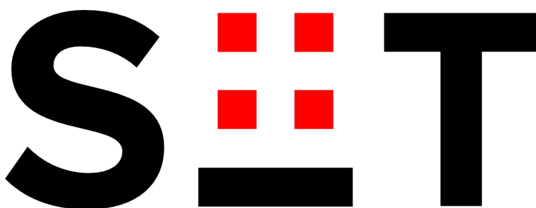


Development of an AI/CV algorithm for recording and tracking user actions on the SIT Alemira Virtual Labs platform

Master's Thesis submitted to the

SIT - Schaffhausen Institute of Technology

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Software Engineering

Computer Vision Algorithms

presented by

Aleksandr Skakun

under the supervision of

Prof. Bertrand Meyer

co-supervised by

Dr. Denis Zholobov

June 2022

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Aleksandr Skakun
Schaffhausen, 5 June 2022

Abstract

This paper presents a description and results of the development of a solution to the User Interface Components Detection problem using Computer Vision algorithms. To solve this problem, test data sets were generated (in total, more than 60,000 images, including their augmented copies (resizing, color, displacement, stretching)), and 17 different models of object detection in the image have been developed and tested. Among them the most successful was selected - a three-stage composite model for recognizing 6 different types of User Interface Components (button, checkbox, folder, icon, text, toggle-switch). The first part of the model is a fast binary classifier for detecting an object in the picture. The second part is a classifier for determining the type of the object. The third part is a detector for recognizing text on the object image. Each of the obtained models gave an accuracy of more than 85%.

To work with the developed model, a simple UI interface was created in the form of a web application, which provides the ability to record user activity (where each user step is recorded and processed using the model described above), manually edit the received activity logs, as well as "passing the log" - step-by-step performing the actions described in the log with their automatic comparison (using image comparison algorithms).

This web application is made in a generalized form, which allows you to supplement it and embed its functionality into other applications using the API.

Application of the results

This dissertation is an industrial thesis made in collaboration with SIT Alemira Virtual Labs. All the software developed in this thesis is planned, if sufficient efficiency is achieved, to be used on this platform.

SIT Alemira Virtual Labs offers a feature called "Lab recorder", which allows the developer to go through the necessary training scenario in a lab environment, while the software keeps track of all his actions and transforms them into human-readable instructions. And later, the steps of this scenario can be tracked when the student passes through the lab in the same way.

According to an agreement with SIT Alemira Virtual Labs, the idea of this master's thesis is to develop specialized Computer Vision based tools/algorithms for tracking user actions in a virtual environment. So this functionality considered to be implemented:

- Recognize which button the user clicks;
- Recognize what he enters and in which fields;
- Check these actions in accordance with the requirements of the teacher;

Depending on the scenario, these actions must either be translated into human-readable instructions (for a laboratory registrar) or verified by comparison with prerecorded actions for tracking and evaluation.

Contents

Contents v

1 Introduction 1

2 Competition Analysis and State-of-the-Art 3
2.1 Old-fashioned approach . . . 4
2.1.1 Template Matching Method . . . 4
2.1.2 Canny Edge's Detection Method . . . 5
2.1.3 Histogram of Oriented Gradients (HOG) . . . 6
2.2 Advanced Machine Learning approach . . . 8
2.2.1 Haar Cascades . . . 8
2.2.2 Multiscale Detection . . . 8
2.2.3 Region-based Convolutional Neural Network (R-CNN) . . . 10
2.2.4 YOLO (You Only Look Once) . . . 11
2.3 Composite models . . . 11
2.3.1 User Interface Element Detector (UIED) . . . 11
2.3.2 Automatic Code Generation Models . . . 12
2.4 Non-CV-based models . . . 13

3 Software Architecture and Design 15
3.1 Web Application Architecture . . . 15
3.2 Database Architecture . . . 17

4 Implementation 19
4.1 Dataset generation . . . 20
4.2 Choosing the architecture of a Computer Vision model . . . 21
4.2.1 Simplified Model's Architecture . . . 21
4.2.2 Sequential Model's Architecture . . . 24
4.2.3 Probability-based Mapping Detection Algorithm . . . 27
4.2.4 UIED Detector Modification . . . 28
4.2.5 Web Application Prototype . . . 29

5 Testing algorithms and results 35
5.1 Detector testing and image comparison . . . 35
5.2 Testing results . . . 37

6 Conclusions **39**
6.1 Further plans 40

Acronyms **41**

Bibliography **43**

Chapter 1

Introduction

Computer vision methods in the modern world are used in a huge number of different fields - from medicine (for example, detecting signs of tumors from images of patients' tissues) to traffic control, and not only. However, some areas still remain almost untouched, and one of these areas is the control of the activities of PC users. It is necessary to monitor user activity in such tasks as:

- Training in the use of software;
- Control of the employee's activity at the workplace;
- Development of parental control systems;
- Reverse engineering;
- etc.

These problems are quite popular and are considered in many companies to improve the quality of work and productivity of employees, reduce lost time, as well as to improve the User Experience of employees.

The solution of these problems usually occurs without the participation of machine learning and computer vision algorithms and manages with standard locking mechanisms, therefore, **the purpose of this work** was to solve these problems using computer vision algorithms and evaluate the effectiveness of such a solution. If the effectiveness of the solution is proven, it will open up new tools for solving the tasks, which reinforces **the relevance of the current work**. All algorithms were developed in the context of the task of "learning to use software" in collaboration with SIT Alemira Virtual Labs. Unlike the other problems discussed above, the selected problem has a number of sub-tasks, most of which can be solved using ML and CV:

1. Recording (by the teacher) the sequence of actions necessary for students to repeat in order to master the skills of interaction with the program;
2. Recognition of objects that were interacted with at each step;
3. Logging of recorded and processed activity and putting it into open access with the possibility of viewing and repeating;

4. Recording by the user of a sequence of actions according to the instructions of the teacher's log;

5. Comparison of each corresponding user step and log;

6. Making a final assessment based on the results of comparing the log and user activity.

A generalization of these sub-tasks in the context of computer vision can be called the "User Interface Components Detection" problem. This problem is difficult to solve mainly because the objects on the screen are extremely diverse, and patterns are too rare for them. Various objects, such as a "button" and an "icon", may be physically indistinguishable for both a computer and a human being, which leads to difficulties in preparing training samples for the model. Therefore, this part of the work - data engineering for training data - is the most important, along with the development of the architecture of the computer vision model.

As a result, this dissertation sets the **following tasks**:

1. Planning and development of the solution architecture, constraints, description of the classes considered and predicted by models and the differences between them.

2. Creation and markup of training, validation and test data sets.

3. Development and testing of computer vision models; finding the optimal model for this task;

4. Development of an algorithm for reading user actions in real time;

5. Development of an algorithm for logging processed information about the user's action and saving logs.

6. Development of an algorithm for evaluating and comparing logs with user actions;

7. Development of a Database architecture for storing information about logs, users, etc.

8. GUI development and testing for using the models described above, in the form of a web application;

9. (As an additional step) An attempt to integrate a web application with the SIT Alemira Virtual Labs system.

There are consistent thorough descriptions of each step, as well as the results of models working with graphs and tables in this report. Also, there is a description of the architecture of the web application (API) and the features of its implementation.

Chapter 2

Competition Analysis and State-of-the-Art

Currently, there are 4 main approaches to solving the UI components detection problem [Chen et al., 2020b]:

- The "old-fashioned" approach, without the use of machine learning;
- "Deep Neural Network" approach, using deep machine learning and advanced machine learning algorithms;
- Combinational approach;
- Programs that do not use image processing.

Each of the approaches has its pros and cons, which should be considered in the context of the task. For example, the articles [Nguyen and Csallner, 2015], [Dixon and Fogarty, 2010] consider the task of reverse engineering applications, which is characterized by increased accuracy to specific objects on the screen. At the same time, almost no attention is paid to image processing time. The article [Chen et al., 2020a] discusses the problem of recognizing text objects on the screen to help the visually impaired. This problem implies special attention to text recognition, while less attention is paid to the exact position of the object and its type.

In this dissertation, the problem of logging committed actions is considered, so it is worth highlighting the main points on which to focus attention:

- **The classification accuracy of the type of object on which the click was made.** This is important in the context of this task, because the user while using the logger to pass the test, must quickly navigate the UI and understand exactly where he should click and what exactly he should do.
- **The text recognition accuracy of the text attached to the object.** Most often, UI objects have a text description - whether it's the name of an icon or an inscription on a button. To improve the user's orientation on the screen, this item is also especially important.
- **The speed of work.** Unlike other tasks, logging should be fast enough, because the average number of clicks per minute from a user, depending on the task, can be from

10 to more than 300 (for example, for pro gamers) [Street et al., 2018], [Chadwick-Dias et al., 2002]. The speed of work is a serious limitation for working with neural networks.

- **Accuracy of image comparison.** The comparison of information from the log and the action performed by the user should also happen quickly enough, and most importantly accurately. The poor quality of the comparative model will make the entire application useless.

On the other hand, the following points are slightly less important for this task:

- **The exact shape of the object**. This task allows us to consider an object not by its exact shape, but by its relation to the environment. If the object is detected less accurately, for example, with extra edges, then this will not greatly affect the UX and the understanding of the log. Moreover, manually correcting the shape of the object will allow the author of the log to correct serious errors.

- **Variety of object types**. Despite the requirement for high accuracy of object type classification, the very presence of a large number of types is not fundamental for the current task. It should be sufficient to generalize all the objects encountered, but not large enough to confuse the user. Moreover, given the importance of the speed of the algorithm, fewer classes, in general, will mean less computing time.

Based on the points presented above, each of the approaches described at the beginning of the chapter will be considered later in this chapter.

2.1 Old-fashioned approach

The first thing to consider is the oldest and most diverse approach to detecting user interface components. There are a huge number of different algorithms for detecting these components in the image, but here will be highlighted the three most popular:

1. Using templates [OpenCV, 2021];

2. Filtering (Laplacian, Gaussian, etc.) and contouring, or "Canny Edge's detection" [Canny, 1986]

3. Histograms of Oriented Gradients [Dalal and Triggs, 2005]

2.1.1 Template Matching Method

The simplest of detection algorithms is simply finding objects in the image by templates. Any object similar to this template (with a degree of similarity above some threshold value) will give a positive outcome. The advantage of this approach is the ease of use and the possibility of successful and fast operation for fixed-size buttons. About 25 years ago, a similar approach was showing excellent results due to highly standardized User Interfaces in operating systems.

Unfortunately, if the button size is not set, then the complexity of finding an object by the template increases quadratically (complexity is $O(width^2 * height^2)$, where *width* and *height* are shapes of the image), and therefore the time spent increases. In addition, the variety of objects in the current OS makes this approach impractical for most tasks. However, it is still

used for reverse engineering of some mobile applications (for detecting the buttons of the built-in keyboard, for example), and it can also be used to compare images and to find attachments to images. An example of template matching can be seen in Figure 2.1 below.

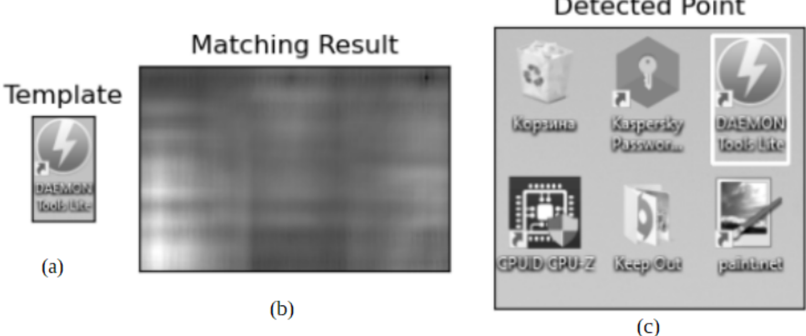


Figure 2.1. Applying the Template Matching method to detect an object (a), the icon, in the image (c). Image (b) is a similarity map in grayscale, on which black represents 100% match.

2.1.2 Canny Edge's Detection Method

The method of filtering and contouring the image is more interesting and complex. It is based on "simplifying" the image and bringing it to a form in which all objects will be represented only by contours. An example of this is shown in Figure 2.2.

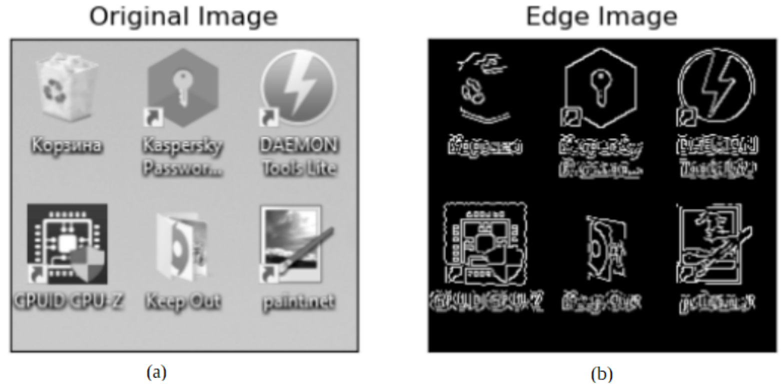


Figure 2.2. Cleaning the image (a) from noise using filtering windows (Gaussians) and applying Canny Edge's method to detect the edges of objects (b).

After this transformation, all objects in the image become separable, which allows using almost any detection method with increased accuracy. This approach is often used to detect non-text elements and is called "Canny edge's detection" [Canny, 1986]. Its meaning is to use Gaussian filtering to reduce noise and further multi-step filtering to determine the boundaries

of objects by searching for the maximum gradient at each point. Usually, improved versions of this technique are used for images [Todd and Mingolla, 1983], which increase the probability of finding edges and the accuracy of their location, however, in the context of UI, where there are no shadows, overlaps, and other interference, the basic mechanism can also be applied. The result of the algorithm is a set of sets of coordinates of points describing the contours found.

This method of image simplification makes it possible to detect boxes containing an image using the usual merging of intersecting contours.

This method is characterized by high speed, however, as mentioned above, it does not go well with text recognition (Optical Character Recognition, OCR). Nevertheless, it allows you to find the boundaries of objects with sufficient accuracy. An example of the operation of such a model is shown in Figure 2.3.

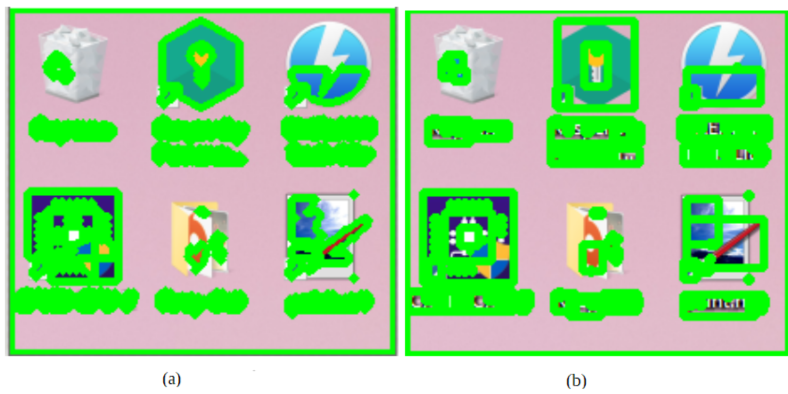


Figure 2.3. Contouring objects based on Canny edge's detection method and the use of hierarchical union (a) and simplification of contours by rectangular approximation (b). There are 134 contours in the images.

Also, this method, unlike the first one, can no longer confirm the presence of the wanted object in the image, however, it allows you to detect objects on it in order to further compare these objects separately using standard image comparison methods.

2.1.3 Histogram of Oriented Gradients (HOG)

This method, like the previous one, is based on simplifying the image before using it directly for object detection. The Histogram of Oriented Gradients method reduces the dimension of the image and removes unnecessary colors and noise from it, leaving only contours. Despite the fact that both the Canny and HOG methods use gradient search, this method is more complex and computationally expensive (due to its relative novelty), however, it produces clearer results, which are often used to train simple machine learning algorithms (such as K-means and Support Vector Machine Classifier [Mallick, 2016]).

The HOG method has 3 main stages:

1. Definition of image gradients;

2. Making a histogram of gradients;

3. Normalization.

In the first stage, horizontal and vertical gradients for each pixel are calculated using window filters. Then these gradients are transformed into polar coordinates (value, angle). At the same time, in the context of this task, it is more advantageous to consider the modulus of gradient values, since the brightness of the object and the brightness of the background may vary relative to each other. An example of the results of the first stage is shown in Figure 2.4.

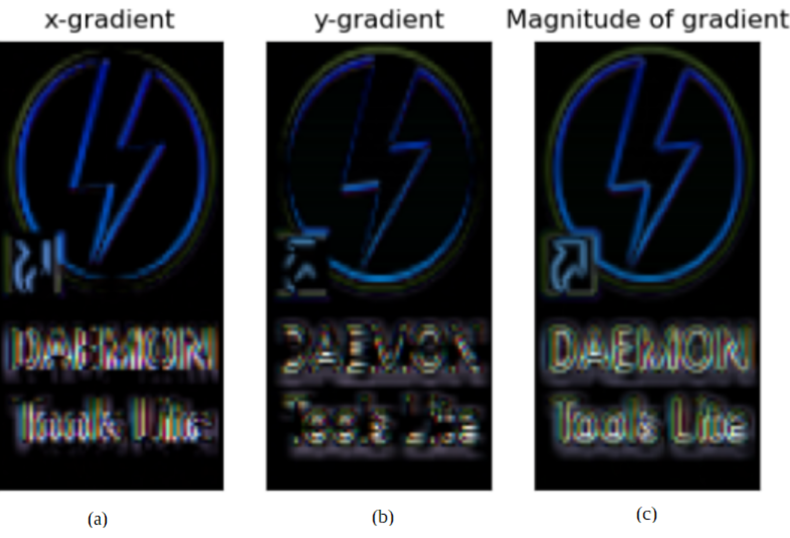


Figure 2.4. Image gradients (in absolute values) for the colored image. (a) - horizontal gradient, (b) - vertical gradient, (c) - full gradient value (normalized).

In the second stage, pixels are grouped into square boxes (most often 8x8 pixels), and the resulting angles are divided into a number of intervals (most often 9). Further, for each group of pixels, the corresponding magnitudes of gradients are added to each interval - histograms are obtained.

In the third stage, the resulting histograms are normalized. This is necessary for preventing the dependence of the histogram size on the brightness of the image and for increasing the quality of further detection. Most often, normalization does not occur pixel by pixel, but by pixel groups, for example, 16x16 pixels. This reduces the total size of the data array.

The resulting data array is a set of gradients of size ($width/16 * height/16 * 9$), which is several times smaller than the original image size, devoid of noise and normalized. Such data is much more convenient for simple machine learning models and gives excellent results in classification and detection.

It is worth noting that data engineering is extremely important for image classification tasks. Over the past 8 years, there has been a serious leap in the development of computer vision technologies precisely because of the focus on preprocessing images and isolating important features from them before classification. For example, best-performance algorithms (based on Convolutional Neural Networks) produce results many times better than other models precisely because of the specific algorithm for obtaining important features in the image (see below).

2.2 Advanced Machine Learning approach

At the time of writing the thesis, there are several varieties of Machine learning algorithms for solving computer vision problems. The most popular of them are:

1. Haar Cascades;
2. Region-based Convolutional Neural Network [Choudhury, 2020]:
3. YOLO (You Only Look Once);

Here is a brief description of these methods below:

2.2.1 Haar Cascades

The method of detecting objects in an image using Haar cascades is a recognized method of detecting parts of a face with only a small number of false positives. Despite the fact that it is, in general terms, a classifier, not a detector, it works extremely fast and does not waste a lot of resources, which makes it a useful and convenient tool for real-time detection. But this method can also be used to recognize other objects that have some specific patterns.

Haar cascades initially represent sets of masks (functions) superimposed on the target image (object) in different sections. Figure 2.5 shows several such masks.

Figure 2.5. Haar-like features (a), (b) and applying one of them to the button (c).

Each function calculates the pixel brightness difference between the black and white parts of the function. The main idea is to find on the images of objects from the training sample such functions (and their positions) that give the most distinguishable from a random result. After finding the set of these functions, the next step is to create a cascade directly. It consists in using the classifier “decision tree” with boosting “AdaBoost” to improve the accuracy of prediction. The depth of such a cascade can reach hundreds of vertices. The meaning of this boosting is to correct tree errors with the help of the next decision tree trained on erroneous data from the previous one. Then the following tree is trained on the errors of this tree and etc., thereby a chain of correcting trees of different weights is obtained.

When an image is fed to the cascade, it applies the selected Haar functions to it and passes the values obtained from them to the root of the first tree. After the decision is made by the first tree, it is “checked” by another tree, etc., as a result, the cascade gives a weighted answer.

2.2.2 Multiscale Detection

It follows from the description above that the cascade works for a single image of an object, however, due to its speed of operation, this function can also be applied to a generalized image

(for example, a PC desktop). The mechanism for searching (detecting) objects in this image is based on the Multiscale Detection technique. Multiscale detection considers an object, as the name implies, on several scales, according to the so-called Pyramid, as showed in the Figure 2.6.

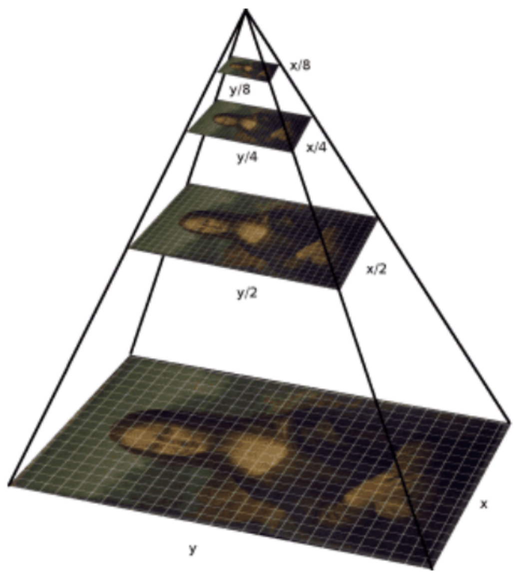


Figure 2.6. Representation of an image pyramid. It contains images at different scales. Image has taken from [Rosebrock, 2020].

After making up the pyramid, many windows of different sizes and shapes are selected, and each of them runs through the images in the pyramid. This is called a “sliding window". At each step of this window, the image in it is fed to the specified classifier, and if it finds a target object in the image, all pixels (taking into account the scale) are marked as containing the object.

After passing through all the windows, the areas (most often rectangular) containing the target object are marked up at all scales. The final result is a set of coordinates of these areas in the image.

This method is essentially a method of converting a classifier into a detector and is applicable not only to the Haar Cascade classifier but also to other classifiers (including those described in the previous approach). It allows you to process images in parallel, but the important point is that the speed of work directly depends on the speed of the classifier, the number of classes, the number of layers of the pyramid, and the size of the sliding window. It is also worth noting that this method is not the only one used to select the ROI (regions of interest) in the image [Gu et al., 2009], [Arbelaez et al., 2009].

Figure 2.7 shows the result of Multiscale Detection using the Haar Cascade, using the example of a desktop image of a Windows 10 PC. Unfortunately, the results are not impressive. Despite the fact that the cascades were trained on a large sample of data (about 20,000 images), the variety of icons is too large and it was difficult to find common features “head-on” for this classifier.

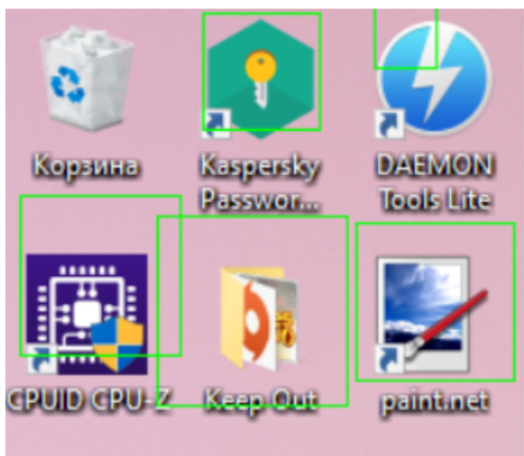


Figure 2.7. The result of applying the Haar Cascades method in conjunction with Pyramid of Sliding Windows. The objects found in the image are highlighted with green frames. A wide variety of icons prevents the cascade from finding patterns in the shape and brightness of objects.

2.2.3 Region-based Convolutional Neural Network (R-CNN)

Region-based CNN is a whole family of algorithms (R-CNN, Fast R-CNN, Faster R-CNN, etc.) based on deep learning, having one distinctive feature - the neural network is used here not to directly predict the class of an object, but to generate a set of features from an image. In general, the pipeline in R-CNN is represented as follows [Girshick et al., 2014]:

1. Extract a number of region proposals;
2. Normalize them (transform to the fixed-length feature vector);
3. Compute CNN features;
4. Classify regions;

Despite the general pipeline, depending on the model, each step can be performed differently. For example, the first step was initially performed using a sliding window. Newer methods involve the use of feature trees (pieces of an object) for each of the classified objects and a complex mechanism based on HOG to find regions of interest (ROI) in the image [Arbelaez et al., 2009]. There are about 2000 such ROI, “candidates” for the title of an object (they are called bounding boxes), on average. Other methods, including those used in medicine, are described here [Carreira and Sminchisescu, 2012], [Arbelaez et al., 2014], [Cireşan et al., 2013].

This is followed by normalization, which can also occur by various methods, including the HOG already described above, and the reduction of the obtained regions of various shapes to the standardized form required by CNN for input data. Most often this is done by stretching and deforming the region.

After that, a multi-layer convolutional neural network for feature extraction is applied. Finally, at the last stage, a string of features obtained from CNN is fed to a classifier (most often Support Vector Machine Classifier, or next deep neural network) to determine the class and its

probability. After applying the classifier to each of the regions, their probabilities and locations are compared, if necessary merging or discarding one of the intersecting regions in which the score is less.

In 2014, this method appeared and showed outstanding results, opening the way for the development of computer vision technologies. Most often, this method is used for detecting live objects in images, but it is increasingly rarely used for real-time detection due to the cost of classification time. The processing time of one image by this model can be up to 40 seconds [Girshick, 2015]. The following method is often devoid of this problem.

2.2.4 YOLO (You Only Look Once)

YOLO is an ultra-fast composite model (45-150 fps), while not conceding in accuracy to many competitors described in the first approach. At its core, YOLO is an ultra-large CNN capable of simultaneously processing the entire image, without the need to divide it into parts (regions) and check each part separately. However, even this model requires some image preprocessing [Redmon et al., 2016].

To understand the detection pipeline, it is worth introducing the concept of **"Anchor-box"** - this is a certain area of the image characterized by its coordinates, width, and height, as well as the probability of an object being in it.

At the very beginning, the neural network, like all the others, should detect the approximate position of objects. Unlike the other models mentioned above, this model does this simultaneously for all classes and across the entire image, thus obtaining bounding boxes.

Next, the image is divided into SxS anchor boxes by grid. For each such box, YOLO considers "Intersection Over Union" (IoU) - the highest overlap of the bounding box and anchor box, divided by non-overlap [Christiansen, 2018]. The obtained values are compared with some threshold, and if they are higher, the anchor box is considered to potentially contain an object, otherwise, it is thrown out of consideration.

After that, the results are grouped and fed as a tensor to the CNN input. The model's response is a tensor of the same size containing predictions about the probability of an object belonging to a particular class in each of the anchor boxes.

2.3 Composite models

Composite models are a combination of a variety of models designed to offset each other's shortcomings, improve predictive ability, or preprocess data before training. In the context of the UI Components Detection task, such models are the most popular, and this work will also provide confirmation of this. For this work, it is important to note the following types of models:

1. User Interface Element Detector (UIED) [Xie et al., 2020], [FlowShare, 2021];
2. Automatic Code Generation Models - a special type of model aimed at generating front-end code from an image (screenshot, figure).

2.3.1 User Interface Element Detector (UIED)

UIED is a complex model consisting of a combination of 5 different DNN and CNN algorithms (such as Fast R-CNN, Faster R-CNN, YOLOv3, CenterNet [Zhou et al., 2019]), as well as several

algorithms for finding boundary regions. This hybrid approach is essentially divided into 2 parts - recognition of text data and non-text objects. This is necessary because different models show the best results for each of the object types.

In the context of recognizing objects on the screen (buttons, icons, switches, scrollers, etc.), they can be presented in a huge number of different shapes and sizes, while the text is quite monotonous. The variety of icons makes the task of classification extremely non-trivial for neural networks prone to retraining. At the same time, old-fashion algorithms can help to deal with this problem.

Figure 4 shows that, for example, HOG makes the text difficult to read, while the icon becomes very recognizable, even if the background was partially similar to it. At the same time, Optical Character Recognition (OCR) models can quickly find texts, but they are difficult to augment to find other objects.

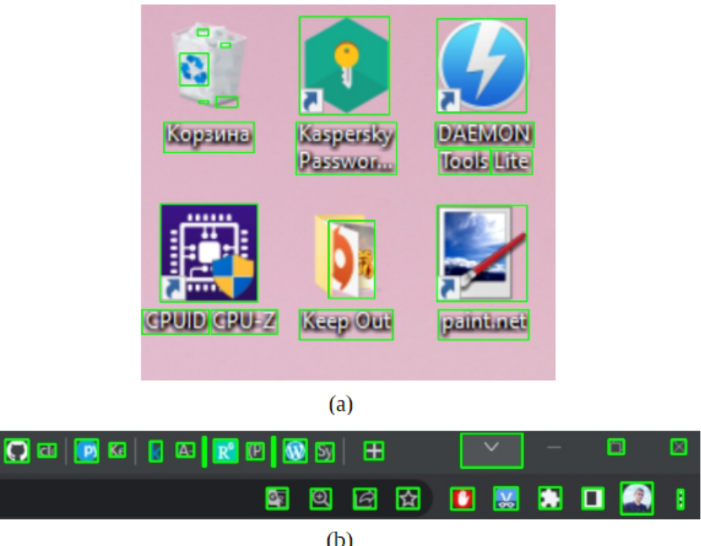


Figure 2.8. Applying the UIED model to detect objects on the desktop (a) and in the browser window (b).

2.3.2 Automatic Code Generation Models

One of the most frequent applications of component detection models is code autogeneration. Unlike the issue discussed in this paper, code autogeneration is mainly used in web design, where the shapes and sizes of objects are limited and are often correlated with the sizes of screens. As a consequence, the models do not address the main issue for current work - the classification of UI elements of the PC. Nevertheless, the consideration of the detecting part of such models is also interesting, and these models themselves are competitive with the current work.

From the considered examples of such models [Biniam, 2020] ,[Nguyen and Csallner, 2015] it was found that all the same models are used for object detection - Faster R-CNN, Single Shot Detector, ResNet (R-CNN).

2.4 Non-CV-based models

In addition to algorithms and programs that use machine learning to recognize objects on the screen, there are other approaches that use browser and system data to obtain information about events that have occurred (such as mouse clicks or keyboard button press). In comparison, the program **Flowshare** [FlowShare, 2021] provides the ability to record screen actions (which partially duplicates the functionality developed in this paper) and to process screenshots with the ability to manually highlight important moments. Even though the screenshots themselves are not processed, the program will often understand where the mouse was clicked and records the result in the "description" field. Often the program even recognizes a clicked button, even if its purpose was not specified, and its description was only in the html code of the web browser (an example of the work in Figure 2.9)

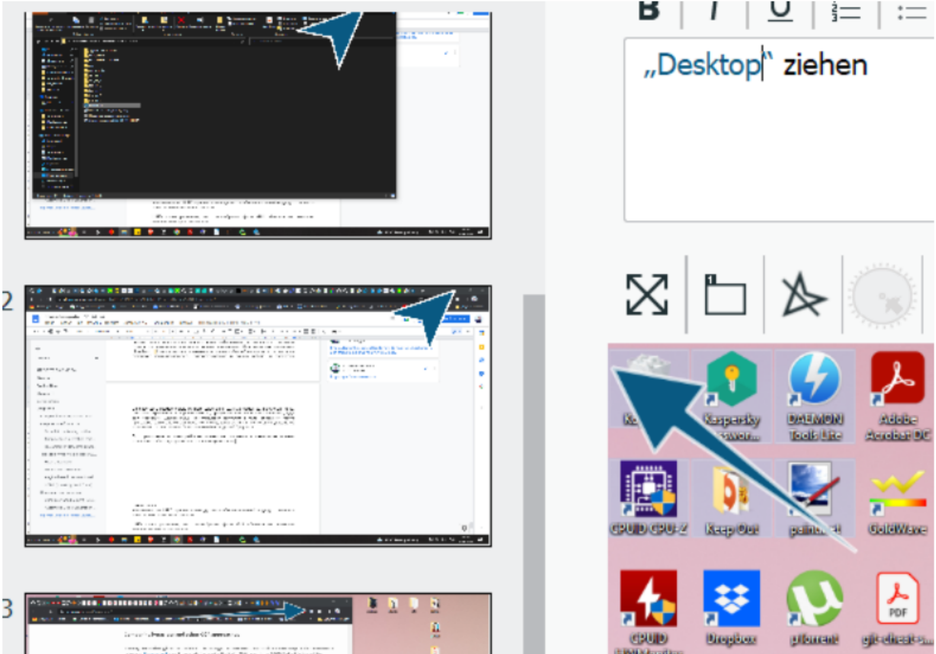


Figure 2.9. Part of the interface of the FlowShare application. On the left is the list of screenshots with the mouse click point. On the right is the object description window and on the bottom right is the current screenshot.

Selenium [Huggins, 2004], on the other hand, is a front-end testing tool often used in web application development. This tool allows you to write tests, sequences of button clicks, links, and other interface objects. It also allows you to listen to events occurring in the browser.

It is currently a widely used toolkit in many fields and the main contender for the task of detecting user actions in the SIT Alemira Virtual Labs project. However, it has a fundamental limitation - it works only with browser objects. For more complex systems, such as virtual machines, this program will not give results, and therefore it is not suitable for this task in full.

RaiMan's SikuliX [Hocke, 2007] is an application similar to FlowShare in many ways, but with one advantage: it allows you to check the user's actions. Unlike FlowShare, where the user simply records his actions as slides and shows them to the workers, later on, Sikuli allows you to

write scripts for the computer that allow you to automate the repetitive actions of the user. The Sikuli interface is a software interpreter that uses as variables the images of the objects needed to click (Figure 2.10). It is a unique program that uses some of the machine vision approaches, such as Template Matching, to navigate the screen and find the necessary UI components. That is, it is used to execute (or verify) actions already recorded, but not to write them down.

```
image_list = [(Red, Color),
              (Blue, Color),
              (Green, Color),
              (Black, Color)]

useless_app = App('C:\\Temp\\UselessApp.exe')
useless_app.open()

for image_pair in image_list:
    wait(2)
    click(image_pair[0])
    if exists(image_pair[1]):
        popup('Test successful...')
    else:
        popup('Test failed...')
```

Figure 2.10. Sample program code in the Sikuli IDE. Image has taken from [Hocke, 2012]

Chapter 3

Software Architecture and Design

3.1 Web Application Architecture

Based on the programs and models discussed in chapter 2, several test model architectures were developed, combining the merits of already known models to solve the UI components detection problem most effectively. In addition, to solve the problem, it is necessary to define system architecture for the interaction of the model with the user activity and the interaction of the model with the web application. For this purpose, a prototype web application with basic functionality was created. Figure 3.1 shows the architecture of the web application. The application under development here is called CVLogger (Computer Vision based Logging system).

The figure shows the general architecture of the project. The architecture of this project is microservices augmented with a streaming block. In addition to the client part, the full implementation of the server part will allow using this application in any other software by the provided API.

The CV Logger UI is the part of the software user interface responsible for interacting with the CVLooger server-side via an API. Depending on the software, it varies, so its implementation is related to its design and architecture.

CVLogger Event Listener is an auto-run script on the system installed in the virtual machine (or on the user's computer). This script listens to the user's actions and sends the data in the CV model to the server for further processing. The data it sends:

- Mouse click coordinates (for clicking) or the number of the keyboard button pressed (when typing);
- Action type (left click, double click, right-click, middle button, scroll, etc);
- Screenshot of the screen during the action.

This script is a potential system security vulnerability because it works directly with user activity data. In some systems, such as MacOS, keyboard key reading is intercepted by the system itself, so such a script is system-specific. However, SIT Alemira Virtual Labs does not work with MacOS virtual machines, so this thesis does not address this issue.

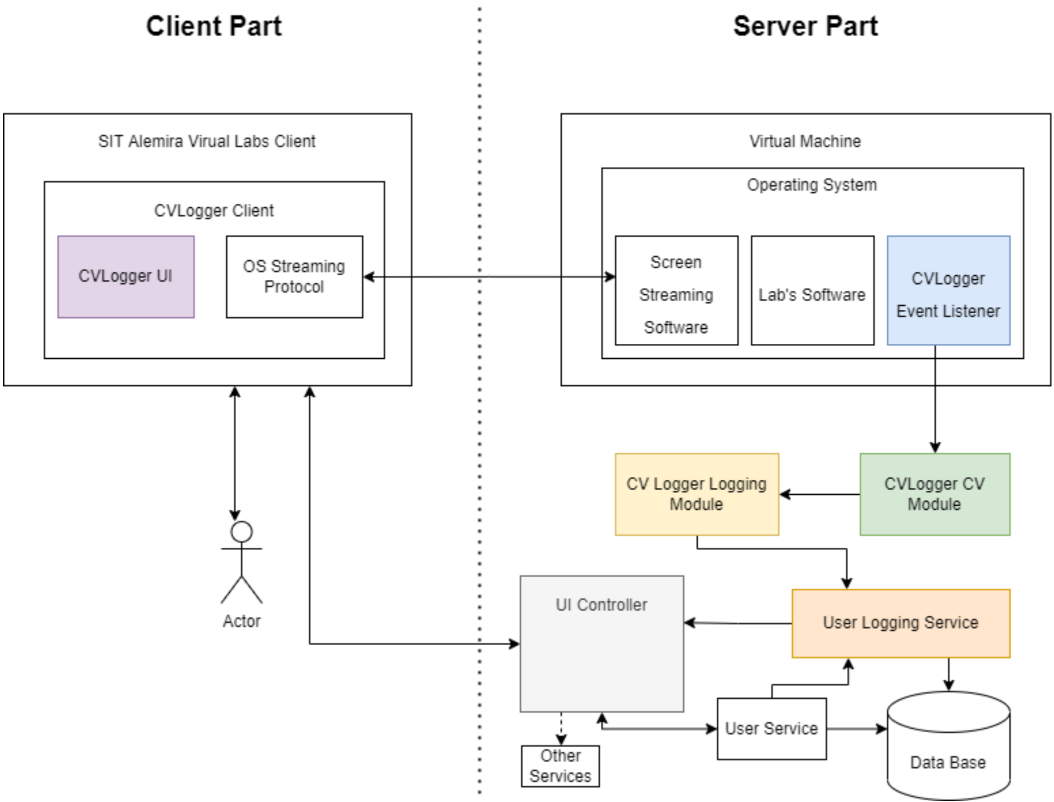


Figure 3.1. Logical View of the architecture of the web application to augment the SIT Alemira Virtual Labs product. The colored boxes indicate the software parts developed in this work: the prototype user interface (purple), the Event Listener (blue), a set of machine learning algorithms for detecting click objects (green), the results logging module (yellow), and the service for interacting with the database (orange).

CVLogger CV Module is a module, which is a set of functions to process images (screenshots) and determine the following parameters of the user's action:

- *Type of action* (click/text selection/scroll). It is inherited from the Event Listener without any changes.
- *Object type*. This parameter describes what object was clicked on. Depending on the model, you can distinguish a different number of object types. For example, "text-containing object/non-text object", or "icon/button/togler/scroller/...".
- *A description of an interaction object*. Many objects in any UI are captioned so that the user can quickly understand the purpose of the object. It can be the name under the icon, or the text of the button. For better User Experience when interacting with logs it was decided to add this text to the description of the log.
- *Images of the object*. This model should be regarded as an object detector for the correct recognition of committed action, hence it should find the boundaries of the interaction object with sufficient accuracy to provide an image of the object to the end-user.

- *Action number.* Due to the asynchrony of the Event Listener, the functions of the CV module are called in parallel, and may not have time to process one event before another occurs. Therefore, it is important to number all incoming events.

CVLogger Logging Module. This module has to pick up a suitable description for the committed event, based on the type of action and the type of interaction object. It is also considered to take previous events into account if necessary.

User Logging Service is a microservice responsible for the interaction of the modules described above with the database (saving, modifying, deleting, and retrieving user and task logging data from the database). It is necessary because logging is a new entity in an application, and specific views and serializers must be created to integrate it into existing systems.

3.2 Database Architecture

Small changes also affect the main program's database itself (new tables need to be created in it, see Figure 3.2). and the UI Controller, because the introduction of the new microservice requires the controller to learn to react to new requests to redirect them to this microservice.

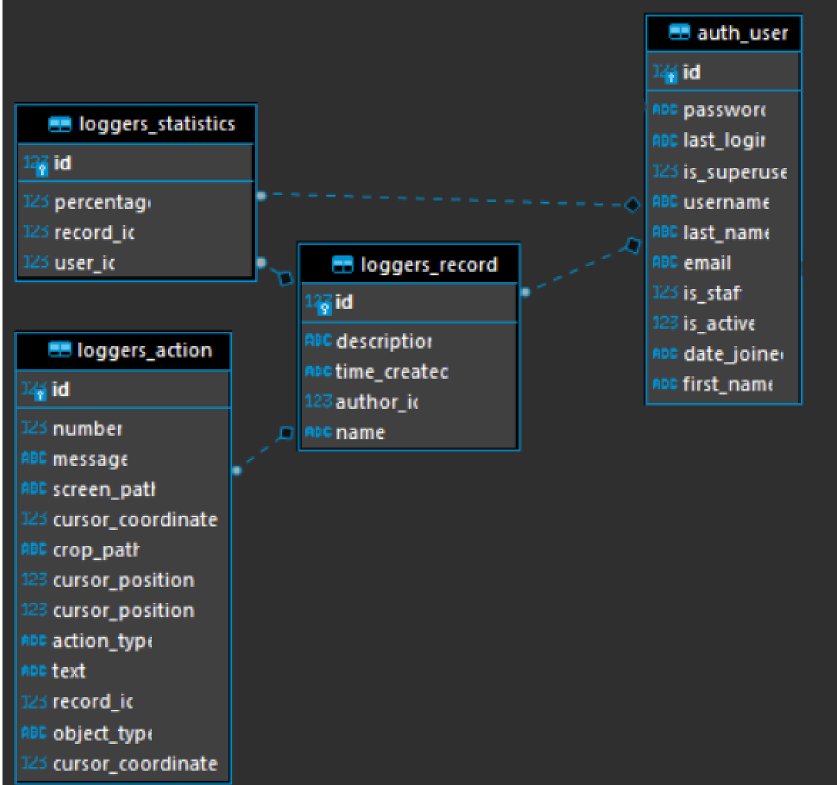


Figure 3.2. Simplified User Logging Service database architecture (without user sessions, permissions, etc.)

The architecture of the database needed to store log data is quite simple and contains only

3 specialized components (+ users table).

The **Record** table (loggers_record) contains basic information about the task to be logged or reproduced. These are:

- The *name* of the task, e.g., "Installing Python on your computer";
- *Task Description*;
- *Author*. It is the field associated with the user table;
- The *creation time* of the instance.

Each committed action is stored as a separate record in the logs in the **Action** table (loggers_action). It contains:

- *Field Number* - the number of the action in a particular task;
- *Message* field - generated message which is essentially a log that will be presented to the users;
- The *Screen Pat*h field is a link to an image (screenshot) in the data storage. The screenshot should be stored to keep the system robust - if the CV-based model returns an incorrect result (e.g. an incorrectly cropped object), it can be corrected manually;
- The *Cursor Coordinates* fields contain click coordinates relative to the upper left corner of the screenshot;
- The *Crop Pat*h and *Crop Position* (X and Y) fields are similar to those described above, and are a reference to the object image (the cropped monitor image) and the coordinates of the click relative to the upper left corner of that image;
- *Action type*;
- The *text recognized* in the object;
- The *ID of the task* in the context in which the action is performed is the field associated with the Record table;
- *Type of object* (according to CV-based model results).

The third table, **Statistics**, contains information about the logging of users. It does not store all user actions, because it is inefficient in terms of memory, but only the result of the whole task. It contains 3 fields:

- The *ID of the user*, who solved the problem;
- The *ID of the problem solved*;
- The *solution percentage* (the number of correct steps vs. the number of total steps). If a user performed more actions than indicated in the log, all events with a number higher than the number of logged actions will be ignored. If fewer actions were taken, the remaining events will be considered unfulfilled, that is, incorrect.

Chapter 4

Implementation

The following technological stack was used to create the CVLogger application: Python 3.8, JavaScript, HTML, CSS, Django 3.1 [Holovaty and Willison, 2005], Vue3.0 [You, 2014] frameworks. The scikit-learn [Cournapeau, 2007], tensorflow [Team, 2015], keras [Chollet, 2015], and openCV [OpenCV, 2021] libraries were used to handle machine learning algorithms in Python.

The application has the following functionality available for embedding:

- User authentication and authorization (can be replaced by a similar system of external software);
- Possibility of creating a new task (new log)
- Ability to view your solutions (solution statistics)
- Ability to view created tasks (logs)
- Display of list of existing tasks (logs)
- Viewing tasks (and a list of actions recorded in them)
- Changing a task (adding actions - logging, deleting a task)
- Changing a recorded action (information, type of object, image, deleting an action)
- Passing through a task

However, the current version of the application also has the following limitations:

- Event monitoring only works correctly on one monitor (monitor #1 for computers with multiple monitors);
- Only 7 types of objects are considered: keyboard button, icon, screen button, text, folder, toggle, and checkbox.
- There is no limitation on task solving. This is an optional functionality that was decided to omit due to its possible uselessness. Now every person (even the author) can go through his tasks as often as he wants.

- Correct processing of entered text is also temporarily absent. At the moment, pressing each button on the keyboard is treated as a separate action.

A separate module is allocated for solving each of the tasks presented in the introduction. A description of solving each step is given below:

4.1 Dataset generation

To train the models under study, more than 60,000 images of various user interface objects and more than 40,000 letters and numbers were generated in various fonts, sizes, and densities:

- Icons (30,000)
- Buttons (35,000)
- Folder icons (20,000)
- Checkbox (2 100)
- Text (2 100)
- Switch (1280)
- Arrow (400)
- Radio button (400)
- Scroll (100)
- Image (3500)
- Letter/number (38 750)

Examples of images from the dataset are shown in the Figure 4.1.

Figure 4.1. Examples of data from the dataset, (a) - letter, (b) - icon, (c) - folder, (d) - button.

For the data set, the Fatkun extension for Google Chrome was used, which allows downloading large arrays of images. So screenshots of screens of mobile devices and PCs were downloaded, as well as sets of UI components in the public domains (Flaticon [Flaticon, 2010], Iconspng [Iconspng, 2016]) and data used in training the models specified in [Sahu, 2020], [MulongXie, 2020].

A script was written for manual processing of downloaded sprites, their cropping, and marking. A script was also written to generate random images (negative samples) and augment data. The uploaded 3,500 background images were randomly cut into images of smaller sizes, shapes, and colors. The color palette of the images was also changed as part of the dataset augmentation. Affine transformations (shift, rotation) were also used for augmentation. The same augmentation algorithm was applied to objects of the Button, Icon, Folder class, and others (except text) to get more samples.

In several models, some classes were merged to increase performance. For example, Radiobutton and Checkbox, similar in appearance, were served to some models as an object of the same class.

To create a dataset for the Optical Character Recognition (OCR) model, a script was written that generates images of letters and numbers based on the fonts available in the system. For each character from the set, there were several images formed by shifting the symbol relative to the center of the image, stretching the symbol, and rotating it.

4.2 Choosing the architecture of a Computer Vision model

After considering possible approaches to solving the UI Components Detection problem, a composite approach was used. The following model architectures were considered:

4.2.1 Simplified Model's Architecture

Simplified Pipeline

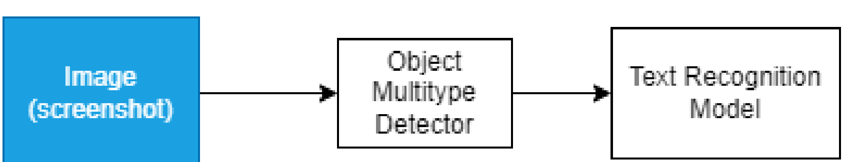


Figure 4.2. Simplified Model's Architecture

The simplified architecture (Figure 4.2) is characterized by the use of a single model for detecting various objects on the screen. The detectors themselves used have image processors, and there is no need to pre-process the image. This architecture is distinguished by the speed of work because detection algorithms are optimized for searching for objects in images of any size, and the image itself does not require preprocessing.

The first detector used in such an architecture was the Haar cascade detector described in Chapter 2. An example of the result can be seen in Figure 2.7 of the same chapter. However, despite its speed, its efficiency turned out to be quite low (see Table 5.1). Up to 15 training epochs of 4,000 background and noise images and 4,000 object images for each cascade were used.

The second option of using the detector for many objects, was a fine-tuned model based on Smart-UI [Sahu, 2020], one of the CNN improvements from the UIED algorithm for generating

web pages from an image. To do this, using the OpenCV Python library, this classifier model (Figure 4.4) was transformed into a detector (using the methods from Chapter 2). The model itself produced quite good results, however, in the images, it could not separate objects from noise. The fact is that such a model is designed for selective classification, and precise bounded objects are used for its training (samples from the dataset against noise images). When the detector checks boundary cases, for example, when only half of the button is located on the classified area, the classifier cannot give an accurate estimate. As a result, despite the fact that the classification of the objects themselves is generally possible (Figure 4.3), it is not possible to determine their location with sufficient accuracy (Figure 4.5).

The number of classes in this model was reduced from 13 (according to the initial settings of the neural network) to 6 (button, icon, folder, checkbox, text, toggle) after fine-tuning. At the same time, the number of neurons on the last layers of the network and the processing time decreased. But on a screen-wide scale, the processing time of the event by the model turns out to be too long (about 40 seconds per 600x600 image, see Table 1), which shows it as inapplicable for this task in this architecture.

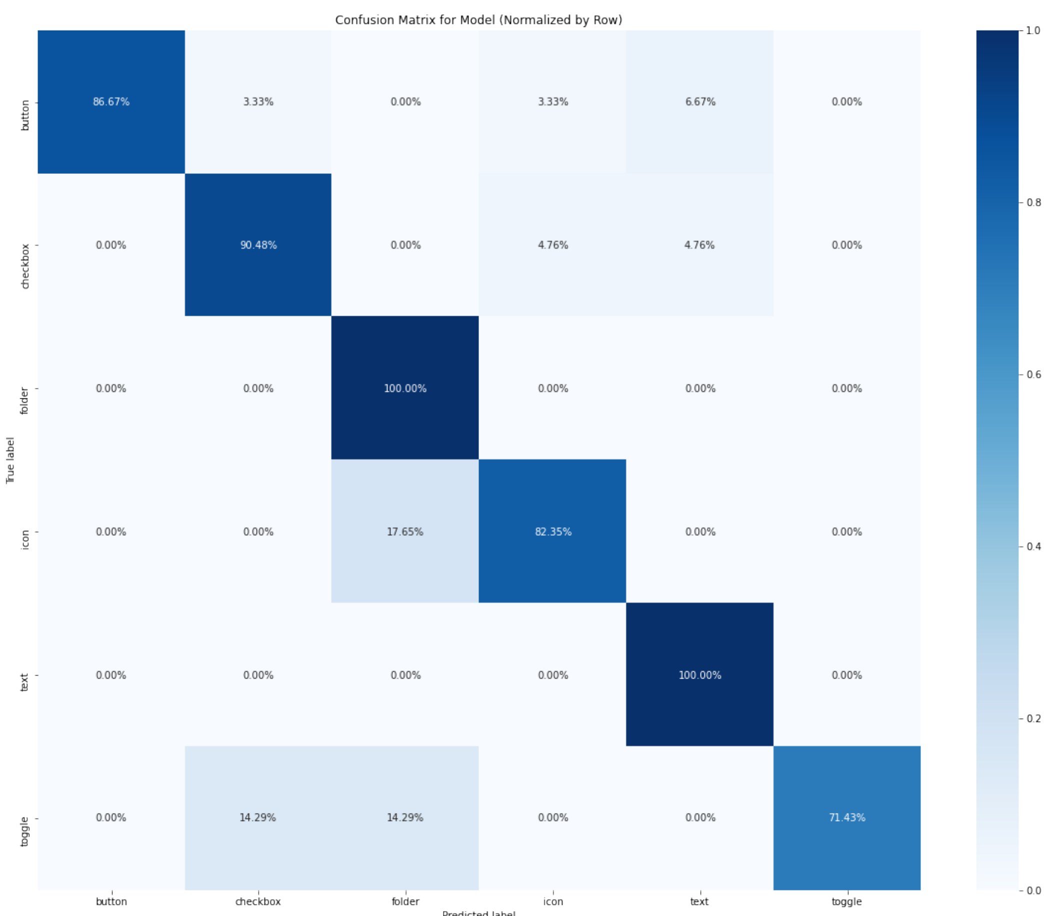


Figure 4.3. Confusion matrix of the CNN SmartUI classifier for 6 classes. Normalized by lines. It can be seen that all classes are predicted, basically, correctly. The average probability for classes is 88.8%.

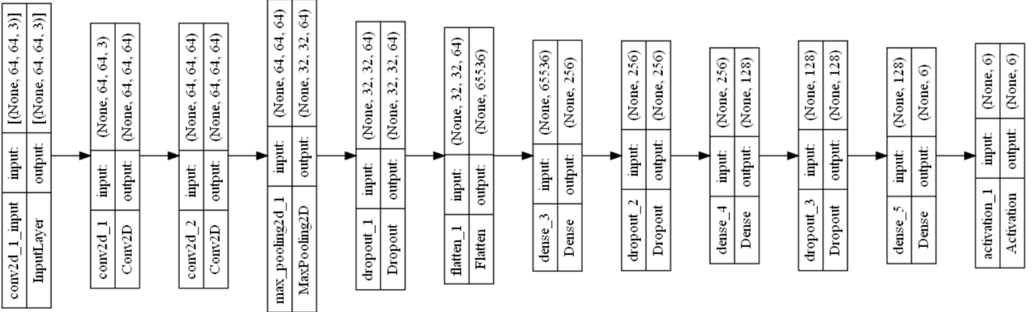


Figure 4.4. A CNN graph of a model based on the Smart UI model.

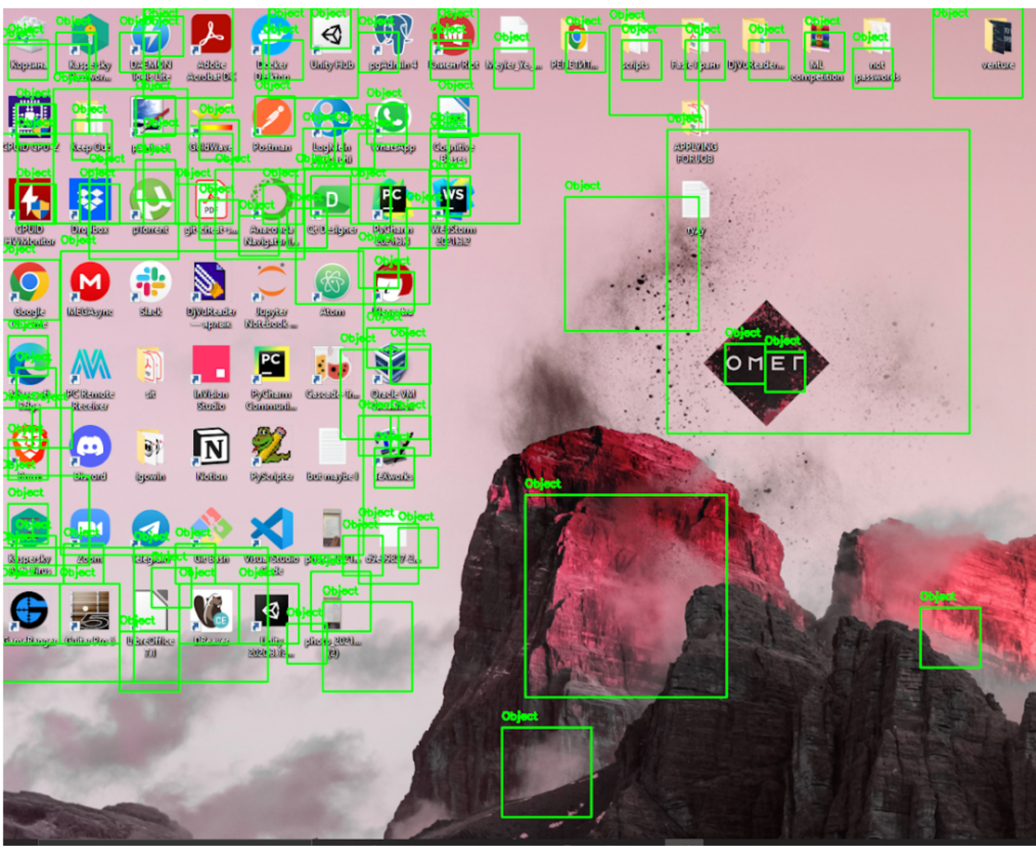


Figure 4.5. Application of CNN-based detection of object types to the screenshot. The detected objects of the "icon" class are highlighted with green frames.

4.2.2 Sequential Model's Architecture

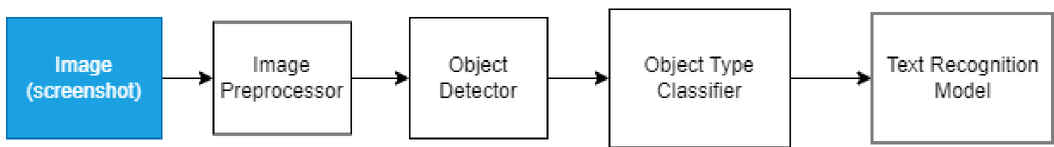


Figure 4.6. Sequential architecture. Detection and then classification of objects in the screenshot.

In the article presenting UIED [Xie et al., 2020], the authors mention that the variety of forms of GUI objects does not allow the use of simple models. To solve this problem and the problem of detection time, an object detector was added to the simplified model, which is a binary classifier

or the simplest detector that checks the presence of objects in the image. Various models were used (Random Forest, Logistic Regression, SVM, Multilayer Perceptron, KNN) and algorithms for generating bounding boxes (HOG-based algorithm, Canny Edges Detection algorithm, UIED detection algorithm).

Testing showed that most models cope with binary classification on object images against noise images quite successfully (average accuracy is about 97%), the test results are presented in Tables 5.3 5.4. The best results were shown by the Support Vector Machine Classifier with adding HOG as image preprocessing step. Unfortunately, the operating time of this model turned out to be too long (more than 40 minutes for a screenshot of size 1920x1080 pixels). The fastest algorithms, Random Forest, Canny Edges detector, and UIED detector coped with the task in about 16 seconds, 0.1 seconds, and 4 seconds, respectively. In this testing, all sklearn-based classification algorithms, like Random Forest and SVM, were tested on a screenshot using the Pyramid of Sliced Windows Technique. They also produced high results, but the calculation time higher that 5 seconds was too long, so it was decided to abandon part of the requirements considered for the project. In particular, one piece of the functionality dropped is the definition of the names of the windows in which the user's action takes place. Whether it's a system window or a browser tab, most often they cover the entire screen, requiring the application to analyze objects of huge size. Using a pyramid of sliding windows, such a task is solved, but as has been shown, too much time is spent.

Without having to view the entire screen, the location can be reduced to some area around the click. To determine the size of this area, the distribution of object sizes was constructed from the object database (Figure 4.7). It has the form of a random gamma distribution. A value covering 90%-quantile of all objects was selected. Rounding up, the window size is 200x200 pixels.

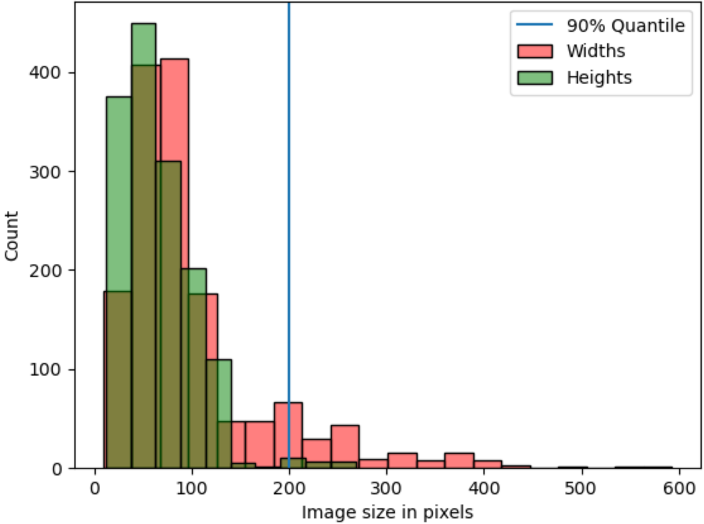


Figure 4.7. Histograms of object size distribution (for all types of objects together).

However, the size of 200x200 pixels is large enough to fit several objects on it. Thus, it does not make sense to submit the entire image to a classifier (Random Forest, SVM, or other machine learning algorithm), and it is necessary to develop an algorithm for more accurate

detection and determination of the boundaries of the object. To solve this problem, an algorithm called Probability-based Mapped Detection (PMD) was developed. A detailed description of the algorithm is presented in the next paragraph, and the results of its testing are presented in Table 5.4. This algorithm worked somewhat worse on average than UIED Detector, because it often cuts off the true object, but it turned out to be much more resistant to dividing the object into many sub-objects. To correct this error, a modification of the UIED algorithm was carried out, which is also described in detail in the following paragraphs.

This part of the model is the most important because the accuracy of the classification of the following models depends on the quality of object detection of these algorithms. Considering that both algorithms work fast enough, in the end it was decided to use a modified UIED detector.

After receiving the image crop containing the intended object, it is fed to the next model - the classifier of the object type. This model is the same convolutional neural network based on Smart UI architecture (Figures 4.3, 4.5). The only difference was its training not on the original image, but on the image of the object transformed into contours using the HOG method (Figure 2.4). This allowed to increase the average classification accuracy to 92.1%.

After this classifier has made a prediction, it is necessary to decide whether to run the most time-consuming model - the text recognition model. Of the 6 classes considered, text can potentially be observed (and be important for understanding the essence of the object) only in four - icon (icon name), folder (folder name), button (text on the button), and text. For instances of other classes (checkbox, toggle), it does not make sense to run the model, which saves time. Another possible approach is to run the model for object recognition and text recognition in parallel. But the operating time changes slightly (from 7 seconds to 5.7 seconds).

OCR (Optical Character Recognition) models such as Keras-OCR [Morales, 2019] and Tesseract-OCR [Smith and Hewlett-Packard, 2005] were chosen as text recognition models for this work. They were fine-tuned on a generated dataset of machine-generated letters and numbers (embedded in the font system). The older model, Tesseract, produced poorer results (see Table 5.2), but it worked a little faster. Nevertheless, considering that the quality of recognition for the convenience of the user will be more important than a small-time delay (which will be reduced due to parallel event processing), preference was given to the Keras-OCR model. To use it, it is also necessary to bring the image of the object into a form that corresponds to the input form expected from the model, while not distorting the text (without using stretching or color changes in the image). To do this, the image was reduced in size proportionally, and inserted on a black canvas with a size of 640x640px in the upper left corner.

The resulting running time of the sequential model is somewhat longer than the simplified one, but it gives much more adequate results that can already be considered in the context of the task.

The last stage of application model development is to compare the objects recorded in the log with the objects that the user interacts with during passing through the log. First, the user's actions are analyzed using the same algorithm that was used to record the log, and then the results of the current log are compared with the results obtained from the user's log. The action and the object of detection are compared. The text on the object is temporarily ignored, but can potentially be considered as an alternative to comparing images of objects. So, for example, a user can click on a certain tab or can open it by a link from a document. The objects in this case will be different, while the text will remain the same. It is supposed to use text comparison using Levenshtein distance [Gad, 2020] with some threshold value, because the accuracy of text recognition by the model may differ from image to image.

It is worth noting that different texts, however, do not mean inequality of objects. The error may be contained, for example, in the first part of the pipeline detector, then the text on the user's image may simply be missed.

4.2.3 Probability-based Mapping Detection Algorithm

Probability-based Mapping Detection Algorithm (PMD) is an algorithm developed while working on this dissertation based on the use of a variety of windows of small sizes and different shapes to create a map of the probabilities of finding object parts on each pixel (or part) of the image. The general idea of the algorithm is as follows:

1. Determine the coordinates of the click and make an array of all the crops of the original image, which have the specified shapes and sizes and at the same time contain the click location;

2. Apply a classifier to each of the crops in this array;

3. Create a map (size up to 200x200) of the probabilities of finding an object in a pixel of the original image. The value in each pixel is calculated as the sum of the classification probabilities from all windows containing this pixel divided by the number of these windows;

4. Using the map, select a window of the minimum size containing as many probability peaks closest to the clicked point as possible.

The resulting algorithm works generally faster than the sliding window algorithm due to the reduced number of windows to check and gives a more accurate result than if to simply combine those windows in which the classifier gave a positive result. An example of the operation of this algorithm is shown in Figure 4.8.

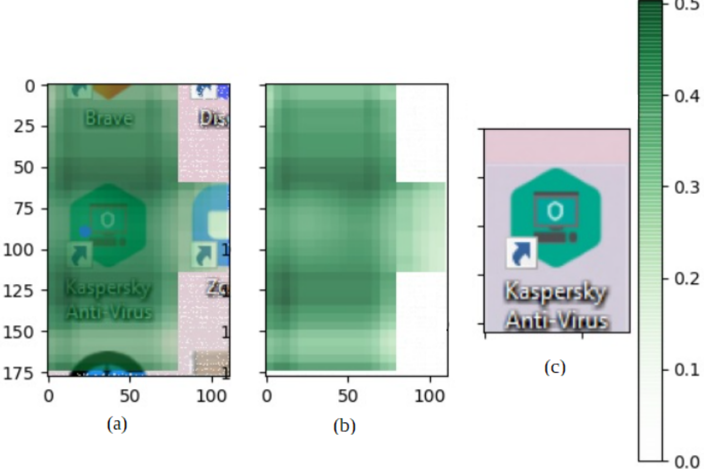


Figure 4.8. The result of the PMD algorithm. (a) - probability map superimposed on the original image. The image is cropped due to being on the edge of the screen, and is also vertically limited by the size of the selected windows; (b) - probability map; (c) - the result of detection after cropping the image.

4.2.4 UIED Detector Modification

To correct the problem of dividing an object into subobjects (Figure 4.9, a), the UIED detector algorithm has been modified. The result after modification can be seen in Figure 4.9, b. The meaning of this modification is to search for neighbors of objects. The whole improvement of the algorithm is described as follows:

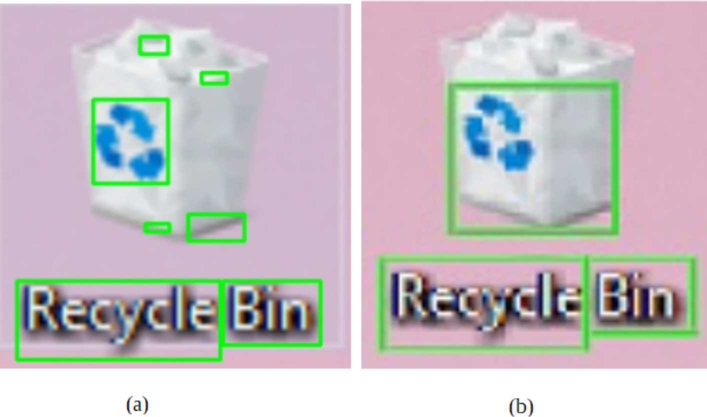
1. The image is processed by the UIED algorithm, the output is a set of coordinates of rectangular shapes.

2. These coordinates are checked for the content of the click location in them.

3. If the click location coincides with the already found form and this form has a size larger than the minimum allowed (a heuristic parameter, an area of 30px is selected in this thesis), this particular form is returned. All objects less than acceptable are removed from consideration.

4. If the coordinates of the click did not hit any of the detected objects, the two objects closest to the click with a size larger than the minimum allowed are considered.

5. Based on the coordinates of the click location and the coordinates of the shapes under consideration, a new minimum-sized area is constructed that accommodates both shapes and the click location.



(a) (b)

Figure 4.9. Possible result of the UIED Detector. In (a) is the fragmentation of objects is clearly noticeable. In (b) this fragmentation is reduced, and all minor objects are ignored.

The number of merging objects is chosen to be equal to two for the reason that UI elements often consist of exactly two parts - an image and a signature. Potentially, this modification can combine them into one object, which will improve the further classification.

4.2.5 Web Application Prototype

The described models and algorithms have been grouped for convenient use via the API for use in the SIT Alemira Virtual Labs web application. As an example of use, the front end of the web application was created(Figure 4.10). Figure 4.11 shows an example of using the application.

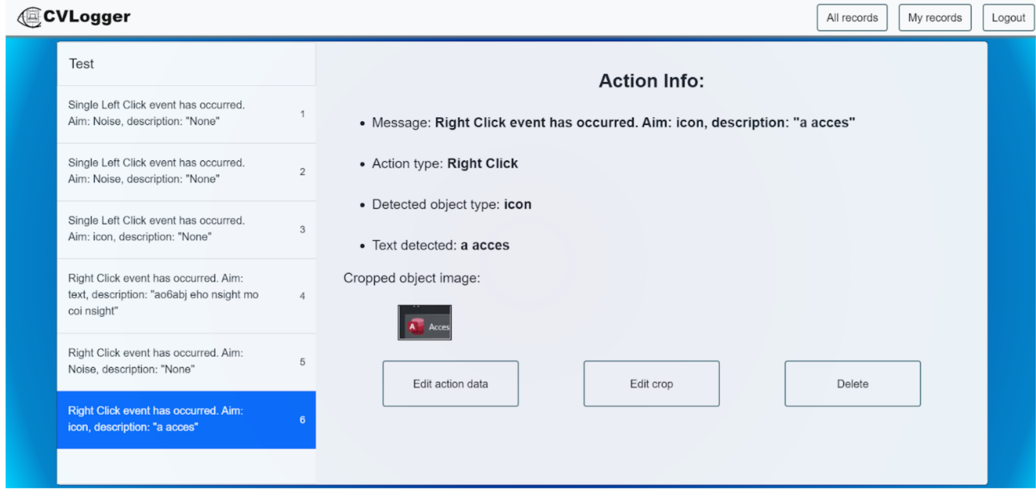


Figure 4.10. The client part of the CVLogger web application prototype. Action info page.

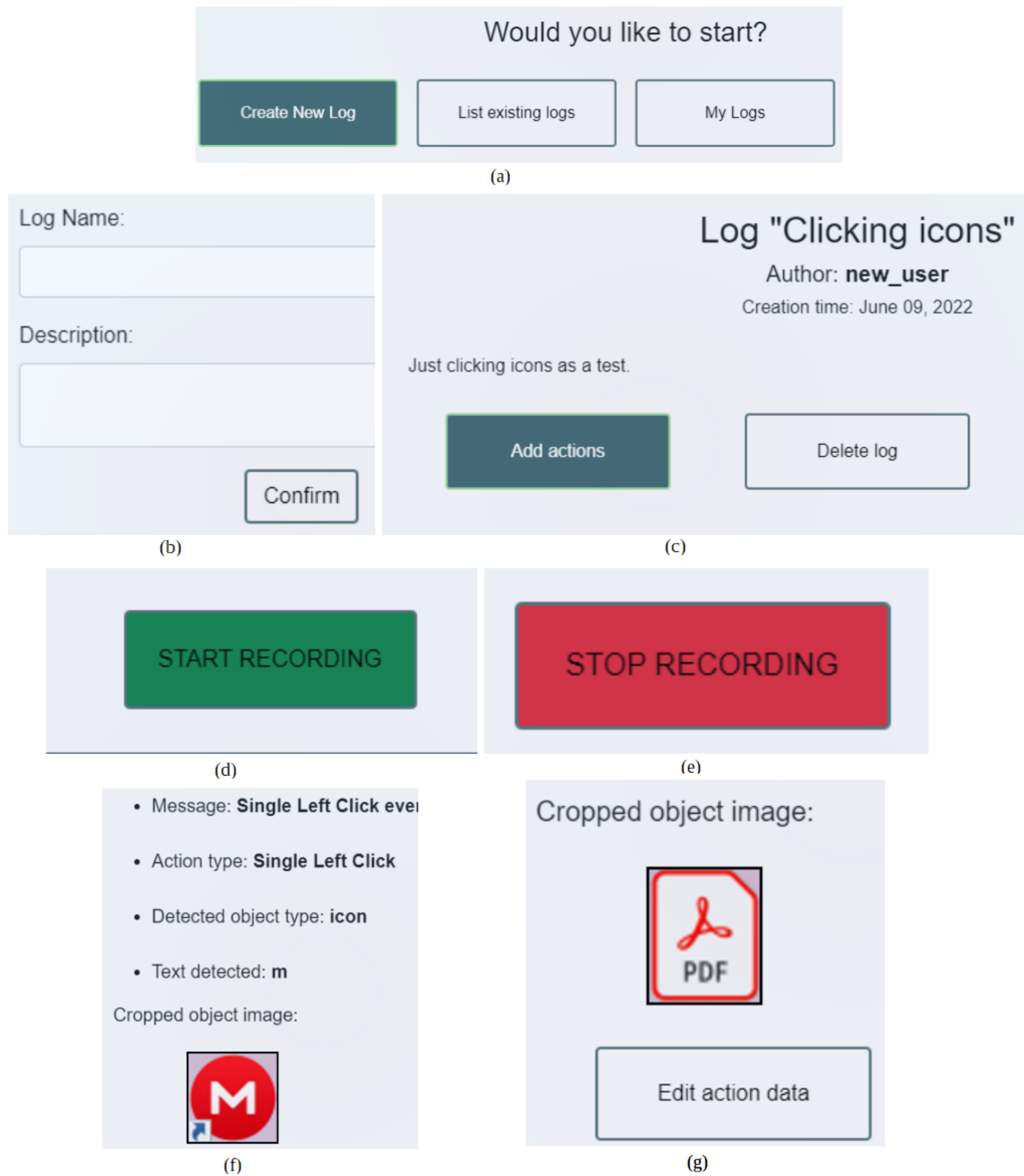


Figure 4.11. The sequence of actions for creating a new log. Figures from (a) to (g) show which buttons to press and what the active part of the UI looks like at this moment.

Description of the sequence of user actions from Figure 4.11:

a) Click the “Create a new log" button.

b) Enter the name and description of the log.

c) After confirmation, you will be taken to the page of the created log.

d) Click “Add actions”. On the page that appears, click “Start Recording".

e) Perform the necessary actions for recording and click “Stop Recording”.

f) You will return to the previous page with information about the log. In the table on the left, there will be a list of recorded actions. Click on them and you will see information about chosen one.

g) If you need to make changes, click “Edit action data".

The API was developed using the Django REST Framework. 4 ORM models were written to interact with the database (Figure 3.2). A serializer has been written to handle CRUD (Create-Retrieve-Update-Delete) requests to these models, as well as views for the following APIs:

- Registering and Authorization;
- Create new log;
- Retreive log by the id;
- Update log (only if you are the author);
- Get log’s actions;
- Retreive action by log’s id and action number;
- Update action’s object image;
- Get current user info;
- Get current user authored logs;
- Get current user statistics about passed logs;
- Request to start record log;
- Request to stop log;
- Request to start passing test;
- Request to stop passing test;

Special attention should be focused on the last four requests. The request to start recording the log currently includes an event listener for mouse and keyboard events, and an event handler function creates a parallel thread for the next function that starts the pipeline of the image processor. After receiving the result from the pipeline, the data, along with the event number, is transmitted to the logger function, which generates a record in the database and directly a log string. The sequence of calls is shown in Figure 4.12, a. An alternative method that was tested was to create a queue instance when requesting the start of a log entry and then launch an event listener that will add events to the queue. In parallel with the listener, the queue check function was run in a separate thread for the presence of an object in it. If there are still objects there, the first object in the queue is taken out of it and processed. Despite the fact that this method is more stable, because it does not allow events that occurred after each other to change places in the database, it takes too much time, because all events are processed sequentially (Figure 4.12, b).

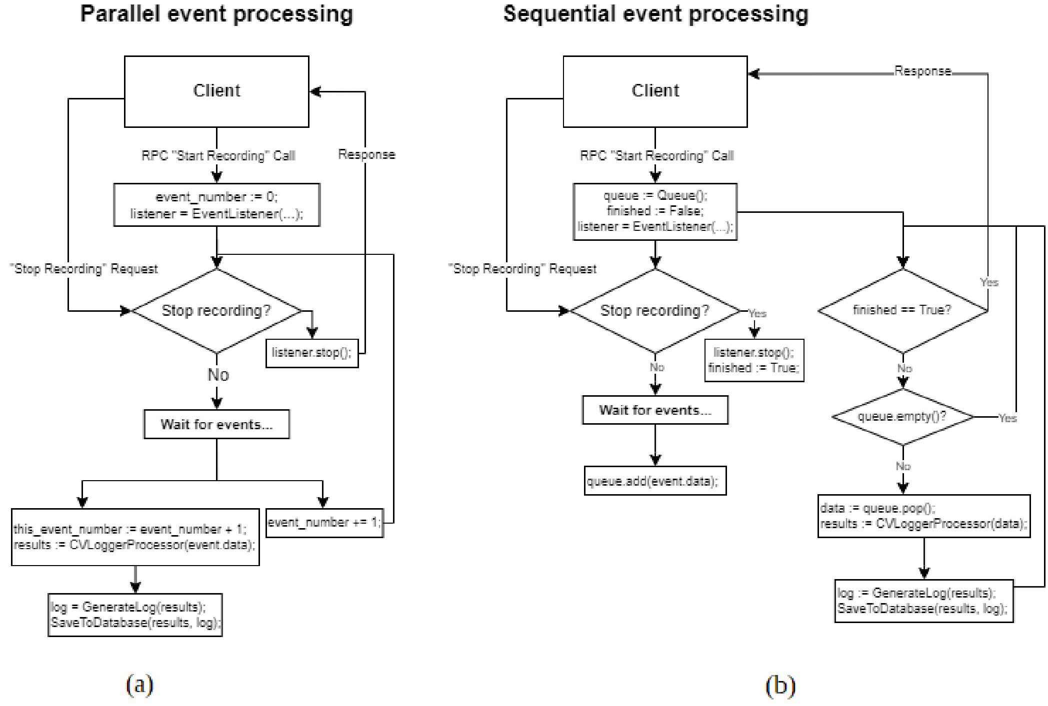


Figure 4.12. Various methods of processing a request to start/stop logging/checking. (a) - parallel event processing; (b) - sequential event processing.

Based on these requests, a web interface was developed that implements them. The front end was developed using the VueJS 3.0 framework. The choice of design tools was made based on the current SIT Alemira Virtual Labs toolkit. For this dissertation, the following pages were added to the client part of the web application prototype:

- Registration and login pages;
- Create Log Page;
- Log page;
- Statistics page;
- Action info page;
- Recording log page;
- Edit action page;
- Edit action object image page;
- Passing log page;

A separate part of the front end at the moment is changing the image of the interaction object. To create such an opportunity for the user, the server needs to store the initial screenshot of the screen during the click. Initially, it was planned to embed this functionality into

the web application interface natively, i.e. using javascript and libraries for drawing images (such as chart.js). However, it was decided to leave this functionality for further development. Instead, a script was written on the server-side that allows you to crop the image (screenshot) in a separate window. It is based on the algorithm noted in the section “Dataset Generation". This solution will work on a local server, but when going into production, the window will not be displayed on the user’s computer, so this functionality will require execution in javascript. However, this functionality is not a central part of the study, so it was omitted during the development process.

Chapter 5

Testing algorithms and results

For machine learning models (binary classifiers) working directly with images dividing them by the type (object or not an object), such as Logistic Regression (LogReg), Random Forest (RF), K-Nearest-Neighbors (KNN), Multilayer Perceptron (MLP) Support Vector Classifier (SVM), the standard binary classification metric f1-score was used. The same metric was used for these algorithms with the preprocessing (applying Histogram of Oriented Gradients method to the images) step. Results of this testing is shown in Table 5.3.

Testing of the detection algorithms specified in Tables 1 and 4, due to the specifics of their work, was carried out in a different way from the algorithms in Table 5.3. For object detectors based on Canny Edge detection and UIED models, as well as for the developed Probability-based Mapping Detection algorithm, an algorithm was developed to evaluate the quality of bounding box detection. This algorithm is described below.

5.1 Detector testing and image comparison

The detected bounding boxes were evaluated using the following algorithm:

1. As test data, images of an object and a random image from the set of "Images", i.e. background, were selected. The image of the object was superimposed into a random position on the background with applied minor affine transformations (stretching in width and height by a random number of pixels (from 0 to 10), changing the color palette);

2. The resulting image was fed to the detector input to determine the bounding box;

3. If several bodies were detected, the one that is closest to the center of the object in the image was selected;

4. The original image of the object and the resulting detection are reduced to the same size (the original size of the object image) and the palette of images is changed to grayscale;

5. Image comparison using the MSSIM metric;

6. Comparison of the result with some threshold selected heuristically (in this thesis, the threshold value of 0.5 is considered).

The Structural Similarity Index (SSIM) [Wang et al., 2004] was used as a metric for checking detection, indicating the similarity of images by comparing their structural similarities. In contrast, another popular MSE (Mean Squared Error) method, which compares images pixel by pixel, in this context can show serious differences in images even if the object frame was selected with an error of only 1 pixel. Another method, Locality Sensitive Hashing, LSH [Mendes, 2019] can solve this problem by splitting images into strips, hashing them, and comparing hashes. However, this method is too sensitive to image cropping and starts to show poor results quickly.

The SSIM method was chosen because it is most resistant to all these transformations. Unlike other described methods of image comparison, this method divides images into windows according to the grid, calculates the result for each window separately according to the Formula 5.1, and then summarizes the results according to the Formula 5.2.

$$\mathrm{SSIM}(x,y)=\frac{\left(2\mu_x\mu_y+c_1\right)\left(2\sigma_{xy}+c_2\right)}{\left(\mu_x^2+\mu_y^2+c_1\right)\left(\sigma_x^2+\sigma_y^2+c_2\right)} \tag{5.1}$$

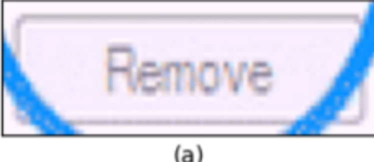
$$\mathrm{MSSIM}(\mathbf{X},\mathbf{Y})=\frac{1}{M}\sum_{j=1}^{M}\mathrm{SSIM}\left(\mathbf{x}_j,\mathbf{y}_j\right) \tag{5.2}$$

In formula 5.1, μ_x and μ_y are the average pixel brightness values in a certain window of image x and image y respectively. σ_x and σ_y are standard deviations from the mean for the same windows, and σ_{xy} is covariance. c_1 and c_2 are constants in this formula. Formula 5.2 shows how the results of each window considered are summed.

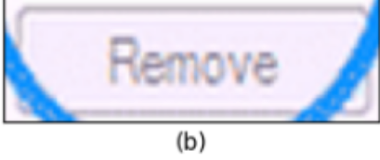
If to select a small number of windows (M<100 in formula 5.2), each window will contain a sufficiently large part of the image, and therefore similar objects in them is more likely will be detected. As the number M increases, the method turns into a pixel-by-pixel comparison. In this dissertation, M=64 was chosen, and each image is divided into windows on an 8x8 grid.

SSIM outputs a result in the range from -1 to 1, where 1 is a perfect match. According to the results of multiple checks, it was found that the most suitable comparison threshold for images of the same object with the optimal number of false-positive results is 0.5. An example of image comparison is shown in Figure 5.1.

SSIM: 0.83



(a)



(b)

Figure 5.1. Comparison of the image of a real object (a) and the one found by the algorithm (b). SSIM = 0.83, therefore, the algorithm considers the detection correct.

SSIM with the same threshold value is also used in this work to compare the detected and stored object's image.

5.2 Testing results

This section shows the results of testing of all tested models.

Model Name	Average detection time, s	Average f1-score	Button	Icon	Folder	Checkbox	Text	Toggle
Haar Cascades	0.02	0.28	0.32	0.11	0.33	0.40	0.07	0.44
Smart-UI based Detector	5.54	0.38	0.42	0.17	0.32	0.32	0.63	0.42

Table 5.1. Results of testing the multi-class detectors of simplified architecture. Average detection time is checking on images of shape 200x200px. F1-score metric is calculated for each class.

Model Name	Average recognition time, s	Precision	Recall
Keras-OCR	6.99	0.50	0.59
Tesseract-OCR	3.12	0.29	0.52

Table 5.2. Comparison of Optical Character Recognition Models.

Model Name	Average detection time, ms	F1-score
LogReg	0.00005	0.77
SVM	10	0.96
RF	0.2	0.95
MLP	10	0.92
KNN	0.2	0.88
LogReg + HOG	**0.0007**	**0.95**
SVM + HOG	10	**0.97**
RF + HOG	**0.3**	0.97
MLP + HOG	97	0.95
KNN + HOG	0.3	0.91

Table 5.3. Comparison of binary classifiers (object vs background) for the object detection task. Studied models (by lines): Logistic regression; Support Vector Classifier; Random Forest; Multilayer Perceptron; K-Nearest-Neighbors; then the same algorithms with Histograms of Oriented Gradients instead of an image on the input. The average detection time is calculated for one classification iteration for one object/noise sample.

Model Name	Average detection time, s	F1-score
Canny Edge's Detector	0.0001	0.46
UIED Detector	0.86	0.83
Probability-based Mapping Detector	1.34	0.67

Table 5.4. Comparison of object detection algorithms (without classification).Average detection time is checking on images of shape 200x200px.

Chapter 6

Conclusions

As a result of this work, the possibility of using computer vision algorithms to process user actions and generate logs, along with their further verification and repetition by other users, was tested. Based on the results of the work, the probability of correct classification of actions is quite high (up to 78%), and the accuracy of text recognition for objects is also high. The event processing time takes about 7.10 ± 0.7 seconds, but due to the parallelism of event processing, this is not a problem for the application. A variety of approaches to image processing and various preprocessing methods open up wide horizons for further research in this direction and increase both the performance of algorithms and their accuracy.

The conclusion from the above is the potential possibility of using an event handler program based on computer vision algorithms for this task. However, for a successful application and a positive User Experience using this application, further research and improvement of detection results are necessary.

Also, the possibility of controlling algorithms for listening to user actions (on a PC, mobile application, or virtual machine) via a web interface was tested. Despite possible problems with data processing time, such systems work stably and allow using the event processing algorithms described.

To perform this work, the following steps were done, compiled together with Denis Zholobov, project manager of SIT Virtual Labs:

1. Developing a CV algorithm for the screen actions recognition;

2. Creating a web application based on the expected design;

3. Integration of the web app and CV algorithms;

4. Testing;

The purpose of this dissertation, to solve the problem of teaching the user to work with software using computer vision algorithms and logging user actions, was completed. A solution has been provided [Skakun, 2022] that allows the user to record their actions in the system, and then other users to use this record to train and compare their actions. This will potentially reduce the time to create training tutorials for beginners in working with software, as well as reduce the learning time, thanks to the step-by-step guide.

6.1 Further plans

After the defense of the dissertation, it is planned to continue work in this direction. List of tasks to be performed in the future:

- Increase the accuracy of object detection;
- Expand the detection area. Introduce algorithms for recognizing windows and nested sections of the interface in which the interaction takes place;
- Develop a full-fledged web application that works with virtual machines;
- Make native implementation of image formatting in a web application.
- Apply models to solve other problems:
  - Training in the use of software;
  - Control of the employee's activity at the workplace;
  - Development of parental control systems;
  - Reverse engineering;
  - Generating UI from an image.

Acronyms

Acronym	Description
CED	Canny Edge's Detector
UIED	User Interface Elements Detection
ML	Machine Learning
CNN	Convolutional Neural Network
R-CNN	Region-based Convolutional Neural Network
PMD	Probability-based Mapping Detection
HOG	Histograms of Oriented Gradients
LogReg	Logistic Regression
SVM	Support Vector Machine
SVC	Support Vector Classifier
KNN	k Nearest Neighbors
MLP	Multi-Layer Perceptron
RF	Random Forest
AdaBoost	Adaptive Boosting
OCR	Optical Character Recognition
CV	Computer Vision
UI	User Interface
GUI	Graphical User Interface
SSIM	Structural Similarity Index Metric
MSSIM	Mean Structural Similarity Index
MSE	Mean Squared Error
API	Application Programming Interface
DB	Database
YOLO	You Only Look Once

Bibliography

Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. From contours to regions: An empirical evaluation. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2294–2301, 2009. doi: 10.1109/CVPR.2009.5206707.

Pablo Arbelaez, Jordi Pont-Tuset, Jonathan T. Barron, Ferran Marques, and Jitendra Malik. Multiscale combinatorial grouping. 2014.

Behailu Adefris Biniam. Automatic code generation from low fidelity graphical user interface sketches using deep learning., 2020. URL `http://ir.bdu.edu.et/handle/123456789/12721`.

John Francis Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986. doi: 10.1109/TPAMI.1986.4767851.

Joao Carreira and Cristian Sminchisescu. Cpmc: Automatic object segmentation using constrained parametric min-cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1312–1328, 2012. doi: 10.1109/TPAMI.2011.231.

Ann Chadwick-Dias, Thomas Tullis, and Michelle Mcnulty. Web usability and age: how design changes an improve performance. *ACM SIGCAPH Computers and the Physically Handicapped*, 2002. doi: 10.1145/960201.957212.

Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Unblind your apps: Predicting naturallanguage labels for mobile gui components by deep learning. *In 42nd International Conference on Software Engineering (ICSE '20)*, 2020a. doi: 10.1145/3377811.3380327.

Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Mulong Xie. Object detection for graphical user interface: Old fashioned or deep learning or a combination? *28th ACM Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020b. doi: 10.1145/3368089.3409691.

François Chollet. Keras, 2015. URL `https://keras.io/`.

Ambika Choudhury. Top 8 algorithms for object detection, 2020. URL `https://analyticsindiamag.com/top-8-algorithms-for-object-detection/#h-1-fast-r-cnn`.

Anders Christiansen. Anchor boxes — the key to quality object detection, 2018. URL `https://towardsdatascience.com/anchor-boxes-the-key-to-quality-object-detection-ddf9d612d4f9`.

Dan C. Cireşan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. Mitosis detection in breast cancer histology images with deep neural networks. *Medical Image Computing and Computer-Assisted Intervention*, 2013. doi: 10.1007/978-3-642-40763-5_51.

David Cournapeau. Scikit-learn, machine learning in python, 2007. URL `https://scikit-learn.org/stable/`.

Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. *International Conference on Computer Vision & Pattern Recognition (CVPR)*, 2005. doi: 10.1109/CVPR.2005.177.

Morgan Dixon and James Fogarty. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. IEEE, 2010. doi: 10.1145/1753326.1753554.

Flaticon. Flaticon, 2010. URL `https://www.flaticon.com/`.

FlowShare. Flowshare®, 2021. URL `https://getflowshare.com/`.

Ahmed Fawzy Gad. Measuring text similarity using the levenshtein distance, 2020. URL `https://blog.paperspace.com/measuring-text-similarity-using-levenshtein-distance/`.

Ross Girshick. Fast r-cnn. pages 1440–1448, 2015. doi: 10.1109/ICCV.2015.169.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014. doi: 10.1109/CVPR.2014.81.

Chunhui Gu, Joseph J. Lim, Pablo Arbeláez, and Jitendra Malik. Recognition using regions. 2009. doi: 10.1109/CVPR.2009.5206727.

Raimund Hocke. Raiman's sikulix, 2007. URL `http://sikulix.com/`.

Raimund Hocke. Sikuli ide: Using pattern objects, 2012. URL `http://testautomationnoob.blogspot.com/2012/`.

Adrian Holovaty and Simon Willison. Django (v3.1), 2005. URL `https://www.djangoproject.com/`.

Jason Huggins. Selenium, 2004. URL `https://www.selenium.dev/`.

Iconspng. iconspng, 2016. URL `https://www.iconspng.com/`.

Satya Mallick. Histogram of oriented gradients explained using opencv, 2016. URL `https://learnopencv.com/histogram-of-oriented-gradients/`.

Ken Mendes. Locally sensitive hashing for near-duplicate image detection, 2019. URL `https://github.com/mendesk/image-ndd-lsh`.

Fausto Morales. Keras-ocr, 2019. URL `https://github.com/faustomorales/keras-ocr`.

MulongXie. Uied, 2020. URL `https://github.com/MulongXie/UIED`.

Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). *30th IEEE/ACM InternationalConference on Automated Software Engineering (ASE)*, 2015. URL https://www.google.com/url?q=https://laptrinhx.com/ui2code-how-to-fine-tune-background-and-foreground-analysis-2293652041/&sa=D&source=docs&ust=1655096275295586&usg=AOvVaw2BMLPpMmgjWzaTE0oPKAK_.

OpenCV. Template matching, 2021. URL https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016. doi: 10.1109/CVPR.2016.91.

Adrian Rosebrock. Turning any cnn image classifier into an object detector with keras, tensorflow, and opencv, 2020. URL https://pyimagesearch.com/2020/06/22/.

Tezan Sahu. Smart ui, 2020. URL https://tezansahu.github.io/smart_ui_tf20/.

Aleksandr Skakun. Github page, 2022. URL https://github.com/AleksTankist111/thesis_CVLogger.

Ray Smith and Hewlett-Packard. Tesseract-ocr, 2005. URL https://tesseract-ocr.github.io/.

R.L. Street, L. Liu, and N.J. Farber et al. Keystrokes, mouse clicks, and gazing at the computer: How physician interaction with the ehr affects patient participation. *Journal of General Internal Medicine*, 2018. doi: 10.1007/s11606-017-4228-2.

Google Brain Team. Tensorflow, 2015. URL https://www.tensorflow.org/.

James T. Todd and Ennio Mingolla. Perception of surface curvature and direction of illumination from patterns of shading. *Journal of Experimental Psychology: Human Perception and Performance*, 1983. doi: 10.1037/0096-1523.9.4.583.

Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. doi: 10.1109/TIP.2003.819861.

Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. Uied: A hybrid tool for gui element detection. *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1655–1659, 2020. doi: 10.1145/3368089.3417940.

Evan You. Vuejs (v3.0), 2014. URL https://vuejs.org/.

Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. 2019.