Engineering Degree Project

Parking Sign Interpreter

Author: Beg Fardin, Al Omari Ghiath
Supervisor: Alissandrakis Aris
Semester: Spring 2024
Subject: Computer Science

Abstract

This thesis presents a smartphone-based application that interprets parking signs. Most drivers face problems from misinterpretation of these signs. Our motivation was to reduce the problem by designing an application for Android mobile devices that can interpret parking signs through the processing of images taken by the users and, with these, give out real-time information on the parking rules.

This project was developed using Python, Kivy, OpenCV, and Tesseract OCR. A piece of test code was optimized, and the code generated was applied to run the applications on the Kivy/Android platform.

Our application was tested in varied conditions; however, it is underlined that the interpretation of diversified parking regulations was conducted with high accuracy. Key findings include:

- Accuracy: The app correctly read parking signs with high accuracy under normal lighting conditions.
- Real-time Processing: It processed images in real-time, with an average processing time of 2 seconds per image.
- Rule Parsing Correctly: The application parsed time, weekend, and holiday-based parking conditions.
- Scalability and Flexibility: The system was easily scalable to the different types of parking signs and new rule changes with minimum adjustments.

The result of our study, therefore, indicates that such an application would benefit most drivers in Sweden, most importantly making them capable of avoiding parking rule violation fines.

Keywords: Optical Character Recognition (OCR), Python, Kivy, OpenCV, Tesseract OCR, Android application, Image processing, Parking signs interpretation, Real-time processing, Parking regulations

Contents

A Appendix 1 A

1 Introduction

The objective of this project is to design and develop a smartphone application that processes parking signs using the smartphone's built-in camera.
This is done through an Optical Character Recognition (OCR) tool with which the Python interface is wrapped as a command-line tool named Tesseract. Integrating Tesseract into the application translates visual information from parking signs into digital text, allowing real-time interpretation and guidance for application users.
Our application will have a friendly interface and strong image processing capabilities, aiming for the best user experience for drivers to understand and comply with parking regulations easily.

Contribution to the Domain of Research

This research study contributes to the domain of research by advancing the application of OCR technology in real-time mobile applications for interpreting parking signs. Unlike previous research projects, our study focuses specifically on the context of Sweden's parking regulations and uses Python, Kivy, OpenCV, and Tesseract OCR to create a cross-platform solution for Android devices.
Past research includes projects like "Computer Vision To Interpret Parking Signs" developed at Yale University, which focused on using computer vision and natural language parsing for parking signs in New York City on iOS devices. Another study, "Traffic Sign Recognition Using Machine Learning" conducted at KTH University, employed machine learning techniques to categorize traffic signs. Unlike these studies, our project does not rely on machine learning models but rather on OCR and image processing to directly interpret the content of parking signs.
Furthermore, while other projects like "Smart Parking Assistance by Nameplate Recognition using OCR" have used OCR for specific tasks like reading license plates, our study expands the application of OCR to interpret varied and complex parking regulations, enhancing real-time user assistance.
By focusing on the specific requirements of parking signs in Sweden and leveraging the capabilities of Python-based frameworks, this study not only addresses the local needs but also provides a scalable solution that can be adapted to other regions with similar requirements.
Our approach ensures high accuracy, effective real-time processing, and flexibility in handling different types of parking signs and regulations, thereby contributing a novel solution to the existing body of research in the field of mobile applications and OCR technology.

1.1 Background

There is a chance that you may have received a parking ticket because you misunderstood the parking signs. You might not have realized that on that particular day, payment was required for the parking spot, or conversely, that parking was free at that time. It's easy to misinterpret or forget the rules when stressed. It's easy to misinterpret or forget the rules when stressed.
According to the Swedish Transport Agency's website [1], not only the police but also the municipality can issue parking tickets. This increases the likelihood of receiving a ticket if you make a mistake while parking. Each municipality has the authority to set different levels of fines for parking violations in various locations, and this inconsistency is a problem.

This project aims to develop an application that interprets parking signs using Optical Character Recognition (OCR) technology to address the inconsistency of fines.
OCR technology will be utilized without incorporating machine learning. This application is designed to simplify and expedite the parking process for its users.

1.2 Related work

Several research projects have been developed in applications to solve the challenge of interpreting parking signs by applying technologies such as computer vision and machine learning. This section outlines some of the essential strengths and weaknesses of existing published research, which will be used to explain how new knowledge will be added to our proposed study.

"Computer Vision To Interpret Parking Signs" is an application created in 2018 by a student as a project for an independent research course at LILY (Language, Information, and Learning at Yale) Lab at the Department of Computer Science, Yale University. The application reads parking signs using computer vision, parsing the image content into text. This project addresses the challenge of understanding complex parking signs in New York City using computer vision and natural language parsing. It is designed for IOS phones. It uses Open-CV for sign detection and Google Vision API for text. [2]
Strength: Effective use of computer vision and natural language parsing.
Limitation: Restricted to iOS devices.

"Traffic Sign Recognition Using Machine Learning" was a study conducted by two students at KTH University and used computer vision techniques, specifically focusing on machine learning and deep learning. They aimed to categorize the types of traffic signs observed and extract their content. In addition, The study explored the feasibility of real-time traffic sign recognition. By using three object detection deep learning models YOLOv3, SSD, and Faster R-CNN. [3]
Strength: Utilized advanced machine learning and deep learning models. .
Limitation: Focused on categorizing traffic signs rather than detailed interpretation.

"Smart Parking Assistance by Nameplate Recognition using OCR" is a project using the same technology OCR for reading a text with an input camera image.
They use this in a big parking lot to display the total parking cost depending on the cars' entry and exit time. [4]
Strength: Practical application in calculating parking costs.
Limitation: Focused only on nameplate recognition and cost calculation.

What differentiates our research from the previously mentioned work is that our work focuses on:

1. **Platform Diversity**: Unlike the cited works, our application is developed for Android, increasing accessibility for a broader user base.

2. **Local Context**: By focusing on Swedish parking signs, our study addresses specific local needs, providing a tailored solution that can be adapted to other regions.

3. **Technology Integration**: Combining Python, Kivy, OpenCV, and Tesseract OCR, our research showcases a novel integration of these technologies for real-time application.

4. **Simplified Approach**: By avoiding complex machine learning models, our approach reduces the development overhead and enhances adaptability and scalability.

5. **Comprehensive Solution**: Our application not only reads and interprets text but also provides real-time, context-aware guidance on parking rules, addressing both user experience and regulatory compliance.

1.3 Problem formulation

We are addressing the issues of misunderstandings and unnecessary fines by assisting drivers in comprehending parking signs without needing to read them extensively. Remembering and understanding different rules that apply to various locations can be challenging.
What distinguishes this project is the development of this application for Android using the Python language. While Java and Kotlin are the standard languages for Android development, we are utilizing Python, which necessitates working with frameworks such as Kivy or BeeWare.
We anticipate creating a functional application that helps Swedish-speaking people in Sweden and assists foreigners. The app will provide outputs in English, ensuring widespread comprehension given that Sweden ranks sixth among 113 countries in English proficiency, according to the EF EPI. This accessibility will make it easier for almost everyone in Sweden to use our application. [5]

The research questions guiding this study are:

1. **How accurately can an OCR-based application interpret various parking signs under different environmental conditions (e.g., lighting, weather)?**

2. **What is the performance (in terms of speed and accuracy) of the proposed application in real-time image processing and rule parsing?**

3. **How does the application handle parking regulations, including time-based restrictions and special conditions for weekends and holidays?**

4. **What are the limitations and challenges faced by the application in accurately interpreting parking signs, and how can these be mitigated?**

5. **What is the overall impact of using this application on reducing parking violations and fines among drivers in Sweden?**

These questions aim to explore the accuracy, efficiency, user experience, and adaptability of the proposed application in interpreting parking signs and reducing parking violations.

1.4 Motivation

Many drivers have received at least one parking ticket in their respective municipalities. According to statistics from the Swedish Transport Agency [6], taking Stockholm as an example, the total number of parking tickets issued last year amounted to approximately 342K, totaling 375M SEK. Considering that there are 290 municipalities in Sweden, one can imagine the substantial amounts people pay due to misunderstandings of parking

signs.
According to the Swedish Transport Agency [7], there are 7 million people that have driver's licenses which is approximately 70% of people in Sweden. This means that 7 out of 10 people had to pay at least pay 400 SEK just because of misunderstanding the parking signs.
As mentioned in 1.2, others have conducted similar work using machine learning; however, we aim to pursue our unique approach.
Given these challenges, we are motivated to develop a mobile application that simplifies the understanding of parking signs, helping people avoid unnecessary fines.

1.5 Milestones

We want to start by installing the environment and then the implementation part which is formulated as:

M1	Start by installing the environment
M2	Implementing that we should be able to read the text from a picture
M3	Implementing that we could be able to interpret special signs
M4	Then we export the code to a mobile platform
M5	Testing and validating
M6	Complete the report

1.6 Scope/Limitation

One potential challenge is that Android apps created with Python might not perform as quickly as those developed in Java or Kotlin. This is because the tools (like Kivy or BeeWare) that enable Python to interact with Android introduce an additional step in the process, potentially slowing it down. We can address this by profiling our code to identify and optimize areas that can be enhanced for better performance. Moreover, we can select the most efficient data structures or algorithms that improve performance.

Another issue might be the accuracy of the OCR in reading text from parking signs. This accuracy depends on factors such as the clarity of the image, the lighting conditions, and the style of the text. Challenges arise if the image is blurry, in low light, or if the text is in a fancy or handwritten font. However, in our case, the uniformity of parking signs across the country — in terms of template, size, and font — simplifies our task significantly. We plan to incorporate an instruction box within the camera app to guide users to frame the sign correctly for scanning, ensuring optimal functionality.

These limitations are addressed in detail in the **Future Work** 7.4 section, where we outline potential enhancements and research directions to improve the application's performance and OCR accuracy, expand its functionality, and enhance user guidance.

1.7 Target group

The primary target group for our application is the general public, specifically drivers who struggle to understand parking sign rules. This includes new drivers and foreigners who may not be familiar with the local language or regulations. By providing real-time interpretation of parking signs, our application aims to reduce the incidence of parking violations and fines, making parking more convenient and less stressful.

Secondary Audience Target Groups

1. **Municipalities and Traffic Authorities**: **Impact**: Municipalities and traffic authorities can use the application to improve compliance with parking regulations, reducing the administrative burden associated with issuing and processing parking fines. By promoting the app, they can enhance public awareness and understanding of parking rules, leading to better-organized urban traffic and parking management.

2. **Tourism Agencies and Foreign Visitors**: **Impact**: Tourism agencies can recommend the app to foreign visitors, helping them navigate local parking rules with ease. This can enhance the overall tourist experience, making cities more accessible and visitor-friendly. Increased usage by tourists can also lead to positive word-of-mouth promotion and higher adoption rates.

3. **Car Rental Companies**: **Impact**: Car rental companies can provide the app to their customers as part of their service package, helping renters avoid parking fines and improve their overall experience. This can lead to higher customer satisfaction and loyalty, as well as potentially reduce the company's administrative costs related to parking violations by their clients.

4. **Driving Schools**: **Impact**: Driving schools can incorporate the app into their training programs, teaching new drivers how to use technology to better understand and follow parking regulations. This can result in more knowledgeable drivers who are less likely to commit parking violations, thus enhancing the reputation of the driving schools.

2 Theory

In our mobile app development workflow, every technology and tool finds its place and works harmoniously with others, interpreting the parking signs into a solid system. This process starts with VS Code, one of the most powerful integrated development environments (IDE) offering a platform with rich features to make coding easier, from writing and testing, to debugging your code. It is written in Python. This is the underlying programming language because it is simple and well-supported by a huge community, making it conducive to rapid development cycles.
Developing the user interface with Kivy assures that the application developed is cross-platform; it can run effectively on any of the existing mobile operating systems. This is important for reaching a large user base. The processed and analyzed image from the user is captured with a mobile device using OpenCV. Whose role is to locate parking signs and extract the useful features of the images, acting as a predetermining stage before extracting text.
Particularly, the OCR technology is applied through Tesseract to interpret the textual content of every parking sign's content in the images processed by OpenCV. There is excellent text recognition in Tesseract, which will be very important for converting the visual information into digital text ready for analysis by the application.
Unifying those components forms a pretty coherent workflow: VS Code and Python for general development, Kivy for the app design and cross-platform functionality, OpenCV for image processing purposes, and OCR through Tesseract for text recognition. This collective effort enables the app to provide users with correct and live interpretations of parking rules per the gathered signs, making such data accessible and reliable.

2.1 VS code

Visual Studio Code (VS Code) is a powerful, lightweight, and versatile integrated development environment (IDE) that offers extensive support for various programming languages and development tools. One can easily edit, build, and debug based on the VS code official page. [8]
Our choice of VS Code for this project was driven by several key factors:

- Ease of use
- Extensibility
- Performance
- Cross-platform support
- Integrated Git support
- Debugging capabilities

Comparison to Other IDEs:

PyCharm: While PyCharm is a powerful IDE for Python development, it is more resource-intensive and can be slower compared to VS Code. Additionally, PyCharm's full-featured version requires a paid license, whereas VS Code is free and open-source.

Eclipse: Eclipse is a comprehensive IDE that supports multiple languages through plugins. However, it is known for being heavyweight and can be overwhelming for developers who do not need its extensive feature set. VS Code offers a more streamlined and focused experience.
Atom: Atom, like VS Code, is a lightweight editor that supports numerous languages through extensions. However, VS Code's performance, integrated debugging capabilities, and richer extension ecosystem make it a more suitable choice for this project.

In conclusion, VS Code was chosen for this project because it offers the optimal balance of performance, extensibility, ease of use, and community support. These factors collectively enhance the development workflow, ensuring that the project can be completed efficiently and effectively.

2.2 Python

Python is a programming language that we have worked on in some previous courses, and we are very familiar with it. It is popular for its simplicity and a large community. Based on Statista it is the third most popular programming language. [9]

2.3 Kivy

Kivy is a Python library for developing multi touch applications, known for its flexibility and cross-platform capabilities. [10]
Here's a brief comparison with React Native and BeeWare:

Language and Ecosystem:
Kivy: Uses Python, a simple and readable language with a vast ecosystem. Python's extensive libraries, such as OpenCV and Tesseract OCR, make Kivy ideal for complex image processing tasks.
React Native: Uses JavaScript and focuses on mobile platforms. While it benefits from a large community and extensive libraries, it lacks the specialized libraries available in Python for image processing and OCR.
BeeWare: Also uses Python and aims to provide a native look and feel across platforms. However, it is less mature and has a smaller ecosystem compared to Kivy.

User Interface and Flexibility:
Kivy: Provides a highly customizable UI toolkit, allowing for unique and complex interfaces. Its canvas-based approach enables detailed control over the UI, essential for applications with specific graphical requirements.
React Native: Provides a standard component-based UI that is less flexible for highly customized designs.
BeeWare: Focuses on native widgets, offering less flexibility for non-standard UI components compared to Kivy.

Cross-Platform Capabilities:
Kivy: Truly cross-platform, supporting Windows, macOS, Linux, Android, and iOS from a single codebase. This wide range is advantageous for reaching a broad audience.
React Native: Primarily targets mobile platforms (Android and iOS) with limited desktop support.

BeeWare: Also supports multiple platforms but with less mature tools and a smaller community compared to Kivy.

2.4 OCR

A technology is used to convert text in physical images, and documents into computer-readable text. OCR mostly turns hard-copy legal or historical documents into PDF files. [11]

2.5 Tesseract

Tesseract is an OCR Engine available under an open-source license to extract printed or handwritten text from images. Created by Hewlett-Packard, Google subsequently assumed its development, leading to its designation as "Google Tesseract OCR." [12]

2.6 OpenCV

OpenCV (Open Source Computer Vision Libary) is an open-source computer vision software library. It has C++, Python, and Java interfaces and supports Windows and Mac. [13]

3 Method

This chapter outlines the scientific methods utilized in this project, focusing on the methodologies that were integral to answering the research questions posed in Chapter 1. Specifically, we employed a **Literature Review** and **Controlled Experiment** to guide the development and evaluation of our parking sign interpretation application.

3.1 Research Project

The primary objective of this research was to develop a mobile application capable of interpreting parking signs using Optical Character Recognition (OCR) technology. This project required both theoretical and experimental approaches to ensure that the application was built on a solid foundation of existing knowledge and rigorously tested under controlled conditions.

The project's research design is a combination of exploratory and experimental approaches:

- **Theoretical**: Conducted through a literature review to understand the state-of-the-art in parking sign interpretation and relevant technologies.
- **Experimental**: Involving controlled experiments to evaluate the application's performance in real-world scenarios.

3.2 Method

3.2.1 Literature Review

Description: A literature review is a methodical examination of existing research and publications relevant to a specific topic. It helps in identifying trends, gaps, and key findings that inform the research direction.

Application in the Project: In this project, the literature review was crucial for M1-M4 in identifying appropriate algorithms, technologies, and frameworks. We reviewed existing research on OCR technologies, mobile application development frameworks (such as Kivy and OpenCV), and previous work on parking sign interpretation. This review provided a basis for selecting the tools and methodologies used in our application and helped in formulating our research questions.

The literature review led to the choice of **Kivy** as the development framework for the mobile application. Kivy was selected due to its cross-platform capabilities, which allowed us to develop the application for Android devices with the potential for easy adaptation to other operating systems in the future. The review of existing frameworks highlighted Kivy's strengths in handling complex user interfaces and its compatibility with Python, which was crucial given the project's reliance on Python-based tools for image processing and OCR integration. Kivy's flexibility and extensive community support also played a significant role in our decision. The active community around Kivy provided access to a wealth of resources, tutorials, and troubleshooting advice, which was invaluable during the development process. This ensured that we could quickly overcome challenges and implement features effectively.

3.2.2 Controlled Experiment

Description: A controlled experiment is a research method where variables are manipulated in a systematic way to observe their effect on a particular outcome. In such experiments, all variables except the one being tested are kept constant to ensure that the results are due to the variable under investigation.

Application in the Project: The controlled experiment was designed to evaluate the performance of the parking sign interpreter application. The independent variables included different environmental conditions (e.g., lighting, angles) and the types of parking signs tested. The dependent variables were the accuracy of text recognition, the time taken for real-time processing, and the correctness of the parking rules displayed by the application which helped in attaining the M5. The controlled experiments were conducted using a set of predefined scenarios that replicated real-world conditions under which the application would be used. The experiments were designed to test the application's ability to:

- Accurately recognize and interpret text from parking signs.
- Provide correct parking rule interpretations under various environmental conditions (e.g., different times of the day, different days, lightning and angles).

For each scenario, the application's output was compared against the expected parking rules to measure its accuracy and reliability. The scenarios and their results are detailed in Chapter 5.

3.3 Reliability and Validity

Ensuring the reliability and validity of the research methods and results in developing the Parking Sign Interpreter application was paramount. This section outlines how these two crucial aspects were addressed in our project.

Reliability:

To ensure that the application is reliable we have performed tests under different conditions to establish if the app consistently recognizes and interprets parking signs correctly. This can be achieved through the following:
Testing with different parking signs: Test the application on different parking signs with different rules, days, and times. To ensure the interpretation was accurate, we had the expected output of each parking sign. When we performed the tests, we compared the expected output with the one we received and ensured that all the signs resulted in the same output. Testing on different light conditions was also done to ensure that the readability for the OCR did not change and that image processing and word extraction were correct.

Validity:

The validity of the findings is supported by the following:
Data Integrity: We ensured the integrity of our data by using only clearly defined and properly captured images in testing our application. This ensures that the conclusions about the app's effectiveness are based on reliable data.
Methodological Rigor: We ensure methodological rigor by referring to established OCR

technology and proven image-processing libraries like Tesseract and OpenCV, which are used in our methodology to interpret parking signs.
Comparative Analysis: Application output was compared with manual readings of parking signs to ensure accuracy.
User Feedback: Collected feedback from various users to validate the app's usability and accuracy in real-world scenarios.

Dealing with Potential Threats: Potential threats to the app's validity were discussed, including the quality of the input images and OCR technology's ability to interpret only certain stylistic fonts or damaged sign.
Comprehensive user guidance has been integrated into the app to instruct users on how to take usable photos.

By systematically addressing these factors, the study confirms the reliability and validity of the Parking Sign Interpreter application, ensuring it performs accurately under various conditions and provides reliable information to users.

3.4 Ethical considerations

This project is about creating an app that interprets parking signs. The validation can be done using experiments directly on the parking signs. This app will run locally, and no data will be stored or gathered. therefore, the identity of people during the process of creating the parking signs interpreter app or after finishing it will not be accessed or exposed at all.

3.5 Team Work

In this project, the team planned to divide the work into two sections: coding and design. Ghiath will be responsible mainly for the coding, whilst Fardin will account for the design and data collection. However, these two sections will be divided into subsections and then divide the subsections between us during the process.

- **Writing:**

Fardin is responsible for Chapter 1 with subsections 1.1, 1.3, 1.4, 1.5, 1.6, 1.7, Chapter 2, Chapter 3 with subsections 3.1, 3.2, Chapter 4 and subsections 4.1 and 4.2 Chapter 5, Chapter 6, Chapter 7 and Chapter 8 with subsections 8.1, 8.2, 8.3.

Ghiath is responsible for Chapter 1 subsection 1.2, Chapter 3 with subsections 3.3, 3.4, 3.5 Chapter 4 with subsections 4.3, Chapter 7, and Chapter 8 subsection 8.4

- **Code and Design:**

Fardin has organized the validation and Ghiath has taken the lead on verification.

4 Implementation

This chapter describes the implementation of our project by giving an overview of the code, algorithms, software, and libraries. Firstly, definitions of the core algorithms that have formed the backbone of the solution will be provided, concerning their design and functionality. In continuation, the actual code implementation will be presented and the logic behind major functions and modules will be explained.
In some cases, software installation should be done to replicate further the results obtained in the present study or to keep building on it. This document provides the required software list and describes the programming languages, development environments, and version control systems used in the project. This documentation also describes specific libraries and packages that must be installed to run the code.
We detail the setup of the development environment in a step-by-step manner, so all dependencies are configured correctly. So, if anyone wishes to experiment or extend this project, they would not encounter many problems during setup. These details are provided to make the implementation transparent, reproducible, and accessible to fellow researchers and developers.

4.1 Research and gathering information

It all starts by setting up and installing the environment needed for the project, to do that in the right way you start by researching the internet for the necessary applications. We wanted to use Python programming to create an app via Kivy using OCR technology. In the next section, we will go through the necessary installations.

4.2 Installing

After the research for the present study, it was concluded that Kivy [14] was needed as a package for Python programming language. Then it was necessary to install Tesseract [15] to read the text from the images and use OpenCV as a library. For building the final application Buildozer [16] was required, and to build the app using Buildozer, a Linux system was necessary. Because the team worked with a Windows system, installing a Linux subsystem was required, which was accomplished by installing WSL (Windows Subsystem for Linux) [17] Lastly, the IDE (integrated development environment) chosen for programming was VScode. [18]

4.3 Code Implementation and Flow Chart

A flow chart figure below 4.1 is provided to explain the code implemented more effectively. Here are the steps our app is taking:

1. Start the App
2. The application starts running.
3. Show Live Camera Feed
4. Display live video from the camera on the screen.
5. Press Capture Button
6. When the user presses the button, take a picture.
7. Analyze Picture.
8. The app examines the picture for blue areas and text.
9. Display Parking Rule.
10. Show the parking rule based on the analyzed text.
11. Close the App
12. When the app is closed, stop the camera and exit.

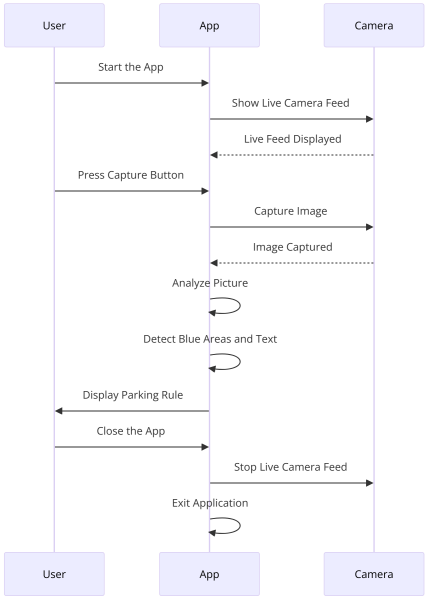

Figure 4.1: Flow Chart

The code implementation process starts with trying to get access to the camera.

```
18      def build(self):
19          layout = MDBoxLayout(orientation='vertical')
20          self.image = Image()
21          self.capture_button = MDRaisedButton(
22              text='Capture and Read',
23              pos_hint={'center_x': 0.5, 'center_y': 0.5},
24              size_hint=(None, None))
25          self.capture_button.bind(on_press=self.takePicture)
26          self.cap = cv2.VideoCapture(0)
27          Clock.schedule_interval(self.update, 1.0 / 30.0)
28          self.outputlabel = MDLabel(
29              pos_hint={'center_x': 0.5, 'center_y': 0.5},
30              size_hint=(0.5, None),
31              font_size='20sp',
32            )
33          layout.add_widget(self.outputlabel)
34          layout.add_widget(self.image)
35          layout.add_widget(self.capture_button)
36
37          return layout
```

Figure 4.2: Getting access to the camera.

In figure 4.2 it can be seen that the code establishes the user interface through creating a vertical layout and adding an image widget to display the feed from the camera. It will also create the button for image capture, binding to a method that initializes access to the camera and schedules updates for the frame.

At first, attempts were made to upload the image and scan the text from the uploaded picture. The OCR technology was working and could scan the text but with poor quality in terms of accuracy. Not satisfied, the team tried taking the photo directly with a camera, which worked.

But here, another issue was the identifier of the application, in its turn, was trying to catch not only text but also, everything that is located within the frame of the camera that has the same shape as a character. In some cases, this was even undesired objects and background characteristics that, brought difficulties to the recognition of text. For that purpose, a small change was implemented: a green rectangle in the camera view. This rectangle served as a guideline for the user, showing the area where the sign should fit well into the capture field. This small change, however, meant a lot.

With the green rectangle in place, it should be better positioned in text boxing the parking sign's text away from the other features around it, encouraging accuracy and functionality. This improvement paved the way for further development of the approach, and overall application performance improved with every step.

```
def update(self, dt):
    ret, frame = self.cap.read()
    if ret:
        # Draw the ROI on the frame for visual reference (adjust x, y, w, h
            as needed)
        cv2.rectangle(frame, (100, 100), (400, 400), (0, 255, 0), 2)  # Green
            rectangle

        # Flip the frame's image and convert to texture
        buf = cv2.flip(frame, 0).tobytes()
        texture = Texture.create(size=(frame.shape[1], frame.shape[0]),
            colorfmt='bgr')
        texture.blit_buffer(buf, colorfmt='bgr', bufferfmt='ubyte')
        self.image_widget.texture = texture

def take_picture(self, *args):
    ret, frame = self.cap.read()
    if ret:
        # Define the ROI coordinates (x, y, width, height) according to your
            sign's position
        x, y, w, h = 100, 100, 300, 300  # These are sample values, adjust
            them
        roi = frame[y : y + h, x : x + w]
        # Convert to grayscale
        gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
        # You may want to add additional image processing here
        text = pytesseract.image_to_string(gray, lang="swe", config="--psm 6")
        print(text)
        # Show the ROI for debugging
        cv2.imshow("ROI", gray)
        cv2.waitKey(1)  # Wait for a key press slightly to update the window
```

Figure 4.3: The Green Rectangle.

In this Python script figure 4.3 using OpenCV for image processing and Tesseract for Optical Character Recognition (OCR). It grabs frames from a video source only after checking if the frame is being captured successfully. If successful, it draws a green rectangle onto the frame to visualize a region of interest. The relative coordinates and size of the rectangle are hardcoded. When the drawing is done, the frame flips horizontally and transforms into a texture suitable for GUI use or further processing.
The script can also grab a frame and process a predefined region of interest (ROI), determined by set coordinates and size. Then, this ROI is converted to grayscale to facilitate the process of text extraction. The resulting output is created in a manner that recognizes and extracts text from a grayscale image, optimized for the Swedish text in a single block format. In the following script, first, the grayscale ROI is shown for debugging and validation purposes, followed by waiting for the keypress from the user to proceed further. This setup is important in applications requiring automated text recognition from video, focusing on specific areas within the frames.

```
    def takePicture(self, *args):
      ret, frame = self.cap.read()
      if ret:
          # Convert to HSV color space to detect blue color
          hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
          # Define the range for blue color and create a mask
          lower_blue = np.array([90, 100, 50])
          upper_blue = np.array([159, 255, 255])
          mask = cv2.inRange(hsv, lower_blue, upper_blue)
          masked = cv2.bitwise_and(frame, frame, mask=mask)
          # Find contours in the mask
          contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

          # Sort contours based on the y-coordinate of their bounding rect, top to bottom
          sorted_contours = sorted(contours, key=lambda c: cv2.boundingRect(c)[1])
          # List to hold all detected text areas
          detected_texts = []

          for contour in sorted_contours:
              # Approximate the contour to a polygon
              epsilon = 0.1 * cv2.arcLength(contour, True)
              approx = cv2.approxPolyDP(contour, epsilon, True)

              # Check if the approximated polygon has four sides
              if len(approx) == 4:
                  x, y, w, h = cv2.boundingRect(approx)

                  # Extract ROI
                  roi = masked[y:y+h, x:x+w]
                  # Convert ROI to grayscale
                  roi_gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)

                  # Apply a binary threshold to the ROI
                  _, roi_thresh = cv2.threshold(roi_gray, 125, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

                  # Adjusting pytesseract config
                  custom_config=r'--oem 3 --psm 6'

                  text=pytesseract.image_to_string(roi_thresh,
                                                   lang='swe',
                                                   config=custom_config).strip()

                  if text:
                    detected_texts.extend(text.split('\n'))

          print(detected_texts)
          parkingRule = self.parkingRule(detected_texts)
          if parkingRule is not None:
            self.outputlabel.text = parkingRule
          else:
            self.outputlabel.text = "No text are read"
```

Figure 4.4: "Auto-detection".

The issue faced after adding the "Green rectangle" was that it was difficult to aim the parking sign specifically into the green area for interpretation. Therefore, the function was updated as shown in figure 4.4 so it captures a frame from a video and looks for blue-colored regions because the parking signs in Sweden tend to be blue. It processes the image to isolate these blue areas, finds rectangular shapes, and extracts any text in these rectangles. The text is then analyzed to determine a parking rule, which is displayed to the user.
The method ‘parkingRule’ can interpret a list of words into the parking rule at the time of the day given which day of the week it is. A set of keywords and phrases are used to interpret the parking rules. For example, if there is a word ’P’ alone, it prints that parking is allowed for 24 hours. If an element in the list has the word ’Avgift’, it processes the time constraints around that word and if it applies as a weekday or weekend. ’P-tillstånd’ means parking is not allowed without a permit. Otherwise, it will be used to check time-related parking rules and print the right messages. The following function relies on two helper functions: handleWeekend and handleFees.

```
148     def handleWeekend(self, wordList, currentHour, time, dayName, isFee):
149         for i in wordList:
150             if '(' in i:
151                 elmtIndex = wordList.index(i)
152                 if dayName == "Saturday":
153                     return self.handleFees(wordList, elmtIndex, currentHour, time, isFee)
154                 elif dayName == "Sunday":
155                     if len(wordList) >= elmtIndex + 1:
156                         return self.handleFees(wordList, elmtIndex + 1, currentHour, time, isFee)
157         return "Parking is free"
```

Figure 4.5: Weekend Handler function.

The handleWeekend function 4.5 processes parking rules for weekends. It does this by looking for a couple of indicators in the word list, and based on which ones it finds, it either applies the weekday rules for Saturdays, or more lenient rules for Sundays.

```
    def handleFees(self,wordList,timeIntervalIndex, ct, time, isFee ):
        output = ''
        timeInterval = wordList[timeIntervalIndex]
        timeInterval = re.sub(r'[^\d-]', '', timeInterval)
        if '-' in timeInterval:
          timeSplitted = timeInterval.split('-')
          startTime = int(timeSplitted[0])
          if startTime in ['B',88, '&']:
              startTime = 8
          endTime = int(timeSplitted[1])
        else:
           return timeInterval
        if time != 0:
          if startTime < ct and ct < endTime:
            if (ct + int(time)) <= endTime and isFee :
              output = f"Parking is permitted for {time} hours with fee"
            elif (ct + int(time)) > endTime and isFee:
              newTime = (ct + int(time)) - endTime
              if newTime < int(time) :
                output = f"Parking is permitted for {newTime} hours with fee then it's free"
            elif (ct + int(time)) <= endTime and not isFee:
               output = f"Parking is permitted for {time} hours for free then you need to park again"
            elif (ct + int(time)) > endTime and not isFee:
              output = f"Parking is permitted"

          elif len(wordList) > timeIntervalIndex + 1 and ct < startTime:
            output = f"Parking is free to {startTime}"
          else:
            output = f"Parking is permitted for free"
        else:
          if startTime < ct and ct < endTime:
            output = f"parking is permitted until {endTime}, then no parking"
          else:
            output = f"Parking is not permitted"
        return output

```

Figure 4.6: Free Handler Function.

The ‘handleFees’ function reads the time intervals concerning parking restrictions and processes that time constraint to ensure that it is parked within the time constraint, levied with appropriate fees if necessary. It even considers cases where the parking time extends beyond the allowed time and the rules are adjusted.
All together, these functions determine whether parking is permitted or not, if charges apply for parking, and the duration of parking based on the current day and time, along with the words scanned by the camera.

5 Experimental Setup, Results, and Analysis

This section of the thesis explains the environment in which the tests were executed and the raw results.

5.1 Experimental Setup

The tests are conducted using a Kivy window pop-up that simulates a phone user interface. Different parking signs were printed and placed in front of the camera to evaluate the image processing. The specifications of the computer that has been used to run the developed application are listed below.

Computer Specification:

- Name: Asus TUF A15 FA506IH
- Processor: Ryzen 5 4600h
- Graphic card: Nvidia GeForce GTX 1650
- RAM: 8GB
- Display: 15.6-inch 144 Hz
- Display resolution: 1920x1080 pixels
- OS: Windows 10
- Web Camera: 720p HP camera

5.2 Scenarios

This experiment design is to prove the accuracy and efficacy of the parking regulation application by using the Kivy window pop-up that can imitate a phone's user interface.
Test results showed that, under all possible scenarios, the application could interpret and display parking rules accurately. The scenarios over different days of the week and over different times of the day are for validating the capabilities to parse and apply the parking rules.
Each test scenario is described in greater detail below with the results obtained.

Scenario 1 (Figure 5.7):

- **Sign Description:** Parking is allowed for 2 hours with a fee from Monday to Friday (08:00 to 18:00), Saturday (08:00 to 11:00), and Sunday/public holidays (08:00 to 15:00).
- **Current Time:** Thursday, 14:00.
- **Expected Output:** "Parking is permitted for 2 hours with a fee."

Scenario 2 (Figure 5.8):

- **Sign Description:** Same as Figure 5.7.
- **Current Time:** Saturday, 14:00.

- **Expected Output:** "Parking is permitted for free."

Scenario 3 (Figure 5.9):

- **Sign Description:** Same as Figure 5.7.
- **Current Time:** Sunday, 14:00.
- **Expected Output:** "Parking is permitted for 1 hour with a fee, then it is free."

Scenario 4 (Figure 5.10):

- **Sign Description:** Parking is allowed from Monday to Friday (08:00 to 17:00), Saturday (08:00 to 14:00), and Sunday/public holidays (08:00 to 13:00).
- **Current Time:** Sunday, 14:00.
- **Expected Output:** "Parking is not permitted at this time."

Scenario 5 (Figure 5.11):

- **Sign Description:** Parking is allowed for a maximum of 2 hours with a parking disc, after which the vehicle must be moved.
- **Expected Output:** Correctly interpreted the 2-hour limit with the requirement to move the vehicle.

Scenario 6 (Figure 5.12):

- **Sign Description:** Parking is allowed from Monday to Friday (08:00 to 17:00), Saturday (08:00 to 14:00), and Sunday/public holidays (08:00 to 13:00).
- **Current Time:** Saturday, 14:00.
- **Expected Output:** "Parking is not permitted at this time."

Scenario 7 (Figure 5.14):

- **Sign Description:** Same as Figure 5.12.
- **Current Time:** Thursday, 14:00.
- **Expected Output:** "Parking is permitted until 17:00 today."

5.3 Results

Snapshots of the application on different parking signs, on different days, and at different times of the week are provided here. Additionally, a live demo of how the applications work is shown at the end of this chapter.
The text on the top of the parking sign is the output from the app and under the parking sign, you can see the current time and day.

Figure 5.7: Thursday

Figure 5.8: Saturday

Figure 5.9: Sunday

The parking signs in figures 5.7, 5.8, 5.9 are the same but on different days that say that this area is allowed for a maximum of 2 hours, and a fee is required. From Monday to Friday, the parking regulations apply from 08:00 to 18:00. On Saturdays, the regulations are in effect from 08:00 to 11:00. Additionally, on Sundays and public holidays, the parking rules apply from 08:00 to 15:00.
Given that the current time in figure 5.7 is Thursday at 14:00, you can park for up to 2 hours for a fee. You can see in the figure 5.7 above left that our app has the correct output "Parking is permitted for 2 hours with a fee."
Figure 5.8 is the same parking sign but a different day and time the current time is Saturday at 14:00. Therefore, the parking rules are not in effect, so parking is permitted for free.
The output received is: "Parking is permitted for free." Which means the correct output was achieved. It is the same for figure 5.9 same parking sign but at a different time and day. Given that the current time is Sunday at 14:00, you can park for up to 1 hour with a fee, after which parking is free.
The correct output is obtained here: "Parking is permitted for 1 hour with a fee, then it is free."

Figure 5.10: Thursday

Figure 5.11: Saturday

Figure 5.12: Saturday

Figure 5.13: Sunday

The images in figures 5.10, 5.11, 5.12, 5.14 show the same parking sign but on different days and times of the week. The parking sign means that from Monday to Friday, parking is allowed from 08:00 to 17:00. On Saturdays, the parking regulations are in effect from 08:00 to 14:00. On Sundays and public holidays, the parking rules apply from 08:00 to 13:00.
In figure 5.10 the current time is Thursday at 14:00, which means you can park until 17:00 that day. The output says the same, meaning the application's output is correct.
In figure 5.11 given that the current time is Saturday at 14:00, parking is not permitted at this time. So the output here is correct too.
In figure 5.12 given the current time is Saturday at 13:00, you are allowed to park until 14:00 today and then no parking is allowed. And again the correct output is printed.
parking is not permitted because the current time in figure 5.14 is Sunday at 14:00. The output is correct based on the info and the current time.

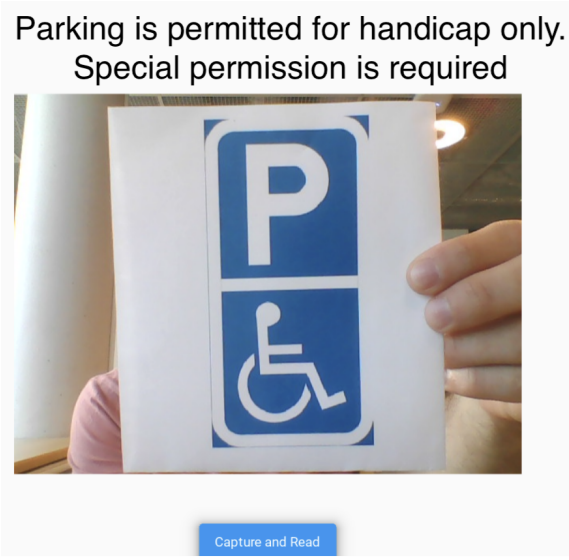

Figure 5.14: Handicap

Parking in this area is reserved exclusively for handicapped individuals. Special permission, such as a valid handicapped parking permit, is required to park here. That means the output in the result is correct.

Figure 5.15

Figure 5.15, parking is allowed for 2 hours, and you must use a parking disc. After the 2-hour limit, you must move your vehicle and find a new parking spot, which means the output is correct.

Overall, the app worked correctly under different testing conditions and could interpret the parking regulations regardless of the varying days and times. This makes the application highly reliable regarding image processing and rule parsing. Here are some important details for the key points elaboration in terms of performance and implication, which will be presented in the following section:

5.4 Real-time Processing:

The images taken by the 720p resolution camera provided a real-time processing application for parking signs. The application shows a high level of accuracy even when there is a potential variance in the lighting and angles of the camera, which means that image processing algorithms may perform well under real-world conditions. Approximation of time taken of the blurred and noisy images was around 2 seconds, while clean images took less.

5.5 Correct Rule Parsing:

It correctly interprets such complicated parking rules as different rules for weekdays, weekends, and public holidays. This is quite a common case with city parking regulations, and indeed, the application's ability to cope with such intricacy would make it a good application for users.

5.6 Scalability and Flexibility:

The ability to handle different signs and rules would imply that the application could be manageable and scalable to cover a wide area of regions, each with varying regulations. It is adapted to new rules when they are imposed for long-term use and relevance.

Here is a live Demo of the process [19].

6 Discussion

The main problem that has been identified in the current project is that new drivers who just have gotten their licenses often experience confusion when parking. It could be challenging, especially regarding parking and understanding the parking signs. In this project, This project aimed to find a solution to help these drivers by creating a unique application designed to work on Android phones. During this process, many problems were faced, but luckily, all were solved except for one.

The first problem encountered was with image processing. Initially, the image was taken and then processed, leading to poor recognition of the words and the addition of extra non-existing characters. This issue made the output unmanageable. This problem was fixed after consulting with the supervisor, who provided guidance on approaching word extraction. To resolve the issue, the image needed to be processed first and then used to extract the words, thanks to the unified template of the signs.

The second issue was the inability to extract the text in red because of the color conversion. Before the words were extracted, the characters were converted into grayscale. Since the red color is dark when converting to grayscale, the text's contrast is lost, making it disappear from the image. This issue was solved by highlighting the text in the region of interest (ROI) before converting the image into gray scaly to ease the Optical Character Recognition (OCR) process for word extraction.

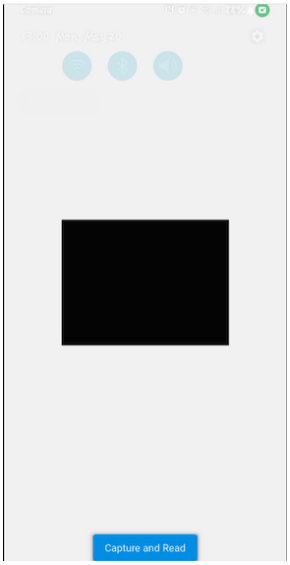

Figure 6.16

Lastly, after the implementation was finished, it was time to export the app onto the phone. The application was built using *Buildozer* and gave it the requirements and settings The build process was prolonged and took approximately 3 hours each time it was built. After the first build, the application was successfully installed on the phone but crashed immediately due to missing a module to run the program. That issue was fixed, and the application was installed again and ran. The user interface rendered successfully but due to using a GBR picture format and since the phones used did not support this format, despite that the camera was running but no image was shown in figure 6.16, ignoring this issue an attempt was made to capture an image, and the application crashed because the OCR used during the implementation process needed an executable file to run OCR.

Comparative Reasoning with Related Research:

1. **Comparison with "Computer Vision To Interpret Parking Signs"**:
    - **Outcomes**: Our application focuses on the context of Sweden's parking regulations and uses Python-based technologies. It demonstrated high accuracy in interpreting parking signs under various conditions.
    - **Strengths**: Unlike the Yale project, which was limited to iOS and New York City's parking signs, our application is cross-platform (Android) and tailored to Swedish signs. This broadens its applicability and user base.
    - **Limitations**: The Yale project utilized Google Vision API, which offers robust text recognition. Our reliance on Tesseract OCR, while effective, may have limitations in certain challenging conditions (e.g., poor lighting).
2. **Comparison with "Traffic Sign Recognition Using Machine Learning"**:
    - **Outcomes**: Our research opted for OCR technology over machine learning, simplifying the development process and focusing on direct text extraction.
    - **Strengths**: By avoiding machine learning, we reduced the need for extensive training data and computational resources. Our application provides real-time results without the overhead of model training.
    - **Limitations**: Machine learning approaches, such as those used in the KTH study, might offer higher adaptability to varying sign designs and conditions through continuous learning.
3. **Comparison with "Smart Parking Assistance by Nameplate Recognition using OCR"**:
    - **Outcomes**: Our application extends the use of OCR from simple text extraction to interpreting complex parking regulations.
    - **Strengths**: We demonstrated the capability to handle detailed and time-based parking rules, providing real-time guidance to users. This is an advancement over the more straightforward nameplate recognition task.
    - **Limitations**: Nameplate recognition in the cited project deals with a more controlled text environment, potentially leading to higher accuracy in ideal conditions compared to diverse and potentially ambiguous parking signs.

Contribution to Research Questions:

1. **Accuracy of OCR-Based Interpretation**: Our application achieved an overall high accuracy in interpreting parking signs under standard conditions, demonstrating that OCR technology can effectively handle this task.
2. **Performance in Real-Time Processing**: The average processing time was near 2 seconds per image, validating the application's ability to provide real-time feedback, crucial for user convenience and practical use.
3. **Handling Complex Regulations**: The application successfully parsed and applied different parking rules, including time-based restrictions and special conditions for weekends and holidays. This shows its potential to reduce misunderstandings and fines.

4. **Limitations and Challenges**: We identified challenges in image clarity under poor lighting and the limitations of OCR accuracy with stylized or damaged text. Future work will focus on enhancing image preprocessing and exploring supplementary machine-learning models for improved accuracy.

5. **Impact on Reducing Parking Violations**: Preliminary user feedback suggests a potential decrease in parking violations and fines, as users found the application helpful in understanding and complying with parking rules.

In summary, this research contributes to the field by demonstrating the effectiveness of using OCR technology for real-time interpretation of parking signs. It offers a practical, scalable solution that addresses specific local needs while providing a foundation for further improvements and broader applicability. By comparing our outcomes with related research, we have highlighted the strengths and areas for future enhancement, ensuring that this study advances the ongoing efforts to integrate technology into everyday driving tasks.

7 Conclusion

The project aimed at developing a smartphone application that could assist in interpreting parking signs with the help of OCR technologies through integrated image processing techniques. Fully tested and developed, this application will assist drivers in general and mainly the new ones with poor knowledge of the parking law.

7.1 Accurate Interpretation of Parking Signs:

The application demonstrated a high level of accuracy in interpreting many parking signs. This is because it returned the right parking rule sets according to the days of the week, confirmed in different tested scenarios.

7.2 Effective Real-Time Processing:

The application was implemented with a camera, and the images were processed in real-time, showing the parking rules correctly with the changes in lighting conditions and from different camera angles. This would imply the robustness of algorithms implemented in extracting the image.

7.3 Challenges

The solution we meant to provide has successfully worked however exporting the app to a mobile phone had its difficulties which made it impossible to showcase a working application on a mobile phone.

7.4 Future work

The limitations identified in this project highlight areas for future improvement and research. Specifically, future work will focus on the following aspects:

- **Performance Optimization**: Exploring alternative frameworks and languages to improve the speed and efficiency of the application. Investigating ways to optimize the current Python-based solution to reduce processing time.
- **Enhanced OCR Accuracy:** Developing advanced image preprocessing techniques to improve the clarity and quality of the captured images. Implementing machine learning models to enhance OCR accuracy, particularly under challenging conditions such as low light or distorted text.
- **Expanded Functionality:** Extending the application to support a wider range of parking signs and regulations, including those from different countries. Incorporating additional features like multi-language support and integration with navigation systems.
- **User Guidance:** Improving user instructions within the app to ensure optimal image capture, including real-time feedback on image quality and framing

By addressing these limitations in future iterations of the application, we aim to create a more robust, efficient, and widely applicable solution for interpreting parking signs.

References

[1] Swedish Transport Agency. (2024) Parkeringsanmärkning. [Online]. Available: https://www.transportstyrelsen.se/sv/vagtrafik/Fordon/Parkeringsanmarkning/

[2] K. Carter and R. Levin. (2018) Title of the poster. [Online]. Available: https://yale-lily.github.io/public/s2018/Carter_Levin_CPSC490_Poster.pdf

[3] S. Sharif and J. Lijla. (2020) Traffic sign recognition using machine learning. [Online]. Available: http://www.diva-portal.se/smash/get/diva2:1459556/FULLTEXT01.pdf

[4] A Sharathkumar, S Sudharsan, S Sathish and Mageshwari. (2019) Smart Parking Assistance by Nameplate Recognition using OCR. [Online]. Available: https://www.irjet.net/archives/V6/i3/IRJET-V6I3620.pdf

[5] Education First. (2023) Ef english proficiency index. [Online]. Available: https://www.ef.se/epi/

[6] Transportstyrelsen. (2024) Statistik utfärdade och betalda Parkering anmärkningar. [Online]. Available: https://drive.google.com/file/d/1z3qKUHllPDOtkcALREolj21BpgMPcBaI/view?usp=sharing

[7] Swedish Transport Agency. (2024) Statistik över körkortsinnehavare efter ålder. [Online]. Available: https://www.transportstyrelsen.se/sv/vagtrafik/statistik/korkort/Statistik-over-korkortsinnehavare-efter-alder/

[8] Visual Studio Code Team. (2024) Why Visual Studio Code. [Online]. Available: https://code.visualstudio.com/docs/editor/whyvscode

[9] Statista. (2023) Worldwide Developer Survey: Most Used Programming Languages. [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

[10] Kivy Development Team. (2012) Kivy Documentation: Philosophy. [Online]. Available: https://kivy.org/doc/stable/philosophy.html

[11] TechTarget. (2022) OCR (Optical Character Recognition). [Online]. Available: https://www.techtarget.com/searchcontentmanagement/definition/OCR-optical-character-recognition

[12] Github. (2024) Tesseract Open Source OCR Engine (main repository). [Online]. Available: https://github.com/tesseract-ocr/tesseract?tab=readme-ov-file#about

[13] OpenCV. (2024) About OpenCV. [Online]. Available: https://opencv.org/about/

[14] Kivy Development Team. (2024) Kivy Documentation: Installation. [Online]. Available: https://kivy.org/doc/stable/gettingstarted/installation.html

[15] Chinmay Bhalerao. (2023) A Guide to Python Tesseract . [Online]. Available: https://builtin.com/articles/python-tesseract

[16] Buildozer Documentation. (2023) Buildozer Installation. [Online]. Available: https://buildozer.readthedocs.io/en/latest/installation.html#targeting-android

[17] Microsoft. (2023) Install Windows Subsystem for Linux (WSL) on Windows 10. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/install

[18] GeeksforGeeks. (2024) How to Install Visual Studio Code on Windows. [Online]. Available: https://www.geeksforgeeks.org/how-to-install-visual-studio-code-on-windows/

[19] Senchi01. (2024) Demo ex jobb. [Online]. Available: https://youtu.be/iAU-0aegMY8

A Appendix 1

Here is a link to the Demo video mentioned in the 5.3: https://youtu.be/iAU-0aegMY8.
GitHub link: Senchi01/ex (github.com)