
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**
**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**


# IDENTIFICATION OF GRAPHICAL USER INTERFACE ELEMENTS FOR ROBOTIC TESTING SYSTEM

**IDENTIFIKACE ELEMENTŮ GRAFICKÉHO UŽIVATELSKÉHO ROZHRANÍ PRO ROBOTICKÝ TESTOVACÍ SYSTÉM**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** **Bc. LUKÁŠ VÁLEK**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. MICHAL ŠPANĚL, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2022**

**Brno University of Technology**
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2021/2022

# Master's Thesis Specification


Student: **Válek Lukáš, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Computer Vision
Title: **Automated Identification of Graphical UI Elements for Robotic Quality Assurance**
Category: Image Processing
Assignment:

1. Get familiar with methods and existing solutions for automatic visual identification of UI elements in images and analysis of UI elements semantics.
2. Prepare a dataset with appropriate scenes with UI elements for your own experiments.
3. Select suitable methods and propose solution for identification of UI elements and analysis of their semantics in your dataset.
4. Implement selected methods and experiment with them. Propose further improvements.
5. Evaluate your results. Demonstrate advantages and disadvantages of your solution and discuss possible future work.
6. Create a short poster or video presenting your work, its goals and results.

Recommended literature:

- According to the superviser's instruction.

Requirements for the semestral defence:

- Items 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Španěl Michal, Ing., Ph.D.**
Head of Department: Černocký Jan, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: November 2, 2021

# Abstract

This thesis explores the issue of graphical user interface (GUI) screen analysis using convolutional neural networks (CNN) and computer vision methods. The thesis aims to create a system which automatically identifies GUI elements based on pictogram and text information for detected components in an input image. A combination of EfficientNetB1 CNN, OCR, and traditional computer vision methods was used to develop the system. A custom dataset which contains 120k pictograms was used to train the CNN. A UI element semantic dictionary was created, which further utilises the text detected by OCR. Finally, a GUI hierarchy analysis subsystem was created to detect and semantically categorise sections in GUI. The resulting system automatically classifies detected pictograms, suggests additional text classes, and separates the GUI screen into hierarchical sections. The system achieves 81.1% UI element identification accuracy and, on average, analyses a single screen in 0.6 seconds. This system automates repetitive processes, thus decreasing needed person-hours. In the future, the system can be further developed to function as a foundation for automated exploratory testing.

# Abstrakt

Tato práce se zabývá problematikou analýzy obrazovek grafického uživatelského rozhraní (GUI) pomocí konvolučních neuronových sítí (CNN) a metod počítačového vidění. Cílem této práce je vytvořit systém, který automaticky identifikuje GUI elementy na základě piktogramových a textových informací pro detekované prvky ve vstupním obrázku. K vývoji systému byla použita kombinace EfficientNetB1 CNN, OCR a tradičních metod počítačového vidění. K trénování CNN byla použita vlastní datová sada, která obsahovala 120 tisíc piktogramů. Byl vytvořen sémantický slovník UI prvků, který dále využívá text detekovaný pomocí OCR. Nakonec byl vytvořen podsystém pro analýzu GUI hierarchie, který slouží k detekci a sémantické kategorizaci oblastí GUI. Výsledný systém automaticky klasifikuje detekované piktogramy, navrhuje další třídy na základě textu a rozděluje GUI obrazovku do hierarchických sekcí. Systém dosahuje 81,1% přesnosti identifikace UI prvků a v průměru zanalyzuje jednu obrazovku za 0,6 sekundy. Systém automatizuje indetifikaci UI prvků, čímž umožňuje zaměstnancům věnovat se jiným činnostem. V budoucnu lze tento systém dále rozvíjet, aby sloužil jako základ pro automatické exploratorní testování.



# Keywords

UI element identification, object identification, EfficientNetB1, neural networks, image classification, semantic analysis of graphical user interface, computer vision, machine learning, semantic dictionary of UI elements, OCR, Python, Keras, TensorFlow, OpenCV

# Klíčová slova

identifikace UI prvků, identifikace objektů, EfficientNetB1, neuronové sítě, klasifikace obrázků, sémantická analýza grafického uživatelského rozhraní, počítačové vidění, strojové učení, sémantický slovník GUI prvků, OCR, Python, Keras, TensorFlow, OpenCV

# Reference

VÁLEK, Lukáš. *Identification of Graphical User Interface Elements for Robotic Testing System.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Španěl, Ph.D.

# Rozšířený abstrakt

V dnešní době představují grafická uživatelská rozhraní (GUI) primární způsob interakce uživatelů s aplikacemi. Atraktivní uživatelská rozhraní (UI) a instinktivní uživatelské zkušenosti (UX) proto hrají obrovskou roli v uživatelsky vnímané kvalitě aplikace. Vývojáři tudíž kladou velký důraz na testování aplikací s GUI. V případech, kdy je zdrojový kód aplikací nebo struktura GUI (HTML) přístupná, mohou testeři psát automatické skripty, které aplikace testují. V případech, jako jsou aplikace do vestavěných terminálů tiskáren nebo software pro automobilový infotainment, však některé interakce, jako třeba kontrola funkčnosti klimatizace nebo ověření vložení papíru do tiskárny nelze posoudit jiným softwarem. Tyto testy musí lidé provádět ručně, jedná se ale o monotónní a opakující se činnosti. Proto existují robotická řešení, která toto testování automatizují. Tato automatizovaná robotická řešení vizuálně detekují stav GUI, pomocí robotických ramen interagují s UI a lze je upravit tak, aby pokryla potřeby různých domén, jako je podpora teplotního senzoru k ověření funkčnosti klimatizace. Tato řešení však musí mít přehled, jaké UI prvky se na obrazovce vyskytují.

Tato práce si klade za cíl vytvořit systém, který ve snímku obrazovky automaticky identifikuje prvky UI, a to rychleji, než by to byl schopen udělat uživatel. Dále se práce pokouší navrhnout řešení pro analýzu GUI hierarchie. Práce také zkoumá širokou škálu metod počítačového vidění pro identifikaci objektů a prozkoumává dostupné datasety s obrazovými daty vhodné pro tuto práci. Práce se dále snaží analyzovat podobná dostupná řešení nastíněného problému.

Za účelem vytvoření systému byly vyvinuty následující komponenty: identifikátor UI prvků, algoritmus pro extrakci textu, sémantický slovník piktogramů a analyzátor GUI hierarchie. Identifikátor UI prvků využívá CNN EfficientNetB1 ke klasifikaci piktogramů ve vstupním obrázku do 61 často používaných tříd, které reprezentují prvky UI. CNN je natrénována na vlastní datové sadě UI prvků obsahující 120 tisíc obrázků v populárních designových stylech. Algoritmus pro extrakci textu používá k detekci textu ve vstupním obrázku interní OCR společnosti YSoft. Text je následně zpracován a přiřazen k odpovídajícím prvkům UI. Sémantický slovník piktogramů je ručně sestavený seznam slov, kterými lze daný piktogram popsat. Tento slovník je porovnán s extrahovaným textem a na základě shody textových informací jsou navrženy třídy UI prvků. Analýza GUI hierarchie využívá tradiční metody CV k nalezení a sémantické kategorizaci sekcí v GUI obrazovkách.

Cílem práce bylo vytvořit systém, který automaticky identifikuje prvky UI na základě piktogramových a textových informací pro detekované prvky ve vstupním obrázku. Tento cíl byl splněn. Výsledný systém automaticky klasifikuje nalezené piktogramy, navrhuje další třídy textu a rozděluje GUI obrazovku do hierarchických sekcí. Systém byl testován s použitím výše zmíněného datasetu, který sloužil jako referenční příklady. U 5 975 anotovaných UI prvků dosáhl systém top-1 přesnosti 70,2% (4 197) a top-5 přesnosti 79,8% (4 769). Sémantický slovník opravil 1,3% nesprávně klasifikovaných prvků, což vedlo k celkové přesnosti identifikace UI prvků 81,1%. 15% (898) prvků bylo identifikováno nesprávně kvůli chybám souvisejícím s OCR. 3,8% prvků bylo nesprávně identifikováno z důvodu jiných chyb. Během procesu hodnocení, který trval 11 minut a 28 sekund, bylo analyzováno 1218 obrázků. V průměru tedy systém zpracuje jeden obrázek za 560 milisekund.

Výsledný systém podléhá několika nedostatkům, která je třeba mít na paměti. Neschopnost použitého OCR správně přečíst jednotlivé znaky a občasné nesprávné umístění ohraničujícího rámečku textu jsou hlavními příčinami chyb systému. Algoritmus detekce piktogramů může zaměnit šum na vstupních obrázcích za hledaný objekt. Analýza GUI hierarchie není kvůli použitým metodám schopna detekovat menší oblasti, jako jsou například seznamy

položek a skupiny tlačítek. Bez ohledu na tato omezení představuje systém použitelnou variantu pro robustní automatickou identifikaci UI prvků. Toto řešení kategorizuje UI prvky do 61 různých tříd, zatímco jiná podobná řešení rozdělují prvky do přibližně 15 tříd. Tento systém navíc dokáže identifikovat ikony z více designových stylů, zatímco jiná řešení se specializují na jediný designový styl. Tento systém také dokáže zpracovat obrázky se značným množstvím vizuálního šumu, zatímco některá jiná řešení se spoléhají na čistý snímek obrazovky. UI prvky jsou navíc rozděleny do hierarchických sekcí pouze pomocí vizuálních informací, zatímco jiná řešení potřebují stromovou strukturu GUI aplikace. V poslední řadě, systém je schopen kategorizovat prvky uživatelského rozhraní pouze na základě textu, který obsahují.

V budoucnu by mohly být provedeny výzkumné práce s cílem zbavit systém jeho nedostatků a dále jej vylepšit. Ke zlepšení přesnosti systému by bylo možné vyzkoušet jiné OCR metody. Kromě toho by mohl být proveden výzkum týkající se rozšiřování sémantického slovníku piktogramů kontrolovanými uživatelskými příspěvky a automatickým překladem slovníku do dalších jazyků. Dále by mohly být zkoumány rychlé a spolehlivé metody pro odstranění Moiré efektu. Rovněž by se mohla zlepšit analýza GUI hierarchie. Místo tradičních metod CV by mohl být natrénován detektor objektů na obecných datových sadách UI prvků. To by systému umožnilo detekovat i konkrétní části, jako jsou seznamy položek a seskupená tlačítka. Dále by tento systém mohl sloužit jako základ pro řešení automatického exploratorního testování. Propojením tohoto systému a systému pro průzkum aplikací by mohlo být vytvořeno řešení, které by provádělo prozkoumávání aplikací pomocí robotického zařízení a vytvářelo grafovou reprezentaci testované aplikace.

Vytvoření systému odhalilo nedostatek veřejně dostupných datových sad, které obsahují UI prvky rozdělené do velmi podrobně rozčleněných tříd. Přístup k takovýmto datasetům by umožnil další otestování tohoto systému, čímž by se zvýšila důvěryhodnost prezentovaných hodnotících dat. Pomocí těchto datasetů by rovněž bylo možné vytvářet další pokročilejší řešení.

Závěrem lze říci, že tento systém ve své současné podobě automatizuje identifikaci UI prvků, čímž zbavuje zaměstnance povinnosti vykonávat tuto repetitivní a rutinní činnost.

# Identification of Graphical User Interface Elements for Robotic Testing System

## Declaration

I hereby declare that this Masters's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Michal Španěl Ph.D. The supplementary information was provided by the members of YSoft AIVA team. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Lukáš Válek
May 17, 2022


## Acknowledgements

I would like to thank my supervisor, Ing. Michal Španěl Ph.D., for his kind guidance, patience and valuable advice. I would also like to thank the YSoft AIVA team for their insightful comments and consultations. Lastly, I would also like to thank my dear ones for their endless support.

# Contents

# Chapter 1

# Introduction

In this day and age, a graphical user interface (GUI) represents the primary way users interact with applications. Therefore, an attractive user interface (UI) and instinctive user experience (UX) play a vital role in users' perceived application quality. Thus, developers place a great emphasis on the testing of said applications. When the application's source code or the GUI structure is accessible, testers can write automated scripts which test the application. However, in instances of printer embedded terminal applications or car infotainment software, some interactions, such as air conditioning and inserting paper into a printer, cannot be assessed by another software. These tests can be performed manually by humans, but they are repetitive and mundane. Therefore, there is a demand for automation of such tests. These automated robotic solutions detect the state of GUIs visually and interact with UIs using robotic arms. Moreover, they can be modified to cover the needs of different domains, like using temperature sensors to verify air conditioning functionality. These solutions need to be aware of UI elements present in GUIs.

This thesis aims to create a system which automatically identifies these UI elements faster than a user. Also, the thesis attempts to propose a solution for GUI hierarchy analysis. Furthermore, it explores a vast array of computer vision methods for object identification and relevant image datasets. Finally, the thesis strives to analyse similar available solutions.

This research is motivated by the above-mentioned circumstances. In addition, it is motivated by the author's personal interest in computer vision, machine learning, and automation.

The thesis is divided into two parts, theoretical and empirical. Chapter 2 discusses existing solutions for UI elements identification and GUI semantic analysis. It also introduces the benefits and limitations of these solutions. Chapter 3 explores the current state of convolutional neural networks to find a suitable method for the system. It also provides an overview of available datasets relevant to this task. The empirical part of the thesis begins with Chapter 4, which outlines the system. Furthermore, all components of the resulting system are proposed, and technical specifications and requirements are defined in this chapter. The implementation of the system, its components, and datasets are described in Chapter 5. The chapter also mentions the inputs, outputs, and limitations of each component. Chapter 6 discusses experiments with the system and its components. Additionally, it evaluates the system, shows achieved results, and proposes future work. The final Chapter 7 summarises the results and concludes the thesis.

# Chapter 2

# Existing Solutions for UI Elements Identification and GUI Semantic Analysis

This chapter explores existing solutions related to the issue of identification of UI elements and GUI semantic analysis. These solutions are compared and contrasted with the task of this thesis. The virtues and limitations of these solutions are discussed.

## 2.1 UI Components Recognition System Based on Image Understanding

Testing of UI designs is an essential part of the application development cycle. Traditionally, GUIs were tested by humans, which is not only very time consuming and expensive but also prone to errors. The authors of this paper, published in 2020, propose a solution to UI components recognition, enabling automatic application testing. [26]

The proposed solution uses image processing technology to detect UI components in the application screenshot. Then it classifies the detected components using a custom CNN. The CNN was trained using a randomly sampled *ReDraw* dataset 3.2. [26]

The screenshot image is converted into grayscale, filtered, thresholded, dilated and *closed*. Then, *Flood-Fill* [1] algorithm is used to differentiate sections in the GUI. Detected GUI sections are then analysed to find hierarchical relationships between them. Then, UI components and GUI sections are separated based on the size of the detected areas. UI components are then classified into generic classes corresponding to the *ReDraw* dataset. Two thousand five hundred images from each class were randomly sampled and divided into training (0.7), validation (0.2) and test (0.1) datasets. The precision of the CNN reached 86.4%. [26] Figure 2.1 shows the approach to UI components classification proposed in this solution.

The system discussed in this paper does a very similar analysis to the approach proposed in this thesis. It classifies found UI components into 14 classes. Also, it detects hierarchy relations within the GUI, albeit it does not use this information. However, the precision of the proposed classifier is somewhat poor, considering the number of classes in the dataset. Also, UI components detection is implemented using traditional computer vision, and the detected contours must be post-processed to produce consistent results.

[1] https://www.techiedelight.com/flood-fill-algorithm/

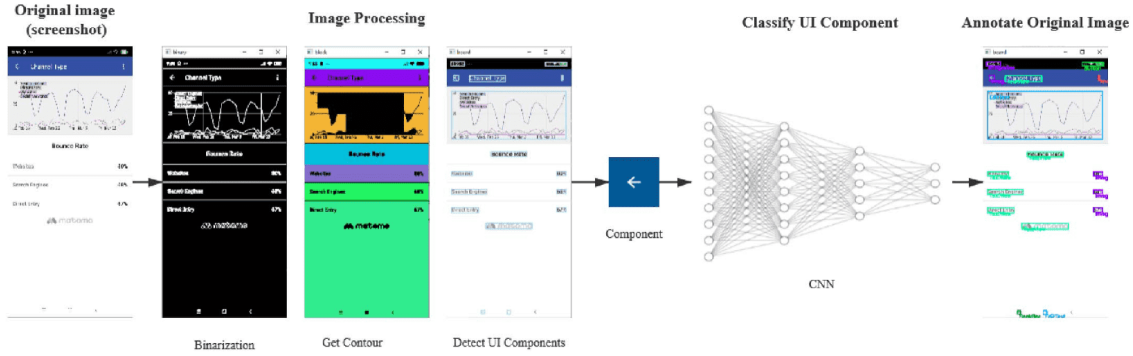



Figure 2.1: Approach to UI components classification. Taken from: [26]

## 2.2 *UIED*: A Hybrid Tool for GUI Element Detection

*UIED* is a toolkit which provides users with a GUI element detection platform. It was published in 2020. It is an image-based approach to GUI element detection. It offers a web interface where users can upload their GUIs, and the system detects and identifies elements in it [32].

It uses *EAST* [2] OCR to detect text in the screenshot image. Next, it uses the *Flood-Fill* [3] and *Sklansky* [4] algorithms to obtain potential layout blocks. Then, edges are detected in the image, which is then converted to a binary form. The connected component labelling algorithm then detects GUI elements. Proposed elements are then classified using a *ResNet-50*, trained on 90,000 GUI elements, divided into 15 classes. The result of the process can be seen in figure 2.2. *UIED* achieves an F1 score of 52% on 5,000 UI images from the *Rico* dataset [32].

This method only classifies elements into 15 classes. Also, the element detection algorithm would not work on noisy images, as the detection relies on clean input images. Moreover, GUIs with open design (not perfectly closed regions) would be marked as one large region.

## 2.3 Learning Design Semantics for Mobile Apps

In recent years, approaches to mine GUI hierarchies, designs and interaction data have been developed. However, this data does not expose what elements on the screen mean and how users use them to accomplish their goals. This solution proposes an automatic approach for generating such annotations for mobile GUIs. [17]

The authors created the lexical UI database in an iterative approach by examining component categories and merging components close in appearance and functionality. For example, *Input* components contain *EditText*, *SearchBoxView* and *TextView*. [17]

A lexical UX database was created by analysing *text buttons* and *icon* elements. The authors extracted 130,761 text buttons from the *Rico* 3.2 dataset. These buttons contained 20,386 unique strings. These strings were then filtered and clustered based on common substrings. Each cluster was then assigned a label. A text string can belong to multiple clusters. Icons were extracted from the *Rico* dataset by finding image components that are

[2] https://pyimagesearch.com/2018/08/20/opencv-text-detection-east-text-detector
[3] https://www.techiedelight.com/flood-fill-algorithm/
[4] http://cgm.cs.mcgill.ca/ athens/cs601/Sklansky2.html

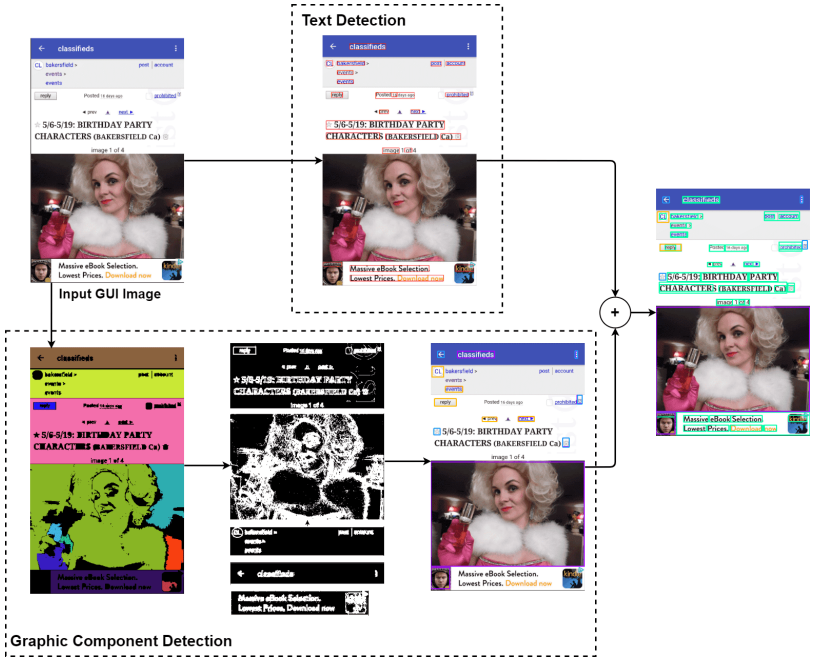



Figure 2.2: *UIED* approach to GUI elements detection and identification. Taken from: [32]

visible and clickable, their area is less than 5% of the total area, and their aspect ratio is no less than 0.75. 49,136 icons were gathered using the mentioned heuristic and were manually divided into 135 icon classes. As icons can have multiple meanings depending on their context, sets of synonyms were created for each icon class. [17]

Non-icon UI components can be classified using the lexical database. Semantic labels for elements in the *Rico* dataset were classified. 1,384,653 elements out of 1,776,412 (77.95%) visible elements were labelled. Twenty-five annotated components per UI on average. [17]

Icons cannot be classified by the code-based heuristic. Therefore CNN was trained to classify images into 99 distinct classes. Previously annotated icons divided into 135 classes were used. Some of the classes were merged to increase support and reduce ambiguity, resulting in 99 classes. Icons were converted to grayscale and whitened before training. GMM model with 128 components was used to distinguish between icons and images that significantly differ from icons in the dataset. As a result, 94% accuracy on the test set (10% of data) was achieved. [17]

By combining both approaches mentioned above, the authors created a system which was used to annotate the whole *Rico* dataset. Code-based patterns were used to detect UI components, and the icon classification was used to add semantic annotations to the dataset. [17] The result of this system can be seen in figure 2.3.

Also, the authors created a convolutional auto-encoder to enable finding similar UIs to the queried one. The result can be seen in figure 2.4.

The system uses a lexical database, which highly correlates with the system presented in this thesis. Also, it uses a CNN to classify images into very *fine-grained* classes. However, this solution relies on a view hierarchy and a screenshot as inputs. As a downside, it cannot analyse applications without access to its source code. Also, it only supports Android applications.

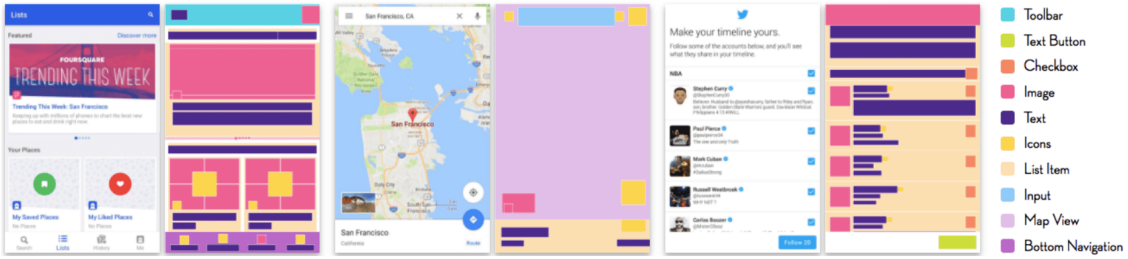



Figure 2.3: Result of the analysis of UIs in the *Rico* dataset. Taken from: [17]

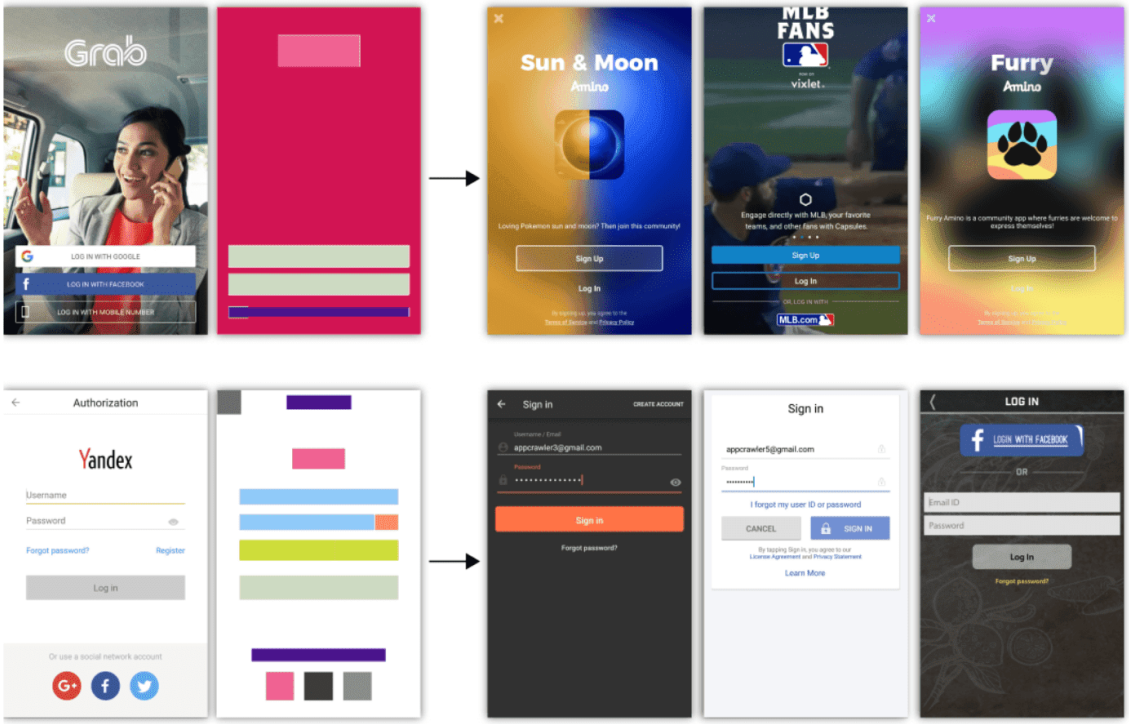



Figure 2.4: Result of UI similarity query. Taken from: [17]

## 2.4 Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels

*Screen Recognition* is a system that, from a single GUI image, creates metadata which describe UI components. Then, the system hands the metadata over to *iOS VoiceOver*, increasing accessibility. The system is designed to run on mobile devices; therefore, it is memory efficient and fast. It was published in 2021. It uses deep learning techniques trained on an *iPhone* application dataset. [34] The approach of the creation of application metadata can be seen in figure 2.5.

The system authors created a dataset of GUIs from 4,239 iPhone applications. First, the top 200 most popular applications of each of the 23 categories (excluding games) were manually downloaded. Then, ten people traversed through GUIs in each application to collect screenshots of visited UIs and their metadata (tree structure, properties of UI elements). However, the collected data was incomplete, so manual annotation had to be done. Forty people annotated all UI elements in the collected screenshots using bounding boxes

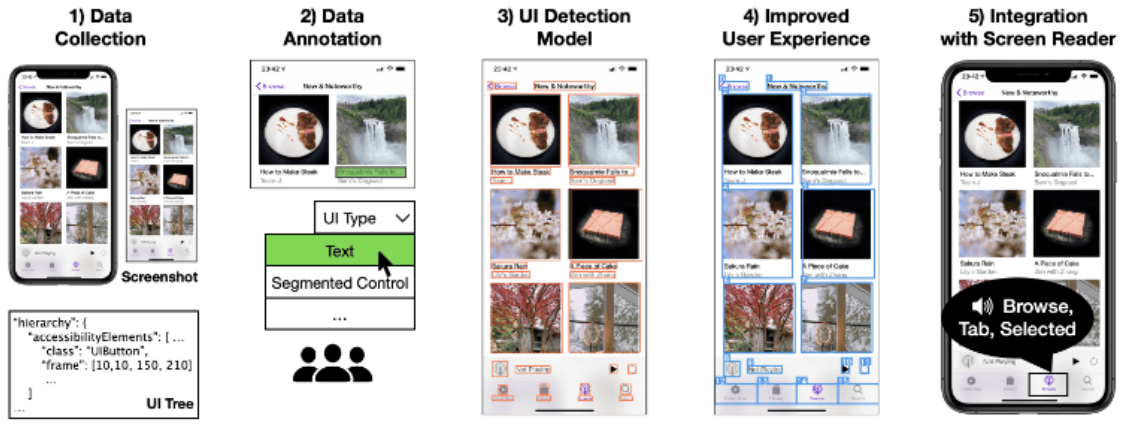



Figure 2.5: Screen Recognition's approach to creating application metadata for screen readers. Taken from: [34]

and identifiers. The dataset was divided into twelve classes: *Checkbox, Container, Dialog, Icon, Picture, Page Control, Segmented Control, Slider, Text, Text Field, Tab Bar Item*, and *Toggle*. These classes were selected based on an examination of 500 sample screens which identified which UI elements are essential for accessibility tools. In total, 77,637 UI screens were annotated. Imbalances in the dataset were solved by using data augmentations. [34]

UI detection model was trained to extract elements from GUI and predict their class. Multiple models like Faster R-CNN and *TuriCreate* model were tried. However, their inference time and memory consumption were not suitable for use in a mobile environment, so SSD model with *MobileNetV1* backbone was selected. The model was trained using four Tesla V100 GPUs for 20 hours (557,000 iterations). Model output is then filtered using *Non-Max Suppression* and setting different confidence thresholds for each class. The model's weighted mean average precision (IOU $> 0.5$) is 87.5%. [34]

The solution then post-processes the inference result to remove extra detections, and finds missing elements using built-in OCR service and *VoiceOver* image descriptors. Next, icon click-ability is predicted using a *Gradient Boosted Regression Trees* model. Then, UI elements need to be grouped because the detector outputs a bounding box for each element separately. The elements are grouped based on hard-coded heuristics empirically acquired from 300 randomly picked samples. Figure 2.6 shows the raw output of the detector (orange) and the grouped, post-processed output (blue). Lastly, a navigation order is inferred using the *OCR XY-cut page segmentation* algorithm, which sorts UI elements in human-readable order. [34]

An evaluation was conducted by 9 participants who used 22 different applications. They were asked to use both *VoiceOver* and *Screen Recognition* and rate the usability of selected applications on a scale from 1 to 5 (higher is better). *VoiceOver* achieved 2.08, while *Screen Recognition* achieved 3.73. [34]

The goal of this system is different to the goal of this thesis. However, there are some overlapping sub-goals like UI element classification, dataset creation, and OCR result post-processing.


Figure 2.6: Grouping and ordering of detected GUI elements. Taken from: [34]

## 2.5 Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps

Modern applications heavily rely on the attractiveness of GUI and UX to attract users and offer them visually attractive UIs. Services like the Google Play store contain functionally identical applications that differ mostly in GUIs and UX designs. Therefore, GUI drafting and prototyping are essential for the creation of GUI-based applications. The prototyping is usually done by graphical designers who use editing software like *Photoshop* to create drafts of the GUI. The design language is usually very similar across multiple facets like websites, mobile applications, and marketing materials. For all these facets, mock-ups are created before committing to the development process. These mock-ups then have to be faithfully translated into code, so that users can enjoy them as they were intended. [18]

Different teams usually carry out the prototyping and development process, which is challenging, time-consuming, and prone to errors. Also, designers often practice an iterative development process, where feedback is collected, and changes are integrated into the mock-ups. Using prototypes is preferred, as more detailed feedback can be collected; however, updating such prototypes by hand proves to be very difficult. Moreover, past work on detecting GUI design violations shows that because of the iterative design practices and development processes. Many startups and innovative companies are creating software prototype applications and could hugely benefit from automated prototyping. [18]

Modern IDEs offer visual GUI creators, allowing designers to create prototypes. However, creating complex and high-fidelity GUIs is difficult, and users are prone to create buggy prototypes. Other GUI design solutions offer collaborative capabilities and interactive previewing on target devices. However, no public solution that offers automatic translation of a mock-up into correct native code for a target platform exists. However, automating such a process is a challenging task. The solution must create valid user interface code from hand-drawn or digitally created design sketches. It has to recognise discrete

objects from mock-ups, categorise them into correct GUI components, and arrange them into proper hierarchical structures. [18]

The design and functionality of GUIs differ dramatically. Hand-made encoding algorithms and heuristics would not be able to perform automatic design prototyping. Therefore, a data-driven machine learning approach, trained on existing GUI applications using application mining software is proposed. The proposed system is divided into three major parts: *detection*, *classification*, and *assembly*. *Detection* is responsible for finding bounding boxes of atomic GUI components in UI mock-ups. Elements are detected using digital mock-up metadata or computer vision techniques (*Canny*, dilation, contours) from image input. Then, each element is classified into its class (button, switch, checkbox, ...). This step is implemented using a custom CNN architecture trained using the dataset mentioned in section 3.2. Lastly, using an iterative K-nearest-neighbours and computer vision techniques, GUI-hierarchies are constructed and translated into target code. [18] Figure 2.7 shows the *ReDraw* approach for automated GUI prototyping.

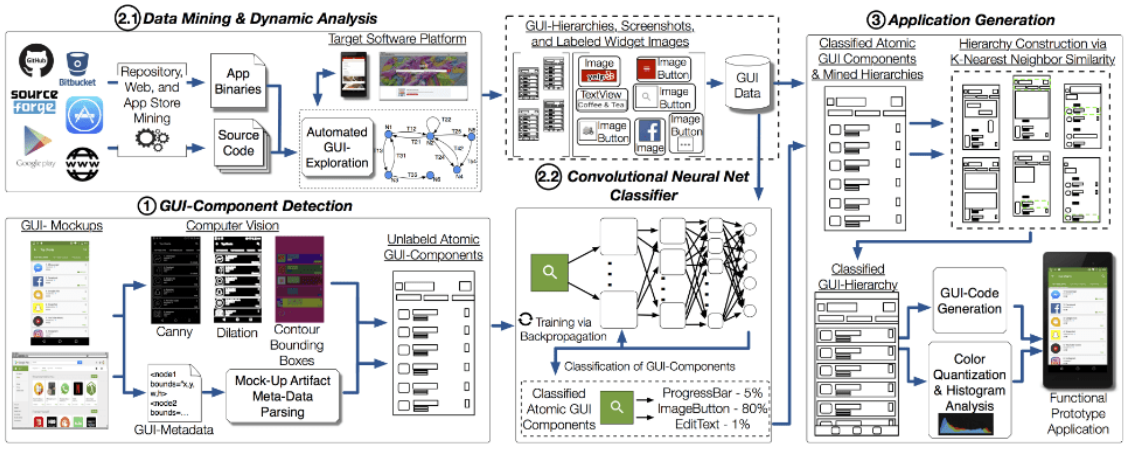



Figure 2.7: The approach of *ReDraw* to automatic GUI mock-up generation. Taken from: [18]

CNN GUI component classifier achieves a top-1 average precision of 91%. Generated applications are visually very similar to their mock-ups, and the code structure is similar to real applications. *ReDraw* outperforms other related solutions for mobile application prototyping like *REMAUI* [19] and *pix2code* [2]. Figure 2.8 shows a comparison between the actual application and the ReDraw's automatically generated application. [18]

The system discussed in this paper shares some similarities with the task of this thesis. GUI element detection and classification of those elements are also necessary for the goal of this thesis. However, the authors of the system created a dataset of 15 most common UI elements, while the system of this thesis requires more *fine* class separation. Also, the GUI hierarchy analysis in this paper relies on a KNN technique constructed by having access to thousands of Android application GUI layouts.

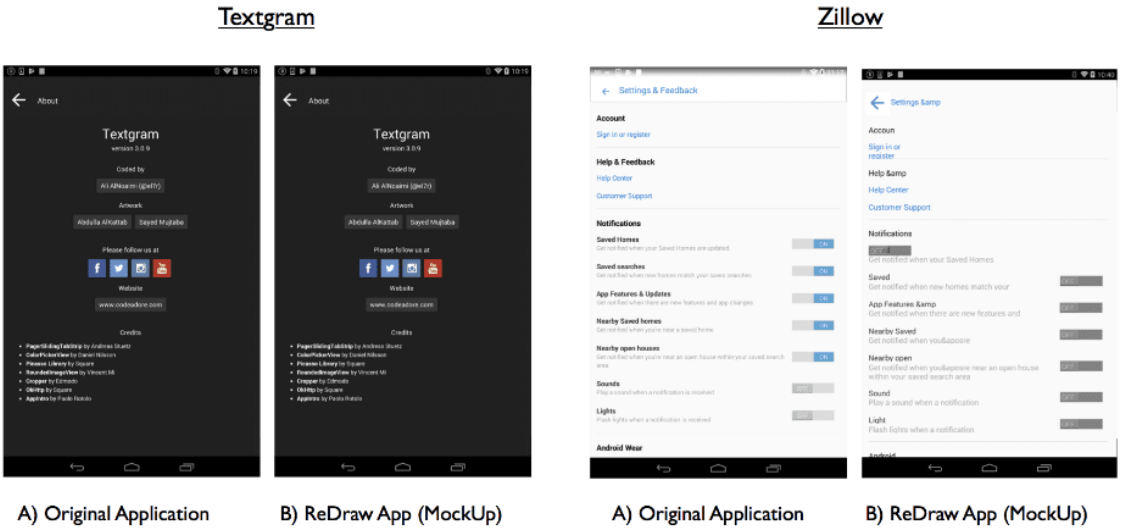



Figure 2.8: Result of *ReDraw* compared to original applications. [18]

# Chapter 3

# Current State of UI Elements Identification

This chapter discusses the current state of neural networks for classification tasks. Also, available image datasets for image classification tasks are mentioned.

In recent years, deep learning methods have become a widely popular solution for, among other, machine learning tasks and image processing tasks. Deep neural networks hugely benefit from the number of available computing resources and a large amount of available data grouped into massive datasets. Deep neural networks surpass traditional machine learning methods in almost every aspect, be it precision, the ability to generalise or the possibility of modifying the network to perform better on a specific task. The traditional machine learning methods are still relevant to this date for specific scenarios, where the algorithm's speed might be more critical than its precision, or when the task is straightforward. These methods are also implemented in almost every machine learning framework and are significantly optimised. [1] However, only deep neural networks relevant for this image classification task are mentioned in this chapter.

## 3.1 UI Elements Identification Methods Overview

### AlexNet

*AlexNet* (61M parameters) is a large scale CNN, published in 2010, with a respectable classification accuracy on the *ImageNet* dataset. *AlexNet* architecture consists of blocks of convolutional layers followed by pooling layers and fully connected layers at the end of the network. Its architecture is similar to the *LeNet* [13] network, but it contains more layers in total. The first block of the network contains 11x11 convolution filters, followed by 5x5 filters. The rest of the filters have a size of 3x3. Two fully connected layers have 4,096 neurons, and the last FC layer has 1,000 neurons. Figure 3.1 shows the architecture of the network. The network had to be trained using multiple accelerators because it did not fit on GPUs of that time. It used dropout and data augmentation to avoid over-fitting. It achieved a top-1 error rate of 37.5% on the *ILSVRC-2010* challenge and won the first place. [12, 22]


Figure 3.1: *AlexNet* architecture. Taken from: [16]

### VGG

*VGG* is a deep CNN architecture that uses small 3x3 convolution filters. Compared to neural networks available at the time of publication of this network, it uses more layers (11-19) of 3x3 convolution filters. [24]

Figure 3.2 shows the architecture of *VGG16* architecture. It is composed of:

- 13 convolutional layers - 5 blocks (64 filters per layer in the first block, double until 512 filters per layer)
- 5 max-pooling layers (2x2, stride 2) - after each block
- 3 fully connected layers - at the end 2x 4096 neurons, 1x 1000 neurons (*ImageNet* dataset number of classes)

Overall, the network has 16 trainable layers, which translates to 138,357,544 parameters. It achieved state-of-the-art detection accuracy on the *ImageNet* 3.2 dataset at the *ILSVRC-2014* competition. It reached 23.7% top-1 (6.8% top-5) validation error compared to the second place which reached 27.9% (9.1%). [24]

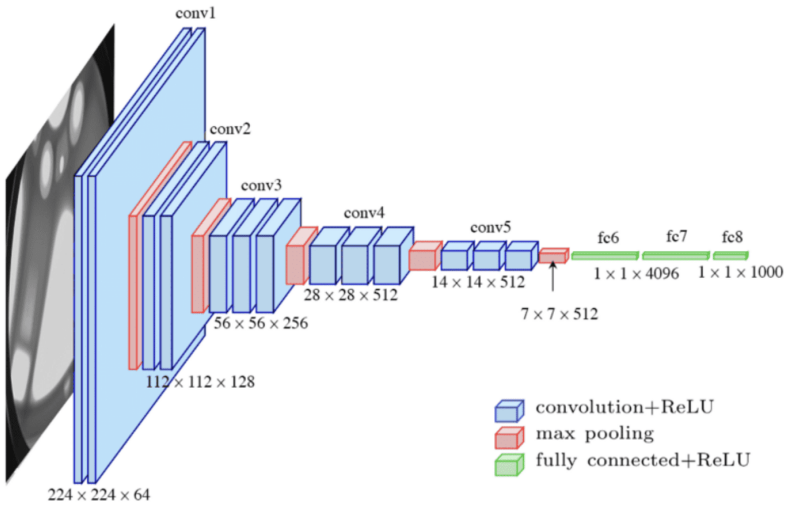


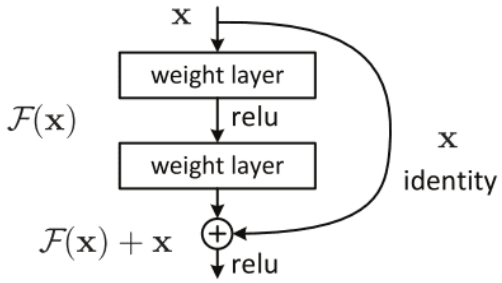

Figure 3.2: *VGG16* architecture. Taken from: www.researchgate.net

*VGG* was, at the time, a revolutionary approach to a deep understanding of image data by replacing larger convolutional filters with 3x3 filters and stacking them in multiple blocks. However, the network is very computationally demanding and is nowadays surpassed in every regard. Therefore, it is not utilised in this thesis.

### ResNet

*ResNet* was published in 2015 as a solution to the problem of vanishing gradients in deep neural networks. The more layers the networks have, the more evident the problem is. As the gradient is back-propagated to upper layers, it gets smaller and eventually reaches zero. Because of that, the network's performance might be lower compared to shallower networks. [7]

*ResNet* solves this issue by introducing residual blocks, as shown in figure 3.3a. Residual blocks contain *shortcut connections* which skip one or more layers. These layers perform *identity* mapping, and their outputs are added to the outputs of stacked layers. They do not add any parameters or computational complexity, and networks using this block can be trained using conventional optimisers. [7] An evaluation of *ResNet* variants and the *VGG-16* network on the *ImageNet* 3.2 dataset can be seen in figure 3.3b. *ResNet-34*, while being much less computationally intensive, achieves better top-1 accuracy then *VGG-16* by 3%.




(a) *ResNet* residual block.

| model | top-1 err. | top-5 err. |
|---|---|---|
| VGG-16 [41] | 28.07 | 9.33 |
| GoogLeNet [44] | - | 9.15 |
| PReLU-net [13] | 24.27 | 7.38 |
| plain-34 | 28.54 | 10.02 |
| ResNet-34 A | 25.03 | 7.76 |
| ResNet-34 B | 24.52 | 7.46 |
| ResNet-34 C | 24.19 | 7.40 |
| ResNet-50 | 22.85 | 6.71 |
| ResNet-101 | 21.75 | 6.05 |
| ResNet-152 | **21.43** | **5.71** |

(b) *ResNet* performance evaluation on*ImageNet* 3.2.

Figure 3.3: *ResNet* residual block and performance evaluation. Taken from: [7]

*ResNet* is a revolutionary approach to deep networks but is outperformed by less computationally intensive architectures nowadays. Therefore, it is not used in this thesis.

### Inception

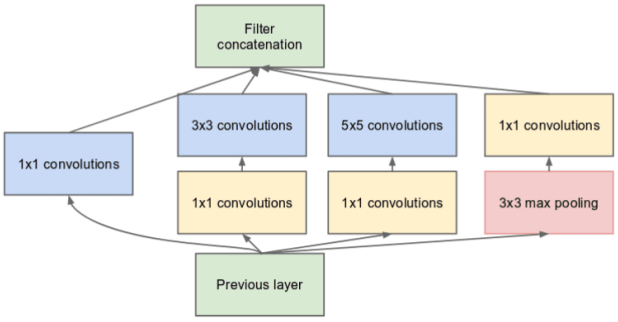
*Inception* architecture, published by Google engineers in 2014, is a solution to the downsides of stacking convolution layers on top of each other. Bigger models are prone to over-fitting, more computationally expensive and have problems with vanishing gradients. Also, in image classification, the right convolution filter size depends on the distribution of the information; a small filter is suitable for local information, while a large filter is suitable for global information. [28, 23]

*Inception v1* (5M parameters) solves those issues by having multiple convolution filters of different sizes: 1x1, 3x3, and 5x5. Also, max-pooling is performed. Outputs of these sub-blocks are concatenated into a single output. To reduce the computational demands of the *Inception* block, 1x1 convolution is performed before 3x3 and 5x5 convolutions, reducing the number of input dimensions. The *Inception v1* block can be seen in figure 3.4a. *Inception v1*, also called GoogLeNet, has nine such blocks stacked on top of each other. Also, two auxiliary classifiers, which propagate auxiliary loss, are added to the network

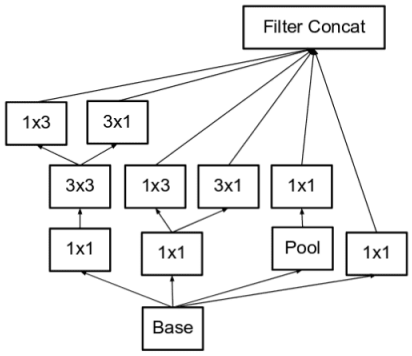
to prevent the vanishing of gradients during training. *Inception v1* won the classification *ILSVRC-2014* challenge. [28, 23]

*Inception v2* and *Inception v3* (24M parameter) architectures replaced the 5x5 convolution filters with two 3x3 filters, which resulted in a computational speed increase. Also, convolution filters of size $n$x$n$ were replaced by a combination of 1x$n$ and $n$x1 filters. These filters further decreased the computational intensity. Moreover, *RMSProp* optimiser, factorised 7x7 convolutions, batch normalisation in auxiliary classifiers, and label smoothing were added to the architecture. The factorised block can be seen in figure 3.4b. *Inception v3* architecture, with all improvements, achieves 21.2% top-1 error, while *Inception v1* achieves 29% on the *ImageNet*3.2. [28, 29, 23]

*Inception v4* (43M parameters) further improves the architecture by updating the *stem* of the network and *reduction blocks*. *Inception-ResNet v2* (56M parameters) introduces residual connections 3.1, enabling the network to converge faster, reducing the training time. [27, 23] Accuracy comparison between the mentioned architectures can be seen in figure 3.4c. *Inception-ResNet-v2* achieves the same accuracy as *Inception-v4*, but it converges faster. The same applies for *Inception-ResNet-v1* and *Inception-v3*.




(b) Inception module with dimension reductions




(a) *Inception v1* block (b) *Inception v3* block

| Network | Crops | Top-1 Error | Top-5 Error |
|---|---|---|---|
| ResNet-151 [5] | 10 | 21.4% | 5.7% |
| Inception-v3 [15] | 12 | 19.8% | 4.6% |
| Inception-ResNet-v1 | 12 | 19.8% | 4.6% |
| Inception-v4 | 12 | 18.7% | 4.2% |
| Inception-ResNet-v2 | 12 | 18.7% | 4.1% |

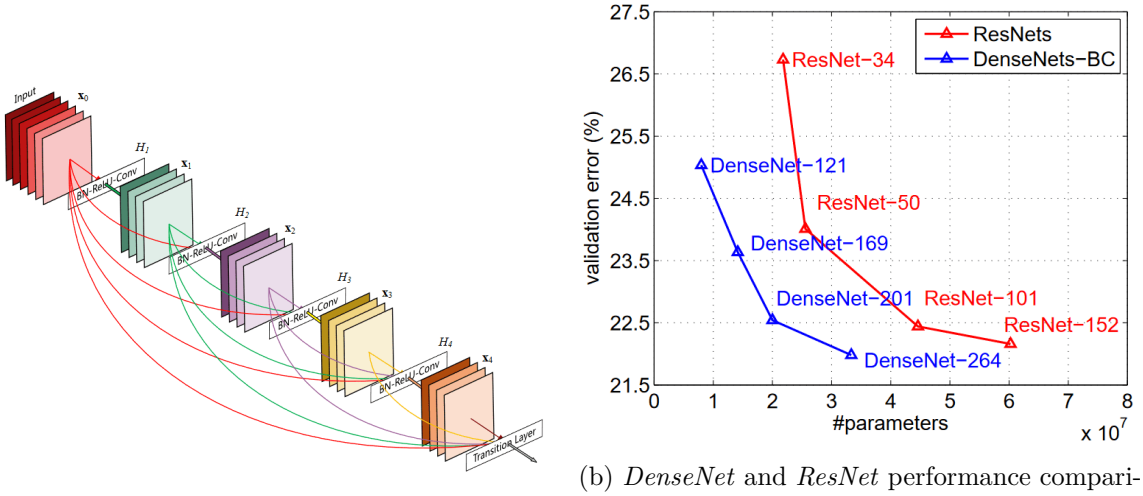
(c) *Inception* versions evaluation on *ImageNet* dataset.

Figure 3.4: *Inception* block and evaluation figures. Taken from: [28, 29, 27]

### DenseNet

*DenseNet* is a CNN, published in 2018, which is inspired by the solutions of *ResNet* 3.1 and *Highway Networks* [25] to the vanishing gradients problem. *DenseNet* approaches the shortcut creation from early layers to later layers by connecting all layers in a dense block (with matching feature-map sizes) directly with each other. The connection between convolution layers is illustrated in figure 3.5a. In the figure, coloured blocks consist of batch normalisation, *ReLU* and 3x3 convolution filters, while the white transition lay-

ers consist of batch normalisation, 1x1 convolution filter and average pooling. In contrast to ResNet, *DenseNet* does not combine features using summation before passing to a layer; instead, features are concatenated. Also, *DenseNet* requires fewer parameters to achieve similar performance as ResNet. Moreover, all layers have direct access to the gradients from the loss function, which helps with the training of deep models. Figure 3.5b shows a comparison between *DenseNet* and *ResNet* on the *ImageNet* dataset. *DenseNet* achieves consistently better validation accuracy while having around half parameters. [8]




(a) *DenseNet* architecture

(b) *DenseNet* and *ResNet* performance comparison

Figure 3.5: *DenseNet* architecture and performance comparison with ResNet. Taken from: [8]

### EfficientNet

*EfficientNet* is a CNN architecture published in 2019 which uses compound coefficient to scale neural networks in width, depth and resolution simultaneously, achieving state-of-the-art performance and efficacy. [30]

The authors studied the impact of scaling one parameter at a time on the network's performance. While scaling in such a way improves the network's performance, it is not optimal because as the input image gets bigger, the network needs more layers to increase the receptive field and more channels to capture more *fine-grained* patterns on the bigger image [30]. Different types of neural network scaling methods can be seen in figure 3.6. Width scaling adds more feature maps at each layer, depth scaling adds more layers to the network, and resolution scaling increases the input image resolution. [30] A comparison between scaling across a single dimension can be seen in figure 3.7a. Scaling only one dimension increases computational complexity dramatically and does not lead to the same accuracy increase.

The *compound scaling method* balances scaling of width, depth, and resolution by scaling with a constant ratio. The ratio of each scaling dimension is determined by a grid search on the original tiny model while keeping the computational complexity increase in mind. The authors found that the best scaling parameters for the baseline network architecture were: depth by 1.2, width by 1.1 and resolution by 1.15. These parameters were fixed as constants, and the baseline network was scaled to obtain *EfficientNet-B1* to *EfficientNet-B7*. The *EfficientNet* network family performance can be seen in figure 3.7. It achieves

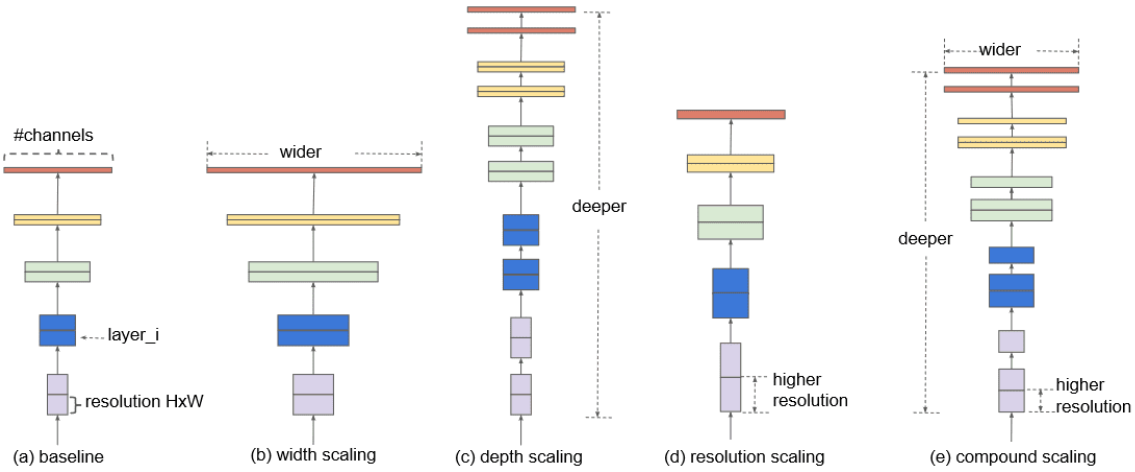



Figure 3.6: CNN scaling methods overview and comparison. Taken from: [30]

state-of-the-art performance on *ImageNet*, *CIFAR-100* and *Flowers* dataset. *EfficientNet* architecture outperforms all other networks, both in accuracy and number of parameters. This technique not only works on the baseline architecture that they used but also on other popular architectures. [30]

Also, using compound scaling, models tend to focus more on relevant regions in the image. An example of it can be seen in figure 3.8.

### SqueezeNet

*SqueezeNet* (0.4M parameters) is a CNN published in 2016, which has the same accuracy as AlexNet while being 50 times smaller. To achieve such performance, the authors employed three main strategies:

- Replace some 3x3 filters with 1x1 filters.
- Decrease the number of input channels to 3x3 filters.
- Downsample late in the network so that convolution layers have large activation maps.

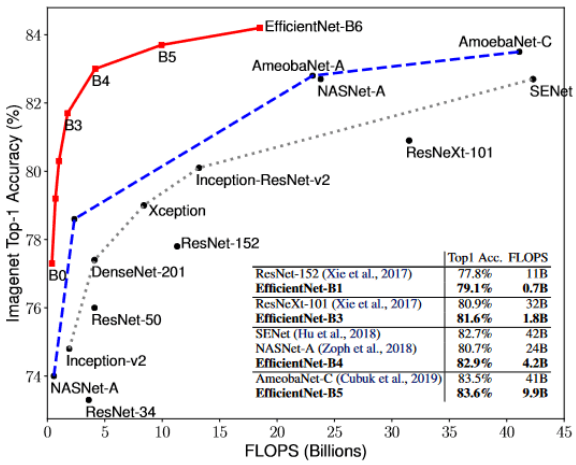
Strategies 1 and 2 aim to decrease the number of parameters of the network while not decreasing its accuracy. Strategy 3 aims to maximise the accuracy of the network. The authors also introduced a *Fire* module. The module consists of a *squeeze* layer (1x1 convolution filter) which feeds into an *expand* layer that contains both 1x1 and 3x3 convolution filters. [10] The *Fire* module can be seen in figure 3.9.

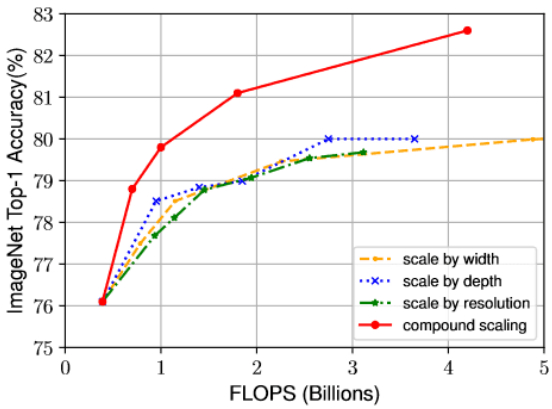
The *SqueezeNet* is built using a convolution layer, followed by eight *Fire* modules and ending with a convolution layer. Authors also experimented with using bypass connections, just like in *ResNet* 3.1. Using a simple bypass, the network achieves 60.4% top-1 accuracy on the *ImageNet* 3.2 dataset, which is 3.2% higher than AlexNet, while being 4.8MB large. [10]

### ShuffleNet

*ShuffleNet* is an efficient CNN architecture designed to run on mobile devices with computation power of around 100 MFLOPs. The authors use two operations: *group convolution*

| Model | Top-1 Acc. | Top-5 Acc. | #Params | Ratio-to-EfficientNet | #FLOPs | Ratio-to-EfficientNet |
|---|---|---|---|---|---|---|
| **EfficientNet-B0** | **77.1%** | **93.3%** | **5.3M** | **1x** | **0.39B** | **1x** |
| ResNet-50 (He et al., 2016) | 76.0% | 93.0% | 26M | 4.9x | 4.1B | 11x |
| DenseNet-169 (Huang et al., 2017) | 76.2% | 93.2% | 14M | 2.6x | 3.5B | 8.9x |
| **EfficientNet-B1** | **79.1%** | **94.4%** | **7.8M** | **1x** | **0.70B** | **1x** |
| ResNet-152 (He et al., 2016) | 77.8% | 93.8% | 60M | 7.6x | 11B | 16x |
| DenseNet-264 (Huang et al., 2017) | 77.9% | 93.9% | 34M | 4.3x | 6.0B | 8.6x |
| Inception-v3 (Szegedy et al., 2016) | 78.8% | 94.4% | 24M | 3.0x | 5.7B | 8.1x |
| Xception (Chollet, 2017) | 79.0% | 94.5% | 23M | 3.0x | 8.4B | 12x |
| **EfficientNet-B2** | **80.1%** | **94.9%** | **9.2M** | **1x** | **1.0B** | **1x** |
| Inception-v4 (Szegedy et al., 2017) | 80.0% | 95.0% | 48M | 5.2x | 13B | 13x |
| Inception-resnet-v2 (Szegedy et al., 2017) | 80.1% | 95.1% | 56M | 6.1x | 13B | 13x |
| **EfficientNet-B3** | **81.6%** | **95.7%** | **12M** | **1x** | **1.8B** | **1x** |
| ResNeXt-101 (Xie et al., 2017) | 80.9% | 95.6% | 84M | 7.0x | 32B | 18x |
| PolyNet (Zhang et al., 2017) | 81.3% | 95.8% | 92M | 7.7x | 35B | 19x |
| **EfficientNet-B4** | **82.9%** | **96.4%** | **19M** | **1x** | **4.2B** | **1x** |
| SENet (Hu et al., 2018) | 82.7% | 96.2% | 146M | 7.7x | 42B | 10x |
| NASNet-A (Zoph et al., 2018) | 82.7% | 96.2% | 89M | 4.7x | 24B | 5.7x |
| AmoebaNet-A (Real et al., 2019) | 82.8% | 96.1% | 87M | 4.6x | 23B | 5.5x |
| PNASNet (Liu et al., 2018) | 82.9% | 96.2% | 86M | 4.5x | 23B | 6.0x |
| **EfficientNet-B5** | **83.6%** | **96.7%** | **30M** | **1x** | **9.9B** | **1x** |
| AmoebaNet-C (Cubuk et al., 2019) | 83.5% | 96.5% | 155M | 5.2x | 41B | 4.1x |
| **EfficientNet-B6** | **84.0%** | **96.8%** | **43M** | **1x** | **19B** | **1x** |
| **EfficientNet-B7** | **84.3%** | **97.0%** | **66M** | **1x** | **37B** | **1x** |
| GPipe (Huang et al., 2018) | 84.3% | 97.0% | 557M | 8.4x | - | - |




(a) *EfficientNet* Scaling methods comparison.

Figure 3.7: *EfficientNet* performance comparisons. Taken from: [30]

and *channel shuffle*. The goal of both of these operations is to reduce computation complexity while maintaining accuracy. [33]

*Group convolution* was introduced in *Xception* and *ResNeXt* networks, but the authors claim that it is not fully utilised in those architectures. Instead, the authors propose to use *group convolutions* in conjunction with a channel shuffle, which enables groups to obtain input data from different groups. The *Shuffle* unit can be seen in figure 3.10a. Image (a) shows a residual block with depth-wise convolution, while in the image (b), 1x1 convolution is replaced by 1x1 *group convolutions*, and a channel shuffle is applied after the first 1x1 *group convolution*. The network can be scaled to the desired complexity. [33]

The accuracy and computational complexity of this model are compared in figure 3.10b. *ShuffleNet* outperforms *VGG-16*, *Inception v1*, *AlexNet*, and *SqueezeNet* while being less computationally intensive. [33]

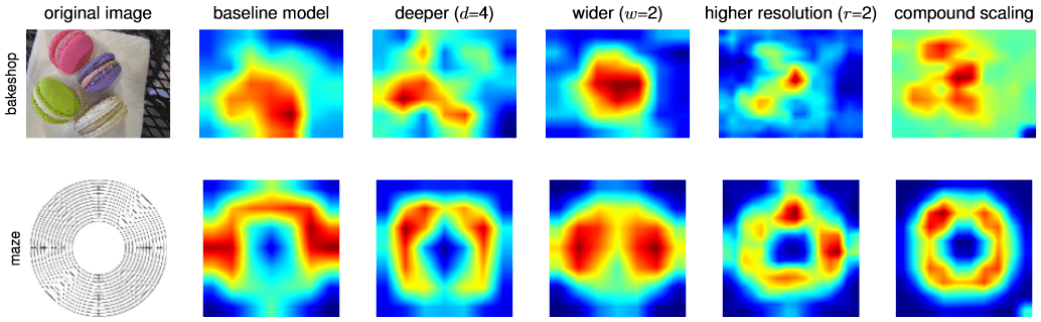



Figure 3.8: Class activation maps while using different scaling methods. Taken from: [30]

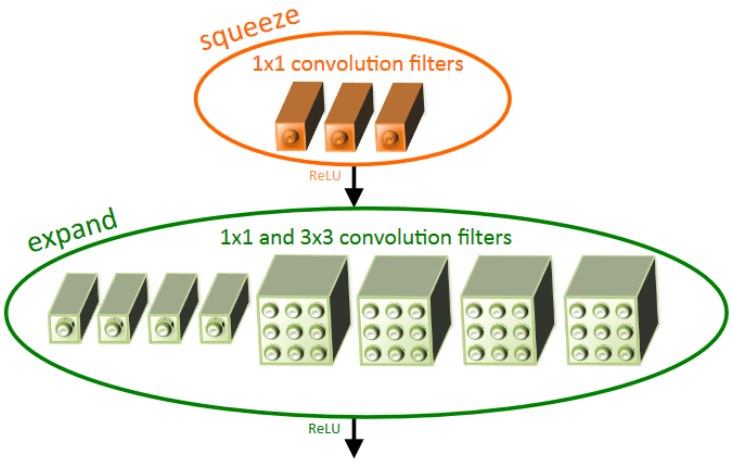



Figure 3.9: *SqueezeNet Fire* module. Taken from: [10]

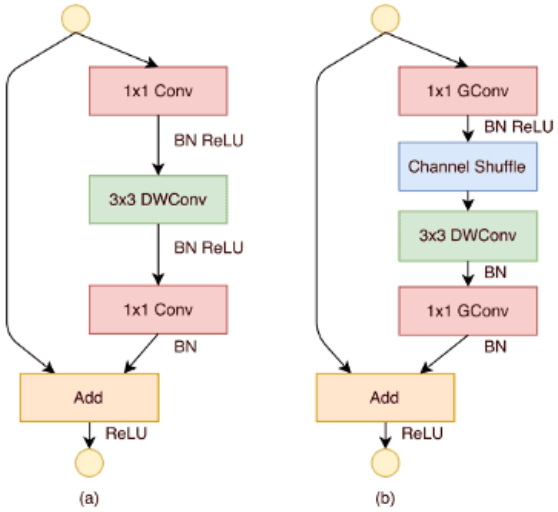
## 3.2 UI Elements Identification Datasets Overview

### ImageNet

*ImageNet* is an image dataset organised into a hierarchy, similarly to *WordNet*. It was created in 2009 to offer researchers an extensive image database to enable proper training of deep neural networks. [6]

Concepts in *WordNet*, which multiple words could describe, are a *synset*. *WordNet* contains more than 100,000 synsets, the majority being nouns. *ImageNet*, at the time of writing, contains 21,840 synsets, and the aim is to provide around 1,000 images to each one. Images of each synset are quality-controlled and hand-made. The goal of *ImageNet* creators is to offer tens of millions of labelled and sorted images to cover most of the *WordNet* concepts. At the time of writing, *ImageNet* contains 14,197,122 images. *ImageNet* project does not own copyrights for images used in the dataset; therefore, it only complies with an accurate list of web images for each synset. Researchers and educators can, under conditions, access get access to the whole dataset. [6]

The most popular subset of the dataset, used for the *ImageNet Object Localisation Challenge*, divides objects into 1,000 classes, 1,281,167 training images, 50,000 validation




(a) *ShuffleNet* block comparison

Table 6: Complexity comparison

| Model | Cls err. (%) | Complexity (MFLOPs) |
|---|---|---|
| VGG-16 [27] | 28.5 | 15300 |
| ShuffleNet 2× ($g = 3$) | 29.1 | **524** |
| PVANET [18] (*our impl.*) | 35.3 | 557 |
| ShuffleNet 1× ($g = 3$) | 34.1 | **140** |
| AlexNet [19] | 42.8 | 720 |
| SqueezeNet [13] | 42.5 | 833 |
| ShuffleNet 0.5× (arch2, $g = 8$) | 42.7 | **40** |

(b) *ShuffleNet* performance comparison

Figure 3.10: *ShuffleNet* architecture and performance comparison on *ImageNet* 3.2. Taken from: [33]

images and 100,000 test images. The size of this subset is 166GB and can be downloaded from *Kaggle* [1].

Figure 3.11 shows an example of two root-to-leaf branches of *ImageNet*. The top row is a mammal subtree, while the bottom row is a vehicle subtree. Figure 3.12 shows

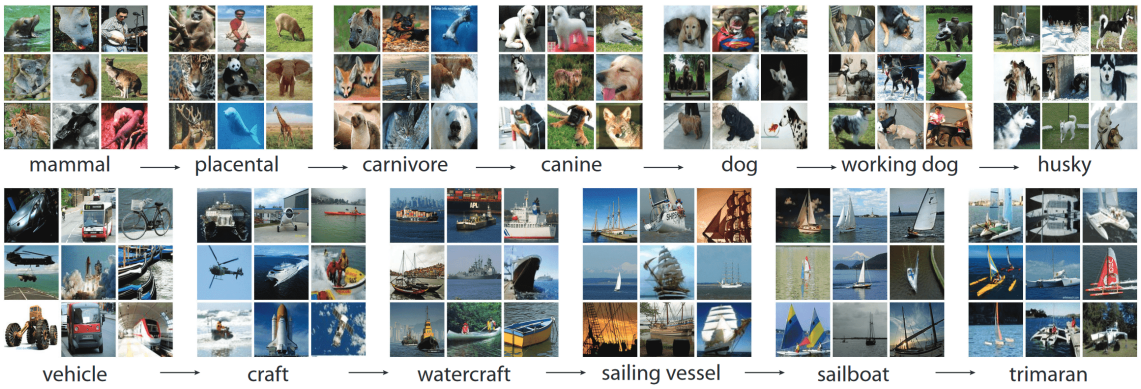



Figure 3.11: Example of two root synsets to leaf branches. Nine random images represent each synset. [6]

a graph-like representation of a root synset *feline*, all the way to leaf synsets like *kitty* and *cougar*. It also shows a root synset of a *stringed instrument*. The more green the word is, the more discriminable the synset is, and vice versa; the more red it is, the less discriminable the synset is.

This dataset hugely helped the advancements in the computer vision industry, and it revolutionised the approach to large scale dataset creations. However, it does not contain any synsets, which could benefit this thesis. Therefore, it is not used.

[1] https://www.kaggle.com/competitions/imagenet-object-localization-challenge/data

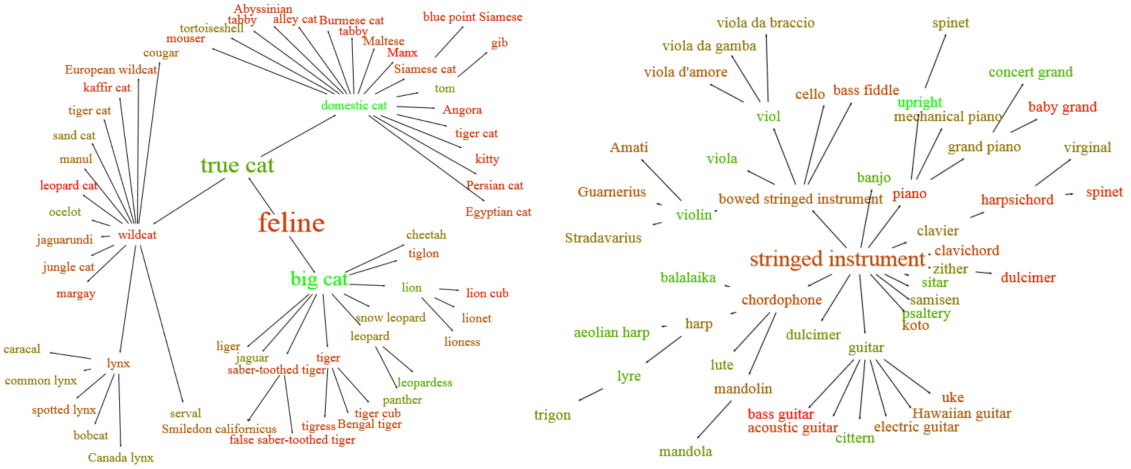



Figure 3.12: Example of a graph representation of a root synset to leaf branches. Taken from: [6]

### CIFAR

*CIFAR* datasets were created in 2009 by labelling a subset of the *Tiny Images* [2] dataset. Two variants of *CIFAR* exist. The smaller one is called *CIFAR-10*. It consists of 60,000 images with a resolution of 32x32. It only has ten classes, and each class contains 6,000 images, 5,000 being training images and 1,000 being test images. The classes are generic real-life objects, mainly vehicles and animals. *CIFAR-100* dataset contains the same number of images, but they are spread across 100 classes, each containing 600 images. 500 of them are used for training, and 100 are used for testing. Also, 20 superclasses group those *fine* classes into *coarser* ones. Each image contains information about its *coarse* and *fine* class. To give an example of how superclasses work: the *people* superclass contains *baby, boy, girl, man*, and *woman* classes. The size of both of these datasets is around 160MB. [11] Figure 3.13 shows classes and image examples of *CIFAR-10* and superclasses of the *CIFAR-100* dataset.

As this dataset contains no GUI or UI element classes, it is not helpful for the task of this thesis.

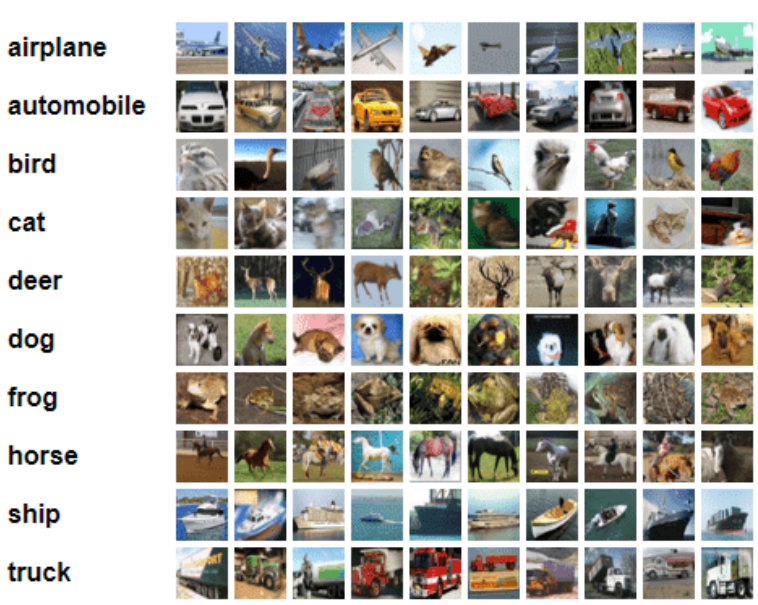
### MNIST

*MNIST* dataset was created in 1998 as a combination of two *NIST* databases. It contains 70,000 images, of which 60,000 are used for training and 10,000 are used for testing. Images are split into ten classes of handwritten digits. Images are black and white with a resolution of 28x28 and are centred by computing the centre of the pixel's mass. The dataset only has around 15MB. [14]

A variant of the *MNIST* dataset is *EMNIST*. It contains both handwritten numbers and handwritten letters. It was created in 2017 from *NIST Special Database 19*. The data has the same structure as the *MNIST* dataset. Six subvariants of the dataset exist:

- *ByClass* - 814,255 characters, 62 unbalanced classes
- *Balanced* - 131,600 characters, 47 balanced classes

[2] https://paperswithcode.com/dataset/tiny-images




(a) *CIFAR-10* dataset classes and images.

**Superclass**
aquatic mammals
fish
flowers
food containers
fruit and vegetables
household electrical devices
household furniture
insects
large carnivores
large man-made outdoor things
large natural outdoor scenes
large omnivores and herbivores
medium-sized mammals
non-insect invertebrates
people
reptiles
small mammals
trees
vehicles 1
vehicles 2

(b) *CIFAR-100* superclasses

Figure 3.13: *CIFAR* dataset overview. Taken from: [11]

- *Digits* - 280,000 numbers, 10 balanced classes
- *MNIST* - 70,000 numbers, 10 balanced classes
- *Letters* - 145,600 letters, 26 balanced classes
- *ByMerge* - 814,255 characters, 47 unbalanced classes

[3]

Other variants like *KMNIST*, *QMNIST*, *3D MNIST* and *BINARIZED MNIST* exist. However, none of the variants is relevant to the task of this thesis.

### ERICA

*ERICA* was published in 2016. It is an interaction mining system that captures both static (UI layout, visual details) and dynamic (user flows, motion details) components of an app's design. It allows mining of this information in a scalable way, without modifying source code or configurations of existing Android applications. *ERICA* takes a human-computer approach to mining, using people to interact with and understand UI designs, and machines to capture the states of those UIs. *ERICA* provides a web interface that controls Android applications. Users interact with that interface, and as they navigate through the applications, *ERICA* detects UI changes and records Android view hierarchies, user events and screenshots. Then it combines them into a unified representation called *interaction trace*. *ERICA* was used to gather *interaction traces* from more than one thousand popular *Google Play Store* applications. They contain more than 18,000 unique UI screens, 50,000 user interactions, 500,000 interactive elements. [5]

*ERICA* depends on users to interact with applications instead of automated solutions for various reasons. First, human interactions with applications are real-world data that an automated crawler cannot provide because it would create unrealistic *interaction traces*.

Also, applications might need user inputs; creating a lot of valid user data across multiple applications is demanding. Furthermore, the UI state has to be captured after it is fully loaded, which is difficult to do automatically, as data can be injected into the already loaded UI asynchronously. Whereas, users usually interact with UIs which are visibly loaded, including the data. [5]

Figure 3.14 shows the creation of *interaction trace* while a user interacts with an online shopping application. Six screenshots and view hierarchies are captured in that example. Also, a *search flow* is highlighted within that interaction trace. [5]

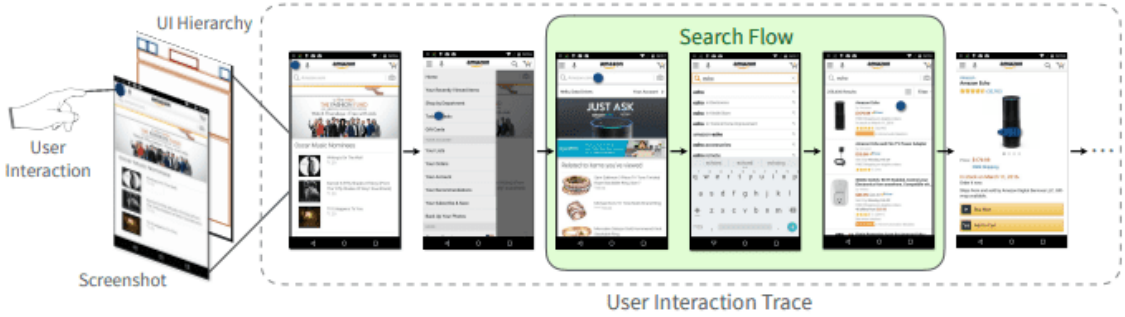



Figure 3.14: User interaction trace in an online shopping application. Taken from: [5]

Although this thesis's task is different from what this dataset aims to provide, it could be used to provide some data it contains. More specifically, if it is possible to extract icon images from it and automatically group them into different classes, it could be very helpful.

### Rico

*Rico* was published in 2017. It is a repository of mobile app designs. It supports five classes of data-driven applications. *Rico* was built by combining crowdsourcing and automation to scalably mine interaction data and design data from Android applications at run-time. The dataset contains data from more than 9,772 Android applications. The mining infrastructure does not need access to the source code, nor does it need to change it in any way. The visual, textual, structural and interactive design properties of more than 72,219 unique UI screens were collected from those applications. When *Rico* users interact with Android applications using the web interface, the system records *user interaction traces*. The traces include screenshots of UIs and interactions performed with them. The size of the dataset is 6GB. [4]

*Rico* is four times larger than the *ERICA* 3.2 dataset and is a superset of *ERICA*'s design information. While *ERICA* contains a collection of UIs encapsulated into *user interaction traces*, *Rico* additionally contains a list of unique UIs gathered across the whole application. This data is helpful for tasks that do not require UIs to be connected together as a sequence. *Rico* can be used for multiple data-driven applications such as Design Search, UI Layout Generation, UI Code Generation, User Interaction Modeling and User Perception Prediction. [4]

As mentioned above, *Rico* uses a human-computer approach to data mining. Humans rely on empirical application knowledge and contextual information to interact with a wide variety of applications. They can get around challenging scenarios; however, they tend to stick to common use cases, therefore achieving low application coverage. Automated agents can crawl through all UIs and edge-cases present in applications. However, agents can

get stuck on screens requiring complex sequences of actions or valid data inputs. By combining these two approaches, *Rico* can achieve better coverage. Humans are unlocking application states locked behind complex UIs, and agents then extensively crawl the whole application. Figure 3.15 shows an example of a crowd worker interacting with an application requiring user input; in this case, a verification code is sent to that phone. The worker can view SMS and e-mail messages that the phone receives to complete the verification process. [4]

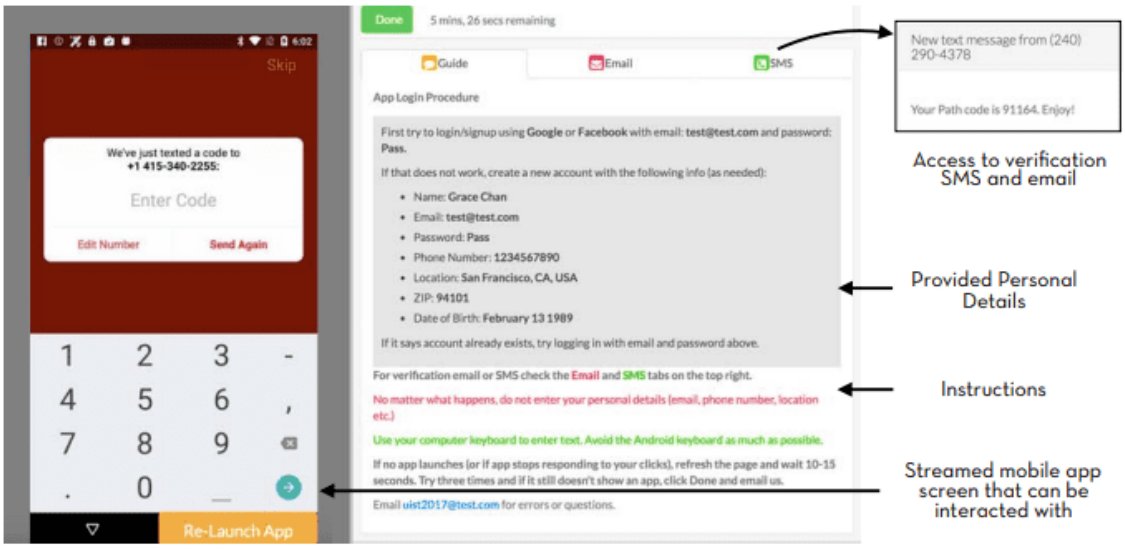



Figure 3.15: Crowd worker's view when interacting with an application requiring user input. Taken from: [4]

Compared to *ERICA* 3.2, *Rico* supports large-scale crowdsourcing over the internet. Crowd workers can remotely authorise in the system, the application is loaded to a phone in the mobile device farm, and the screen of the phone is streamed to the worker's browser. When workers interact with the application through the web interface, these commands are sent to the phone, performing the action. When crowd workers complete a complex action like data input, their actions are stored. Automated crawlers use those stored input data and replicate crowd workers' actions to inspect the whole application automatically. [4] Over time, the dataset will become outdated as new applications are constantly produced, designs are changing, and ways of interacting with applications might also change. User input is still necessary to get around complex UIs, so workers must keep interacting with new applications to keep the dataset updated. [4]

Although this thesis's task is different from what this dataset aims to provide, it could be used to provide some data it contains. More specifically, it could be beneficial if it is possible to extract icon images from it and automatically group them into different classes.

### Enrico

*Enrico* (Enhanced *Rico*) was created in 2020. It is a curated subset of *Rico* 3.2, containing 1,460 UIs and 20 design topics [15].

*Rico* 3.2 is the largest public dataset of mobile app designs. However, as *Rico* was created semi-automatically, it is very noisy. Authors of the *Enrico* dataset claim that only 10% of the UIs in the *Rico* dataset can be deemed high-quality design examples. The most

commonly identified problems were: a mismatch between app screenshot and wireframe, a mismatch between view hierarchy and wireframe, no wireframe available or empty image, and considerable overlaps among wireframe elements. As a result, around 90% of UIs were ruled out to create a smaller but high-quality dataset [15].

First, 10,000 UIs from the *Rico* dataset were randomly sampled and were evaluated, whether they were a good or a bad design. Eventually, 1,460 UIs were approved, and 20 UI design topics were found in those UIs. Then, these UIs were assigned to design topics, creating a smaller but higher quality version of *Rico* [15].

Although this thesis's task is different from what this dataset aims to deliver, it could be used to provide some data it contains. More specifically, it could be beneficial if it is possible to extract icon images from it and automatically group them into different classes. However, the *Rico* 3.2 dataset is better suited for this, as it is larger than *Enrico* and the increase in data quality of this dataset is not relevant for this thesis.

### ReDraw

*Redraw* was published in 2018 as a dataset of GUI components of Android applications. It consists of 15 categories, 1,382 UI pictures and 191,300 annotated GUI components. [18] The dataset is used to train CNNs that automatically detect GUI elements in mock-up images 2.5.

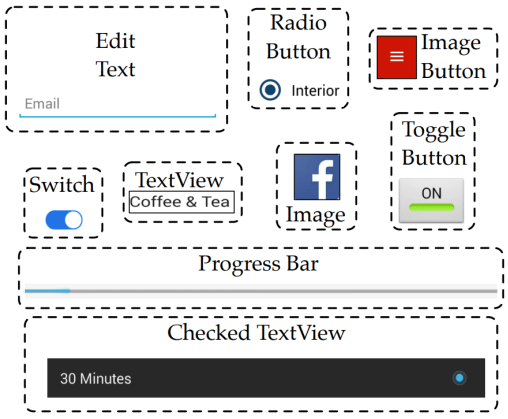
All categories of *Google Play* [3], except games, were extracted, resulting in 39 categories. Then, 240 top applications from each category were downloaded, which resulted in 8,878 unique applications. Each application was then explored, and GUI-related information from each screen was stored. Each application was navigated using a *Depth-First-Search* [4] manner, exercising tappable components. GUI-related information that describes the hierarchy, element type and coordinates, other properties displayed on the screen and its screenshot were saved for each unique screen. Only fifty actions were allowed per application to finish the exploration in a feasible time. 19,786 unique app screens containing 431,747 GUI components were gathered. Then, applications that contained web technologies and *Unity* [5] were removed, as they could not be analysed appropriately. Also, Screens in a landscape mode, screens containing only *Layout* components and screens containing *WebViews* were removed. Lastly, components with less than 200 instances were removed, and cases when bounding boxes were incorrect or cases when components were made up of a single colour were solved. After this filtering, 191,300 labelled GUI components from 6,538 applications remained. These components were split across 15 classes. However, these classes were imbalanced, so data augmentations were used to balance the dataset. Figure 3.16 shows examples and class distribution of the GUI components dataset. [18]

This dataset serves a very similar purpose as is needed for this thesis: GUI components classification. However, the data is split across 15 generic components, while the system implemented in this thesis needs to be able to classify UI element pictograms into much *finer* groups.

### Annotation Tools Overview

Using an appropriate annotation tool is very important to an effective annotation of datasets. Therefore, a short research on available free annotation tools was conducted. The table

[3] https://play.google.com/store
[4] https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph
[5] https://unity.com/




(a) A subset of the *ReDraw* training dataset.

| GUI-C Type | Total # (C) | Tr (O) | Tr (O+S) | Valid | Test |
|---|---|---|---|---|---|
| TextView | 99,200 | 74,087 | 74,087 | 15,236 | 9,877 |
| ImageView | 53,324 | 39,983 | 39,983 | 7,996 | 5,345 |
| Button | 16,007 | 12,007 | 12,007 | 2,400 | 1,600 |
| ImageButton | 8,693 | 6,521 | 6,521 | 1,306 | 866 |
| EditText | 5,643 | 4,230 | 5,000 | 846 | 567 |
| CheckedTextView | 3,424 | 2,582 | 5,000 | 505 | 337 |
| CheckBox | 1,650 | 1,238 | 5,000 | 247 | 165 |
| RadioButton | 1,293 | 970 | 5,000 | 194 | 129 |
| ProgressBar | 406 | 307 | 5,000 | 60 | 39 |
| SeekBar | 405 | 304 | 5,000 | 61 | 40 |
| NumberPicker | 378 | 283 | 5,000 | 57 | 38 |
| Switch | 373 | 280 | 5,000 | 56 | 37 |
| ToggleButton | 265 | 199 | 5,000 | 40 | 26 |
| RatingBar | 219 | 164 | 5,000 | 33 | 22 |
| Spinner | 20 | 15 | 5,000 | 3 | 2 |
| **Total** | 191,300 | 143,170 | 187,598 | 29,040 | 19,090 |

**Abbreviations for column headings:** "Total#(C)"=Total # of GUI-components in each class after cleaning; "Valid"= Validation; "Tr(O)"= Training Data (Organic Components Only); "Tr(O+S)"= Training Data (Organic + Synthetic Components).

(b) Class distribution of *ReDraw* dataset.

Figure 3.16: Examples and class distribution of the *ReDraw* dataset. Taken from: [18]

with findings can be seen in figure 3.17. Each feature is evaluated with one of three options: yes, no, and somewhat. Three of the tested tools, *CVAT* [6], *Label Studio* [7], and *VOTT* [8], were discovered to be useful for annotation of large datasets with many (30+) classes. They offer look-up of classes, file management, assisted annotation, and a large variety of export formats. Each of these tools come with its advantages and disadvantages, but *CVAT* was selected for the purposes of this thesis.

| Features | CVAT | Label Studio | LabelImg | LabelMe | VGG | VOTT |
|---|---|---|---|---|---|---|
| Images | yes | yes | yes | yes | yes | yes |
| Video | yes | yes | no | yes | yes | yes |
| Cooperation | yes | yes | no | no | no | no |
| Assisted annotation | yes | yes | no | no | no | yes |
| Detection | yes | yes | yes | yes | yes | yes |
| Segmentation | yes | yes | no | yes | yes | yes |
| Classification | somewhat | yes | somewhat | yes | yes | somewhat |
| Many export formats | yes | yes | no | no | somewhat | yes |
| File management | yes | yes | no | no | no | somewhat |
| Web version (online) | somewhat | no | no | yes | yes | yes |
| Easy installation | somewhat | yes | yes | yes | yes | yes |
| Suitability for a large dataset | yes | yes | no | no | no | somewhat |
| Suitability for many classes | somewhat | yes | somewhat | somewhat | no | somewhat |

Figure 3.17: Feature set comparison of popular free annotation tools

[6] https://cvat.org/
[7] https://labelstud.io/
[8] https://github.com/microsoft/VoTT

# Chapter 4

# Draft of UI Elements Identification and GUI Semantic Analysis System

This chapter outlines the UI elements identification method, the GUI hierarchy analysis method, the text extraction algorithm, and the semantic dictionary. Also, a technical specification is made. This chapter separates the research and the implementation part of the thesis.

## 4.1 Objectives and Requirements for the Resulting Solution

The objective of the resulting solution is to identify UI elements of a device, which are captured by a digital camera. The system expects an image of the device's screen and a list of detected elements on that screen as input, and for each member of that list, it should return its class, text, and to which GUI hierarchy section it belongs. The system should be able to distinguish between similarly looking pictograms that represent actions users can do with the device. If the areas containing buttons also contain text, the system should be able to extract that text and use it to enhance the class prediction. A draft of the output of the system can be seen in figure 4.1. In the figure, blue bounding boxes represent input button areas, green text is the result of the classification, $p$ is a pictogram class, and $t$ is a text class. The colourful bounding boxes and text represent the result of the GUI hierarchy analysis.

## 4.2 Technical Specification of the Resulting System

Based on the needs of *YSoft AIVA*, the system must meet the following requirements:

- The system processes an input image, without a GPU accelerator, faster than a user could.
- The system works if the UI element both contains and does not contain text.
- The system must offer multiple predictions for a pictogram.
- The system must be able to process input in a parsable file.

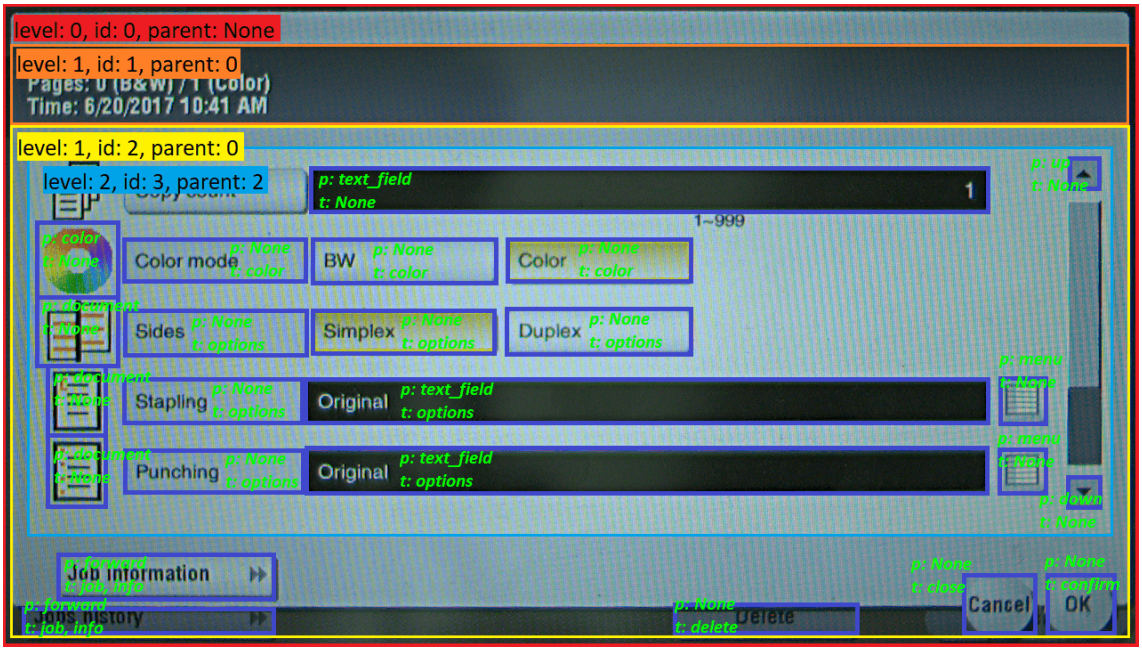



Figure 4.1: Draft of the output of the system.

- It must be possible to use trained weights of neural networks in different ML frameworks.
- The system's output must be a parsable and sendable file.

## 4.3 System Outline

The system consists of pre-processing phase and two main phases, button classification and GUI hierarchy analysis. The diagram of the system outline can be seen in figure 4.2. The system first detects if the input button areas contain any text, and if they do, the text is extracted and used to improve the accuracy of the button's class prediction. If any text is found, the button area is cropped to exclude it. The UI elements are then cut from the input image, batched, and their class is inferred. Next, for those UI elements that contained text, their predicted class is possibly corrected based on the text they contained, using the semantic dictionary. The last phase of the system is the GUI hierarchy analysis. The whole input image is analysed, and each button is assigned its hierarchical section.

## 4.4 Text Extraction Algorithm Outline

The system uses OCR to retrieve what text is present on the screen and where is it located. OCR that is used is an internal *YSoft* service that, for a given image, returns all detected words and their coordinates. Each word is then assigned to an element that the word intersects the most. Then all words in an element are merged into sentences if they are close enough together. The text provides additional information about the class of the button by matching the text with a hand-made dictionary of synonyms. There can be multiple proposed text classes for each button.

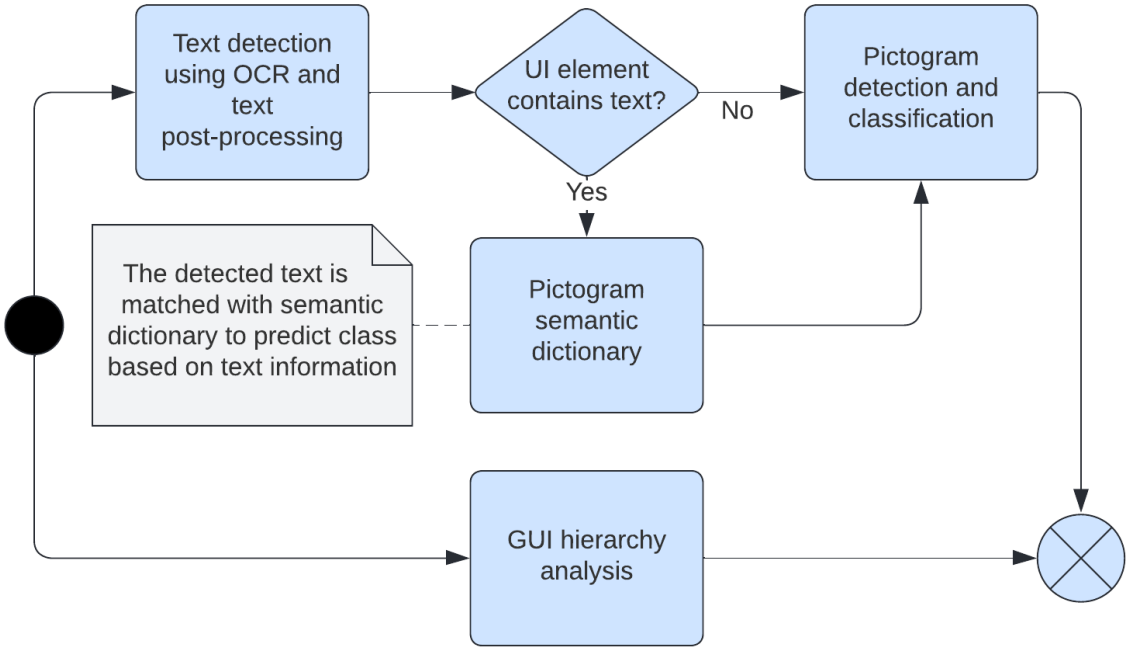



Figure 4.2: Diagram showing the system and its components.

It is also crucial to know if any text is present in the button area. If no text is present, the whole button area is sent to the pictogram classifier. On the other hand, if some text is present, another processing step is done to find only the area of the pictogram. This step must be done because the classifier is only trained on pictogram images without any text whatsoever. The pictogram detection algorithm is discussed further in section 4.5.

The implementation of this algorithm is discussed in section 5.4. An example of the OCR's output looks like this:

```
{"text": "Memory",
 "position": {
  "x": 1142.5,
  "y": 101.666664
  "width": 59.999996,
  "height": 15.833333}}
```

## 4.5 Pictogram Detection and Identification Outline

As mentioned in section 5.2, the selected CNN used for pictogram identification is Google's *EfficientNetB1*. There are three possible scenarios for how the button's identification can happen. If a button does not contain any text, the whole button's area is classified, and no pictogram detection is needed. However, if a button contains some text, the text has to be excluded from the classification. To accomplish this, traditional computer vision techniques are used to find edges that are not part of the text. If such edges are found, they are enwrapped into a bounding box area that is then classified by the selected CNN. However, if there is no pictogram found in the button, no classification is done, and only the text information is used, as mentioned in section 4.6.

The classification result is the five most probable class predictions and it is also included in the system's output. The exact implementation is discussed in section 5.5. An example of the classification output looks like this:

```
1: ALERT probability: 75.139%
2: TEXT_FIELD probability: 11.741%
3: DOWN probability: 11.629%
4: POWER probability: 0.411%
5: DOCUMENT probability: 0.365%
```

## 4.6 Semantic Dictionary Outline

The semantic dictionary is necessary to identify buttons that contain only text information. In cases where there is both some text and a pictogram, the text information is used to propose other possible classes of the button, may the classifier fail, or help select the most probable class of top predictions from the classifier. For each classifier class, the dictionary contains a list of words that are synonyms for the class's name and words which can also be used to describe the given pictogram class. Synonyms can be present in multiple classes to make more accurate class predictions. All proposed predictions are then assigned to the button and are a part of the system's output. The semantic dictionary is hand-crafted and, for the purposes of this thesis, only supports English text. A corpus manager is used to find relevant synonyms for each class. Also, some synonyms are gathered empirically from printer screens. Implementation of the semantic dictionary is discussed in section 5.6.

An example of synonyms for the *info* class would look like this:

```
'info': ['info', 'information','notify', 'about', 'help', 'check']
```

## 4.7 GUI Semantic Analysis Outline

GUI semantic analysis is responsible for a hierarchical separation of the GUI screen. It takes the whole screen image as an input and outputs the detected hierarchical structure. The screen is processed using traditional machine learning techniques, and the analysis works under the assumption that the screen has distinct edges that separate different parts of the design. Such assumption can be made because most of input GUI images contain edges that separate different sections of the screen, as can be seen in figure 4.3. Other image features, like colour, could also be used to distinguish different GUI sections; however, those are not as reliable because GUIs in some designs have the same background colour throughout the whole screen and only lines of different colours separate sections. Such examples can be seen in figures 6.1 and 6.3. When the image is pre-processed (edges are exaggerated), the *OpenCV* function *FindContours* is used to obtain found contours. They are then filtered to remove small contours, and largely intersecting ones are merged. When this analysis is done, each button is assigned to the inner-most part of the hierarchy it intersects. The implementation of this analysis is discussed further in section 5.7.

An example of the result of the analysis can be seen in figure 4.3. *level* refers to the level of nesting, *id* refers to an identifier of the GUI section, *parent* refers to the identifier of the parent GUI section. A structured description of this example looks like the following:

```
{"coordinates": 'some x,y,w,h coords',
```

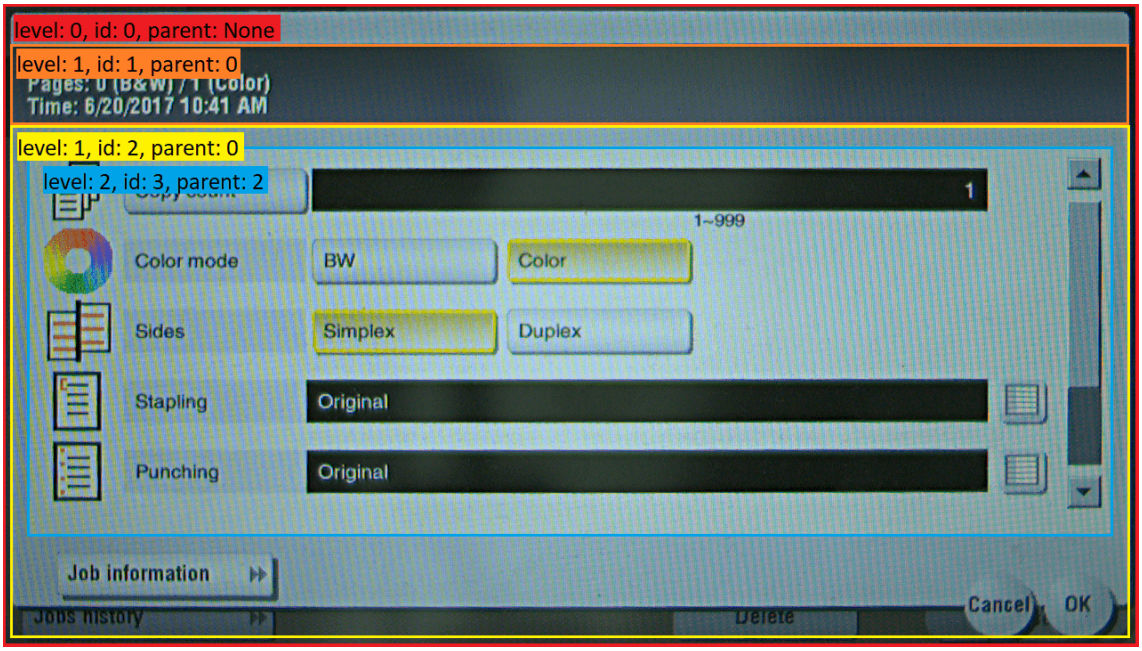



Figure 4.3: GUI semantic analysis draft example.

```
"parent_id": None,
"buttons": [...],
"id": 0,
"children": [{
   "coordinates": 'some x,y,w,h coords',
     "parent_id": 0,
     "buttons": [...],
     "id": 2,
     "children": [{
         "coordinates": 'some x,y,w,h coords',
         "parent_id": 2,
         "buttons": [...],
         "id": 3}]},
   {"coordinates": 'some x,y,w,h coords',
     "parent_id": 0,
     "buttons": [...],
     "id": 1,
     "children": []}]}
```

The level is represented as *parent/children* relation in a tree structure. Each GUI section also contains its id, its parents id, a list of buttons that belong to that section and a list of child sections.

## 4.8 System Input and Output Outline

The system expects an input image of a device's screen and a file with a parsable structure, as mentioned in section 4.2, that contains coordinates for each found GUI element in that

image. Also, for each button, the file must contain a general class to which it belongs. The valid general classes are *StaticText*, *EditText*, *Button*, *RadioButton*, *Switch*, *CheckBox*, *StaticImage*, and *ImageButton*. If the button contains a different label, it is ignored.

As mentioned in the section 4.2, the system's output is a parsable file that contains every information obtained during the processing of the input. The output contains this information:

- hierarchy information
  - hierarchy coordinates
  - hierarchy id
  - hierarchy parent id
  - list of buttons inside it
  - list of child hierarchies
- button information
  - button coordinates
  - original button label
  - button text and its coordinates
  - button class
  - five most probable classes
  - suggested text classes
  - relative pictogram area within the button

The exact implementation is discussed in section 5.8.

# Chapter 5

# Proposed System Implementation

In this chapter, the implementation of the proposed system is discussed. The creation of datasets needed for the training of the system is described. Text extraction algorithm, pictogram detection and identification, semantic dictionary and GUI semantic analysis implementations are described in this chapter. All steps of these algorithms, along with their inputs and outputs, are mentioned.

## 5.1 Programming Language and Frameworks Selection

Nowadays, there are multiple programming languages that are suitable for machine learning tasks, but Python [31] still has the largest community and the most available resources and frameworks that help programmers focus on their specific tasks, ultimately increasing productivity. Although Python's runtime is very slow, thanks to the number of optimised frameworks available, all time-sensitive computation is done outside of Python, making the whole system run very fast. For those reasons, Python was selected to be the implementation language of the system. The selected ML frameworks were *TensorFlow* and *Keras*. *Keras* framework contains an implementation of Google's convolutional neural network *EfficientNetB1* that was, as mentioned in section 5.2, selected for pictogram identification. Nowadays, *Keras* exclusively uses *TensorFlow* as a backend, so it was also indirectly used.

Other frameworks worth mentioning that were selected are *NumPy* [20], *OpenCV* [21] and Anytree [1]. *NumPy* was selected for its fast array manipulation capabilities, whereas *OpenCV* was selected for its image manipulation capabilities. These frameworks are written in a low-level programming language and are compiled. Python only serves as a high-level API, which makes its use convenient, but at the same time, the runtime is very fast. Anytree, a tree data structure implementation, was selected to implement the hierarchical analysis of the GUI screen. The hierarchical analysis is discussed in detail in section 5.7.

## 5.2 UI Elements Identification Method Selection and Implementation

### Selection of the UI elements identification methods

Based on the research of methods suitable for UI elements identification, which is conducted in section 3.1, the selected CNN for this system is *EfficientNetB1* 3.1. The *EfficientNet*

[1] https://anytree.readthedocs.io/en/latest/

family of networks contains 7 variants of the same architecture, scaled to fit different needs. The *B1* variant of the network was selected for its relatively low computational complexity and high classification accuracy compared to other members of its family, and also different methods. Due to open formats for ML models, for example like *ONNX* [2], the model and the trained weights of the network can be loaded and used in any other modern ML framework that supports the standard, even if the framework does not contain the implementation of said network.

### Compilation of the network

*Keras*'s implementation of the *EfficientNetB1* was used as it allows to load the model with custom weights and without a top layer. Pre-trained weights, called *NoisyStudentB1NoTop* [3], were used to speed up the training process. The *EfficientNetB1* was loaded without its top layer so that the network's output could be customised. Nevertheless, to keep the network architecture the same, some layers (those that were not loaded) were added back, followed by a dense layer to match the number of classes.

- *GlobalAveragePooling2D* layer
- *BatchNormalization* layer
- *Dropout* layer with 0.2 probability
- *Dense* output player with 61 neurons (number of classes) and *Softmax* activation function

Adam with default learning rate was used as an optimiser function. Sparse categorical cross-entropy was used as a loss function. The compiled network has 6,658,500 parameters, of which 1,431,181 are trainable.

### Implementation

The definition and training of the *EfficientNetB1* convolutional neural network are located in the file *efficientnetb1_training.py*. Running the scripts starts training the network. The script requires that both the training and the test datasets are stored in a folder *./dataset/train*, respectively *./dataset/test*. The weights are saved after each epoch if the validation accuracy exceeds the maximum achieved accuracy so far. These weights are stored in the file *efficientnetb1_model.hdf5*. If there is such a file in the same folder as the script, the weights are loaded, and training starts again. After each training phase, a plot showing its progress is shown. At the end of the training, the model is evaluated, and the results are printed into the terminal.

## 5.3 Pictogram Classifier Dataset Creation

As mentioned in section 3.2, there is no available dataset suitable for UI elements classification into *fine* classes; therefore, a new dataset had to be created. The dataset partly consists of UI element cuttings of printer touch screen images taken by a camera and partly by UI element icons in different designs downloaded from a website.

[2] https://onnx.ai/
[3] https://www.kaggle.com/ipythonx/efficientnet-keras-noisystudent-weights-b0b7

### Printer dataset creation

*YSoft* provided the original images of printer screens. The dataset consists of 1,218 images annotated into six classes. This number of classes was insufficient for identifying different pictogram meanings, so new, more detailed annotations were created in the dataset annotation tool 3.2 called *CVAT* 5.1. Icons were distributed among 61 classes A.1, which resulted in 6,796 icon annotations. Before annotations were cut from the original image, augmentations were applied to each annotation. Both, the original and the augmented images can be seen in figure 5.2. Augmentations that were used to enhance the dataset were:

- rotation by +- 5°
- scale by +- 15%
- shear by +- 0.005
- translation by 8% of the icons width and height

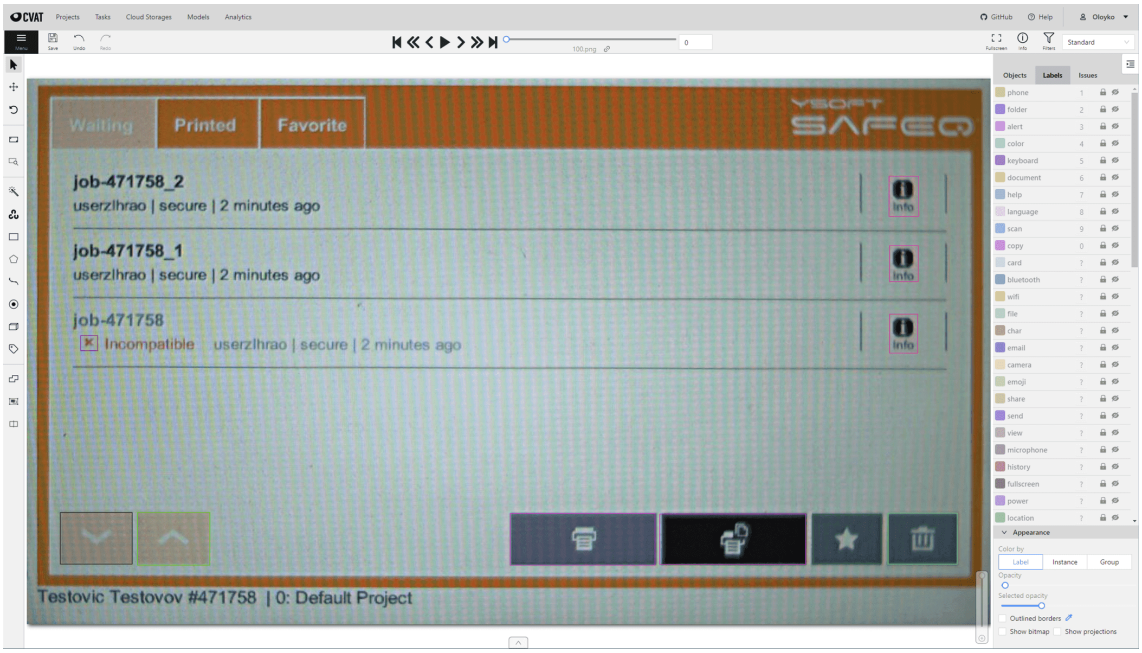


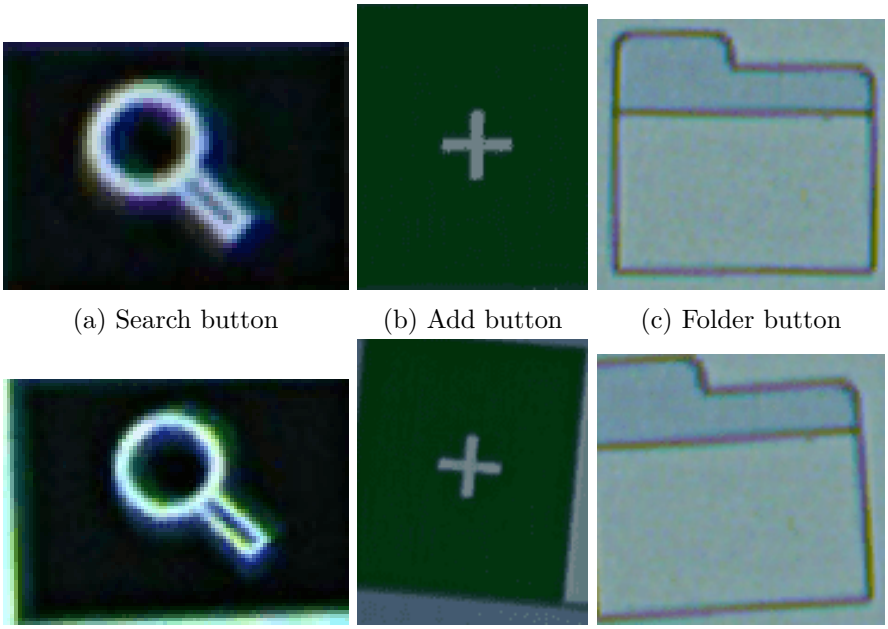

Figure 5.1: Printer image annotated in *CVAT*.

Also, during training, a 10% contrast shift was used. For each icon, five new icons were generated using the augmentations mentioned above, with values assigned by the uniform distribution. This step could have been done in *Keras* during training but that would have resulted in black corners due to a lack of surroundings information in the icon cuttings. This process resulted in a final printer dataset with 40,776 images. The images were kept in original resolution to save disk space. The images are resized to a resolution, as required by the classification neural network, during training by *TensorFlow* dataset loader functions.

The creation of the printer dataset is implemented in the file *printer_dataset_creation.py*.

**Printer dataset image filtering** Since the images from the printer dataset were captured by a camera; they suffer from the *Moiré* pattern [4]. This effect is generally periodical

[4]https://www.focuscamera.com/wavelength/what-is-the-moire-effect-in-photography-how-to-avoid-it/



(a) Search button (b) Add button (c) Folder button

(d) Augmented search button (e) Augmented add button (f) Augmented folder button

Figure 5.2: Printer dataset buttons

and creates unwanted edges that could reduce the accuracy of classification. If the effect were to be removed from images in the printer dataset, it would also have to be removed from each image classified in a production environment. This means that the filtering needs to be fast as the system needs to work in real-time. To remove the noise, three different approaches were tried. Different tried filtering approaches are discussed in section 6.1.

### Generic UI pictogram dataset creation

More varied data had to be gathered to teach the classification neural network to recognise icons in different design styles. One such dataset that contains UI icons in different design styles exists [5], but it was not used because the images are in *jpg* format, which is not suitable for background replacement.

The images were downloaded from a website [6] that contains millions of icons in different design domains. 3,402 black icons in different design styles with alpha channels were downloaded from the website. These icons are without noise and look too perfect in comparison with the input data of the system, so image augmentation was used to expand the dataset and make it visually correspond with actual input data.

**Background replacement** Background cuttings with different brightness levels, later sorted into dark and light groups, were extracted from the original printer dataset. Then each icon, which was initially only black, was recoloured to dark grey, light grey and white colour. Also, black icons were left. As mentioned in the paragraph below, these recoloured icons were augmented and then placed on top of the extracted backgrounds. Light icons

[5] https://www.kaggle.com/testdotai/common-mobile-web-app-icons
[6] https://icons8.com/

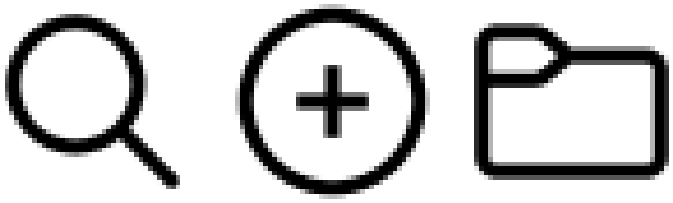


Figure 5.3: Examples of downloaded icons

were merged with dark backgrounds, and dark icons were merged with light backgrounds. From a respective brightness group, which background was used is determined at random with uniform distribution. Examples of downloaded icons and extracted backgrounds can be seen in figure 5.4. Figure 5.5 shows examples of the finished icon dataset.

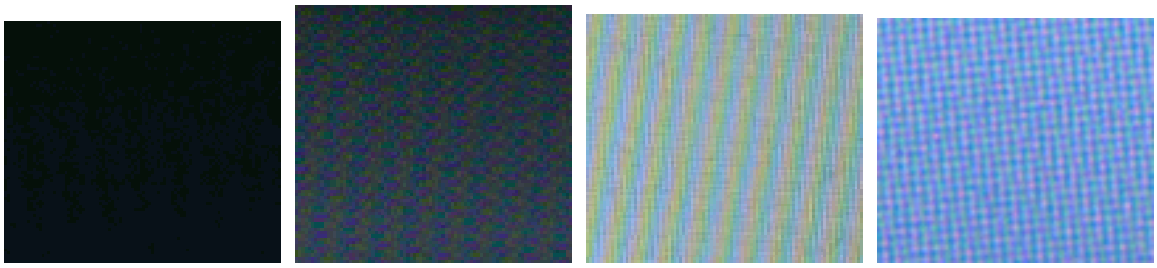


Figure 5.4: Examples of extracted backgrounds

**Icon augmentation** Augmentations were applied to each icon before icons were placed on top of the background image. Augmentations that were used to enhance the dataset were:

- rotation by +- 5°
- scale by +- 15%
- shear by +- 0.005
- translation by 8% of the width and height of the icon

Also, during training 10% contrast shift was used. Five new icons were generated using the augmentations mentioned above, with values assigned by the uniform distribution for each icon. This step could have been done in *Keras* during training, but that would have resulted in black corners due to a lack of surroundings information in the icon cuttings.

This process resulted in a final generic dataset with 81,624 images. The images were kept in original resolution to save disk space. The images are resized to a resolution, as required by the classification neural network, during training by *TensorFlow* dataset loader functions.

The final dataset for UI elements identification was created by merging the printer icon dataset with the generic icon dataset. In total, this dataset contains 122,016 images divided into 61 classes.

The generation of this part of the dataset is located in the file *icons_dataset_creation.py*.

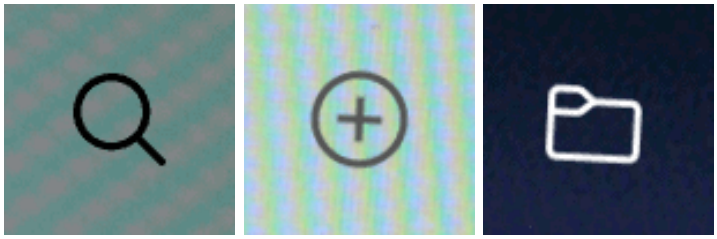


Figure 5.5: Examples of combined icons with backgrounds

## 5.4 Text Extraction Algorithm Implementation

As mentioned in section 4.4, the processing of text present on the screen is essential for the proper working of the whole system. The system uses *YSoft*'s internal OCR to read text on the screen.

The OCR expects an image encoded into *base64* format and sent in the payload of a request. If the request fails, an empty string is returned. If the request succeeds, a JSON is returned. The JSON contains a list of found words, which are in this format:

```
{ "text": "Memory",
  "boundingBox": {
    "position": {
      "x": 1142.5,
      "y": 101.666664},
    "size": {"width": 59.999996,
    "height": 15.833333}}}
```

The *text* key holds the text value of the word. The *boundingBox/[x/y]* key holds the top-left coordinates of the text bounding box, while the *size/[width/height]* key holds the width and the height of the bounding box. The OCR's response is cached to avoid overloading of the service.

Each word in the JSON response is then added to a button object that it intersects with. If the button's original class is not *[StaticText/EditText/Button]*, the word is not added to it as the other classes are not expected to contain any text and would, in most cases, result in an error.

Next, each button's text is merged into sentences. The merging is based on the distance between each word. The threshold is $0.1 * button\_width$ , respectively $0.1 * button\_height$ for horizontal merging, respectively vertical merging. The whole purpose of the merging, even though it is not useful for further analysis, is to connect words that relate to each other, as this result is more useful for end-users.

The result of the whole algorithm so far can be seen in figure 5.6. The blue bounding box represents the whole element area, while the green bounding box represents the area of a sentence.

The last text analysis phase matches the button's text information with the semantic dictionary 5.6. Each button's sentence is split into words, and each of those words is checked to whether it matches any values in the dictionary. Each word is converted to lowercase before this process. Keys of values that matched with words are added to the button's text class prediction list.


Figure 5.6: Printer screen image with merged text.

The text class prediction is mainly used to classify buttons that contain text but do not contain a pictogram. This helps massively helps automation, as users do not need to annotate these text buttons manually, or for cases where more text classes were suggested, they can only select one class from the suggested list, saving time and limiting human errors. Also, as the text classification uses the same 61 classes as pictogram classification, another use case is cross-checking these two results for buttons that contain both text and pictogram. The effectiveness of result cross-checking is mentioned in the evaluation section 6.7.

An example of the text classification can be seen in figure 5.7.

**Limitations** One huge limitation is the OCR's inability to read buttons with only one letter or a number. Numbers are often correctly read, but the OCR tends to merge them with nearby numbers on its own, making them overlap over multiple buttons. As a result, letters are often incorrectly read or even not detected at all. A solution to this problem might be using a different OCR, but the same problem, albeit not as apparent, occurred using *Google*'s *Cloud Vision OCR* [7].

Another approach is ignoring buttons that are part of the software keyboard. However, the detection of the keyboard would have to be hard-coded, which is not desired as more errors could appear, or done using a neural network or traditional computer vision, which is outside the scope of the thesis. The problem can be seen in figure 5.8.

**Implementation** The request creation and retrieval are implemented in the file *text_analysis.py*, in `OcrApiCommunicator` class. The intersection check is implemented in the function *check_if_near_bb* in *the utils.py* file. The text merging is implemented in the method *merge_ocr_result* of *ButtonCutting* class. Finally, the button's text semantic dictionary matching is implemented in the function *get_text_classes* in the file *text_analysis.py*.

[7] https://cloud.google.com/vision/docs/ocr


Figure 5.7: Text classification example.

## 5.5 Pictogram Detection and Identification Implementation

The algorithm described in this section is responsible for the identification of pictograms present in the input image. It uses *EfficientNetB1* CNN as a classification method to distinguish pictograms into 61 classes. The training of the network is described in section 6.2. Also, traditional computer vision methods are used to detect a region with the pictogram in cases where the UI element contains some text. However, as the classifier is trained on a dataset containing only pictograms, classifying UI elements with text would result in poor classification accuracy.

The input of the algorithm is a UI element, which has already passed through the text analysis. The whole UI element area is classified if it does not contain any text. Otherwise, a pictogram detection occurs.

**Pictogram detection** First, the element is converted to grayscale and thresholded using *Otsu*'s method [8]. Then, both *Sobel-x* and *Sobel-y* are applied to the binary image, resulting in two images with found edges. Then, for both images, contours are found and filtered to eliminate the ones that are too small. Contours that were eliminated were erased from the binary images. Now that the noise was removed, a morphology operation *dilate* is applied three times to both images. The binary images are then merged together using a *bitwise or* operation. In the combined image, contours are found again. Contours that are smaller than 1% of the element area are eliminated. Also, contours that intersect with the text of the element are eliminated. Then, those contours located at the borders (10%) of the area are eliminated. Lastly, if the height or the width of the contour is ten times longer than the other, it is also eliminated. Finally, those contours that got through all tests are merged into one singular area. If that area is larger than 4% of the total area of the element, the result of the detection are coordinates of the supposed pictogram. If no

[8] https://learnopencv.com/otsu-thresholding-with-opencv/

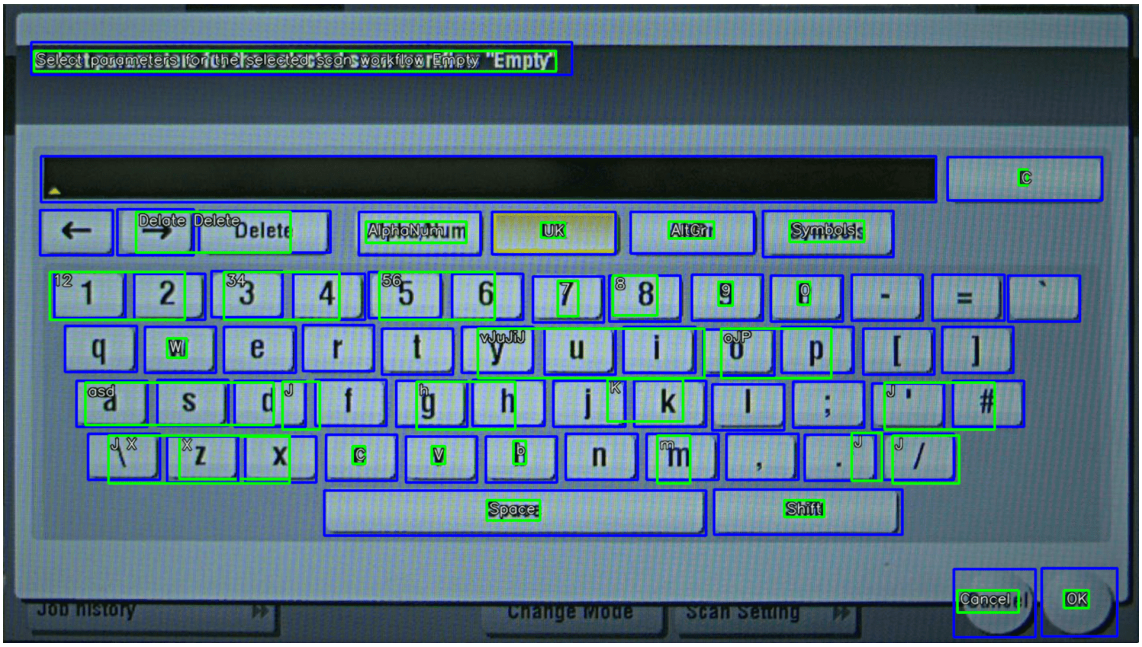


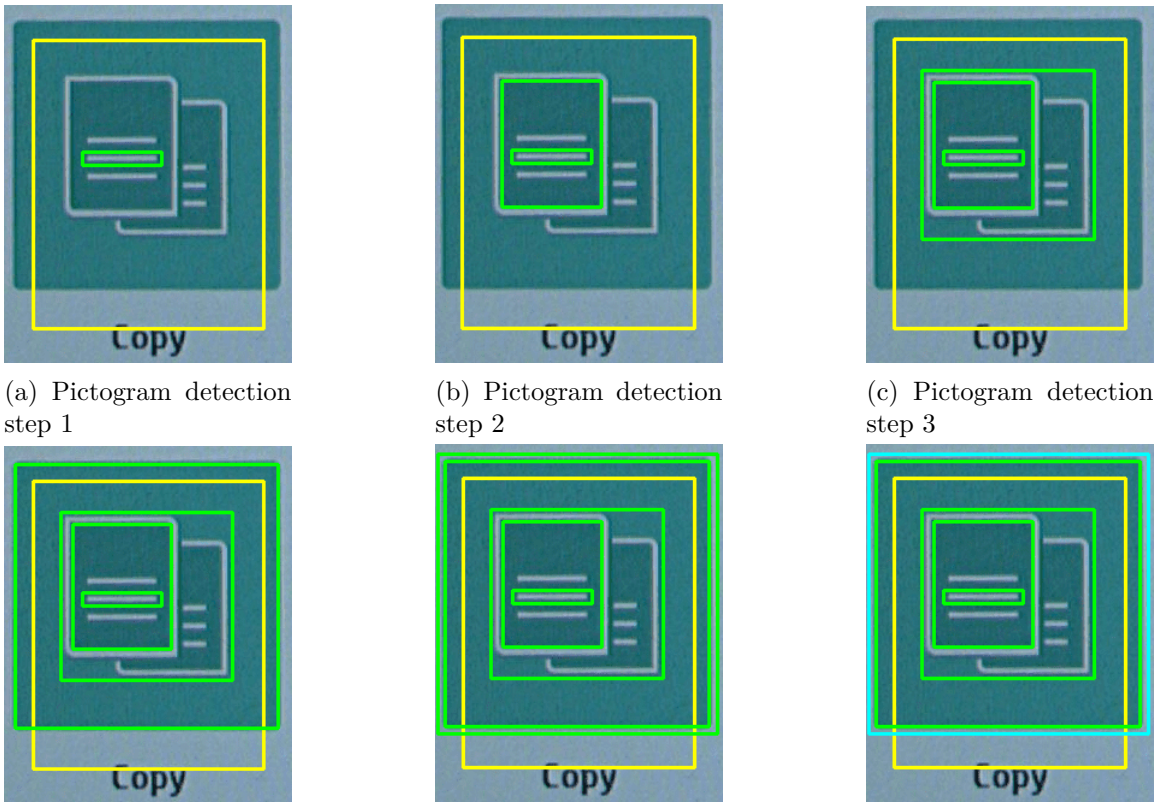

Figure 5.8: An example software keyboard screen image failed text detection.

contours passed to the end, the element is not classified. The detection process can be seen in figure 5.9. In that figure, the yellow bounding box shows the region outside of which contours are ignored, the green bounding boxes are contours that passed through all tests, and the blue bounding box is the final pictogram area that is created by merging the green areas. Steps one 5.9a through five 5.9e show contours that passed through all tests and step six 5.9f shows the output of the detection.

**Pictogram classification** The classification input is either the whole area or the area of the supposed pictogram, as discussed above. First, the pictogram is resized to 240x240 resolution to fit the input layer size of the *EfficientNetB1*. Then, the pictogram is classified, resulting in 61 probabilities. Finally, only the five most probable classes are stored and are part of the system's output. The classification output of the button shown in figure 5.9 can be seen below.

```
Top 1 pred: DOCUMENT with~prob: 51.41647458076477%
Top 2 pred: COPY with~prob: 48.373040556907654%
Top 3 pred: CALENDAR with~prob: 0.15204616356641054%
Top 4 pred: PRINT with~prob: 0.04938848433084786%
Top 5 pred: FOLDER with~prob: 0.004582103429129347%
```

**Limitations** As the input images are very noisy and the *Moiré* effect creates a lot of strong edges, pictograms are detected in elements that do not contain any pictogram. This error is only present in cases when the element contains some text which also means that the semantic dictionary is used and can propose correct classes, mitigating the error. Also, sometimes no pictogram is detected even though it should. This happens in cases where the pictogram contrast is very low as the algorithm is tuned to ignore the noise. The semantic dictionary mitigates this error as well.




(a) Pictogram detection step 1

(b) Pictogram detection step 2

(c) Pictogram detection step 3

(d) Pictogram detection step 4

(e) Pictogram detection step 5

(f) Pictogram detection step 6

Figure 5.9: Example of pictogram detection steps

The implementation of algorithms mentioned in this section is located in the file *pictogram_detection_identification.py*.

## 5.6 Semantic Dictionary Implementation

The semantic dictionary is used to perform a text-based button classification. The button's text information is used to find dictionary values that match the word. If such a match succeeds, the value's key is returned as the button's class. This algorithm is explained in section 5.4.

The semantic dictionary is implemented simply as a Python dictionary data structure. Its keys are equivalent to the pictogram classifier classes A.1. Values of these items are words that could also be used to describe images in the pictogram dataset grouped under the same class label. For example, a plus sign icon, which is assigned to the *add* class, could verbally be described as *plus, add, insert, join, new*, etc. Some words were added to the dictionary empirically. Then, *Word Sketch* [9] corpus manager system was used to enlarge the dictionary systematically. There are some duplicate values as a word can describe multiple class labels. For example, the word *internet* can describe icons that belong to both *web* class and *wifi* class, even though these classes are two disjunctive sets. The whole semantic dictionary can be seen in figure A.2.

[9] https://app.sketchengine.eu

Other classes can be added to meet domain-specific requirements for text button classification. In this thesis, text classes like *job* and *waiting* were added to increase classification accuracy in the printer domain.

**Limitations** As this dictionary was hand-made, it does not scale well. All changes and corrections have to be made manually. Also, in this thesis, the only supported language is English, and the translation to other languages would mostly also have to be made manually.

## 5.7 GUI Semantic Analysis Implementation

GUI semantic analysis separates the GUI into distinct, hierarchically ordered sections. It uses traditional computer vision methods to filter the image, convert it into a binary form and find the aforementioned hierarchical sections. The draft of this analysis is discussed in section 4.7.

The analysis expects the whole GUI image as an input. The image is then converted into grayscale, a *bilateral* filter (*neighbour pixel diameter* = 5, *colour space sigma* = 25, *coordinates space sigma* = 75) and then a Gaussian blur with a kernel size of 3 is applied to it. The blurring is necessary to eliminate the *Moiré* as much as possible and improve further analysis. The grayscaled and blurred image, which can be seen in figure 5.10, is then passed into the *Canny* detector (lower hysteresis threshold = 10, upper hysteresis threshold = 80, aperture size = 3, L2Gradient = True) to detect edges present in the filtered image. Next, the morphology operation *close* with a 5x5 kernel of ones is applied to the picture to remove background noise and close gaps in found edges. The last pre-processing phase adds a rectangle around the whole image to help correctly detect *root* contour in the last step. The result of the pre-processing can be seen in figure 5.11. The function from *OpenCV* *findContours* using *RETR_TREE* and *CHAIN_APPROX_SIMPLE* parameters is used for this pre-processed image. It returns all detected contours, approximated as rectangles, in a tree-like structure. Each detected contour contains its id, the id of its parent contour, the id of its previous contour, and the id of its next contour on the same level.

However, only those contours representing GUI sections are necessary for this analysis; therefore, the contours need to be filtered. First, contours that only take up 10% of the total image area are eliminated. Next, intersection over union is calculated for each contour pair. Those pairs whose IoU is larger than 0.92 are merged. During the merging process, contour id and parent id values are combined into a list that contains them.

Next, the filtered contours are stored in a tree data structure using the *AnyTree* [10] library. The root node is the root contour that was artificially created at the edges of the image. Children of the root node are all contours that match any of its parent ids with the root node's ids. The example image of this hierarchy analysis can be seen in figure 5.12. Each section contains its level of nesting, id (list of merged ids) and parent id. The hierarchy of the example image can be described as the following:

```
[123, 0]
    |--[28]
         |--[44, 38]
```

[10] https://anytree.readthedocs.io/en/latest/

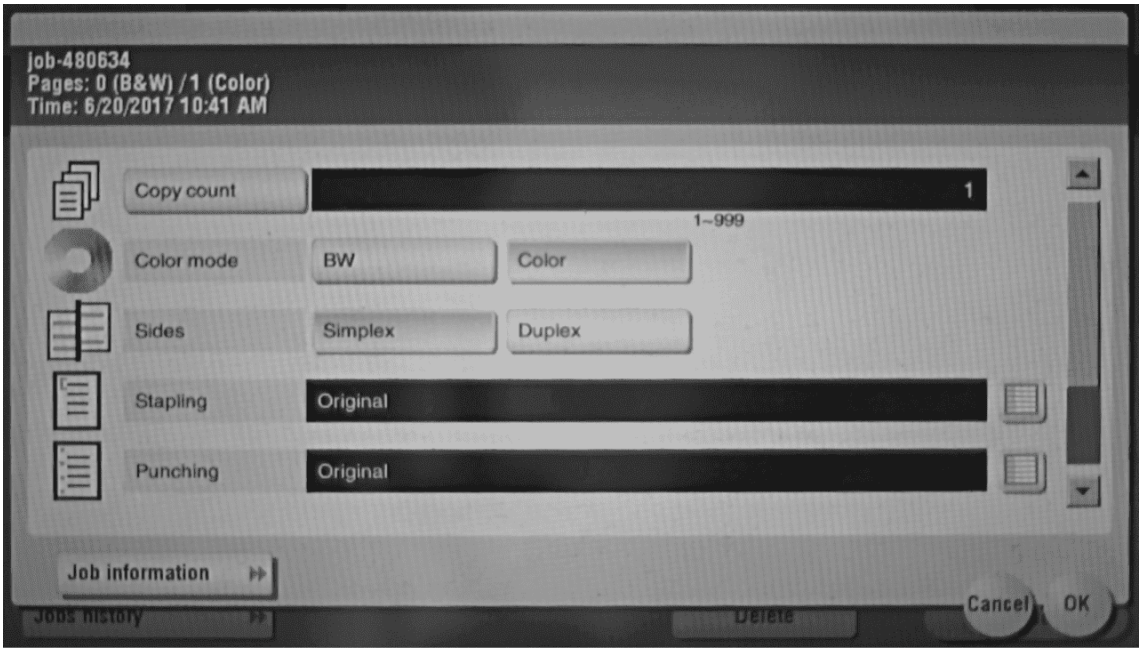



Figure 5.10: An example of a blurred image before edge detection.

Lastly, all buttons are assigned to the inner-most hierarchy section where they intersect. The detected hierarchy in the system's output is represented as a tree. The root hierarchy is also a root in the JSON output, and the children are stored as a list under the *children* key. Each hierarchy section in the JSON also contains a *buttons* key which holds information about each button that belongs to that section. The output of the system is further discussed in section 5.8.

**Implementation** The algorithm discussed in this section is implemented in functions in the file *hierarchy_analysis.py*. The function *find_contours* processes the input image using computer vision methods and finds contours in the image. The function *filter_merge_contours* is responsible for merging contours with very similar areas and filtering areas that are too small. The function *build_tree* converts detected contours into a tree data structure. Finally, the function *assign_button_hierarchy* assigns a button to the correct hierarchy.

**Limitations** Figuring out the correct parameters for used computer vision methods is really difficult as a slight change of one parameter can make the analysis unsuccessful. As those parameters are hard-coded, the analysis might fail when external conditions like lighting, screen resolution, or camera resolution change. Therefore, parameters were set to enable the algorithm to successfully analyse images with significant defects. As a downside, smaller details in the screen are not detected, so GUI sections with low contrast edges are ignored. Still, sometimes noise is detected as a GUI section, resulting in a wrong output. Also, as the hierarchy analysis is based on an edge detector, GUI screens with a minimalist and clean design might result in an output with only the root section present. However, it is not a huge problem, as in that case, human eyes cannot detect any edges either, and the GUI is structured and works differently.

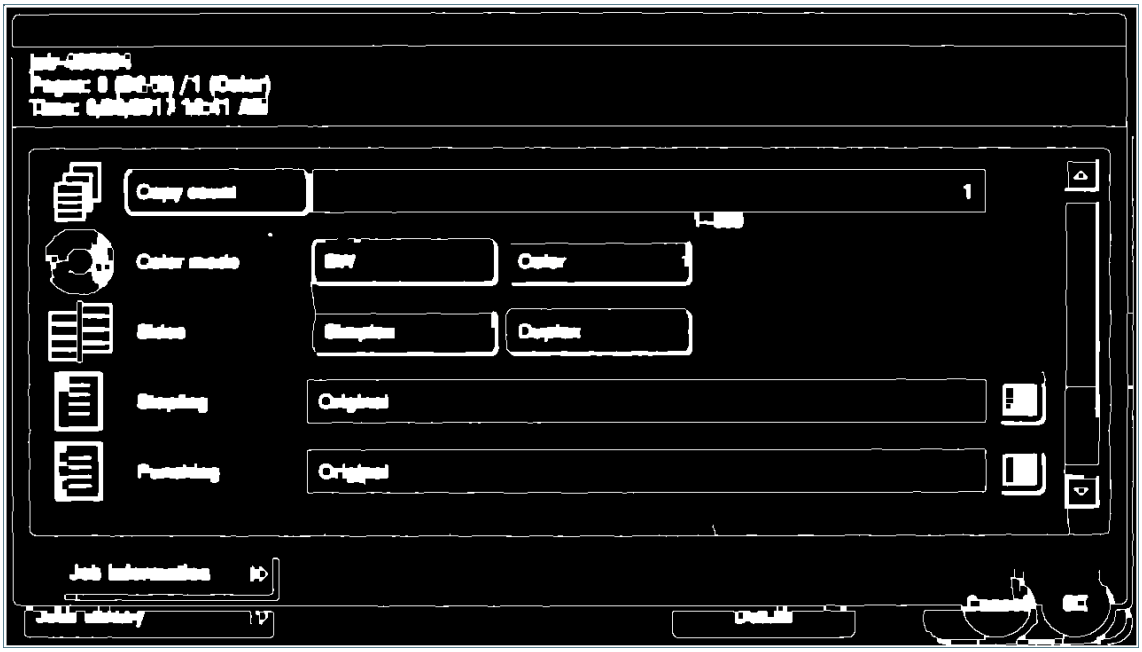


Figure 5.11: An example of an image after *Canny* edge detection and morphology close operation.

## 5.8 System Input and Output Implementation

### Input

The input of the system is a combination of a GUI image and UI element detections for that GUI. The button detections, which are produced by *YSoft*'s internal service, are stored in a file in Pascal VOC [11] format. These detections contain the coordinates of each element along with its class. The class must be one of the following: *StaticText, EditText, Button, RadioButton, Switch, CheckBox, StaticImage*, or *ImageButton*. An example of the input detections can be seen in figure A.3. The whole visualised input can be seen in figure 5.13. Blue bounding boxes indicate areas of detected buttons, and the text represents the original class of the element. When both files are loaded, `ButtonCutting` object is created for each element. The object contains all information gathered about the UI element during the analysis of the screen. It also contains methods to output the information in a structured way.

### Output

After analyses discussed in this chapter are done, the GUI hierarchy sections are stored in a tree structure, and each section contains all buttons that belong to it. The tree is then traversed, and its stored information is converted into a JSON file. The output file is too large to include in the thesis, but the file's structure with descriptions can be seen in figure A.4.

[11] https://deepai.org/dataset/pascal-voc

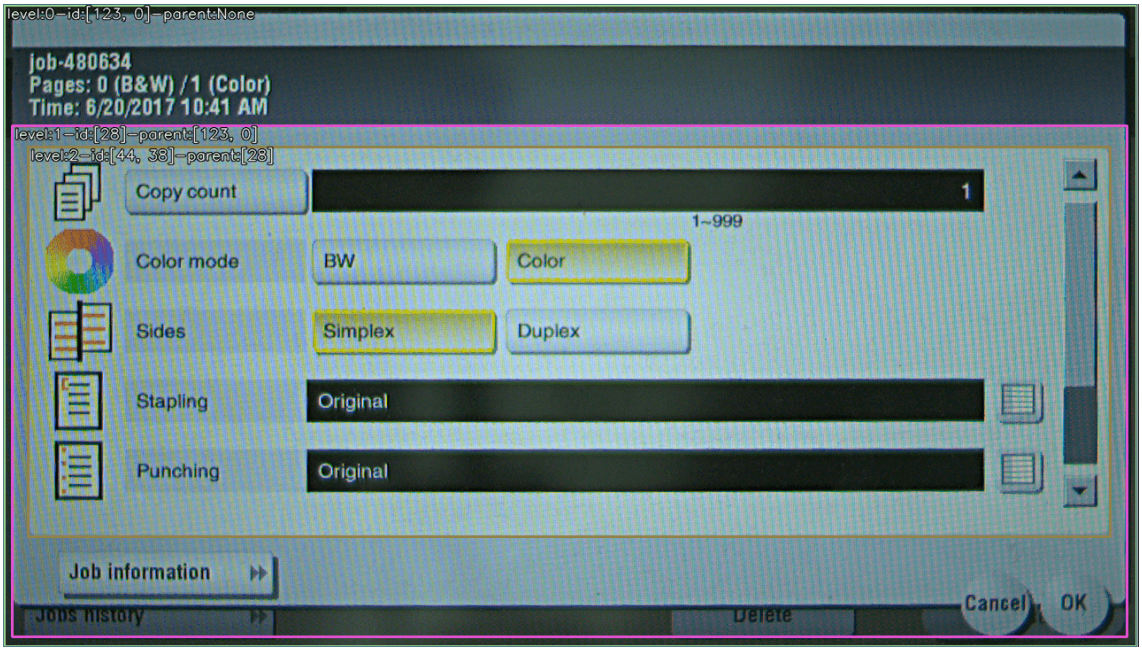



Figure 5.12: An example image after the hierarchy analysis is complete.

A visualised example of the system's output can be seen in figure 5.14. In the figure, blue (purple [12]) bounding boxes represent the input UI element area. Green bounding boxes, along with the text inside of them, represent the detected text. Red (purple) bounding boxes represent the supposed area with a pictogram. The word in the top left corner of the blue (purple) area is a predicted class of that pictogram. The second row in the top left corner is a list of predicted classes using the text information if there is any text in the element. Bounding boxes with other colours represent the hierarchy sections. Each section contains information about its nesting level, identifier, and parent.

[12] if the pictogram area is the same as the input element area, blue and red colour are overlapping → purple

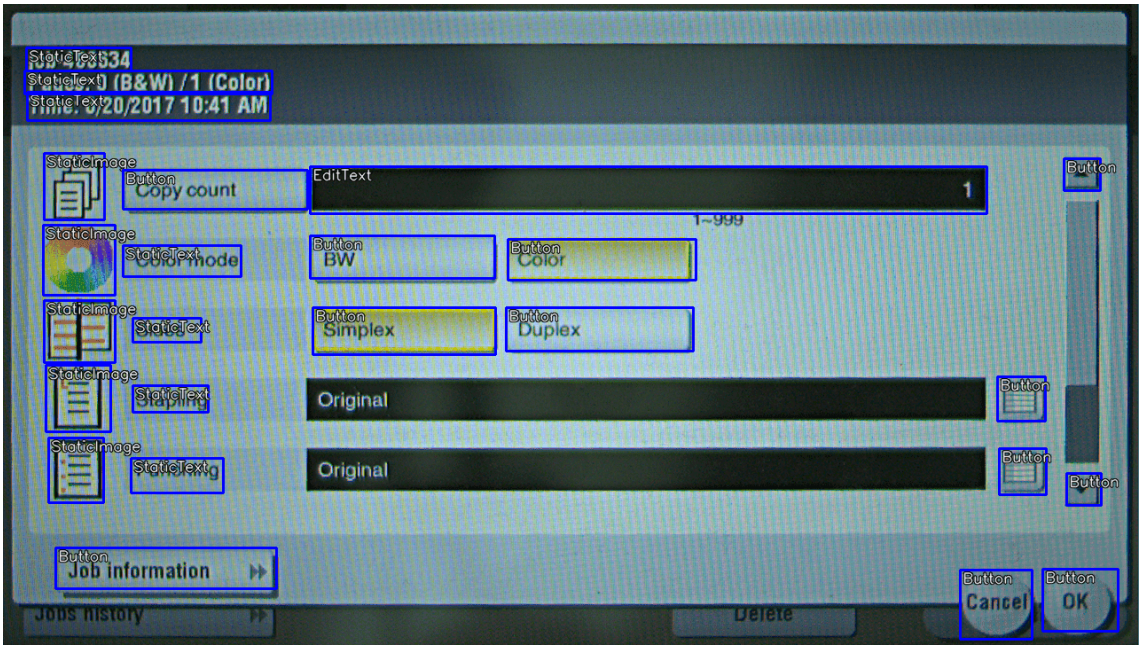



Figure 5.13: The visualised input of the system.

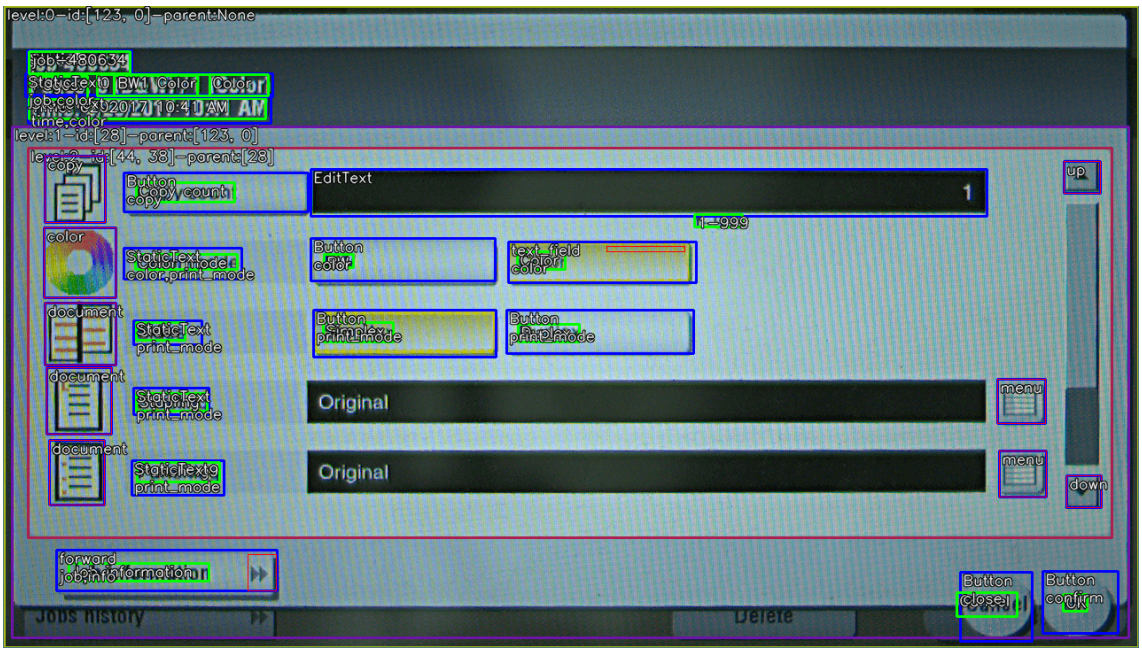



Figure 5.14: The visualised example of the output of the system.

# Chapter 6

# Experiments

In this chapter, experiments with the system and its components are mentioned. Then, the system's performance is evaluated. Finally, future work with the system is discussed.

## 6.1 Dataset Experiments

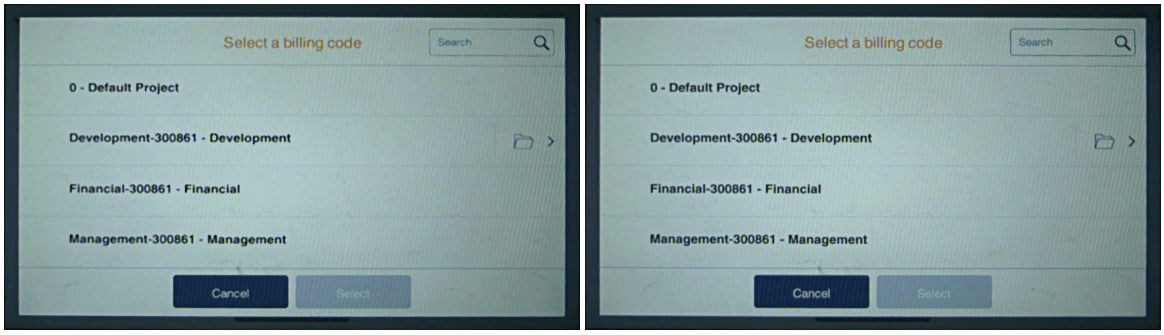
**Image resizing** *EfficientNetB1* expects the input image to have 240x240 resolution, while images in the dataset have a variable resolution. Images created using the icon website usually have 50x50 or 100x100 resolution, but images from the printer domain have a variable resolution. The whole dataset cannot be stored in memory during training, so a single image is resized multiple times. To mitigate that, each image was resized to the target resolution during the creation of the dataset. Then, the training time was measured and compared to the training time of the dynamic rescaling. The training took precisely the same time, so the idea was dropped, as the dataset was around 2.5 times larger.

**Dataset augmenting with noise** During the dataset creation, augmentations like rotation, scale, shear, and translation were used to augment and enlarge the dataset. Another way to augment the dataset is by adding noise to it. Gaussian noise, salt and pepper noise and colour noise were tried to expand the dataset further. These noise techniques were used on top of augmentations used in section 5.3 and the dataset was three times larger. However, as the input images are very noisy already and the accuracy of the classifier is 99% in all metrics, no improvements were noticed. Moreover, the training took three times longer for the same amount of epochs. Also, the original dataset was acquired using an HD camera, and the actual input of the system has a lot less noise nowadays. Therefore, no additional augmentations were used.

### Experiments with printer dataset filtering

**Local filtering** Both *Median* filter [9] and *Gaussian* filter [1] were tried to reduce the *Moiré* pattern. However, due to the nature of the effect, none of the methods was able to significantly reduce the effect while the rest of the image was degraded. Examples of such filtering can be seen in figure 6.1. Therefore, due to these facts, these methods were not used.

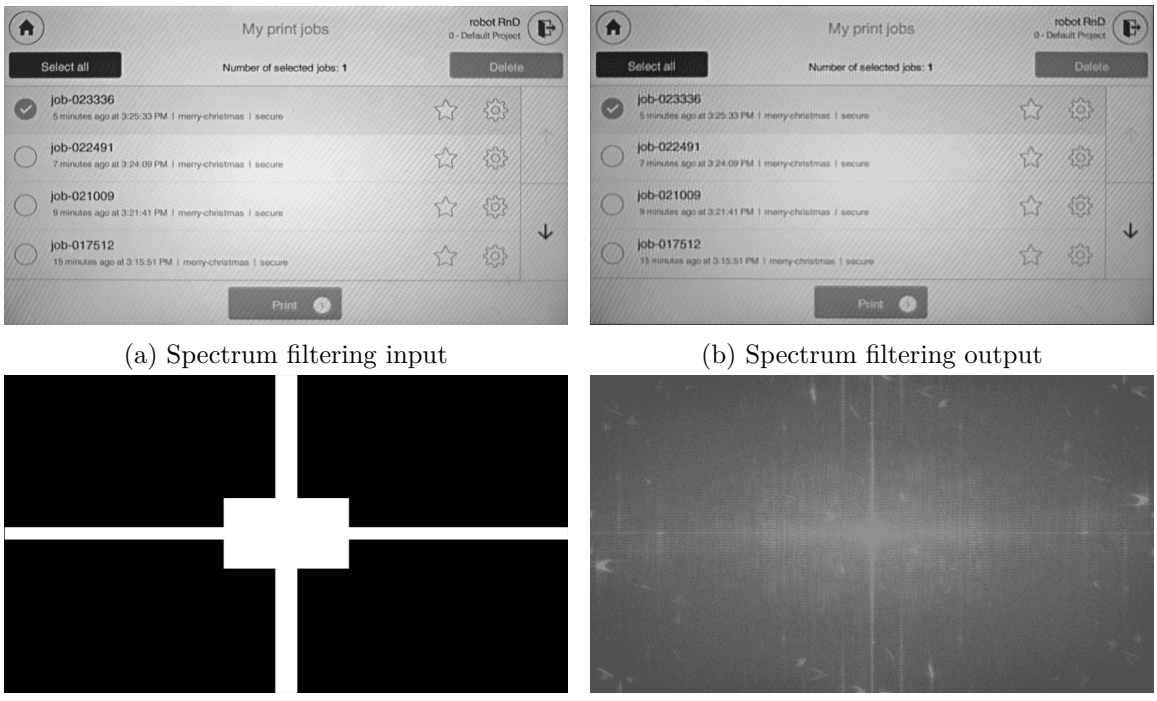
[1] https://www.geeksforgeeks.org/apply-a-gauss-filter-to-an-image-with-python/




(a) *Median* filter (b) *Gaussian* filter

Figure 6.1: Local filtering comparison. The original image (with some additional irrelevant information) can be seen in figure A.4

**Filtering in spectrum** Due to *Moiré*'s pattern periodical nature, spectrum filtering was tried to remove or reduce the unwanted effect. The printer image was converted to spectrum using *Fast Fourier Transform* [2]. Then a mask that should keep the main frequencies along both axes and remove *Moiré* Pattern peak frequencies was applied to the image. However, this approach did not, in most cases, result in a clear image. The *Moiré* pattern in the images was not perfectly symmetrical and was usually overlapping with the main frequencies of the image. The filtered image still retained the pattern, but the edges of the buttons were blurred. The input image, image in the spectrum, used filtering mask, and filtered image can be seen in figure 6.2. Due to the ineffective filtering, this method was not used.




(a) Spectrum filtering input (b) Spectrum filtering output

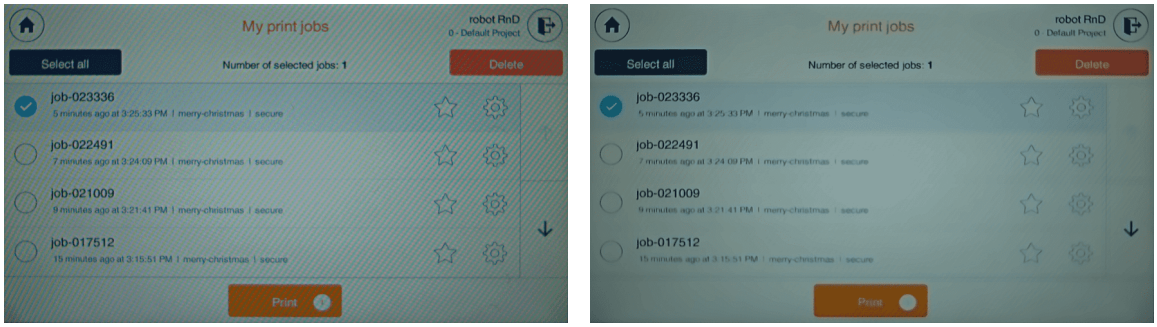
(c) Spectrum filtering mask (d) Input in spectrum

Figure 6.2: Spectrum filtering process example

[2] https://docs.scipy.org/doc/scipy/tutorial/fft.html

Filtering in the spectrum is implemented in the file *filtering_spectrum.py*.

**Non-local means filtering** Because there are similar areas in typical screen images, non-local means denoising was tried to remove the *Moiré*'s pattern. A function that implements this algorithm is a part of *OpenCV* and is called *fastNlMeansDenoisingColored*[3]. The parameters that yielded the best filtered image were: *h=10*, *hColor=30*, *tempWinSize=7*, *searchWinSize=21*. After the filtering, images were usually a little blurred, so an *Unsharp masking* [4] sharpening algorithm was used. The result of this method can be seen in figure 6.3. However, this method's performance is very poor; one image takes about 2 seconds to filter, which is not acceptable in real-time applications.




(a) non-local_before (b) non-local_after

Figure 6.3: Non-local means filtering example

*Non-local Means* filtering is implemented in the file *filtering_nonlocal_means.py*.

## 6.2 *EfficientNetB1* Network Training

*EfficientNetB1* CNN implementation is discussed in section 5.2. The creation of the dataset used for the training of the network is described in section 5.3. The dataset was split into three parts:

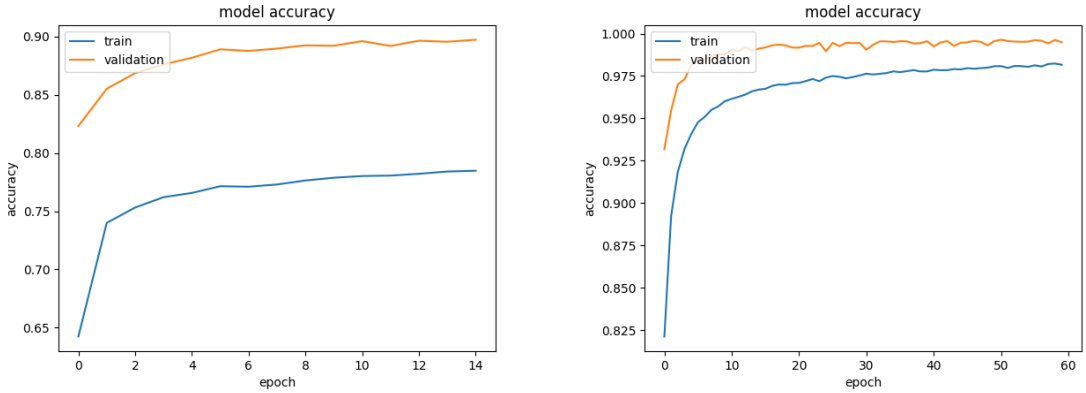
- 10% test data
- 9% validation data
- 81% train data

A function *image_dataset_from_directory* from *Keras*'s *utils* module was used to load, split, resize and batch the images. A contrast shift of 10% was used during training to augment the data.

The training was split into two parts. Firstly, the model was kept frozen, and only the newly added layers were trained for fifteen epochs. The accuracy [5] of the model after this training was 79% on training data and 90% on the validation data. The training accuracy is lower than the validation accuracy because dropout layers are only used during the training process to avoid overfitting and improve model generalisation. The plot

[3] https://docs.opencv.org/4.x/d5/d69/tutorial_py_non_local_means.html
[4] https://scikit-image.org/docs/dev/auto_examples/filters/plot_unsharp_mask.html
[5] (TruePositive+TrueNegative)/Total samples

of the training can be seen in figure 6.4a. Secondly, the last twenty layers were un-frozen, and the model was trained for another eighty epochs. After this training was completed, the accuracy of the training data was 98%, while the accuracy of the validation data was 99%. These results can be seen in figure 6.4b. One epoch took around six minutes to finish on the *GTX 1070*. After training, a classification report was generated, which can be seen in table 6.1. In the table, accuracy is the number of correct predictions divided by the total number of samples. Macro average is the sum of a metric of each class divided by the number of classes. Weighted average considers how many samples each class has, so fewer samples in one class means that its precision, recall and f1-score have less of an impact on the weighted average for each of those metrics [6].




(a) 15 epochs, frozen network

(b) 65 epochs, unfrozen layers

Figure 6.4: Training progress of *EfficientNetB1*

| metric | precision | recall | f1-score | support |
|---|---|---|---|---|
| accuracy | X | X | 0.99 | 15431 |
| macro avg | 0.99 | 0.99 | 0.99 | 15431 |
| weighted avg | 0.99 | 0.99 | 0.99 | 15431 |

Table 6.1: Table showing the accuracy of the trained *EfficientNetB1* model.

## 6.3 Text Extraction Experiments

Text extraction is significantly dependent on the quality of the OCR. In this case, *YSoft*'s internal OCR was used to detect text. In this section, both successful and unsuccessful extractions are shown. In those images, a blue bounding box is a UI element's area, and green bounding boxes are the detected text. The list of words in the top left corner are class suggestions based on the detected text in the UI element. This component cannot be evaluated on its own automatically because no ground truth to which the result could be compared exists. The creation of such ground truth, on a larger scale to provide meaningful result, would be very time consuming. Therefore, only the number of UI elements with suggested

[6] https://towardsdatascience.com/choosing-performance-metrics-61b40819eae1

text classes is calculated. Out of 19590 UI elements, 11172 (57%) contained text information. Out of those elements, which have some text information, 5387 (48.2%) elements were suggested with at least one text class.

### Successful text extraction examples

Text extraction is considered successful when there is some text in the image, and it is read correctly or when there is no text in the image and no text is detected. Figure 6.5 shows examples of successful text extractions and text class predictions. Images 6.5a through 6.5e show text extractions when the buttons both contain and do not contain a pictogram. Image 6.8c shows a modal window which contains buttons with pictograms, whereas image 6.5g shows buttons with text only. Image 6.5h shows a whole screen with all text correctly read.

### Unsuccessful text extraction examples

Text extraction is considered unsuccessful when there is some text in the image, and it is misread or when there is no text in the image and some text is detected. Figure 6.5 shows examples of unsuccessful text extractions. In image 6.6a, text was correctly detected but was misread. The image 6.6b contains no text, but some text was detected. The images 6.6c and 6.6d contain both text and a pictogram but the OCR mistakes the pictogram for a letter. The image 6.6e is also incorrect because the OCR merged numbers in the image together. The image 6.6f contains a whole software keyboard in which most of the letters were read by the OCR but were incorrectly merged. If the OCR includes a pictogram in the detected text area, it is not a big problem because the semantic dictionary usually corrects the error. However, when the OCR detects text in a UI element with only a pictogram, it disables the pictogram detector's ability to find it, therefore causing an error.

## 6.4 Pictogram Detection Experiments

Multiple methods for pictogram detection were tried during the implementation of the pictogram detector. The methods mainly differed in the way edges are detected. Tried edge detectors were the *Sobel* operator, *Laplacian* operator and *Canny*. Out of those methods, edge detection using *Sobel* operator resulted in the smallest number of errors. This component cannot be evaluated on its own automatically because no ground truth to which the result could be compared exists. The creation of such ground truth, on a larger scale to provide meaningful result, would be very time consuming. Therefore, an estimate of at least 90% successful pictogram detection is made, based on the visual evaluation during the implementation of this subsystem. This claim can be substantiated by the low number of classification errors in the evaluation of the whole system 6.7. The figures below show the result of pictogram detection on elements containing text. The yellow bounding box indicates an area in which pictograms are being detected, the green bounding boxes indicate areas where contours were found, and the blue bounding box indicates an area where pictogram was detected.

### Successful pictogram detection examples

Pictogram detection is considered successful when the area contains a pictogram, which is detected without text being a part of that area. Also, when the UI element only contains

text, the correct output is an empty detected area. Figure 6.7 shows UI elements and their binary representation, which were detected correctly.

### Unsuccessful pictogram detection examples

Pictogram detection is considered unsuccessful when the area contains a pictogram which is not entirely inside the output of a bounding box or when text is also part of the output. Also, when the UI element only contains text and the output is not an empty area, the detection is also unsuccessful. Figure 6.8 shows UI elements and their binary representation of failed pictogram detection analysis. The image 6.8a and image 6.8e were incorrectly analysed because the right edge of the button intersects with the detection area. The image 6.8c was incorrectly analysed by the OCR as it did not detect the letter *f*; therefore, pictogram detection determined it to be a pictogram. Image 6.8g contains a lot of noise, as seen in its binary representation, which was mistaken for a pictogram. Overall, most pictogram detection errors are caused by wrong OCR or button edges intersecting with the detection area.

## 6.5 GUI Semantic Analysis Experiments

The GUI semantic analysis cannot be evaluated automatically as no datasets that could be used as a ground truth exist. Therefore, examples of cases where the analysis succeeds and struggles are shown in this section. Also, in cases where the analysis struggled, a pre-processed image is shown to help understand what went wrong. As the expected input of the analysis is a noisy image, filtering algorithms are used before edges are detected. However, during this process, edges with small contrast are also softened, making them invisible for edge detection methods. Larger versions of images shown in this section can be seen in section A.5.

### Successful analysis examples

GUI hierarchy analysis is considered successful when all main GUI sections are detected correctly, and no additional incorrect GUI sections are detected. The figure 6.9 shows examples of successful analyses. Images 6.9a through 6.9c were analysed perfectly. All main GUI sections were also detected in figure 6.9d but the item list was not detected, as was accounted for above. For the reasons mentioned above, the GUI semantic analysis cannot be evaluated automatically on a large number of inputs. Nevertheless, an estimate of 80% successful rate was made.

### Unsuccessful analysis examples

GUI hierarchy analysis is considered unsuccessful when at least one GUI section is incorrectly detected. Figure 6.10 shows examples of unsuccessful analyses, each example showing a different cause. Image 6.10a is considered incorrectly analysed because, even though three sections were detected, the innermost section contains all UI elements. The correct result would be if the inner-most section originated just below the *Scan workflows* button, separating the tab's content from the tab button itself. The error is, in this case, caused by *OpenCV*'s function *findContours* which did not return the correct contours for a correct binary image 6.10b. Image 6.10c was incorrectly analysed because the edge detection process did not detect edges separating GUI sections present in the image. As mentioned

above, the filtering and edge detection is tuned to ignore the noise, which means that edges with small contrast are ignored. Edges were correctly detected for image 6.10e but function *findContours* found incorrect contours. Image 6.10g is an example of edge-less GUI design where this analysis fails. These cases are a grey area and more of a limitation than errors, as the analysis correctly did not find any sections (based on edges), but humans intuitively sense how the GUI is structured.

## 6.6 System Overview

The system is separated into multiple folders and files based on the functionalities it implements. Systems components:

- *GUIElementsAnalyzer/AnnotationsParser.py* - a file containing the class responsible for loading input and creating button cutting
- *GUIElementsAnalyzer/ButtonCutting.py* - a file containing the class storing button cuttings and methods that alter them
- *GUIElementsAnalyzer/config.py* - a file containing configuration settings
- *GUIElementsAnalyzer/draw_bb_utils.py* - a file containing functions that visualise the output of subsystems
- *GUIElementsAnalyzer/hierarchy_analysis.py* - a file containing functions responsible for GUI hierarchy analysis
- *GUIElementsAnalyzer/logger.py* - a file containing logger definition
- *GUIElementsAnalyzer/main.py* - a file containing the main system loop
- *GUIElementsAnalyzer/pictogram_detection_identification.py* - a file containing functions responsible for pictogram detection and identification
- *GUIElementsAnalyzer/text_analysis.py* - a file containing functions responsible for text analysis
- *GUIElementsAnalyzer/utils.py* - a file containing utility functions
- *GUIElementsAnalyzer/test_data* - a folder containing image data and its original annotations
- *GUIElementsAnalyzer/evaluation_data* - a folder containing correct annotations
- *efficientnetb1_training.py* - a file containing *EfficientNetB1* compilation and training functions
- *icons_dataset_creation.py* - a file containing functions responsible for the creation of icons dataset
- *printer_dataset_creation.py* - a file containing functions responsible for the creation of printer dataset
- *dataset* - a folder containing the dataset used for *EfficientNetB1* training

- *utils.py* - a file containing utility functions
- *filtering_nonlocal_means.py* - a file containing filtering experiments using non-local means method
- *filtering_spectrum.py* - a file containing filtering experiments using non-local means method
- *noisy_student_efficientnet-b1.h5* - a file containing pre-trained weight used to make the training faster
- *efficientnetb1_model.h5* - a file containing the weights of *EfficientNetB1* after training
- *README.md* - a file containing information how to run the system and its parameters

## 6.7 System Evaluation

### Accuracy

Individual system components cannot be evaluated independently automatically, as there are no ground truth results with which the output could be compared. The creation of such ground truths on a large scale to provide meaningful results for each component of the system would be incredibly time-consuming. Also, for example, ground truths for UI element classes suggested by text information are highly subjective, as each person might assign an element with different classes. Therefore, to evaluate individual components, the results would have to be checked manually. To provide reasonable results doing the manual evaluation, hundreds of samples would have to be checked, which is also very time-consuming. Therefore, the system is evaluated as a whole, using the annotations of the printer dataset as a ground truth. This evaluates cases where the element contains only a pictogram or both; a pictogram and text. Elements that only contain text are not annotated in that dataset, so only a count of suggested text classes was calculated for them. Also, the hierarchy analysis cannot be evaluated automatically, so it is evaluated manually on its own in section 6.5.

The whole printer screen dataset (1,418 images) and the annotations (61 classes) were used as an input; however, the class information of the element was ignored during the analysis. The system was run normally, and after each image was analysed, the system's output was compared with the annotation class. The following metrics were tracked:

- *total_btn_cuttings* - total number of elements in the input file
- *buttons_with_text* - total number of elements in the input file that contain text
- *text_based_class_suggestions* - number of non-annotated elements for which text class was found
- *total_annotated_btn_cuttings* - total number of annotated elements in the input file (elements with a pictogram and possibly text)
- *correctly_top1_classified_cuttings* - number of elements classified using its pictogram (correct class the same as top 1 prediction)
- *correctly_top5_classified_cuttings* - number of elements classified using its pictogram (correct class is in top 5 predictions)

| | |
|---|---|
| Total button cuttings | 19,590 |
| Total button cuttings with text | 11,172 (57%) |
| Text based class suggestions | 5,387 (48.2%) |
| Total annotated button cuttings | 5,975 |
| Top 1 button classification | 4,197 - 70.2% |
| Top 5 button classification | 572 - 9.6% (79.8%) |
| Text classified buttons | 76 - 1.3% |
| OCR errors | 898 - 15% |
| Classification errors | 232 - 3.8% |

Table 6.2: Table showing the evaluation of the system.

- *correctly_text_classified_cuttings* - number of elements where pictogram classification failed, but text classification suggested the correct class
- *ocr_error* - number of errors probably caused by wrong OCR text detection
- *classification_error* - number of errors probably caused by wrong classification

The result of the evaluation can be seen in table 6.2. 19,590 elements were in the input images. Out of those elements, 11,172 (57%) contained text, and out of those elements with text, 5,387 (48.2%) elements were assigned with classes using the semantic dictionary 5.7. Only 5,975 elements were annotated (61 classes), and all those elements contain a pictogram. Out of those elements, 4,197 - 70.2% were correctly classified, 572 - 9.6% (79.8%) had their ground truth in the top 5 predictions, 76 - 1.3% were incorrectly classified (pictogram) but text-based classification suggested the correct class, 898 - 15% were incorrectly classified due to OCR errors and 232 - 3.8% classified incorrectly due to other errors. Overall, the system achieves 81.1% accuracy in classifying the correct class of the UI element.

### Performance

The performance was measured on the system with the following specifications:

- CPU - Intel i7 7700k @ 4.8Ghz - affects the performance of the whole system, as most CV algorithms run on CPU
- GPU - Nvidia GTX 1070 @ 1911 Mhz - affects the performance of the pictogram classification using *EfficientNetB1*
- Storage - M.2 NVMe SSD - affects image and annotation files loading times
- Network - 1Gbps download/50Mbps upload (wired) - affects communication speed with the OCR service
- OS - Windows 10 64bit

The whole evaluation process, where 1,218 images were analysed, was finished in 11 minutes and 28 seconds. This translates to 1.77 images per second, 28.5 UI elements per second or in other words, one image per 560 milliseconds. As the program ran on a single thread, the CPU usage was around 20%.

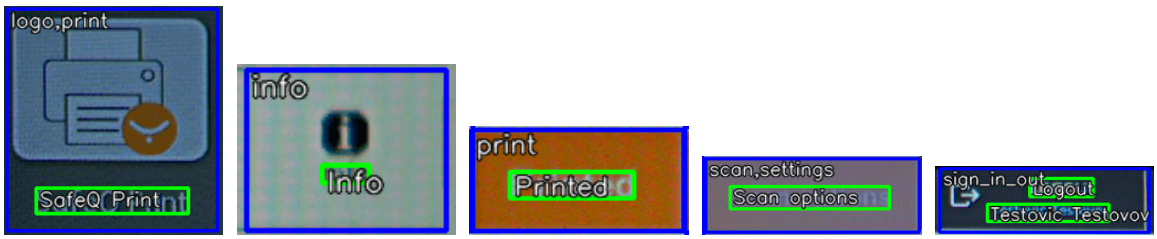
## 6.8 Future Work

The system could be improved by using an OCR capable of correctly detecting letters and numbers in the software keyboard and correctly detecting text in noisy images. Alternatively, an object detection neural network trained on a dataset of software keyboards could be used to detect and exclude them from OCR processing. OCR performance and pictogram detection could be improved by figuring out how to remove the *Moiré* effects in sub 0.1 seconds.

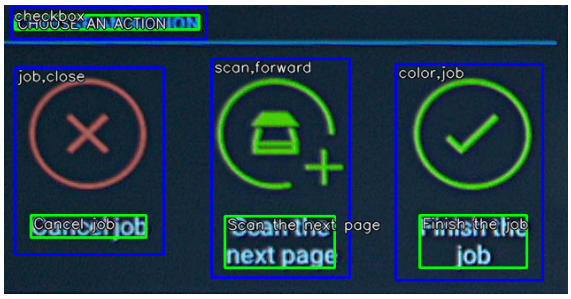
Moreover, the semantic dictionary could be implemented in a more scalable and expandable way, for example, as a web service where users can suggest new synonyms and admins could review the suggestions. Also, an automatic dictionary translation to other languages would save many person-hours. Lastly, the detection and correction of spelling errors caused by wrong OCR analysis would improve the accuracy of text class prediction.

Hierarchy analysis performance could be improved by using an object detection neural network trained on a GUI design dataset like *Rico* 3.2 or *Enrico* 3.2. However, a custom dataset with hierarchy annotations for printer and web GUIs would have to be created. Moreover, hierarchy analysis could be improved by detecting item lists, button groups, and other interconnected UI elements as hierarchy groups.

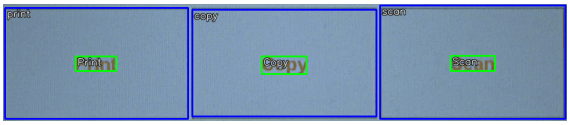
This system enables robotic solutions to identify UI elements on the screen without the need of human interventions. Therefore, the system can be used in collaboration with an application exploration system to create a solution for automatic exploratory testing. An application exploration system controls a robotic device and, with the knowledge of the UI, generates a graph representation of the tested application.



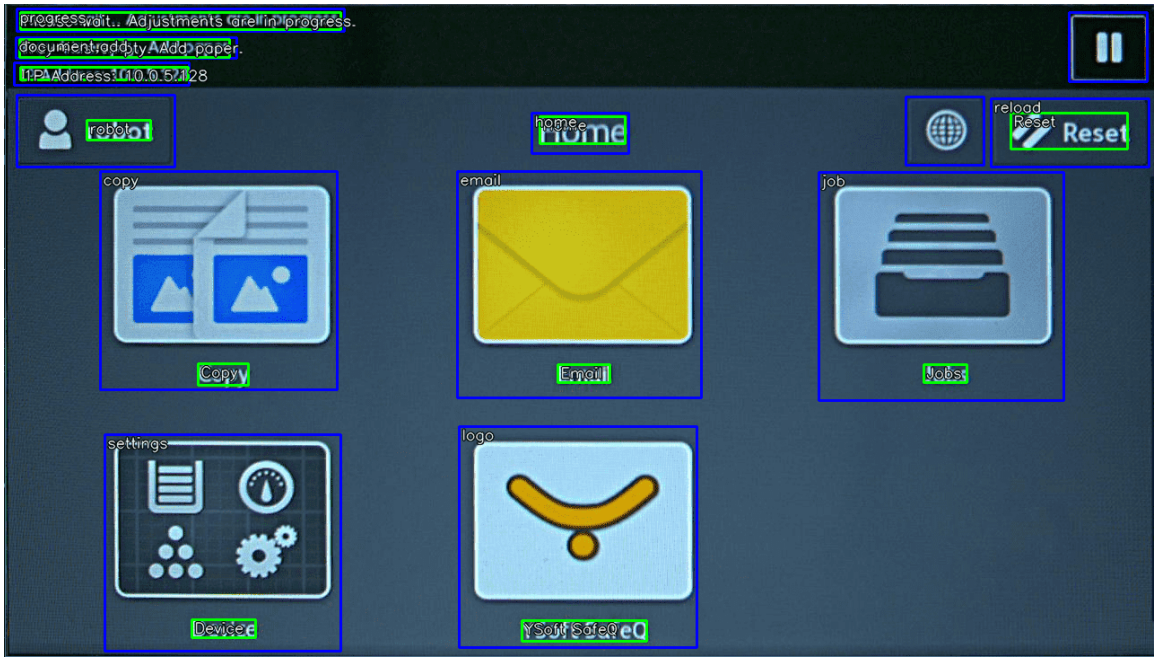

(a) (b) (c) (d) (e)




(f) Modal window

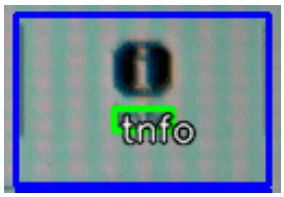


(g) Only text buttons

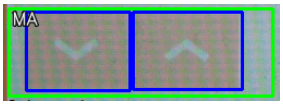



(h) The whole screen detected

Figure 6.5: Examples of successful text extractions



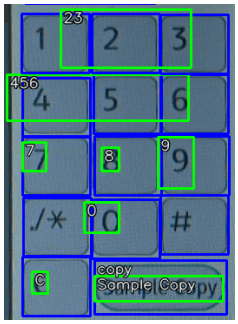

(a) Incorrectly read



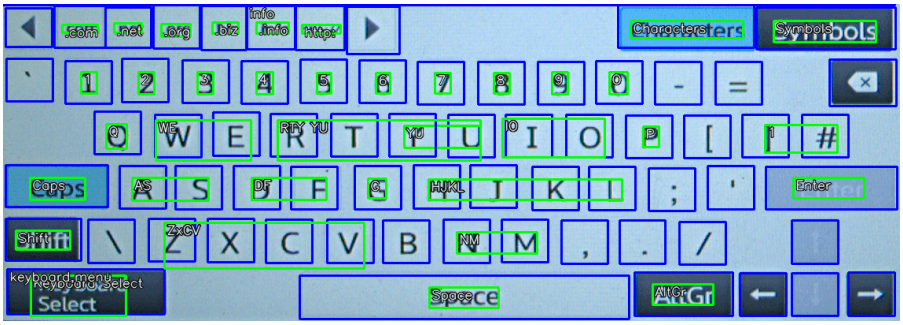

(b) Wrong detection


(c) Wrong detection


(d) Wrong detection

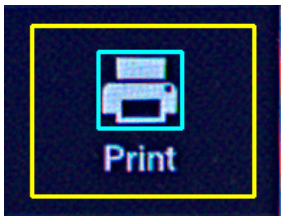



(e) Merged

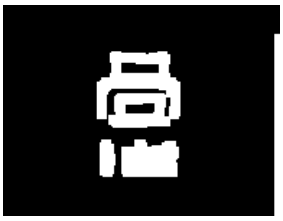



(f) Merged incorrectly, some letters not detected

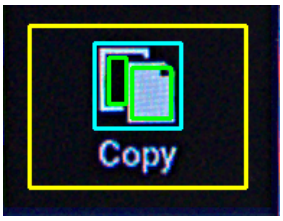
Figure 6.6: Examples of unsuccessful text extractions




(a) Pictogram detected



(b) Binary image of 6.7a




(c) Pictogram detected

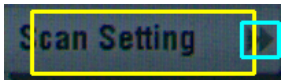
(d) Binary image of 6.7c


(e) Pictogram detected

(f) Binary image of 6.7e


(g) Pictogram detected

(h) Binary image of 6.7g




(i) Pictogram detected

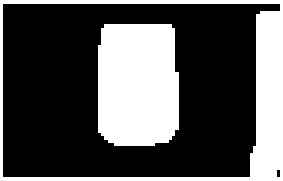
(j) Binary image of 6.7i


(k) Pictogram detected

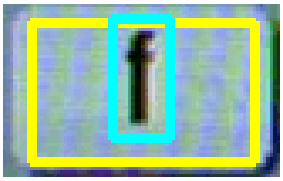
(l) Binary image of 6.7k

Figure 6.7: Examples of successful pictogram detections


(a) Edge detected



(b) Binary image of 6.8a

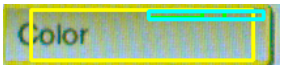



(c) Letter detected

(d) Binary image of 6.8c


(e) Edge detected

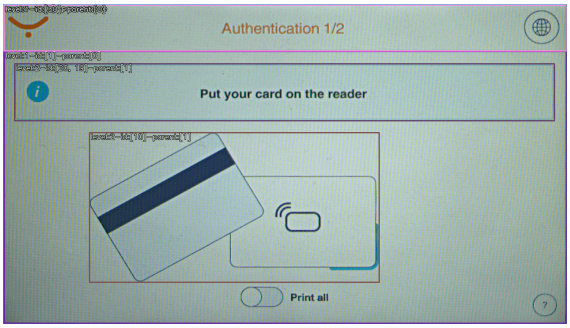
(f) Binary image of 6.8e



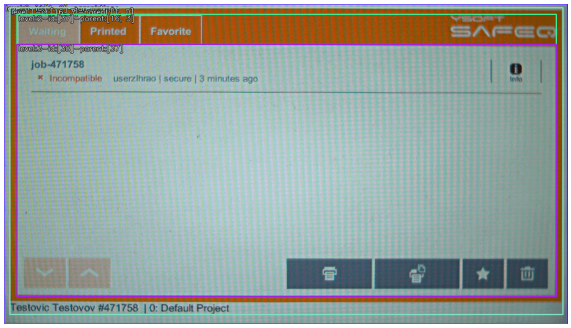

(g) Noise detected

(h) Binary image of 6.8g

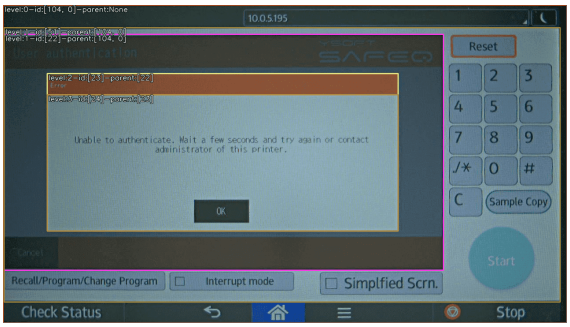
Figure 6.8: Examples of unsuccessful pictogram detections



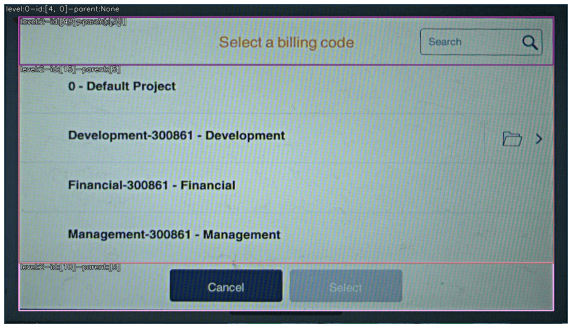

(a) All GUI sections detected




(b) All GUI sections detected

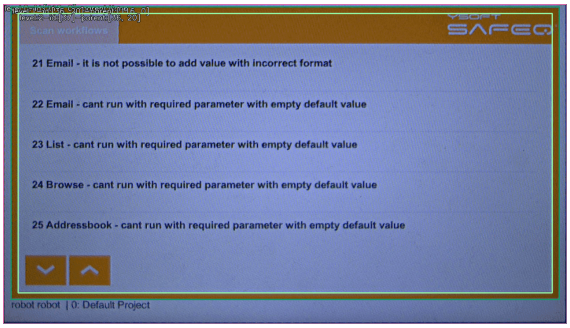



(c) All GUI sections detected

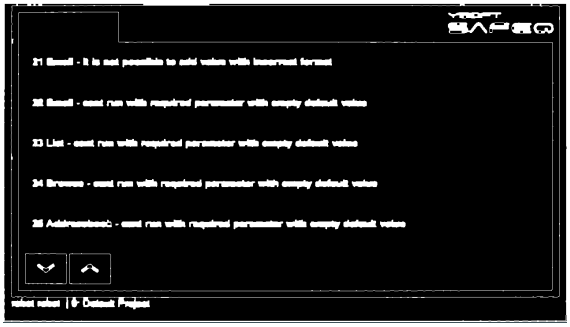



(d) All GUI sections except the item list detected

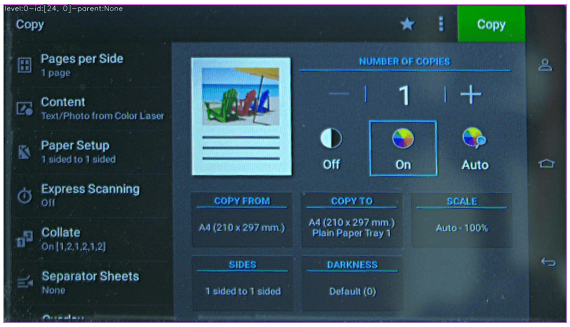
Figure 6.9: Examples of successful hierarchy analyses



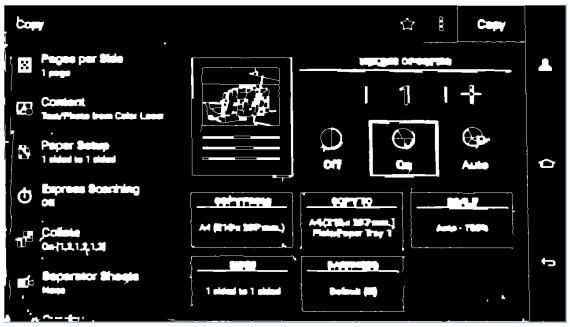

(a) All elements in one hierarchy



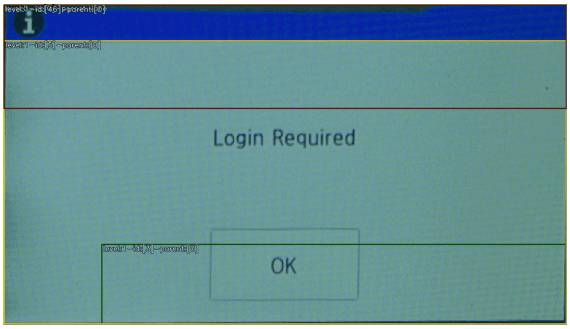
(b) Pre-processed binary representation of 6.10a




(c) No hierarchies found (2 clearly visible)

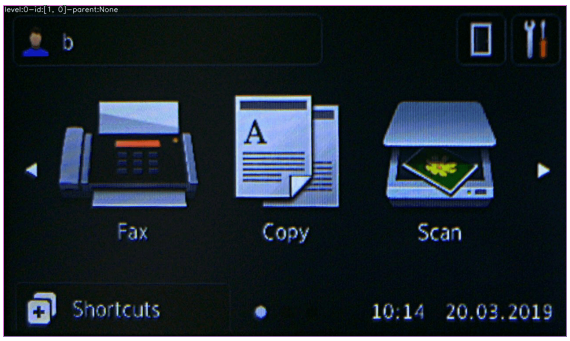


(d) Pre-processed binary representation of 6.10c

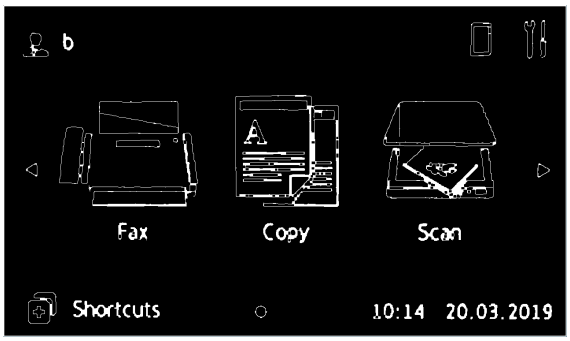



(e) Incorrect hierarchy sections detected.


(f) Pre-processed binary representation of 6.10e




(g) Edge less GUI - no hierarchies found




(h) Pre-processed binary representation of 6.10g

Figure 6.10: Examples of unsuccessful hierarchy analyses

# Chapter 7

# Conclusion

This thesis pursues the issue of graphical user interface (GUI) screen analysis using convolutional neural networks (CNN) and computer vision methods. The thesis aimed to create a system which automatically identifies GUI elements based on pictogram and text information for detected elements in an input image. This aim was accomplished.

In order to create the system, the following components were created: UI elements identifier, text extraction algorithm, pictogram semantic dictionary, and GUI hierarchy analysis. The UI elements identifier uses *EfficientNetB1* CNN to classify pictograms in the input image into 61 classes of frequently used UI components. The CNN is trained on a custom UI element dataset of 120k images of popular design styles. The text extraction algorithm uses *YSoft*'s internal OCR to detect text in an input image. The text is then post-processed and assigned to corresponding UI elements. The pictogram semantic dictionary is a handmade list of words which can be used to describe a pictogram. The dictionary is matched with the extracted text to propose UI element classes based on its text information. Lastly, GUI hierarchy analysis uses traditional CV methods to find and semantically categorise sections in GUI screens.

The resulting system automatically classifies detected pictograms, suggests additional text classes, and separates the GUI screen into hierarchical sections. The system was tested using the aforementioned dataset as a ground truth. For 5,975 annotated UI elements the system achieved top-1 accuracy of 70.2% (4,197), top-5 accuracy of 79.8% (4,769). The semantic dictionary corrected 1.3% incorrectly classified pictograms, resulting in UI elements identification accuracy of 81.1%. 15% (898) elements were identified incorrectly because of OCR related errors. 3.8% of elements were incorrectly identified because of other errors. During the evaluation process, which took 11 minutes and 28 seconds, 1218 images were analysed. Hence, on average, the system processes one image in 560 milliseconds.

The resulting system is subject to a few limitations which should be borne in mind. The primary cause of the system's errors is the inability of the used OCR to correctly read single characters and its occasional incorrect text bounding box placement. Also, the pictogram detection algorithm can mistake the noise in input images for a sought-after object. Lastly, GUI hierarchy analysis is, due to the methods used, not capable of detecting smaller sections like item lists and button groups. Notwithstanding these limitations, the system represents a viable option for a robust automatic UI elements identification. This solution categorises UI elements into 61 distinct classes while other similar solutions separate elements into around 15 classes. Moreover, this system can identify icons from multiple design styles while other solutions specialise in a single design. In addition, the system can process screen images with substantial visual noise, while some other solutions rely heavily

on a clean screenshot. Furthermore, UI elements are separated into hierarchical sections using only visual information while other solutions need the view tree of the application. Finally, the system can categorise UI elements solely based on the text it contains.

Future research work might be undertaken to eliminate the system's limitations and to further improve it. Different OCR solutions could be tested to improve the accuracy of the system. Moreover, research on the pictogram semantic dictionary curated expansion, and automatic translation could be conducted. Furthermore, fast and reliable methods for *Moiré effect* removal could be researched. Also, the GUI hierarchy analysis could be enhanced. Instead of using traditional CV methods, an object detector could be trained on general UI elements datasets. This would enable the system to detect specific sections like item lists and grouped buttons. Lastly, this system might serve as a foundation for an automatic exploratory testing solution. Combining this with an application exploration system could create a solution which generates a graph representation of tested applications, utilising a robotic device.

The creation of the system revealed a dire need for publicly available datasets containing UI elements categorised into very *finely* separated classes. Having access to more diverse datasets would enable further comparisons, thus increasing the credibility of the presented evaluation data. Also, using these datasets, new advanced solutions could be created.

In conclusion, the system in its current state automates UI elements identification, thus eliminating the need for human employees to engage in repetitive, mundane activities.

# Bibliography


[1] Arai, K. and Kapoor, S., ed. *Advances in Computer Vision*. Springer International Publishing, 2020. Available at: https://doi.org/10.1007%2F978-3-030-17795-9.

[2] Beltramelli, T. *Pix2code: Generating Code from a Graphical User Interface Screenshot*. arXiv, 2017. DOI: 10.48550/ARXIV.1705.07962. Available at: https://arxiv.org/abs/1705.07962.

[3] Bhattacharyya, J. *6 mnist image datasets that data scientists should be aware of (with python implementation)*. Dec 2020. Available at: https://analyticsindiamag.com/mnist.

[4] Deka, B., Huang, Z., Franzen, C., Hibschman, J., Afergan, D. 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*. New York, NY, USA: Association for Computing Machinery, 2017, p. 845–854. UIST '17. DOI: 10.1145/3126594.3126651. ISBN 9781450349819. Available at: https://doi.org/10.1145/3126594.3126651.

[5] Deka, B., Huang, Z. and Kumar, R. ERICA: Interaction Mining Mobile Apps. In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. New York, NY, USA: Association for Computing Machinery, 2016, p. 767–776. UIST '16. DOI: 10.1145/2984511.2984581. ISBN 9781450341899. Available at: https://doi.org/10.1145/2984511.2984581.

[6] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. et al. ImageNet: A large-scale hierarchical image database. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, p. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[7] He, K., Zhang, X., Ren, S. and Sun, J. *Deep Residual Learning for Image Recognition*. arXiv, 2015. DOI: 10.48550/ARXIV.1512.03385. Available at: https://arxiv.org/abs/1512.03385.

[8] Huang, G., Liu, Z., Maaten, L. van der and Weinberger, K. Q. *Densely Connected Convolutional Networks*. arXiv, 2016. DOI: 10.48550/ARXIV.1608.06993. Available at: https://arxiv.org/abs/1608.06993.

[9] Huang, T., Yang, G. and Tang, G. A fast two-dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. 1979, vol. 27, no. 1, p. 13–18. DOI: 10.1109/TASSP.1979.1163188.

[10] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J. et al. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model*

*size*. arXiv, 2016. DOI: 10.48550/ARXIV.1602.07360. Available at: https://arxiv.org/abs/1602.07360.

[11] Krizhevsky, A. *The CIFAR dataset*. 2009. Available at: https://www.cs.toronto.edu/~kriz/cifar.html.

[12] Krizhevsky, A., Sutskever, I. and Hinton, G. ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems*. january 2012, vol. 25. DOI: 10.1145/3065386.

[13] Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. 1998, vol. 86, no. 11, p. 2278–2324. DOI: 10.1109/5.726791.

[14] LeCun, Y., Burges, C. and Cortes, C. *The mnist database*. 1999. Available at: http://yann.lecun.com/exdb/mnist/.

[15] Leiva, L. A., Hota, A. and Oulasvirta, A. Enrico: A High-quality Dataset for Topic Modeling of Mobile UI Designs. In: *Proc. MobileHCI Adjunct*. 2020.

[16] Li, F.-F., Johnson, J. and Yeung, S. *Fei-Fei Li, justin johnson, Serena Yeung - Stanford University CS231N*. 2019. Available at: http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture09.pdf.

[17] Liu, T. F., Craft, M., Situ, J., Yumer, E., Mech, R. et al. Learning Design Semantics for Mobile Apps. In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: Association for Computing Machinery, 2018, p. 569–579. UIST '18. DOI: 10.1145/3242587.3242650. ISBN 9781450359481. Available at: https://doi.org/10.1145/3242587.3242650.

[18] Moran, K., Bernal Cárdenas, C., Curcio, M., Bonett, R. and Poshyvanyk, D. *Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps*. arXiv, 2018. DOI: 10.48550/ARXIV.1802.02312. Available at: https://arxiv.org/abs/1802.02312.

[19] Nguyen, T. A. and Csallner, C. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, p. 248–259. DOI: 10.1109/ASE.2015.32.

[20] Oliphant, T. *NumPy: A guide to NumPy* [USA: Trelgol Publishing]. 2006–. [Online; accessed <today>]. Available at: http://www.numpy.org/.

[21] OpenCV. *Open Source Computer Vision Library*. 2015.

[22] Patel, K. *Architecture comparison of Alexnet, vggnet, ResNet, inception, DenseNet*. Towards Data Science, Mar 2020. Available at: https://towardsdatascience.com/architecture-comparison-of-alexnet-vggnet-resnet-inception-densenet-beb8b116866d.

[23] Raj, B. *A simple guide to the versions of the inception network*. Towards Data Science, Jul 2020. Available at: https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202.

[24] Simonyan, K. and Zisserman, A. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv, 2014. DOI: 10.48550/ARXIV.1409.1556. Available at: https://arxiv.org/abs/1409.1556.

[25] Srivastava, R. K., Greff, K. and Schmidhuber, J. *Highway Networks*. arXiv, 2015. DOI: 10.48550/ARXIV.1505.00387. Available at: https://arxiv.org/abs/1505.00387.

[26] Sun, X., Li, T. and Xu, J. UI Components Recognition System Based On Image Understanding. In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2020, p. 65–71. DOI: 10.1109/QRS-C51114.2020.00022.

[27] Szegedy, C., Ioffe, S., Vanhoucke, V. and Alemi, A. *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*. arXiv, 2016. DOI: 10.48550/ARXIV.1602.07261. Available at: https://arxiv.org/abs/1602.07261.

[28] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. et al. *Going Deeper with Convolutions*. arXiv, 2014. DOI: 10.48550/ARXIV.1409.4842. Available at: https://arxiv.org/abs/1409.4842.

[29] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z. *Rethinking the Inception Architecture for Computer Vision*. arXiv, 2015. DOI: 10.48550/ARXIV.1512.00567. Available at: https://arxiv.org/abs/1512.00567.

[30] Tan, M. and Le, Q. V. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv. 2019. DOI: 10.48550/ARXIV.1905.11946. Available at: https://arxiv.org/abs/1905.11946.

[31] Van Rossum, G. and Drake, F. L. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN 1441412697.

[32] Xie, M., Feng, S., Xing, Z., Chen, J. and Chen, C. UIED: a hybrid tool for GUI element detection. In:. November 2020. DOI: 10.1145/3368089.3417940.

[33] Zhang, X., Zhou, X., Lin, M. and Sun, J. *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices*. arXiv, 2017. DOI: 10.48550/ARXIV.1707.01083. Available at: https://arxiv.org/abs/1707.01083.

[34] Zhang, X., Greef, L. de, Swearngin, A., White, S., Murray, K. I. et al. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021.

# Appendix A

# Additional Figures

## A.1 List of Classes

add, alert, back, black_white, bluetooth, calendar, camera, card, chat, checkbox, close, color, confirm, copy, decrease, delete, document, down, dropdown, edit, email, emoji, favorite, filter, folder, forward, fullscreen, help, home, info, keyboard, language, location, lock, logo, menu, microphone, pause, phone, power, print, profile, progress, radiobutton, reload, scan, search, send, settings, share, shop_cart, sign_in_out, slider, switch, text_field, time, up, usb, visibility, web, wifi

## A.2 Semantic Dictionary

```
{'add': ['add', 'insert', 'sum', 'append', 'add-on', 'attach', 'adjoin',
        'join', 'affix', 'connect', 'include', 'prepend', 'zoom-in',
        'zoom in', 'new', 'plus'],
'alert': ['alert', 'aware', 'caution', 'warn', 'observant', 'canny',
          'notify', 'error'],
'back': ['back', 'retreat', 'reverse', 'previous', 'former', 'backwards',
         'withdraw', 'backtrack', 'prior', 'undo', 'return'],
'black_white': ['black_white', 'white', 'black', 'monochrome', 'greyscale',
                'grey', 'black and white', 'white and black'],
'bluetooth': ['bluetooth', 'handsfree', 'roaming'],
'calendar': ['calendar', 'date', 'date picker', 'date-pick', 'date-picker',
             'anniversary', 'appointment', 'meeting'],
'camera': ['camera', 'photo', 'record', 'shoot', 'lens', 'cinema', 'movie',
           'film', 'video', ],
'card': ['card', 'identification', 'game', 'payment', 'pay', 'credit',
         'cash', 'authentication', 'credentials', 'paid', 'finance', ],
'chat': ['chat', 'message', 'conversation', 'chatter', 'converse', 'talk',
         'speak', 'communication', 'note'],
'checkbox': ['checkbox', 'tickbox', 'check', 'uncheck', 'choose'],
'close': ['close', 'shut', 'cancel', 'closed', 'decline', 'no', 'cancel',
          'revoke', 'quit', 'exit', 'leave', 'abandon', 'retire'],
'color': ['color', 'hue', 'colouring', 'paint', 'colour', 'palette', 'can',
          'colorful', 'color-picker', 'color picker', 'red', 'green',
```

```
          'blue', 'yellow', 'cyan', 'magenta', 'bw', 'finish'],
'confirm': ['confirm', 'ok', 'done', 'verify', 'validate', 'affirm',
            'approve', 'assure', 'yes'],
'copy': ['copy', 'copied', 'duplicate', 'imitate', 'imitation', 'repeat',
         'replicate', 'reproduce', 'rewrite', 'fake'],
'decrease': ['decrease', 'minus', 'zoom out', 'diminish', 'less', 'smaller',
             'small', 'lessen', 'depreciate', 'shrink'],
'delete': ['delete', 'destroy', 'exclude', 'remove', 'cut out', 'omit',
           'cut', 'drop', 'erase', 'trash', 'backspace'],
'document': ['document', 'paper', 'log', 'report', 'cite', 'chronicle',
             'newspaper', 'certificate', 'pdf'],
'down': ['down', 'downward', 'downgrade', 'declining', 'descending',
         'falling', 'dropping', 'download'],
'dropdown': ['dropdown', 'popdown', 'pulldown', 'submenu', 'sub-menu',
             'pop-down', 'pull-down', 'listbox'],
'edit': ['edit', 'revise', 'update', 'pencil', 'pen', 'draw', 'comment',
         'change'],
'email': ['email', 'e-mail', 'e mail', 'gmail'],
'emoji': ['emoji', 'smile', 'face', 'smiley', 'sad', 'avatar', 'avatar'],
'favorite': ['favorite', 'star', 'remember', 'save', 'saved', 'favourite',
             'like', 'heart', 'preferred', 'template', 'templates',
             'favorited', 'favourited'],
'filter': ['filter', 'order', 'sort', 'funnel', 'descending'],
'folder': ['folder', 'file', 'file-picker', 'folder-picker', 'filepicker',
           'folderpicker', 'store', 'save', 'store', 'stored'],
'forward': ['forward', 'right', 'move', 'fast-forward', 'skip',
            'fastforward', 'next', 'next page', 'redo', 'proceed',
            'follow'],
'fullscreen': ['fullscreen', 'maximize', 'maximise', 'enlarge', ],
'help': ['help', 'question', 'support', 'inquire', 'about', 'aid', 'assist',
         'assistance', 'advice', 'advise', 'guide'],
'home': ['home', 'return', 'go-home', 'house'],
'info': ['info', 'information', 'notify', 'about', 'help', 'check'],
'keyboard': ['keyboard', 'type', 'write', 'console', ],
'language': ['language', 'translate', 'translator', 'dictionary', 'speak',
             'speaker', 'sound', 'localization', 'localisation',
             'languages'],
'location': ['location', 'gps', 'position', 'locate', 'area', 'region',
             'spot', 'locality', 'navigation', 'navigate'],
'lock': ['lock', 'key', 'unlock', 'password', 'pin', 'code', 'dpassword',
         'authentication'],
'logo': ['logo', 'brand', 'icon', 'symbol', 'ysoft', 'y-soft', 'hp',
         'xerox', 'teams', 'km', 'safeq', 'osa', 'osaa'],
'menu': ['menu', 'option', 'select', 'toolbar', 'submenu', 'application',
         'applications', 'popup'],
'microphone': ['microphone', 'speak', 'mute', 'voice', 'mic', 'recorder',
               'transmitter'],
'pause': ['pause', 'stop', 'halt', 'interrupt'],
```

```
'phone': ['phone', 'call', 'dial', 'hang-up', 'contact', 'call up', 'ring',
          'buzz', 'ring up', 'telephone', 'fax'],
'power': ['power', 'start', 'stop', 'sleep'],
'print': ['print', 'imprint', 'printer', 'printed'],
'profile': ['profile', 'person', 'figure', 'silhouette', 'portrait', 'user',
            'username'],
'progress': ['progress', 'loading', 'loaded'],
'radiobutton': ['radiobutton'],
'reload': ['reload', 'refresh', 'reset'],
'scan': ['scan', 'scanner', 'ocr', 'scanning'],
'search': ['search', 'inspection', 'exploration', 'investigate', 'inspect',
           'examine', 'check', 'magnifier', 'browse'],
'send': ['send', 'commit', 'post', 'ship'],
'settings': ['settings', 'options', 'tools', 'set-up', 'setup', 'service',
             'fixing', 'fix', 'cog', 'cogged', 'screwdriver', 'wrench',
             'setting', 'utility', 'device'],
'share': ['share', 'upload', 'shared', 'uploaded'],
'shop_cart': ['shop_cart', 'shopping', 'buy', 'cart', 'shopping-cart',
              'basket', 'punnet', 'cart', 'shop'],
'sign_in_out': ['sign_in_out', 'login', 'register', 'sign-in', 'sign-out',
                'logout', 'leave', 'disconnect', 'registration',
                'authentication', 'log out', 'log in'],
'slider': ['slider'],
'switch': ['switch'],
'text_field': ['text_field, text-field, text, text-area,textbox, text-box'],
'time': ['time', 'duration', 'alarm', 'clock', 'stopwatch', 'history'],
'up': ['up', 'top', 'upwards'],
'usb': ['usb', 'wired', 'connector', 'connection'],
'visibility': ['visibility', 'visible', 'hidden', 'show', 'hide'],
'web': ['web', 'website', 'internet', 'globe', 'server', 'servers'],
'wifi': ['wifi', 'connection', 'signal', 'internet', 'connect', 'wi-fi'],

'job': ['job', 'jobs'],
'waiting': ['waiting'],
'print_mode': ['simplex', 'duplex', 'original', 'default', 'stapling',
               'punching', 'sides', 'mode']}
```

## A.3 System Input - UI Element Detections Example

```
<annotation>
  <folder/>
  <filename>printer/1.png</filename>
  <source>
    <database>Unknown</database>
    <annotation>Unknown</annotation>
    <image>Unknown</image>
  </source>
  <size>
    <width>1280</width>
    <height>724</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>StaticText</name>
    <bndbox>
      <xmin>325.2275390625</xmin>
      <ymin>437.06640625</ymin>
      <xmax>937.20361328125</xmax>
      <ymax>502.0888671875</ymax>
    </bndbox>
  </object>
  <object>
    <name>StaticImage</name>
    <bndbox>
      <xmin>553.9541015625</xmin>
      <ymin>241.9990234375</ymin>
      <xmax>723.0124969482422</xmax>
      <ymax>387.3433532714844</ymax>
    </bndbox>
  </object>
  <object>
    <name>Button</name>
    <bndbox>
      <xmin>507.673828125</xmin>
      <ymin>634.6279296875</ymin>
      <xmax>769.2936096191406</xmax>
      <ymax>710.8189544677734</ymax>
    </bndbox>
  </object>
</annotation>
```

## A.4 System JSON Output Example

```
{"name": [1],                          // hierarchy section id
 "coordinates": [x,y,w,h],             // hierarchy section coordinates
 "parent_id": None,                    // parent hierarchy id
 "buttons": [],                        // list of buttons in this section
 "children": [{                        // list of nested hierarchy sections
   "name": [9],                        //
   "coordinates": [x,y,w,h],           //
   "parent_id": 1,                     //
   "buttons": [{                       //
     "original_label": "Button",       // input class of the button
     "xmin": int,                      // starting x position
     "ymin": int,                      // starting y position
     "xmax": int,                      // ending x position
     "ymax": int,                      // ending y position
     "button_text": [{                 // text of the button
       "text": "Close window",         // text value
       "boundingBox": {                // bounding box of the text
         "position": {                 //
           "x": float,                 // start x pos of text bb
           "y": float},                // start y pos of text bb
         "size": {                     //
           "width": float,             // width of the text bounding box
           "height": float}}}],        // height of the text bounding box
     "button_class": "close",          // most probable class of the pict
     "top_class_predictions":[         // top five probable predictions
       ["close", 0.999981880],         // name of the class and its prob
       ["back", 0.0000151827125],
       ["sign_in_out", 0.0000027493],
       ["up", 0.00000092499],
       ["alert", 0.00000009206],
     "text_class": ["close"],          // list of text class predictions
     "rel_pictogram_area": [x,y,w,h]// coordinates of the pictogram
}]}]}
```

## A.5 GUI Hierarchy Analysis Examples

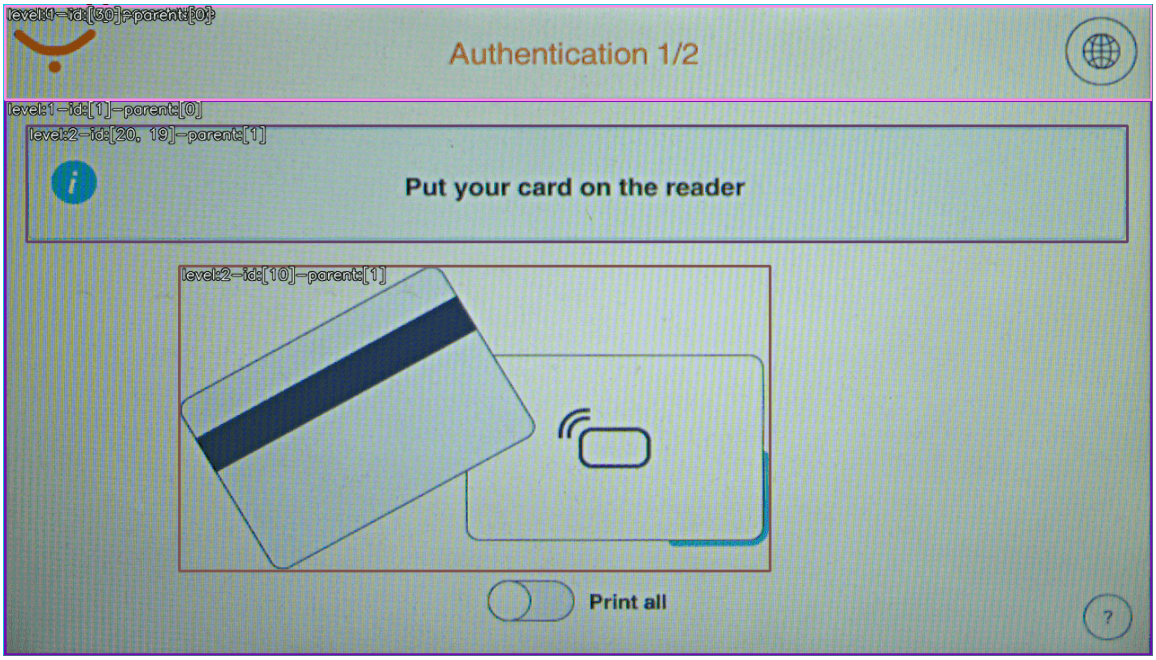



Figure A.1: All GUI sections detected

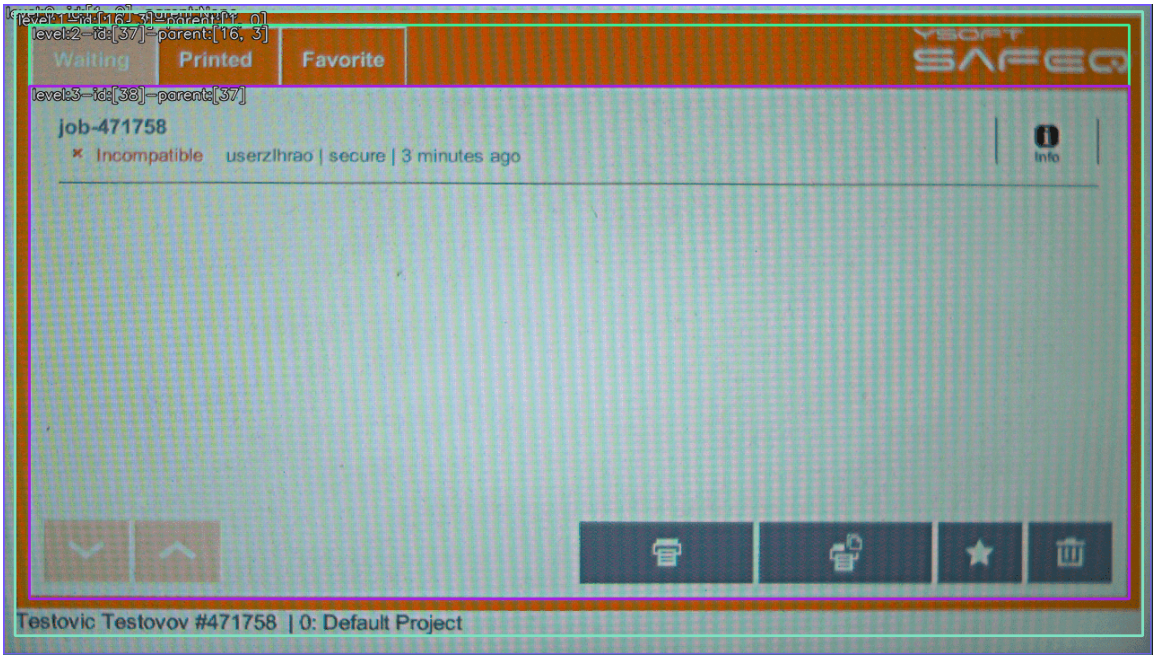



Figure A.2: All GUI sections detected

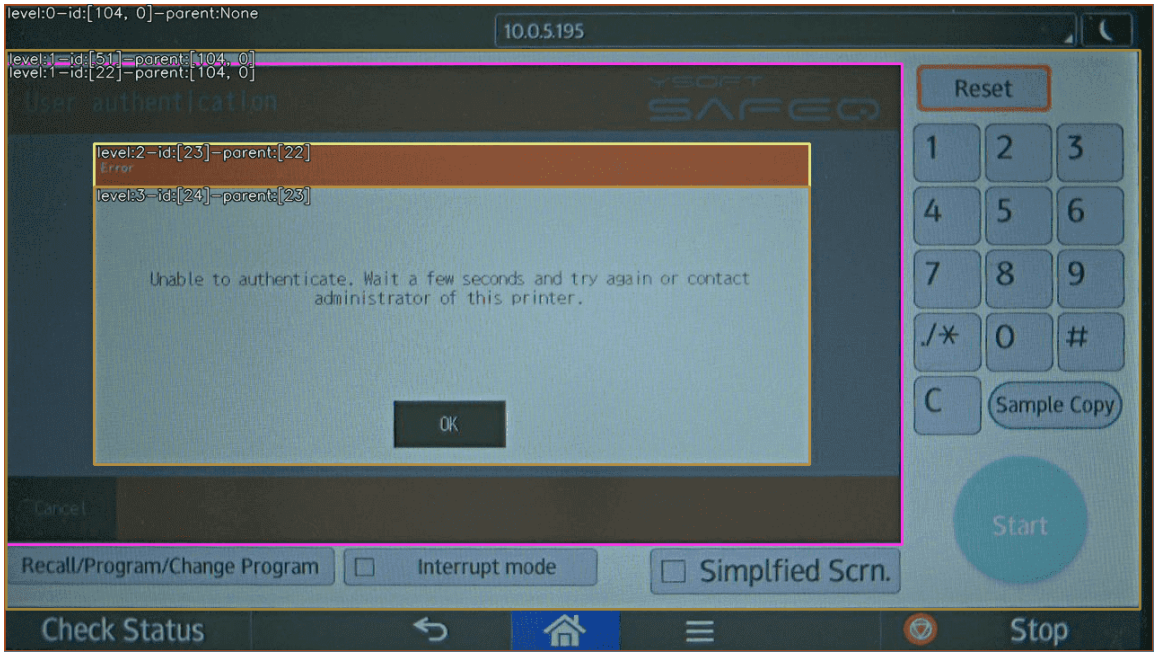



Figure A.3: All GUI sections detected

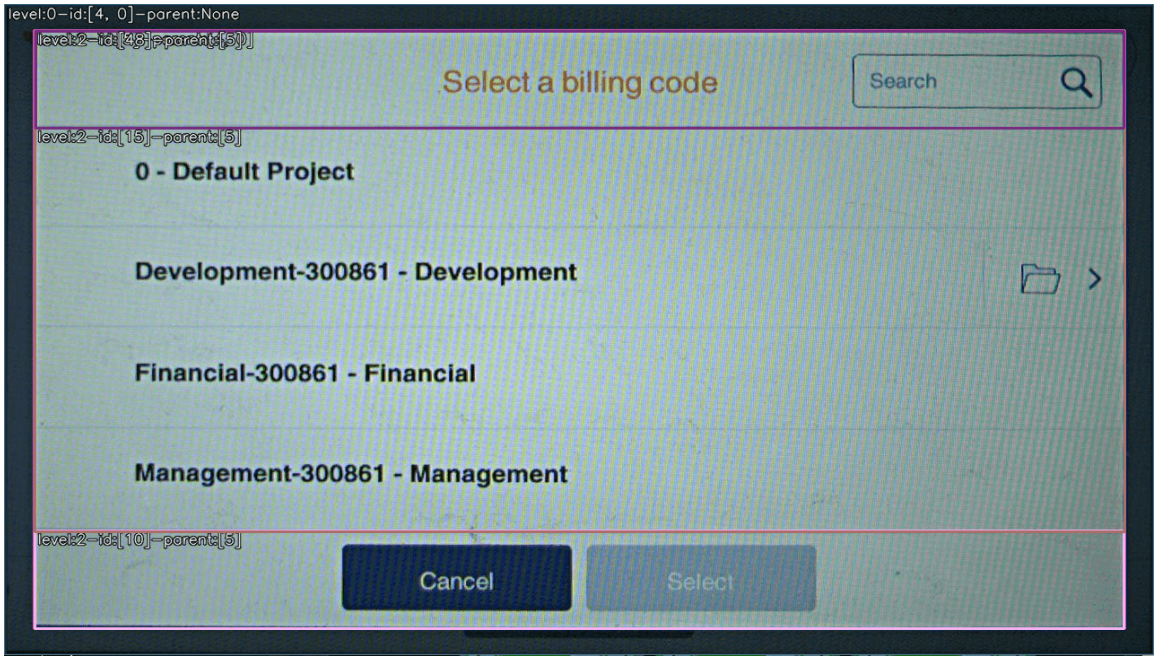



Figure A.4: All GUI sections except item list detected

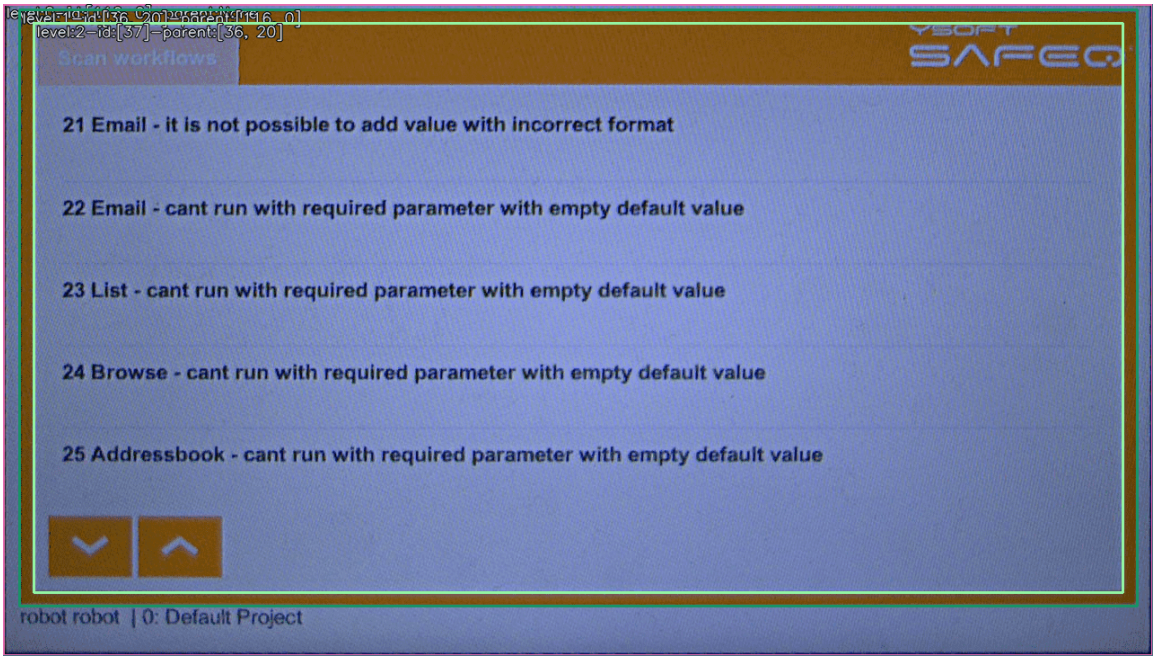



Figure A.5: All elements in one hierarchy

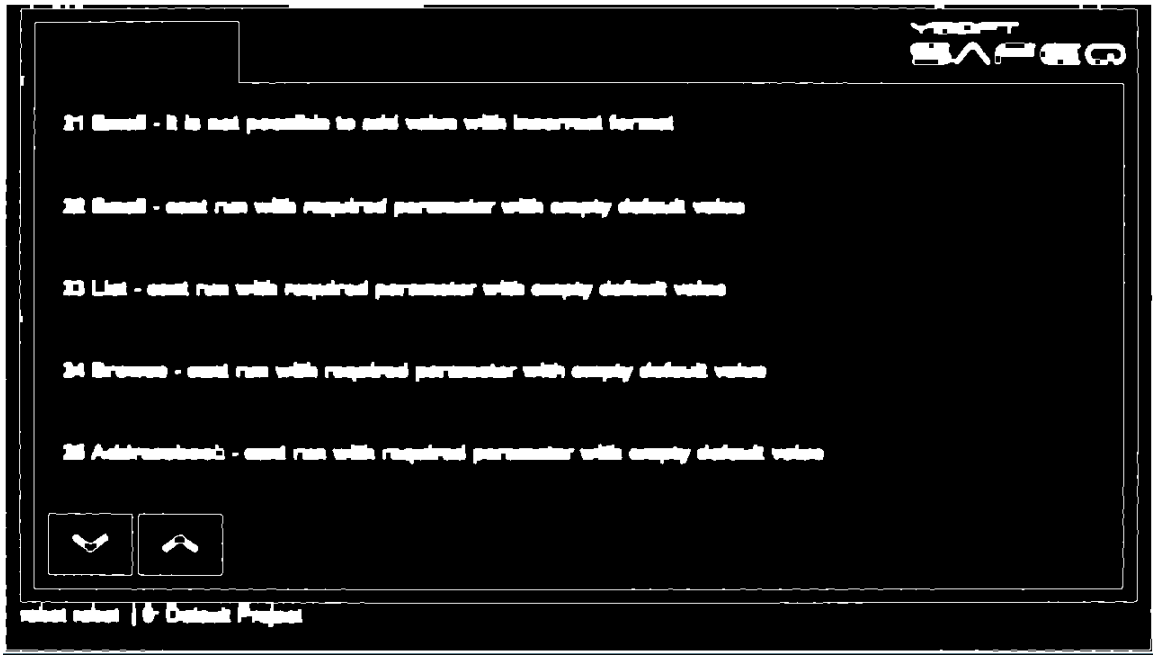


Figure A.6: Pre-processed binary representation of A.5

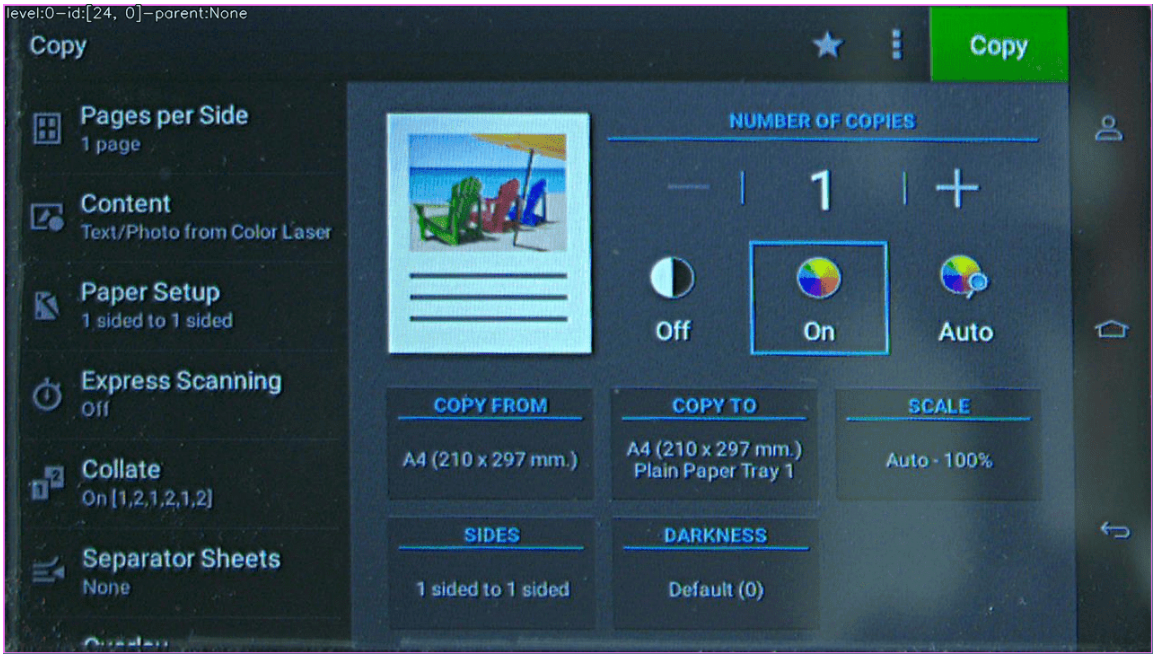



Figure A.7: No hierarchies found (2 clearly visible)

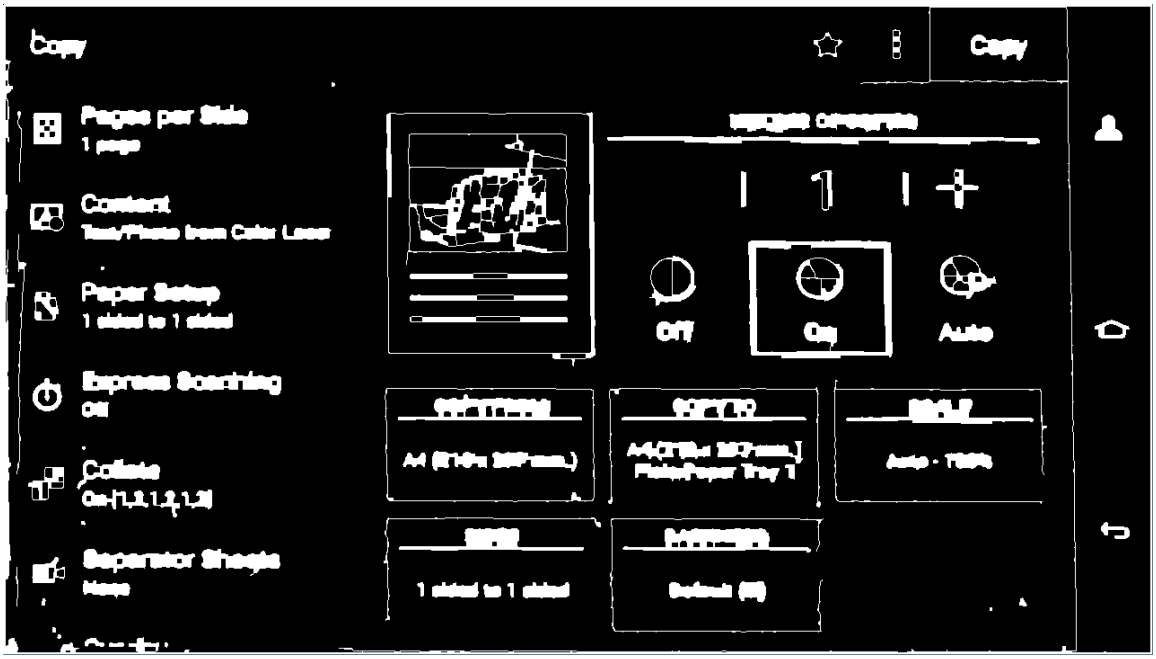


Figure A.8: Pre-processed binary representation of A.7

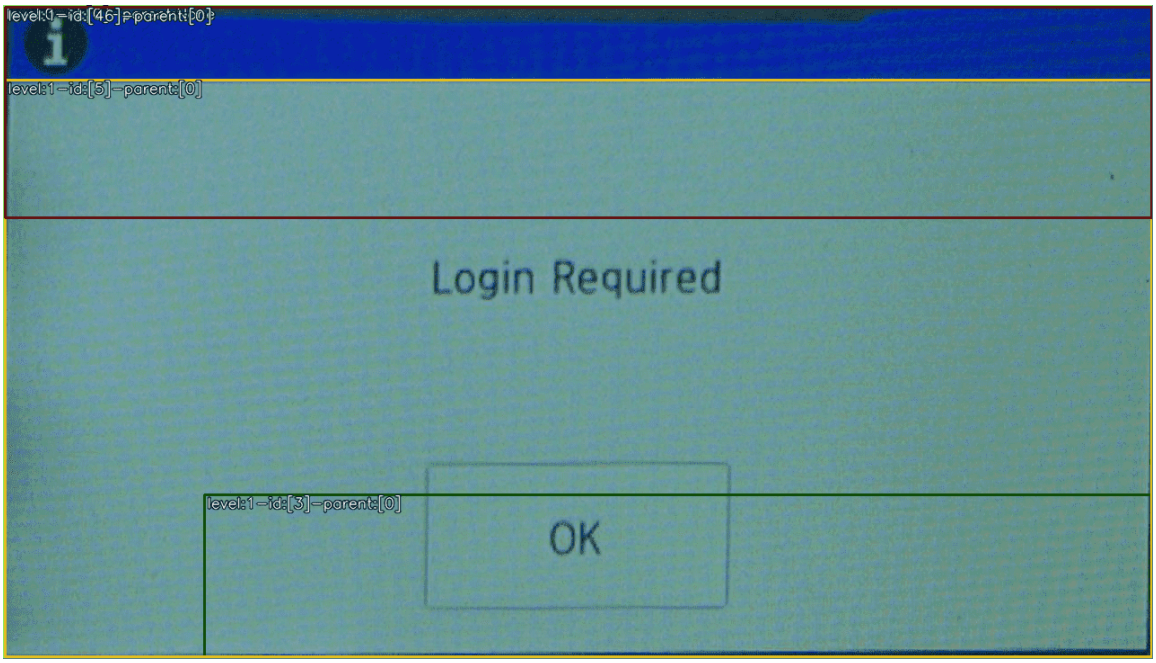



Figure A.9: Incorrect hierarchy sections detected


Figure A.10: Pre-processed binary representation of A.9

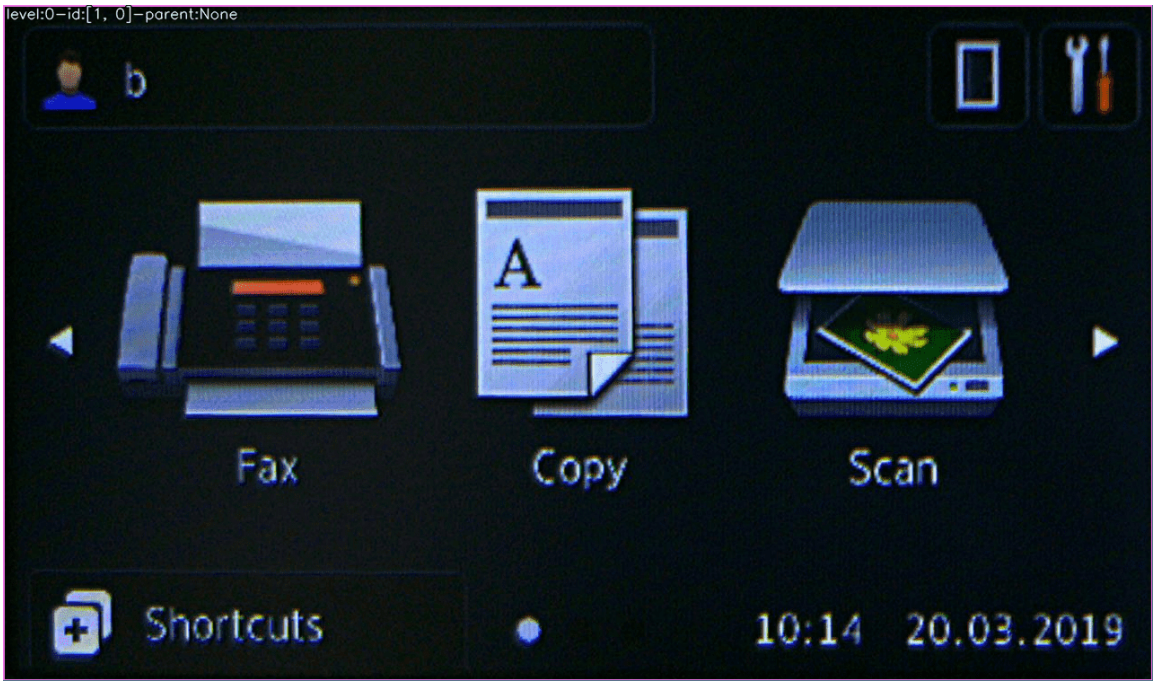



Figure A.11: Edge less GUI - no hierarchies found

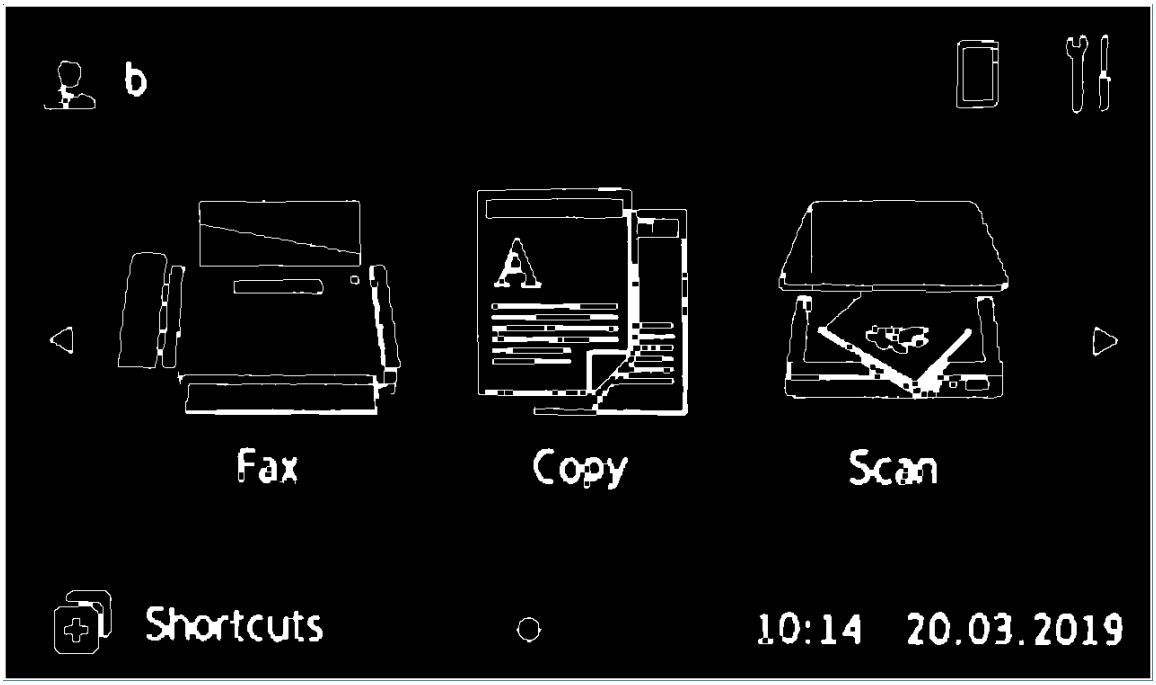



Figure A.12: Pre-processed binary representation of A.11