine learning, introduces a more accurate way of computer vision as it uses a more sophisticated algorithm called artificial neural network (ANN). An ANN mimics the human brain as it consists of millions of interconnected processing nodes that correspond to similar behavior to neurons in the brain. Therefore, ANNs are broadly used to make generalizations and inferences. Also, it is one of the best techniques to find patterns, make predictions and reveal the relationships between the input and the output of any given problem.

There are several types of neural networks depending on the context of their use; however, convolutional neural networks (CNNs) [12] are the ones that are most utilized in object detection. CNNs vary based on the procedure they follow. For instance, in region-based convolutional neural networks (R-CNN), an image is divided into region proposals and then CNN is applied for each region to extract the most fundamental features. Although this approach produces good results, the training phase is long and time-consuming since it applies CNN to each one of the regions one by one. On the other hand, Fast R-CNN [13], which is proposed to solve the aforementioned drawbacks of R-CNN, performs faster object detection by feeding input images to CNN rather than feeding the region proposals individually. However, You Only Look Once (YOLO) [14] is one the most popular algorithms for object detection. YOLO stands out from the above-mentioned algorithms as it seeks parts of the image rather than complete image and predicts the bounding boxes of objects and their class probabilities.

### 3.3 Datasets

There are several datasets in the literature that are used in various GUI-related studies. The Rico dataset [15] is a collection of design data that reveals the graphical, textual, structural, and interactive design attributes of Android apps. These attributes can be convenient to support data-driven applications in various ways such as design searching and UI layout generation. This dataset consists of 72000 individual UI screens from more than 9700 Android applications, which are created based on 27 different app categories.

Last but not least, the VINS dataset [6] is a collection of annotated UI images and it consists of wireframe images, screenshots from iPhone and Android apps. The UI designs in the dataset are from the Uplabs [16], and UI screens from the Rico dataset. It comprises 4800 pairings of UI design images with their matching XML files in Pascal-VOC format. The original study [6] where the VINS dataset was first introduced aimed at building an object detection-based visual search framework for a given mobile UI image. The evaluation result of the framework produced promising results in terms of GUI element detection accuracy and visual search precision.

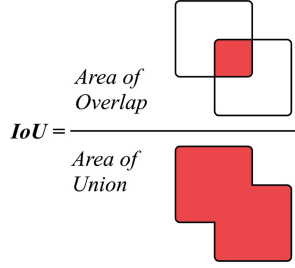
### 3.4 Object Detection Evaluation Metrics

Object detection evaluation metrics reveal how well the object detection model is performing. Accordingly, precision is the ratio of correctly detected elements to all the elements detected by the model. It is a measure of how accurate a model is at predicting positive samples. As it is depicted in Eq. 1, it is calculated by dividing the true positives by the total number of positive predictions. Also, recall is a measure of how many relevant objects are detected, which is calculated by dividing the true positive elements by the total number of true positives and false negatives - see Eq. 2.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \quad (1)$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \quad (2)$$

On the other hand, the intersection over union (IoU) (Fig. 1) is the ratio that is obtained by dividing the number of pixels in the intersection between a ground-truth object and a predicted object, with the the number of pixels in the union. It is mostly used as an evaluation metric for tasks such as object detection and segmentation. Also, the average precision (AP) metric is often used to assess the accuracy of object detectors such as the Faster R-CNN and YOLOv5. AP is calculated as the area under the precision-recall curve.

$$IoU = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


**Fig. 1.** Intersection over union formula

## 4 Analysis and Design

In order to create a custom GUI detection model, a collection of mobile UI images must be either newly generated or an existing dataset should be used. Furthermore, these images should be grouped in a way that they contain predominantly a specific GUI element - where each image group becomes a training class for the specific GUI element. These classes can be seen as the objects that are intended to be discovered by the object detector.

Considering that this study aims to evaluate accuracy and the average precision of the new YOLOv5 model, there was a need for a similar study where the same classes and evaluation phases can be followed. Accordingly, the training image dataset and the algorithm that was suitable for our study was the work of Bunian et al. [6]. Therefore, all the image classes that need to be generated for this study, required to follow suit the structure of Bunian et al.'s work. For the very same reason, instead of creating our own dataset of UI images, the VINS dataset which was used in Bunian et al.'s study is obtained and split into multiple subsets - i.e. training images, validation images and testing images. In order to keep the consistency of the comparative work, this model was also to be trained, validated and tested with the very same number of UI images of the same image dataset. In the comparative study [6], the authors make use of 4543 out of 4800 VINS UI images; where the total 4543 is split into three smaller groups as 3643 for training, 450 for validation and 450 for testing.

Taking into consideration the above image numbers, the system that would be training the model should at least have a decent amount of CPU and GPU resources. There are multiple paths that could be taken to train this model - such as local PC-based or cloud-based training approaches. Each one of these options do contain their own advantages, such as for the prior, all the data is stored locally and training could last as long as required. The latter does not involve running a local server or inhouse PC, which can be distracted by hardware failure or power outage. However, if the training process is conducted on cloud, then the researcher is limited with the amount of service time that is given by the cloud provider, which can be a cumbersome task if the training phases take long periods of time and require re-setup/reconfigure every time.

## 5 Implementation

The following section details the preparation and development steps taken to achieve the proposed GUI element detection model. Accordingly, to train the prototype model, a Mac mini, with 3 GHz 6-Core 8<sup>th</sup> generation Intel core i5 and 32 GB memory, is used to carry out all the operational steps necessary - such as data preparation and model training. As mentioned in the previous section, the same training can be conducted on Google's Colab cloud solution, however, when this option was tried initially, the time required to train the model increased substantially, since the Colab virtual machine lifetime is limited to 12 h and required reconfiguration to continue from the last checkpoint. Hence, the implementation of the prototype proceeded with an inhouse Mac mini.

First and foremost, the implementation started with data preparation, which involved the conversion of the Pascal VOC annotation files to a YOLOv5 compatible format. Thereafter, the GUI element detection model training phase is performed.

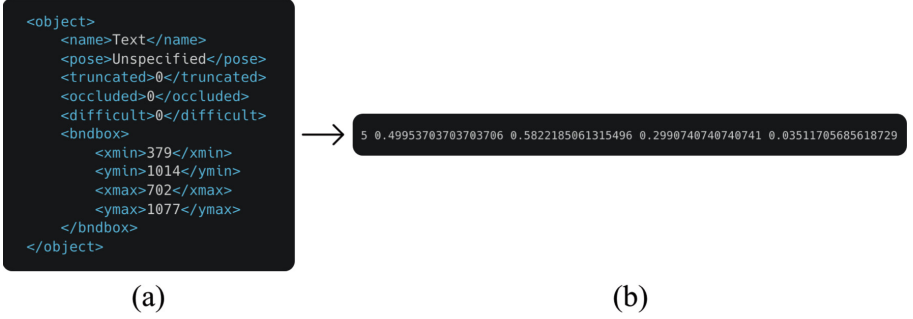
### 5.1 Preparing the Dataset

The work by Bunian et al. [6] has two phases: (a) to identify and locate UI elements in any given screen capture image using a trained model by the SSD algorithm; (b) to use the initially created VINS image dataset to compare with the given screen capture image and identify the most similar images based on UI element hierarchies. Since this work only aims to evaluate and compare the precision of UI element identification and location using a YOLOv5 trained model, it will try to faithfully follow the steps of the first phase (a) of Bunian et al. Therefore, all class sizes and names used in the implementation will comply with the existing study.

In line with the Bunian et al. study, the total number of UI images in the dataset is reduced from 4800 to 4543. This reduction is achieved by removing the existing wireframe images in the original dataset. Also, to match the dataset structure with the existing study, 4543 UI images are split into three sets for training, validation and testing - respectively having the ratio of 80%, 10% and 10%. Furthermore, the naming convention of the classes that are used for training were kept in compliance with the existing study so that the results of the new model can be compared directly. Accordingly, the names of the classes generated are *Background Image*, *Checked View*, *Icon*, *Input Field*, *Image*,

*Text, Text Button, Sliding Menu, Page Indicator, Pop-Up Window, Switch and Upper Task Bar.*

The VINS dataset contains annotation files along with the mobile UI images. The annotations are stored in XML files using Pascal VOC format. However, YOLOv5 demands annotations for each image in the form of a TXT file, with each line describing a bounding box. Therefore, a Python script is run in the directory where the dataset is located, to find each of the XML files (Fig. 2a) and convert them to TXT format (Fig. 2b).



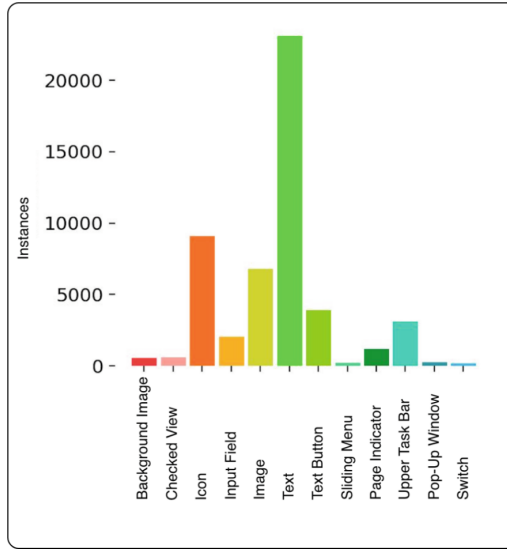
**Fig. 2.** An example of conversion from Pascal VOC XML to YOLO TXT

## 5.2 GUI Element Detection Model

The YOLOv5 algorithm is employed to create a custom GUI element detection model. It is deemed appropriate since YOLO family algorithms tend to be fastly deployable - requiring limited number of library and dependency setup; but on the other hand, resulting with a swift and accurate real-time object detection architecture.

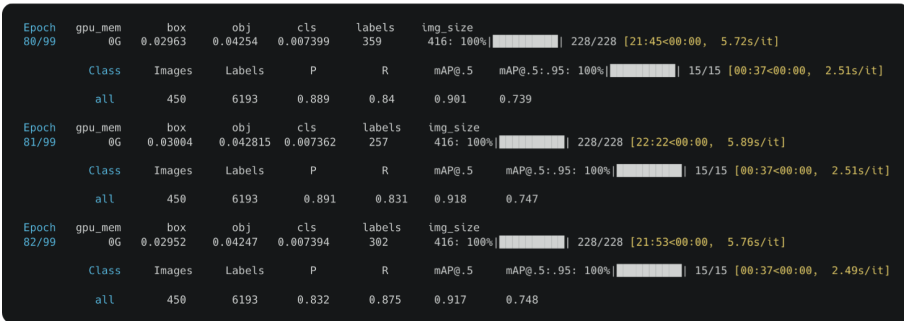
Accordingly, initially the YOLOv5 repository, which includes its open-source implementation, is cloned to the computer. Then, the YOLOv5 libraries and dependencies are installed. The YOLOv5 algorithm takes the required configuration data from two *yaml* text files - namely *train.yaml* and *data.yaml*. The *train.yaml* includes parameters that describe the model architecture. In order to speed up the training process, there are a couple of pre-trained models (*YOLOv5n*, *YOLOv5s*, *YOLOv5m*, *YOLOv5l* and *YOLOv5x*) within the repositories. For the purpose of UI image training, in our study *YOLOv5s*, which is the smallest and fastest model available, is used as the reference starting point. Hence, the pretrained weights of *YOLOv5s* are used as the beginning of the training process to avoid an overfitting of the model. In the second configuration file - *data.yaml* - the training, validation, and test set paths on the local PC are provided. Also a parameter called *nc* is set, which declares the number of classes in the dataset in both of the *train.yaml* and *data.yaml* files. Considering the total number of classes - GUI element types - that is utilized as part of this training process, the *nc* value is set to 12.





**Fig. 3.** Distribution of class labels in training dataset

After the *yaml* files are configured, the model training phase is initiated. The above figure (Fig. 3) shows the distribution of the total 50413 annotated GUI instances in 3643 training images, which are split into 12 classes. The new model is trained (Fig. 4) for 100 epochs and with a batch size of 16 training images; and as a result, it took 38 h and 33 min to fully train the GUI element detection model.



**Fig. 4.** A snapshot from the training phase

After the training phase is completed, the resulting model is used to detect GUI elements in a given image. This process is achieved by invoking the Python detector script, which is provided as part of the YOLOv5 repository. The detector script is fed with an UI image (Fig. 5a) and after running the trained model, it infers the relevant UI items and their location on images and saves the annotated image (Fig. 5b) to a specified destination folder.

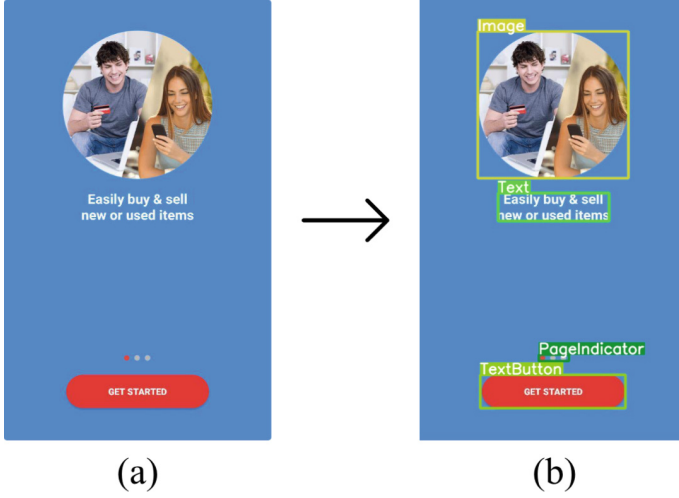


Fig. 5. Running inference on an image using the trained model

## 6 Evaluation

Since one of the main objectives of this study is to train and evaluate a newly released object detection model, which in this case is used to detect and classify GUI elements in UI images, this section initially draws plots using PASCAL VOC (IoU = 0.5) and MS COCO [17] (AP@[0.5:0.05:0.95]) evaluation metrics. The rest of the section presents the class base average precision findings and compares with the benchmark study.

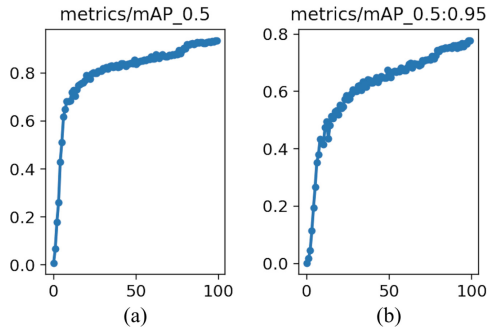


Fig. 6. Accuracy results of model training on the validation set

After each epoch during the training process, the validation set is used to evaluate the model. This evaluation is specifically observed to optimize and fine-tune the model hyperparameters for the subsequent epoch of training. Thus, it is ensured that the training proceeds in the right direction without any issues. Accordingly, the total of 450 UI images that are selected as the validation set are utilized to calculate the mAP values at specific IoU thresholds. As a result, the validation findings show that the model has

achieved an mAP (IoU = 0.5) of 93.3% at the last epoch of the training process (Fig. 6a). Considering the MS COCO dataset challenge, which introduces a new evaluation metric (AP@[0.5:0.05:0.95]), during the training process APs were calculated at 10 IoU thresholds between 0.5 and 0.95 with a step size of 0.05. The average of the 10 calculated APs achieved the mAP of 77.6%. The two calculated mAP results conducted on the validation set show that the model incrementally gives better results in classifying the GUI elements by each epoch.

The second part of the evaluation entails the comparison of this newly trained model with the existing benchmark study's results. Therefore, after the model training has ended, the Python script, which is provided as part of the YOLOv5 repository, has been deployed on the selected 450 testing UI image dataset. After this process, an AP value is generated per input class, which then their average is calculated to obtain the overall mAP.

Based on the calculations described above, Table 1 depicts the average precision findings for each one of the classes in our prototype and the benchmark study by Bunian et al. As can be clearly seen, the superiority of the model that is trained with the YOLOv5 is apparent to the naked eye. To put in numbers, the average precision of 10 out of 12 classes has yielded better results compared to the Bunian et al. [6] model. Even though the YOLOv5 model has achieved slightly lower AP score in the *Background Image* and *Sliding Menu* classes than the Bunian et al. [6], it successfully has achieved higher average precision rates in smaller GUI elements, such as *Icon*, *Checked View* and *Page Indicator*. Overall, it is calculated that the proposed YOLOv5 model has outperformed the Bunian et al. [6] model's mAP by 15.69%.

**Table 1.** Average precision (AP) at IoU of 0.5 for each of the 12 class labels on the test set

Class label	AP (%) at IoU = 0.5	
	Bunian et al. [6]	YOLOv5
Background image	89.33	84.62
Checked view	44.48	67.59
Icon	50.5	89.58
Input field	78.24	94.57
Image	79.24	83.87
Text	63.99	96.3
Text button	87.37	98.83
Sliding menu	100	99.5
Page indicator	59.37	97.96
Pop-Up window	93.75	95.52
Switch	80	97.16
Upper task bar	90.4	99.49
<b>Overall mAP (%)</b>	<b>76.39</b>	<b>92.08</b>

The findings have been further explored through the confusion matrix (at IoU = 0.5) to evaluate the matching success of the test UI images to the corresponding 12 classes - see Fig. 7. The figure indicates that all the *Sliding Menu* and *Upper Task Bar* containing images have success matched with the correct class. Following this, the second best performance of matching with the relevant classes have been achieved by the test UI images that contain *Text*, *Text Button*, and *Page Indicator* instances. However, UI images that contain *Checked View* and *Switch* instances have not performed as well. This problem is believed to be related to the size of these UI elements in relation to the image in general. Hence, it can be said that smaller UI elements tend to be harder to detect and locate. Lastly, it has been observed that some akin and look-alike UI types such *Icon* and *Image* can be falsely predicted. As a result, the newly generated model falsely predicted %18 of *Icons* as *Images*; similarly, falsely predicted %9 of *Images* as *Icons*.

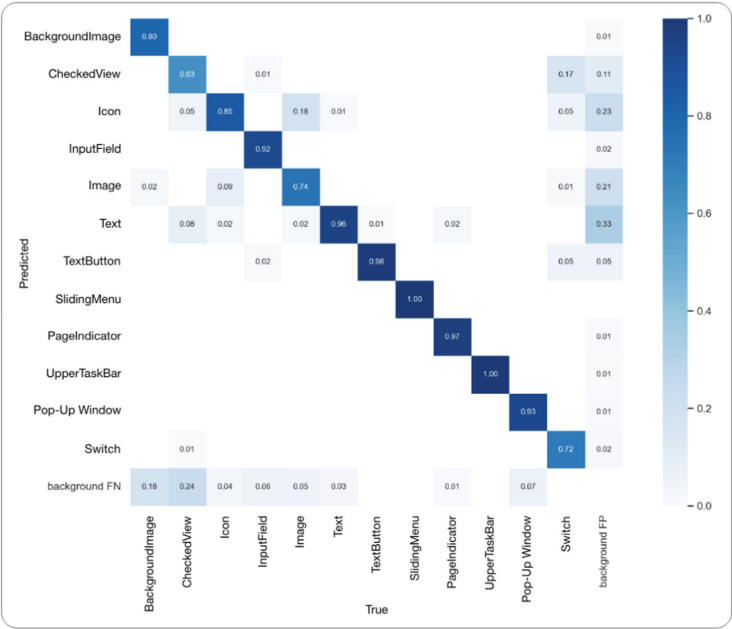


Fig. 7. Confusion matrix based on 450 test images

## 7 Conclusion and Future Work

Overall, this study proposes, trains and develops a GUI element detection model that is making use of the YOLOv5 algorithm and VINS dataset. The original dataset contains 4800 UI images and their annotation files, which is later reduced to 4543 images to make it comparable with the Bunian et al. [6] study. Accordingly, a total of 4543 images are split into 3 subsets for training, validation, and testing purposes. Furthermore, the

accompanying VINS dataset annotation files are converted to YOLO TXT format. The model is trained using the newly proposed YOLOv5 algorithm that employs the pre-selected training dataset. Then in line with the benchmark study, 450 images are used for validation and the same number of images for testing. Accordingly, it is observed that the model has given 15.69% better mAP than the model used in the Bunian et al. [6] study. Overall, the comparison showed that the newly created model has performed better than the benchmark study in 10 out of 12 classes.

Based on the experience gained throughout the evaluations, it is foreseen that if the classes used for training were more evenly distributed the final results could have been much more accurate. Even though the newly generated model outperforms the set benchmark study findings, there is still room for improvement in the detection of small GUI elements. It is believed that this could be achieved by tiling the dataset [18] or using a different YOLO-based algorithm specifically developed for small object detection. The success of this phase would inevitably lead to the next step which involves automated UI file generation and UI design regeneration for multiple mobile platforms.

## References

1. Puiu, T.: Your smartphone is millions of times more powerful than the Apollo 11 guidance computers. ZME Science (2021). <https://www.zmescience.com/science/news-science/smartphone-power-compared-to-apollo-432/>. Accessed 15 Jan 2022
2. Ceci, L.: Number of apps available in leading app stores as of 1st quarter 2021. Statista (2022). <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>. Accessed 30 Jan 2022
3. Statista Research Department: Revenue of mobile apps worldwide 2017–2025, by segment. Statista (2021). <https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/>. Accessed 02 Feb 2022
4. Chen, J., 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. ACM (2020)
5. Xie, M., et al.: UIED: a hybrid tool for GUI element detection. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1655–1659. ACM (2020)
6. Bunian, S., et al.: VINS: visual search for mobile user interface design. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pp. 1–14. ACM (2021)
7. Nguyen, T.A., Csallner, C.: Reverse engineering mobile application user interfaces with REMAUI (T). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 248–259. ACM (2015)
8. Chen, S., et al.: Automated cross-platform GUI code generation for mobile apps. In: IEEE 1st International Workshop on Artificial Intelligence for Mobile, pp. 13–16. IEEE (2019)
9. OpenCV About. <https://opencv.org/about/>. Accessed 13 Feb 2022
10. Apple Developer Vision. <https://developer.apple.com/documentation/vision>. Accessed 16 Feb 2022
11. Google Colab. <https://colab.research.google.com>. Accessed 1 Jan 2022
12. Zhiqiang, W., Jun, L.: A review of object detection based on convolutional neural network. In: Proceedings of the 36th Chinese Control Conference, pp. 85–112. IEEE (2017)

13. Girshick, R.: Fast R-CNN. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1440–1448. IEEE (2015)
14. Redmon, J., et al.: You only look once: unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 779–788. IEEE (2016)
15. Deka, B., et al.: 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, pp. 845–854. ACM (2017)
16. What is Uplabs? <https://www.uplabs.com/faq>. Accessed 18 Feb 2022
17. Lin, T.-Y., et al.: Microsoft coco: common objects in context. In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds.) ECCV 2014. LNCS, vol. 8693, pp. 740–755. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-10602-1\\_48](https://doi.org/10.1007/978-3-319-10602-1_48)
18. Unel, F.O., et al.: The power of tiling for small object detection. In: Proceeding of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 582–591. IEEE (2019)