FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

MLKit Text Recognition Evaluation

Matheus Pereira Gonçalves

Mestrado em Engenharia Informática e Computação

Supervisor: António Miguel Pontes Pimenta Monteiro

March 13, 2022

© Matheus Gonçalves, 2022

MLKit Text Recognition Evaluation

Matheus Pereira Gonçalves

Mestrado em Engenharia Informática e Computação

March 13, 2022

Resumo

Reconhecimento de Texto é uma área da visão por computador que foca na extração de texto numa imagem. Com o aparecimento de dispositivos inteligentes, houve a necessidade dos computadores reconhecerem e transmitirem com precisão a informação para o utilizador. Com reconhecimento de texto conseguimos, por exemplo, fazer com que carros inteligentes detetem sinalização rodoviária ou traduzir documentos físicos para o mundo digital. No mundo do desenvolvimento de software, existem ferramentas que auxiliam os desenvolvedores na implementação de reconhecimento de texto. Contudo, maior parte destas ferramentas são pagas ou necessitam de um pré-processamento de imagem mais trabalhado (ABBY e Tesseract). Escolhemos então, uma ferramenta nativa do Android para implementar este reconhecimento nos dispositivos móveis. MLKit é um kit de desenvolvimento de software (SDK), desenvolvido pela Google, que fornece soluções de aprendizagem por máquina a dispositivos móveis. Desenvolvemos uma livraria Android capaz de traduzir imagens de faturas para texto. Além disso, adaptamos a livraria para suportar múltiplos casos de leitura, facilitando o trabalho do desenvolvedor quando implementa novas funcionalidades. Apesar de não haver muita pesquisa sobre o MLKit, testamos o algoritmo de reconhecimento com um dataset desafiador: Total-Text. Os resultados provam que o MLKit está longe da precisão atingida por algoritmos do estado da arte, com apenas 33.73% de precisão para palavras completas. Contudo, MLKit é adaptado para fornecer um execelente tempo de execução nos dispositivos móveis. Além disso, provou ser melhor para reconhecer text frontal e horizontal, enquanto teve problemas em reconhecer texto curvo e multi orientado. É uma ferramenta grátis, compatível com múltiplas plataformas e fácil de implementar, sendo a solução perfeita para efeitos comerciais e escaneamento de simples documentos.

Abstract

Text Recognition is a computer vision field that focuses on extracting text present in images. With the appearance of intelligent devices, there was this need for computers to interpret the physical world accurately and to transmit information to the user. Text recognition enabled, for example, intelligent cars to detect road signs or computers to translate entire printed documents to a company's database. In the software development world, there are many tools that developers can use to implement text recognition. However, most of these are paid services or require significant effort in image pre-processing (ABBY and Tesseract). We chose an Android native tool to implement on-device recognition. MLKit is a Software Development Kit (SDK) developed by Google that brings machine learning solutions to mobile devices. We develop an Android library capable of translating invoice images to text. Furthermore, we adapt the library to embrace multi use-cases, facilitating the developer work when implementing new futures. Although there is little research on MLKit, we tested it against a challenging dataset: Total-Text. The results proved to be far from benchmark algorithms, with a precision of about 33.73% for full words. However, MLKit is adapted to provide mobile devices with excellent execution time. Not only that, MLKit best recognizes horizontal and frontal text while it struggles with multi-oriented and curved instances. It is free, multi-platform compatible and easy to implement, making it the perfect tool for commercial purposes and simple document scans.

Agradecimentos

Quero agradecer à minha familia e namorada por me terem acompanhado nesta jornada de que me orgulho.

Matheus Gonçalves

Contents

1 Introduction **1**
1.1 Context and Motivation . . . 1
1.2 Objectives . . . 2
1.3 Methodology . . . 2
1.4 Document Structure . . . 2

2 Text Recognition **3**
2.1 Computer vision . . . 3
2.2 Text Detection . . . 4
2.2.1 Proposal-based . . . 4
2.2.2 Part-based . . . 5
2.2.3 Instance Segmentation . . . 5
2.3 Text Recognition . . . 5
2.4 On-Device Recognition . . . 5
2.5 Datasets . . . 6
2.5.1 Synth90k . . . 6
2.5.2 IC13 . . . 7
2.5.3 COCO-Text . . . 7
2.5.4 Total-Text . . . 7

3 Library Requirements **9**
3.1 Functional Requirements . . . 9
3.2 Quality Requirements . . . 9
3.3 Text Recognition Tools . . . 10

4 MLKit Evaluation **11**
4.1 MLKit . . . 11
4.2 Metrics . . . 11
4.2.1 Text Detection . . . 12
4.2.2 Text Recognition . . . 13

5 Implementation **15**
5.1 Android Development . . . 15
5.1.1 CameraX by Jetpack . . . 17
5.1.2 Fragments . . . 17
5.1.3 Android User Interface . . . 17
5.1.4 Kotlin . . . 18
5.1.5 Text Recognition API . . . 19

5.2 Library Architecture 20
5.2.1 MVVM 20
5.2.2 Library Scheme 21
5.3 Data filtering 23

6 Results **25**
6.1 Invoice Payment Recognition 26
6.2 Custom Use Case 28
6.3 Performance Evaluation 31
6.3.1 Data Results 32

7 Conclusions and Future work **38**

References **40**

List of Figures

2.1 STEM fields for computer vision 4
2.2 Ground truth annotation for an image in Total-Text 8

4.1 Text Recognition API Overview 12
4.2 Recall and precision summarized 13
4.3 Different match types between bounding boxes 13

5.1 Android stack components 16
5.2 Simple UI example 19
5.3 Text Recognition API output 20
5.4 Android MVVM pattern 21
5.5 Library architecture 22
5.6 Regex matching inconsistency 24

6.1 Demo application MainActivity 25
6.2 Activity Recognition steps 26
6.3 TextRecognitionActivity with default overlay fragment 27
6.4 Custom Use Case activity 31
6.5 Performance Activity 32
6.6 Distribution of Characters and Words in Total-Text Dataset 35
6.7 EDW distribution 36
6.8 EDC distribution 37

List of Tables

2.1 Dataset Benchmarks 6

Listings

4.1 Levenshtein Distance pseudo code [37]. 14
5.1 Simple UI declared in XML 18
6.1 CustomUseCase activity 28
6.2 Text Recognition API call 29
6.3 Get the closest reference hit to label 30

Glossary of Acronyms and Abbreviations

AAR	Android Archive
API	Application Programming Interface
CNN	Convolutional Neural Networks
GT	Ground of Truth
JVM	Java Virtual Machine
KNN	K-Nearest Neighbor
LSTM	Long-short Term Memory
MLKit	Machine Learning Kit
MVVM	Model View ViewModel
OCR	Optical Character Recognition
STEM	Science, Technology, Engineering and Mathematics
SVM	Support Vector Machines
SDK	Software Development Kit
UI	User Interface

Chapter 1

Introduction

In the information era we live in, smartphones have become one of the most used systems worldwide. They enable users to quickly produce and share any information across a network. Consequently, a large amount of information populated the world-wide web, or as we commonly know, the internet. One of the challenges this era face resides in the ability of a computer to interpret physical data. Thus many enterprises want to build bridges between the real and digital world for accessibility. One of these bridges is called **Text Recognition**.

1.1 Context and Motivation

ITSector is an enterprise that builds financial solutions for the information market. Most of their clients prioritize the development of mobile banking solutions. Essential services such as money movement or invoice payment are a must in these applications. However, modern applications prioritize user experience. Inserting information manually from a printed invoice can become time-consuming. With modern technology, we can agile the process of extracting data from the physical world by using the smartphone camera. Therefore, one can point the camera at a financial document, and the application would read and filter the information that needs to continue with the financial operation.

Making mobile applications are always a challenge since we are not always assuming a user's role when implementing features. Providing ways for a user to navigate with ease and feel comfortable with the product can improve satisfaction and eventually increase the company's revenue.

It was initially proposed to implement a simple application capable of using the device's camera to read an invoice payment method. However, the enterprise needs this feature to apply to other use cases. In order to apply the solution to multiple use cases, it requires us to isolate the development of such feature and implement it to be dynamically used to read different data. Although this document is more developer-focused, one individual with little user experience on mobile apps can simply understand the developed components.

1.2 Objectives

This work aims to develop an Android Library capable of enabling developers to quickly implement text recognition features in their applications. Nevertheless, the developed solution must be tested in order to find any bottlenecks that would compromise user satisfaction. Later on 3 we specify the requirements for our product, as well as the technology adopted.

1.3 Methodology

We start by investigating solutions in the text recognition scene. These can be applied to our implementation, so it is better to contextualize the challenges and replicate them with our resources. Furthermore, an ITSector engineer will transmit the specific requirements for the product to-be and the associated restraints we have to overcome.

With all the requirements aligned, we can choose the tools we need to build the solution. Consequently, an architecture sketch can be drawn to represent the items we need to develop and their interactions.

To test the final product, we will use standard metrics with state-of-the-art systems for text recognition. These can transmit the capabilities and constraints of the developed solution. Thus one can understand its limits and applicability when implementing new projects.

1.4 Document Structure

Apart from this one, we present an additional 6 chapters:

- **Chapter 2: Text Recognition**: In this chapter, we present the state-of-the-art solutions for the text detection and recognition problem.
- **Chapter 3: Library Requirements**: The requirements of our library are specified in this chapter.
- **Chapter 4: MLKit Evaluation**: Quick introduction to MLKit, followed by how we will test it.
- **Chapter 5: Implementation**: The final product environment, tools and architecture are described here.
- **Chapter 6: Results**: In the results chapter, we will show the final product in action as well as the testing results.
- **Chapter 7: Conclusion and Future Work**: In this self-explanatory chapter we resume all the document contents and deduce the conclusions.

Chapter 2

Text Recognition

This chapter presents the research done on the fields that embody text recognition technology by the scientific community. First, we will look at the significant area in which it is inserted (Computer vision) followed by its methods and applications.

2.1 Computer vision

As the name suggests, computer vision is the scientific field that deals with how computers perceive and interpret visual content such as images or video. For us humans, it is pretty simple to retrieve information about the environment around us using our vision capability and process that given information in our brain and come up with a label for each object we see. However, for a computer to classify an image or even a frame from video, it must go through a set of processes that incorporate and consider many different research fields, as shown in figure 2.1.

Many STEM fields are present and each one of them contributes in many stages of computer vision, the main ones are the following [15]

- Low-level vision: Image processing for feature extraction. Feature extraction removes redundancy from an input image by dividing it into segments, only retrieving the important big portions to be analysed.

- Intermediate-level vision: Concerned with getting abstract information about an image. A low-level recognition includes image segmentation (partitioning it into different segments/-parts).

- High-level vision: Deals with pattern recognition and is capable of assigning a label to an image, known as **Image Classification**

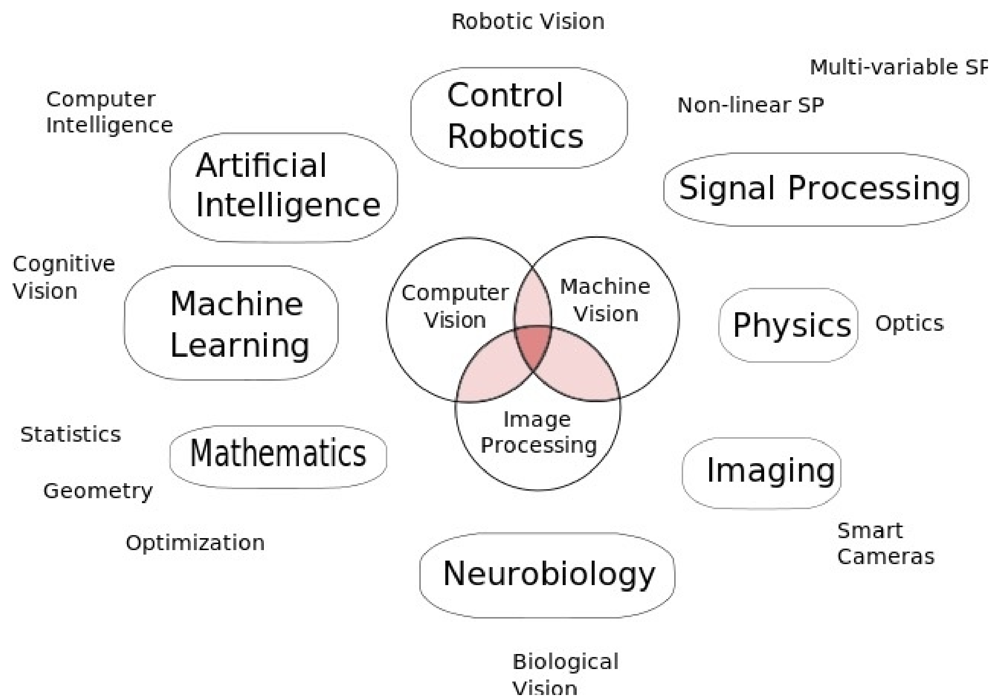


Figure 2.1: STEM fields for computer vision

2.2 Text Detection

The previous 2 low levels are part of the problem that *Text Detection* approaches. Detecting text in an image can influence the following text recognition step. So, it is imperative to detect text instances present in an image accurately.

There are two types of text detection algorithms: Regression-based and instance segmentation based. Furthermore, regression algorithms can also be divided into: proposal-based and part-based [45, 8, 25]. These deep learning techniques take the input image as a chain of pixels that are then passed into the network nodes where processing occurs. The final layer of the network should provide us with pixels that contain text.

2.2.1 Proposal-based

This type of algorithm is characterised by outputting a region (usually a rectangle) where it thinks an instance of an object is. An example is the RCNN (Regional Convolutional Neural Network). Many adaptations of this algorithm have appeared in the research community. The famous Fast R-CNN [48] is part of this group and is widely used since its execution time is pretty low. When it comes to text, some researchers understood the rotational behaviour of text as a challenge and created the R2CNN [20].

2.2.2 Part-based

Part-based algorithms extract regions from images and try to create links between them. These links are what is called a context. CTPN (Connectionist Text Proposal Network) [41] is one of these cases. It uses recurrent neural networks allied with short-term memory networks (LSTM), which are used for linking text based on proximity. CTPN successfully detected vertically disposed words in an image.

2.2.3 Instance Segmentation

Previous algorithms retrieve boxes of detected text. This variation only differs by post-processing the output box. Once the region is retrieved, the algorithm segments the output into words, text lines, or characters. We can achieve this by using Fully Convolutional Neural Networks (FCN). One good example was the text detection algorithm submitted by Yao et al. [47] which outperformed state-of-the-art detection algorithms for multi-oriented and curved text instances.

2.3 Text Recognition

Once the text has been detected and the interest regions have been determined, we can start feeding these cropped instances to a text recognition algorithm. Solutions in the past performed individual character level recognition by using HOG (Histogram Oriented Gradients) descriptors [43]. These count occurrences of the gradient in a character image. The gradient is then used to calculate the difference between a set of characters in a database. The character that shows the least difference is chosen. This solution performed well on documents with no character variations. However, it struggled for scene text recognition on natural imagery.

Most recently, researchers are implementing attention-based encoder-decoder frameworks [9]. These require a large amount of data to train and time to learn. However, it proved to be accurate in recognising scene text imagery. As the name suggests, the framework is separated by an encoder and decoder stage that the image goes through. The regions of interest are gathered by an CNN/LSTM network (basically a part-based detection algorithm) in the encoding phase. Then it passes through an AN (Attention Network), which aligns the output results with the training data. This allows for the algorithm to fit features of the output with the data that we provide. Furthermore, the output string can even pass through a Natural Language Processing module to verify and fix any lexicon errors.

2.4 On-Device Recognition

Deep learning algorithms use a model to store the progress of feature learning. These trained models can be transferred to other devices to recognise without re-training the entire network again. Allowing for development and intensive training with desktop systems, which perform

better than smartphones in typical situations, and the subsequent migration of these to mobile environments. Thus, the previously mentioned solutions can then be used in a mobile device.

2.5 Datasets

To properly evaluate MLKit Text Recognition, we first need to gather some images to feed the algorithm and evaluate the output based on some ground of truth[1]. The popularity of supervised learning tasks, which are heavily data-driven, triggered an immense search for labelled data and solutions that are not entirely dependent on big data [50, 5, 1]. Not only that, but a set of AI competitions worldwide needed some standard on how to compare the contestant's solutions. Thus datasets were created. In these competitions, the dataset is the starting point for competitors, which need to produce a solution that can ultimately get a higher success rate. Chen et al. surveyed the text recognition benchmark datasets [8] as well as the methodologies that contestants found, adapted or created to improve the text recognition on those datasets. Some of these datasets are shown in table 2.1. Typically a dataset is divided into two parts: the train set and test set. The train set is used to, as the name suggests, to train deep learning approaches. In our case, we use the train set and the ground of truth it provides to evaluate MLKit's implementation.

Table 2.1: Dataset Benchmarks

Datasets	Language	Images			Type
		Total	Train	Test	
Synth90k	English	~9000000	-	-	Regular
IC13	English	561	420	141	Regular
COCO-Text	English	63686	43686	10000	Irregular
Total-Text	English	1555	1255	300	Irregular

There are two types of recognition datasets, regular and irregular. Regular datasets contain frontal and horizontal text images. Irregular datasets contain curved, multi oriented and perspectively distorted text, which represent a challenge for recognition among researchers [10, 46, 39]. First, we show a couple of datasets that proved crucial in improving the recognition success rate over the years. Then we give a more detailed description of the adopted dataset for MLKit evaluation 2.5.4.

2.5.1 Synth90k

Synth90k [19] is a synthetic dataset with over 9 million 32x100 images. Each image contains a word out of a 90k dictionary. The generated images contain mostly front and horizontal text with

[1]Ground of truth is information that is known to be accurate or trustworthy, provided by direct observation and measurement

little to no distortion. Instead of the typical approach of character recognition, this dataset focus on providing enough data to perform word-based recognition. Since deep learning approaches require a large amount of data to be accurate, Synth90k came to fill the void of the standard realistic/small datasets.

2.5.2 IC13

The IC13 is a dataset submitted by the ICDAR2013 robust reading competition [22]. This dataset contains images selected from a variety of web and email pages, with a total of 561 images, 100x100 resolution each and 5003 whole words. It has pixel-level ground of truth, i.e. each character was manually "painted" with pixels [13], making it the ideal dataset for a segmentation-based solution.

2.5.3 COCO-Text

The COCO-Text is a dataset containing over 63k scene images, and it is based on MS-COCO [42]. These images were gathered without having text in mind. Thus, almost 50% of the dataset images contain no text, the other part being mainly irregular text. Furthermore, the ground of truth of these images embed not only the label text and word bounding boxes (that are common in text datasets), but also the category of text (handwritten or machine printed), a binary classifier for legible and illegible text and the type of script language (English, French, etc). It was one of the benchmark datasets for multi-oriented text recognition.

2.5.4 Total-Text

Total-Text is another scene text recognition focused dataset [12]. It was created to fill the need for total curved text scene images that were missing in other benchmark datasets such as ICDARs and COCO-Text. Researchers found solutions on how to properly detect oriented text, one of these being the famous Fast Region-based Convolutional Network [16, 48, 20] which prove to be faster and more accurate than previous R-CNN, SPPnet detection algorithms. Consequently, many Fast-RCNN variations appeared for proper text detection, even for curved instances [49]. Total-text and other curved, multi-oriented datasets establish a challenge for researchers to detect curved text and properly recognize it.

The dataset contains 1555 total images, of which 1255 correspond to the annotated train set that we will use to evaluate MLKit implementation. Each image can contain one or more instances of text with the proper ground of truth, as shown in figure 2.2. In this dataset, we have the text instances annotated with:

- x and y coordinates for the 10 polygon vertices (standard across all instances).
- Text label.

```
x:[214 250 287 324 362 349 320 290 261 231]
y:[325 298 291 297 320 347 332 324 331 346]
label: ASRAMA
orientation: c

x:[ 53  69  87 103 120 121 104  87  72  56]
y:[446 445 443 443 443 456 459 458 458 458]
label: PERUNDING
orientation: h

x:[ 53  76 100 125 148 149 126 100  76  51]
y:[463 459 457 457 458 480 482 483 481 483]
label: HENRY
orientation: m
```

...

Figure 2.2: Ground truth annotation for an image in Total-Text

- Orientation type: "c" for curved, "h" for horizontal and "m" for multi-oriented text.

Compared to the other datasets, this one enables evaluation in three different orientation domains, i.e., we can test the algorithm and get different results for either horizontal, multi-oriented or curve text. Furthermore, we can analyze the results and pinpoint any features or bottlenecks (if the case) based on the data segmentation.

Chapter 3

Library Requirements

In this chapter, we specify the requirements for the developed library. These were transmitted by an ITSector representative along the 3 months of planned development. We first show the core functionalities that our library must provide, followed by some quality requirements. Finally, we align the requirements with tools on the market that we can use to implement recognition.

3.1 Functional Requirements

According to Ruth, Malan et al. [27], functional requirements "capture the intended behavior of the system. This behavior may be expressed as services, tasks or functions the system is required to perform". Hence we can list the following features:

- **FRQ1**: The library must have a global Android component capable of doing recognition when passed an image.
- **FRQ2**: Adapt the first component to filter information.
- **FRQ3**: Develop an Android component that uses the camera hardware functionality to read imagery in real-time.
- **FRQ4**: Create an entry point for user interface customization for the camera view.

3.2 Quality Requirements

Quality or non-functional requirements are a subset of requirements that deal with usability, maintainability, reliability, portability, and efficiency of the final product. These can be somewhat ambiguous or difficult to detect and significantly impact the final product's success rate [6, 32]. From the banking perspective, we can name a few quality requirements regarding our library:

- **QRQ1**: The library must be secure
- **QRQ2**: Implementation scalability over new camera functionalities

- **QRQ3**: Appealing UI/UX

When we talk about security and privacy in software, we are worried about sensitive data that might be leaked to third parties, and from the banking standpoint, that cannot happen. So, how can we minimize or even prevent that leak from happening? One way to look at it is to keep the computation on-device, i.e., the mobile phone does the fundamental computation and recognition offline. This way, we can mitigate man-in-the-middle attacks utilizing the more advanced hardware specifications out of the mobile device.

In terms of scalability, our library must provide a simple way of feeding an image to an algorithm. Our implementation will use the image for text recognition, but we must build the ground block for other algorithms such as face recognition or even a barcode reader.

Finally, our solution must be enjoyable for the user. Since different applications can have different use cases for the camera, we provide an entry point for the developer to create a custom user interface. Therefore we implement the core functionality of the screen with the camera view, a default user interface, and we teach the developer how to add a customizable new UI. As for user experience, expect the product to be fast, easy to use and non-blocking. We go further on details later on chapter 5.

3.3 Text Recognition Tools

With all the requirements aligned, we can choose technologies that best fit our goals. These can be cloud or on-device solutions. Since cloud services require an image to be transmitted over a network, it compromises **QRQ1**. All that is left is finding a technology that works offline. ABBY Fine Reader and Tesseract are popular OCR (Optical Character Recognition) engines designed to read and identify data on printed documents [40, 35]. However, ABBY is a paid service and will not be used for this solution. Furthermore, Tesseract, which Google owns, proved to be challenging when implementing for Android devices. Not only that, it requires a lot of image pre-processing and training to be accurate.

The Android developer community is focused on using a complete solution for detecting and recognizing text. It is called **MLKit**[1] and Google developed it to respond to the demand of vision features in Android and IOS applications. Since it is targeted to the Android environment, it is undoubtedly the chosen technology. In the following chapter, we overview the technology on their official website.

[1] MLKit official website: https://developers.google.com/ml-kit

Chapter 4

MLKit Evaluation

4.1 MLKit

MLKit is a mobile SDK developed by Google that encapsulates the on-device machine learning expertise in a more friendly, developer-oriented way. It provides many solutions, namely face detection, pose estimation, object detection, and text recognition for Android and IOS devices. This technology allows developers to quickly and effectively implement machine learning features in their applications. In this case, developers can use the Text Recognition API to recognize text in any Latin-based character set[1]. Under the hood, this API stays a black box to the community, i.e., there is no academic work published by the Google team on the adopted implementation strategy.

Below is the overview of the API (Application Programming Interface) on the MLKit website 4.1. An API is a service that the programmer can use to request and receive a response. In this case, the *TextRecognitionAPI* provided by MLKit, will take an image as input and output the recognized text. As shown, the library supposedly has great execution time and small application size; hence one can implement recognition without worrying too much about the final application size. Upon installing the application on the device, all the necessary components of the API are downloaded via Google Play Services, an android application that handles many background tasks. Although we do not get much *a priori* information from the website, we take a deeper look at how we communicate with the API in chapter 5.

4.2 Metrics

Metrics are the essential part of evaluating MLKit's implementation. It is the way to extract meaning from the result data. Thus we can analyze results and make conclusions. The metrics used in previous studies [25, 8, 22, 48, 30, 21, 29] are divided into two parts. One set of metrics focuses on evaluating text detection solutions, while the other evaluates the output string from

[1] Source: https://developers.google.com/ml-kit/vision/text-recognition

Text Recognition API	
Description	Recognize Latin-script text in images or videos.
Library name	`com.google.android.gms:play-services-mlkit-text-recognition`
Implementation	Library is dynamically downloaded via Google Play Services.
App size impact	260KB
Initialization time	Might have to wait for library to download before first use.
Performance	Real-time on most devices.

Figure 4.1: Text Recognition API Overview. Source: https://developers.google.com/ml-kit/vision/text-recognition/android

the recognition. The outputs are evaluated based on the ground of truth annotation provided by datasets.

4.2.1 Text Detection

To evaluate text detection/localization, researchers compare the bounding box from the ground of truth with the bounding box of the algorithm output with two simple measures:

$$Recall(Gi,Di) = \frac{Area(Gi \cap Di)}{Area(Gi)}, \in [0,1]$$

$$Precision(Gi,Di) = \frac{Area(Gi \cap Di)}{Area(Di)}, \in [0,1]$$

$$Gi = \text{Ground of truth object list rectangles}$$

$$Di = \text{Detected object list rectangles}$$

According to Wolf et al. [44], "recall illustrates the proportion of the ground truth rectangle which has been correctly detected, and precision decreases if the amount of additional incorrectly detected area increases."

Furthermore, these two metrics are combined to give us a generic final score:

$$f - score = 2 \times \frac{Precision \times Recall}{Precision + Recall}, \in [0,1]$$

Although we are not evaluating the text detection model from MLKit, we will possibly encounter some inconsistency between the GT and MLKit detection bounding boxes. These are summarized in figure 4.3. As a solution, we will standardize all these mismatches into one match

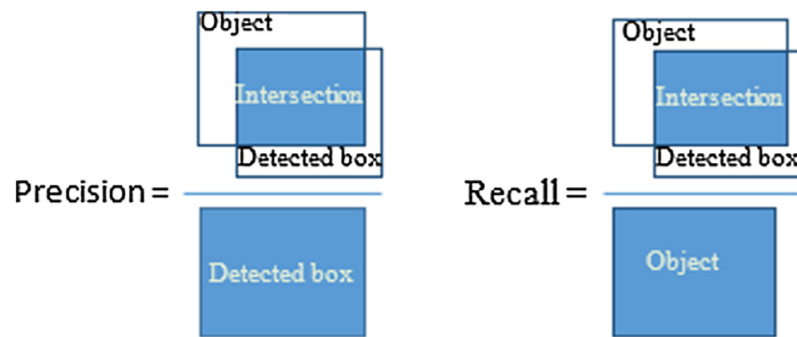


Figure 4.2: Recall and precision summarized [23]

by using the position of the MLKit's bounding box and Total-Text polygon to check if there are multiple objects of either type inside one another. From there, we merge or split based on the output of the geometric function.

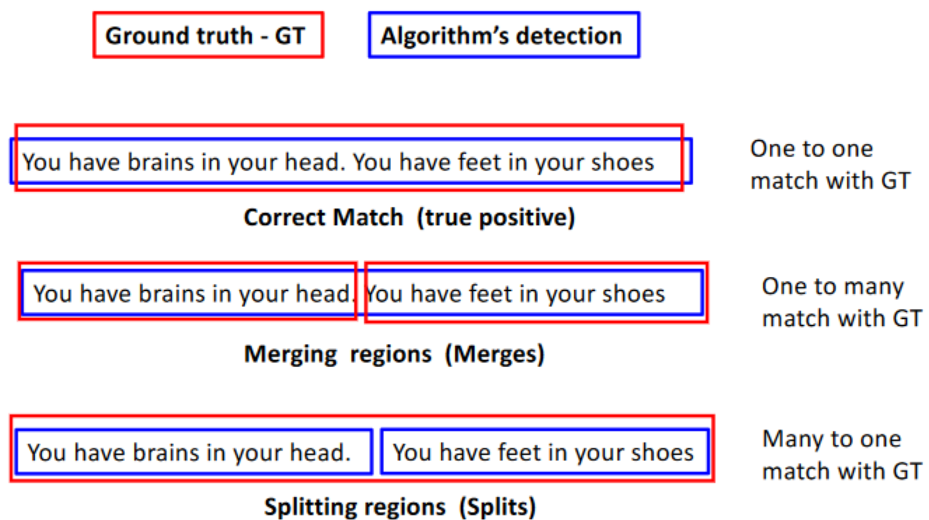


Figure 4.3: Different match types between bounding boxes [17]

4.2.2 Text Recognition

The recognition phase gives us an output string of the algorithm's content for a given bounding box. Researchers use a traditional formula to compare the GT and output string to evaluate this step. In the ICDAR competition [22, 21] the standard metric is the edit distance, also called Levenshtein Distance [17]. This comparison can be at character and word level, for example:

Character edit distance:

$$ED_c(cat, cop) = 2$$

For the above two strings to be equal, we will need to replace two characters. This formula also counts character insertions and deletions with an equal cost for the three operations. In listing 4.1 we have the pseudo code on how to calculate this distance, which we applied in 5.

```
int LevenshteinDistance(char s[1...m], char t[1...n])
 //d is a table with m+1 rows and n+1 columns
 declare int d[0...m,0...n]
 for i from 0 to m
  d[i,0] := i
 for j from 0 to n
  d[0,j] := j
 for i from 1 to m
  for j from 1 to n
   if s[i] = t[j] then cost := 0
   else cost := 1
   d[I,j] := minimum(
    d[i-1,j] +1, //deletion
    d[i,j-1] +1, //insertion
    d[i-1,j-1] +cost //substitution
 return d[m,n]
```

Listing 4.1: Levenshtein Distance pseudo code [37].

Word edit distance:

$$ED_w(cat, cop) = 1$$

$$ED_w(job, job) = 0$$

The word edit distance is a binary measure (only evaluates to 0 or 1) per instance and is used to evaluate how many complete words the algorithm fails. If the two strings are equal, it measures 0, otherwise 1.

The previous measures represent the number of errors per instance when evaluating with Total-Text. Thus, to get a global view of the results in all instances of the dataset, we have to normalize these metrics:

$$NormED_c = \frac{totalED_c}{numCharacters} \in [0, 1]$$

$$NormED_w = \frac{totalED_w}{numWords} \in [0, 1]$$

As previously stated in 2.5.4, the different orientation text instances in the dataset enable a separated evaluation; hence we calculate three normalized edit distances each for a different orientation.

Chapter 5

Implementation

The android library implementation is specified in this chapter. We will start by looking at the planning stage to have a higher picture of the environment, architecture and tools adopted. An android contextualization is necessary to justify the choices made for structure and technology. That said, we will give an insight into android development.

5.1 Android Development

Android is an operating system available in a worldwide variety of mobile devices. Based on Unix type systems, it provides an environment for application development. Development in android is made possible thanks to a software development kit (SDK) provided by Google itself [14, 24]. Thanks to this SDK, we can use many of the stack components 5.1 available through the operating system. In our case, the camera is an essential part of our project. Therefore the application must communicate with the Linux kernel to provide the requested feature. Although we mention applications as the higher stack level for android components, we must not forget that an android library is also at the higher level of the stack. It provides code/feature reusability across multiple applications and does not differ too much from a basic application.

Android apps are developed using four basic components [36]:

- Activity
- Service
- Content Provider
- Broadcast Receiver

Activities represent a single screen where we define the logic and interface for that given instance. Application's architecture is mainly formed by interconnected activities i.e, one activity can launch another to form a back stack (the back button will pop the current activity returning to

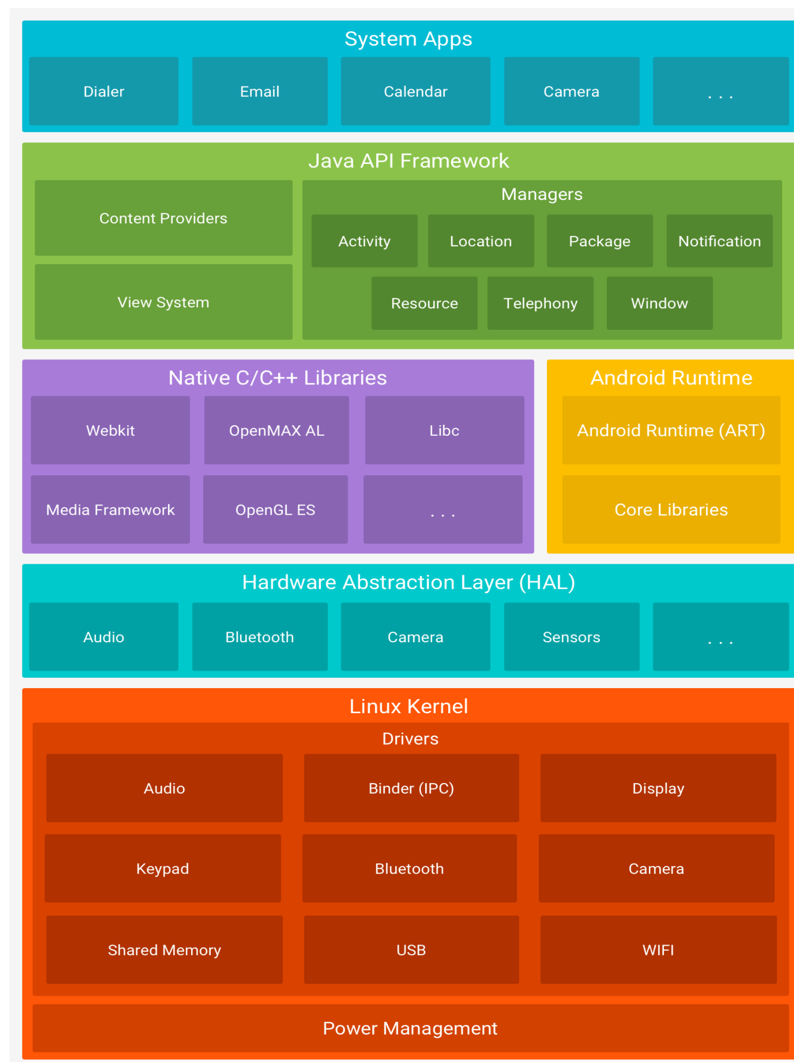


Figure 5.1: Android stack components

the one who started it). Activities are the only application component we will be using to build our library; hence one can launch one activity to recognize text in an image.

Services are background processes without a user interface. They are ideal for long term processes like fetching streaming data. Although they lack UI, they can communicate with activities to notify events through notifications and toast messages (alerts that appear in the current active activity).

Content providers allow data sharing between different applications. One can implement a content provider to enable other apps to add, remove or query data. This data sharing is made possible due to the Universal Resource Identifier (URI) [1], which is defined by the content provider we want to access.

[1]"A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource" [4]

Finally, a broadcast receiver is a mechanism that allows applications to handle broadcasts. The broadcast receiver is registered by an application that specifies which type of data it must receive. If the broadcast message matches the data type of the registered receiver, it will be invoked regardless of whether the application is running or not.

In older years of android development, a content provider could be used to launch the system camera app, capturing the image we need for analysis outside of our application. Using content providers was the standard approach until a set of libraries and tools provided by Jetpack[2] were developed. This innovation brought camera features to be developed totally inside one's application.

5.1.1 CameraX by Jetpack

As already mentioned above, CameraX is a support library that makes implementing camera views on android easier, compatible across multiple devices and with less boilerplate code [18]. However, one significant functionality of this library stands out, the image analysis functionality. This feature enables the application to take the information directly from the camera buffer and feed that same information onto an image processing algorithm like MLKit. Thus we improve the overall user experience by taking away the responsibility of capturing an image. Instead, we charge the developer to filter information from the set of overtime images.

5.1.2 Fragments

Another important aspect of our library focuses on the use of fragments. Fragments are modular pieces with their user interface and logic that are embedded within an activity [36]. It was developed to further satisfy the necessity of reusing components across one or multiple applications or activities. These fragments are nothing more than pieces that can be glued (or destroyed) to form a responsive activity. In this case, we use the capability of fragments to build a custom overlay on top of our camera view. Therefore developers can customize the user interface of the pre-built recognition activity as we will see in 5.2.

5.1.3 Android User Interface

A user interface comprises all the visual aspects and interaction endpoints available in a system. Therefore, a key part of android application development relies on designing and creating these interfaces. As previous stated, activities and fragments hold this visual capability. Interfaces are declared in the format of an XML [3] file. The XML file that holds their visual information is inflated when activities or fragments are created. The listing 5.1 shows a simple example of a UI declaration. The figure 5.2 reflects the XML declaration after being inflated onto an activity.

[2]Jetpack website: https://developer.android.com/jetpack

[3]XML (Extensible Markup Language) is a textual markup language disposed in a tree structure that holds various types of data [34]

The top-level tag *LinearLayout* is a *ViewGroup*. While the nested ones (*TextView* and *Button*) are referred as *Views* [36].

Views (also known as widgets or components) are the basic unit of an interface. Everything that is displayed on the screen is built by sub-classing the Android View class. The SDK provides some pre-built components like the *Button* or the *TextView* that we saw in 5.2.

ViewGroups are containers that manage the disposition of its children views in the interface. The most common one is the *LinearLayout* that organizes the child views in single rows or columns based on the declared orientation (vertical or horizontal). The view group we will be using to dispose items in the camera screen is the *FrameLayout*. A *FrameLayout* allow for developers to allocate views in some regions of the screen based on declared gravity's (top, bottom, center, right and left), which is all we want for a simple and effective user interface.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3                android:layout_width="match_parent"
4                android:layout_height="match_parent"
5                android:orientation="vertical" >
6      <TextView android:id="@+id/text"
7                android:layout_width="wrap_content"
8                android:layout_height="wrap_content"
9                android:text="Hello, I am a TextView" />
10     <Button android:id="@+id/button"
11             android:layout_width="wrap_content"
12             android:layout_height="wrap_content"
13             android:text="Hello, I am a Button" />
14 </LinearLayout>
```

Listing 5.1: Simple UI declared in XML

5.1.4 Kotlin

Kotlin is a statically-typed programming language that runs on the java virtual machine (JVM). Developed by the Google team back in 2011, Kotlin's main features allow developers to write less, more readable and error prune code. Before Kotlin appeared in the scene, android applications used Java. Thus was the necessity for a modern language that could embrace object-oriented programming, functional programming, and other features of modern languages. As of 2017, Google announced Kotlin as their official programming language for android development. Due to the interoperability with Java, Kotlin can be introduced progressively alongside the existing Java blocks of an application, avoiding the cost of a brute force migration. It also provides null safety, less verbosity, lambdas and higher-order functions [28, 36, 2, 7, 31]. Even though Java and Kotlin can be used simultaneously, it is best practice to build new android features mainly on Kotlin. Hence the current implementation of the recognition library is built using Kotlin solely.

Figure 5.2: Simple UI example

5.1.5 Text Recognition API

The *TextRecognition* API is called passing an image as input. The *CameraX* image buffer provides us with an instance of an *Image* object that contains all the information transmitted by the camera's device. Afterwards, we feed the image instance to the API, returning an object of type *Text*. As shown by figure 5.3, the *Text* object is divided into hierarchies. Thus one *Block* contains one or more *Line* elements and one *Line* element is composed by one or more *Element* objects. These hold the same type of information. For example, in a *Block* object, we have the following properties:

- **Text**: The recognized text in the whole block in string format.
- **Frame**: Characterized by a *Rect* object, it holds 4 values that compose the limits of the rectangle (left, top, right, bottom).

- **Corner Points**: More detailed information regarding the vertices of the detected text box. Instead of 4 values per box, it holds the complete 4 coordinates of the rectangle.
- **Recognized Language Code**: Code of the detected language, for English it would output "en".
- **Lines**: Child elements of a *Block*. In the case of a *Line* object, this property would contain *Elem* objects.

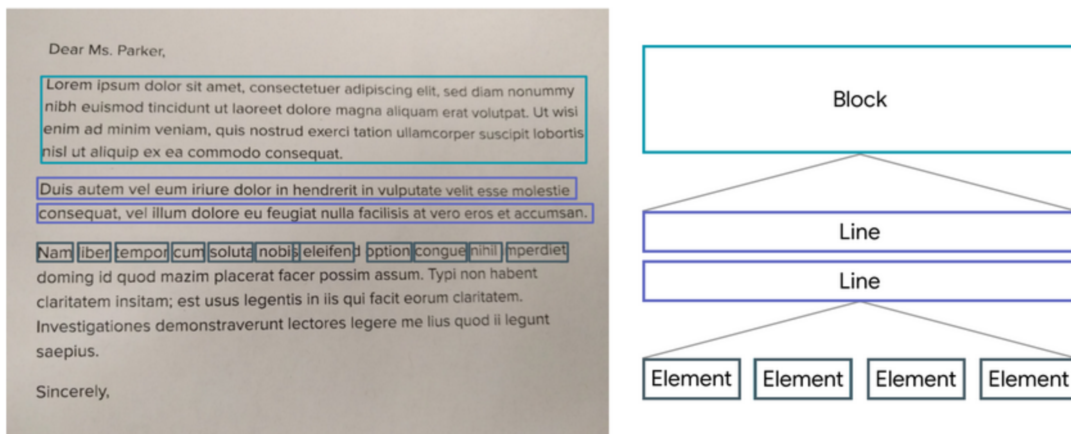


Figure 5.3: Text Recognition API output. Source: https://developers.google.com/ml-kit/vision/text-recognition

5.2 Library Architecture

In this section, we discuss library architecture. First, we will look at a prevalent design pattern for android applications called MVVM. Then we establish a global structure for our library, pinpointing each element's responsibility. Finally, we discuss the data filtering approach adopted.

5.2.1 MVVM

Model view view model or simply MVVM is a design pattern applicable to android development. As the name suggests, it comprises three elements: model, view and view model. The model holds the data and can be stored in a database or created at runtime. The view model is the model of a given view, i.e. manages its state and can call background processes like retrieving data from a website or database. The view manages the user interface and can issue commands to the view model that require some computation effort. This design pattern helps enterprise development in many ways but, most importantly, makes it easier to test software due to the separation of concerns. Furthermore, it enables code reusability (the same view model can be used across many views) [26, 38].

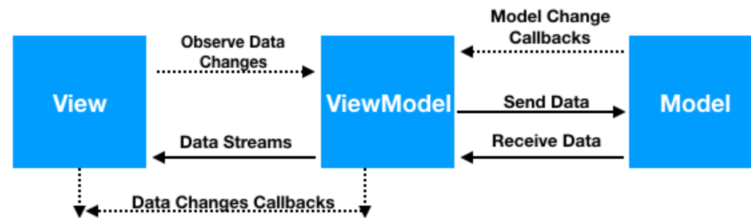


Figure 5.4: Android MVVM pattern

As illustrated by figure 5.4, the view model works as the glue between the presentation and the data that exists on the application. This pattern is the most critical element since it holds all the logic to retrieve data from the model and link it to the view.

5.2.2 Library Scheme

With all the requirements aligned and chosen our design pattern, we present the adopted architecture for the library 5.5.

- **CameraActivity**: Activity that handles camera permissions, launches the CameraX image analysis buffer and is responsible for inflating the user interface. This top-level activity is created for many camera use cases, not only text recognition, avoiding repeated code.
- **activity_camera.xml**: The activity XML file that describes the interface for camera use cases. It is composed by a top level *FrameLayout* that contains two childs:
  - *PreviewView*: View that is used by the CameraX library to inflate the camera view.
  - *FragmentContainerView*: Fragment container that is used for different UI use cases.
- **TextRecognitionActivity**: Child activity of CameraActivity. It manages the overlay fragment and is the entry point for the text recognition operation.
- **TextRecognitionViewModel**: The view model will handle the background operation for the actual text recognition. It calls the function for translating an image into text from the TextRecognitionLogic object and communicates the translated text to the activity.
- **TextRecognitionLogic**: Singleton Kotlin object that stores all the recognition logic. Thus enabling accessibility from any point in the application. It is responsible for initializing the recognizer instance from MLKit and handling recognition requests through function calls. The most generic function takes an image, calls the MLKit Text Recognition API, and returns an instance of a class called *Text* which is a segmentation of the text in the image organized by blocks, lines or elements. Each block, line or element have an associated

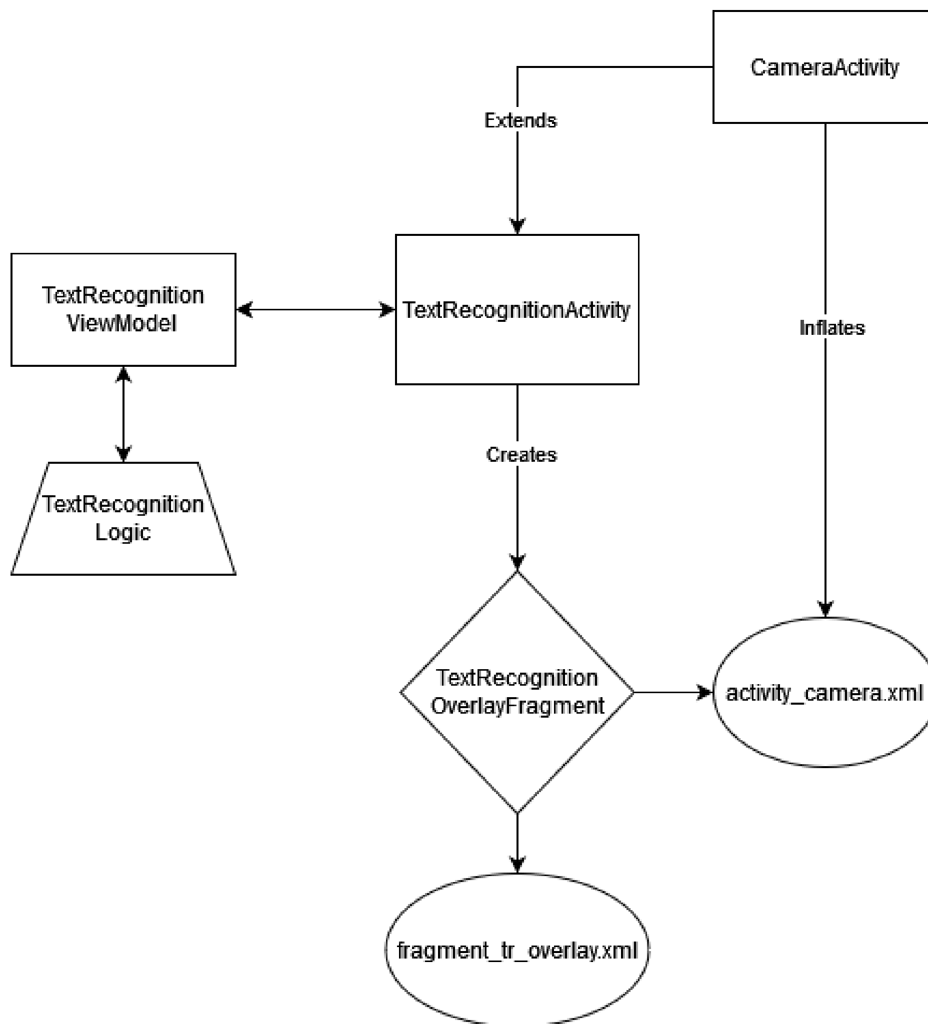


Figure 5.5: Library architecture

rectangle formed by four coordinates, which the text detection model computes. Hence one can use the coordinates to filter information based on proximity and geometric logic.

- **TextRecognitionOverlayFragment**: This fragment is responsible for UI customization on top of the camera preview. It has it's own interface (declared in *fragment_fr_overlay.xml*) and can be inflated on the **activity_camera.xml** through the *FragmentContainerView*. The fragment interface has a top level *FrameLayout* with two components:
  - *top_level_action_bar.xml*: The action bar sits on top of the layout and has it's own separate declaration file. Contains two buttons:
    - *Close Button*: Closes the current activity.
    - *Flash Button*: Ideal for low light conditions. If a device does not have a lantern, this button is not enabled.

- *ImageView*: The image view is an image located at the center of the screen. It is used purely for visual aesthetics. Since the image preview occupies the whole screen, it is nice to have the center of the device as a reference for pointing to data.

5.3 Data filtering

A critical aspect of the implementation focuses on filtering the information present in an image. The MLKit algorithm takes an image and outputs all the detected text in the form of a string, a sequence of characters. Our approach is to take the output string and extract information through regular expressions (RegEx) [4].

The *TextRecognitionActivity* allows developers to filter information using regex. Before launching this activity, we can define two arguments for this purpose:

- **HashMap<String,String>**: An Hashmap is a <key, value> data structure. It is used to tell the algorithm which information we want to search. For example, in the invoice payment method, we need an HashMap with three entries:
  - <"reference",`"\\d{3}\\s*\\d{3}\\s*\\d{3}"`>
  - <"entity",`"\\d{5}"`>
  - <"value", `"\\d+,\\d+"`>

  The first entry filters the reference field. Its regex translates to 3 blocks of 3 digits (`\\d{3}`) separated by zero or more blank spaces (`\\s*`). Entity is defined by 5 sequenced digits (`\\d{5}`). Finally, the value of the invoice is characterized by having one or more digits (`\\d+`) on the right and left side of the comma `","`.

- **(Optional) Array<String>**: An array is a data structure used to store multiple values of one type. Here we define our regex strings for labels that we want to detect in a single frame before filtering with the HashMap. Using the invoice recognition example, we can define 3 regex strings that detect the reference, entity and value labels. Once these placeholders are detected, we proceed to filter the HashMap entries. If this array argument is not included, it goes straight to HashMap filtering.

Although this is the easiest way to deploy text recognition in an application, it has one major drawback. Let us assume that our Regex matches multiple entity values as illustrated by 5.6. In the face of this situation, the way the activity is implemented will retrieve the last detected occurrence of the regex pattern. Hence, in chapter 6 we analyze a detailed custom implementation of recognition with the relative position in mind.

[4] Regex is a form of identifying characters or a chain of characters in text through pattern matching [33]

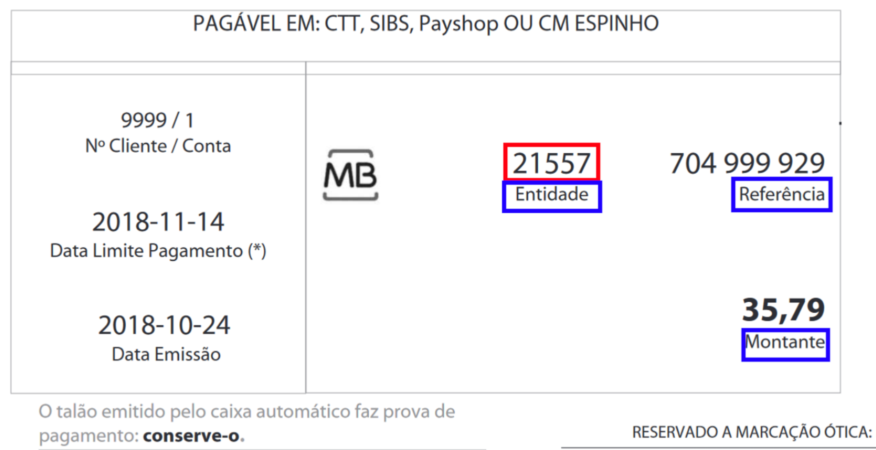


Figure 5.6: Regex matching inconsistency

Chapter 6

Results

In order to test our implementation, we exported the library to an AAR (Android Archive) [11] which encapsulates and compresses all the developed components into a single file. Furthermore, we create a demo application and imported the aar library file onto our new project. The figure 6.1 shows the main activity once we launch the demo application, which contains three buttons:

Figure 6.1: Demo application MainActivity

- **MB Text Recognition**: Launches the invoice payment recognition use case.

- **Custom Use Case**: The custom use case button launches an activity that demonstrates how to use camera features, UI customization and customized data filtering for countless use cases.
- **Performance Eval**: Runs an activity that begins the task of evaluating the MLKit's algorithm on Total-Text dataset 2.5.4.

6.1 Invoice Payment Recognition

When clicking the *MB Text Recognition* button, we are presented with a new activity that contains three fields. Our goal is to fill these fields with the correct data through the recognition library. In order to do that that we launch the default *TextRecognitionActivity* of our library. As explained in 5.3 previously, this is the easiest way of implementing recognition.

After launching the *TextRecognitionActivity*, a background process calls the *Text Recognition API* and our regex filters its output. Furthermore, the activity returns the filtered data so we can use it to fill the blank spaces in our *ScanDocumentActivity* as shown by 6.2. It takes up to 1 to 2 seconds on average to recognise the data in the document.

Figure 6.2: Activity Recognition steps. Left and Right: *ScanDocumentActivity*, in the middle: *TextRecognitionActivity* with default overlay fragment.

The overlay fragment that lives inside the *TextRecognitionActivity* is composed by 3 components as shown by 6.3. This UI provides the user with a close button to terminate the current activity. Furthermore, the flash button enables recognition in low light environments but is only visible if

the device has an embedded lantern. The image view is purely used for aesthetics and gives a central reference of the device when pointing to data.

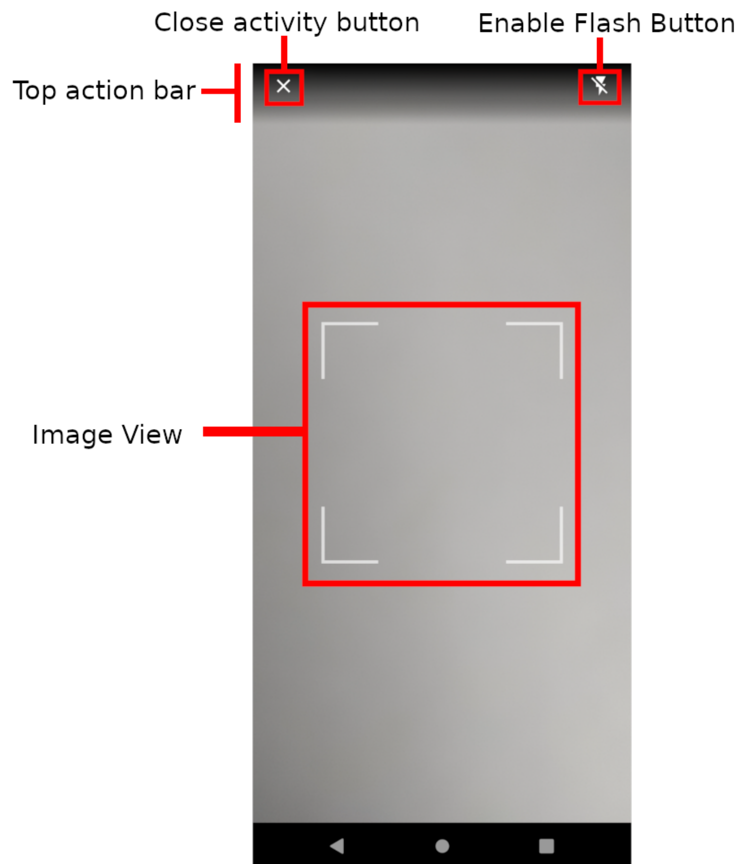


Figure 6.3: TextRecognitionActivity with default overlay fragment

The only effort the developer has, in this case, is to specify the arguments that define the data to return, i.e, the **HashMap** and the optional **Array** described in 5.3. The algorithm that filters the content based on regex pattern matching has a temporal complexity of $\mathscr{O}(B)$, with B being the number of output blocks of MLKit. This implementation makes the response time as fast as we can. However, we can encounter some inconsistency if multiple regex matches occur inside the same block. In this case, the regex returns only the first encountered match for the same block. If multiple values are detected across multiple blocks, it is returned the first occurrence of the last block detected. Nevertheless, it is a good and fast solution for parsing documents with little to no noise. However, our library is an entry point for customisation and a vehicle for implementing multiple use cases. These, of course, are more time consuming and require a little more effort. However, it is best to isolate each case and take full advantage of the support functions we built.

6.2 Custom Use Case

In order to demonstrate the customisation that our library provides, we decided to construct a new activity. This new activity extends the *CameraActivity* of our library. Hence, it reduces the boilerplate code and time to initialise the camera and its view. Listing 6.1 shows the initialisation of our custom use case activity.

```
class CustomUseCase : CameraActivity() {

    /**
     * Custom View Model
     */
    private val mViewModel: CustomViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        lateinit var overlay: CustomOverlayFragment

        super.setupCamera({ imageProxy ->
            //You can set whatever function for image analysis
            mViewModel.analyzeImage(imageProxy)
            imageProxy.close()
        },{ cameraInstance ->
             //OPTIONAL: Use the Camera instance for more features like enabling flash.

            /**
             * Here you can use whatever fragment you want for UI/UX inside the
             * camera activity, just make sure to call [setupOverlay] so the fragment
             * can bind to the container
             */
            overlay = CustomOverlayFragment.newInstance()
            super.setupOverlay(overlay)
            overlay.initFlashFeature(cameraInstance)
        })
    }
}
```

Listing 6.1: CustomUseCase activity

As we can see, implementing an activity with camera features has become much more accessible and concise. Our parent activity (*CameraActivity*) handles the layout and camera initialisation. In the *setupCamera* method, we have a callback that allows passing an instance of *imageProxy*, which is an image object, to any function we would like for analysis. Furthermore, we can use a *cameraInstance* to enable many features of our camera through *CameraX*. Here we declare a new overlay fragment, passing it to the parent activity to be inflated in the container.

The analysis function is declared in our *CustomViewModel*. It starts by wrapping all the computation inside a **coroutine**, which is a lightweight thread that can run in the background without affecting the UI. We decided to implement a more robust solution for the invoice payment method

in this function. To do that we first call the text recognition function that returns the *Text* object as described in 6.2. Then we loop through the blocks and lines of our *Text* variable giving us a time complexity of $\mathcal{O}(L*B)$, with L and B being the number of lines and blocks, respectively. We do not parse on *Element* level due to matching inconsistencies. Taking the reference string "001 002 003" as an example, we would have 3 different *Elements* representing 3 characters each. Therefore our regex matching would not work. Not only that, the algorithm would have a different time complexity and, therefore, a longer execution time.

```
/* Call recognition function and filter results by line */
TextRecognitionLogic.getImageText(image).also { text ->
    //Loop through block and lines
    text.textBlocks.forEach { block ->
        block.lines.forEach { line ->

        /**
        * 1. Check for labels in frame
        */
            refLabelRegex.find(line.text, 0)?.let {
                labelRef = line
            }

            entLabelRegex.find(line.text, 0)?.let {
                labelEnt = line
            }

            valLabelRegex.find(line.text, 0)?.let {
                labelVal = line
            }

        /*
        * 2. Check hits for actual regex values
        */
            refRegex.find(line.text, 0)?.let {
                referenceLineHits.add(line)
                referenceRegexHits.add(it.value)
            }

            entRegex.find(line.text, 0)?.let {
                entityLineHits.add(line)
                entityRegexHits.add(it.value)
            }

            valueRegex.find(line.text, 0)?.let {
                valueLineHits.add(line)
                valueRegexHits.add(it.value)
            }
        }
    }

}
```

Listing 6.2: Text Recognition API call

Inside the loops, we have two major blocks of code, adequately identified with numbers 1 and 2 in 6.2. The first one checks for the label occurrences like "Entidade" or "Referência" and saves the line where each label is located. Furthermore, we register any hits for regex matching for the three fields in the next block. In each field, we will save the line and the value of the regex matching. With all the data gathered, we can start filtering results.

Listing 6.3 shows how we filter the many reference hits. Here we start by checking if the label has been successfully detected by performing a null check on variable *labelRef*. Then we verify if we have occurrences of reference values on our *referenceLineHits*. If so, we immediately call an auxiliary function inside our library called *getClosestLine* which will take the line where the reference label was detected and the lines where we detected possible reference values. The function will return the closest line to the label. Hence one can get the value of the closest reference. Although we are only showing the filtering of the reference value, the other fields hold the same logic.

```
/* Filter reference hits to return the closest one to the label */
labelRef?.let { refLabel ->
    if (referenceLineHits.size > 0) {
        val closestHit = GeometricUtils.getClosestLine(
                refLabel.boundingBox,
                referenceLineHits)
        closestHit?.let {
            val index = referenceLineHits.indexOf(closestHit)
            reference.postValue(referenceRegexHits[index])
        }
    }
}
```

Listing 6.3: Get the closest reference hit to label

With all the background logic done, the activity is notified and shows the correct values for our fields, even if the image has noise. Let us take figure 5.6 and do a scan, this time with our customised solution. Figure 6.4 illustrates the result once we point the mobile device to the invoice. Although it detects the 5 digit number "99929" as a possible entity value, it is discarded since it is far from the label.

Although this solution is more time consuming and more complex to implement than the first one presented here, it reflects perfectly the support our library provides. The extension capability of our *CameraActivity* allied with the initialisation of the *TextRecognition* API through the *TextRecognitionLogic* and the custom user interface entry as a fragment allows for developers to implement less and concise code. Furthermore, we encourage implementing each use case separately, with particular attention to the relative position of data across many document layouts.

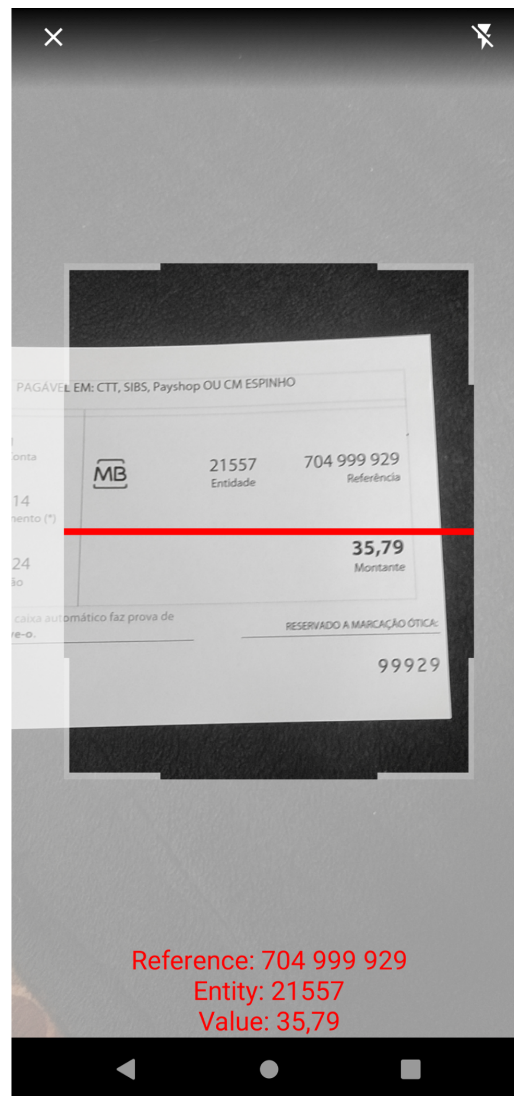
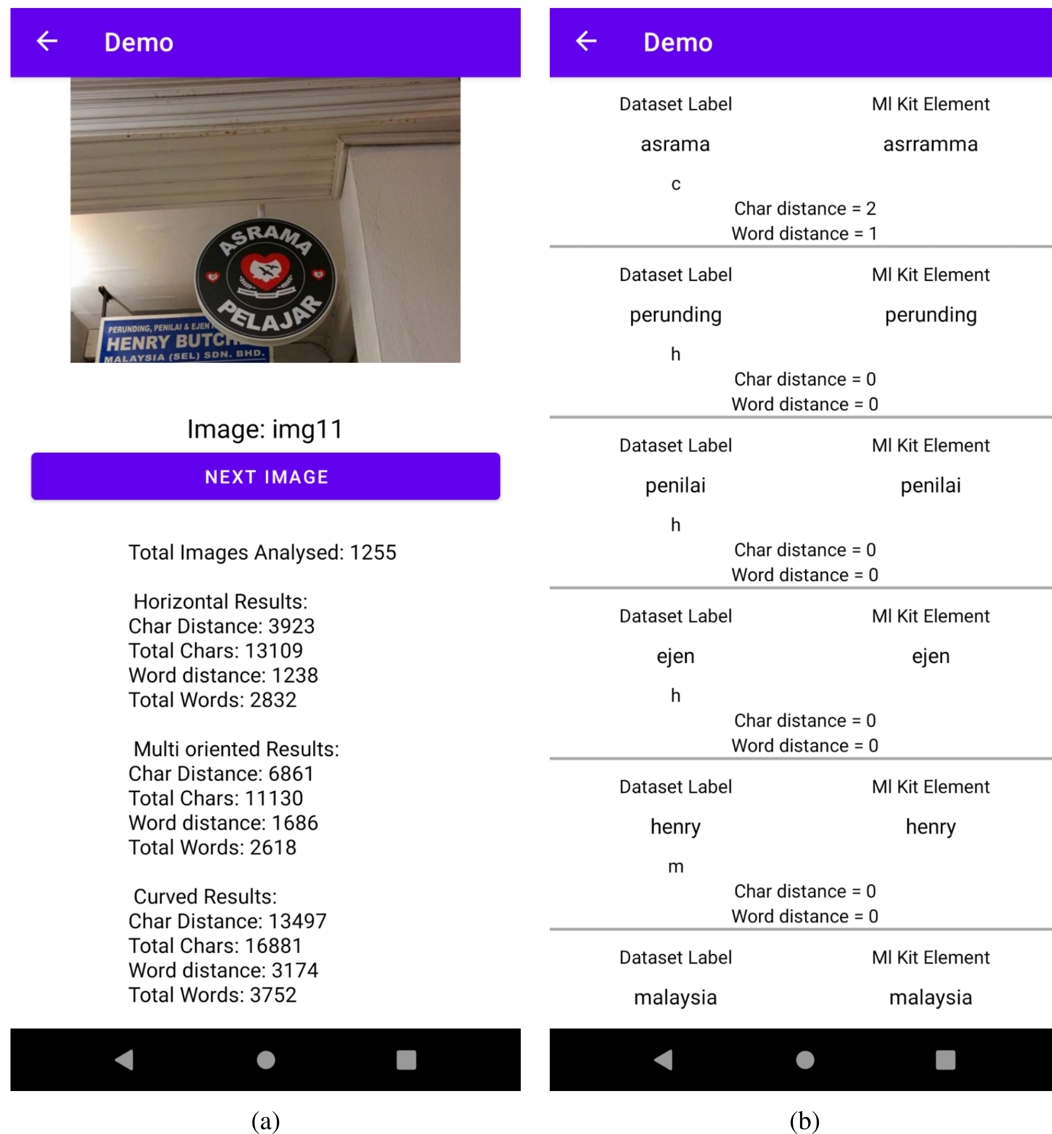


Figure 6.4: Custom Use Case activity with custom overlay fragment

6.3 Performance Evaluation

This section will present the results of MLKit's API on the Total-Text dataset. Once we click on the *Performance Eval* button, it starts a background process that transverses all images in the dataset, parsing different match types between the GT and the algorithm detection 4.2.1. After standardising the GT and MLKit output, we calculate the recognition metrics 4.2.2. Figure 6.5 shows the activity once all the computation is done. Here we can navigate through all images using the next and previous buttons. In each image, the global variables displayed will not change. However, the list we see in 6.5b will show, for the selected image, all GT instances alongside the parsed *Elements* from MLKit.



(a) (b)

Figure 6.5: (a) On top the image view, followed by the image name and a button used for navigation between image instances. Furthermore at the bottom, we print some global variables from computation results. (b) Scrollable list with GT words compared to MLKit *Elements*

.

6.3.1 Data Results

From the 1255 images, we were able to analyse 9202 words and 41120 characters. The resulting distribution among the different classes of GT instances (Horizontal, Multi-Oriented and Curved) is represented by 6.6. Although Total-Text is a curved focus scene detection dataset, it has a good distribution for other types of text instances. As we can see, the character and word distribution do not deviate much from each other. Meaning we have a good character composition across word instances.

6.3.1.1 Word Recognition

In the word dimension of our data, we verified that out of 9202 complete words, MLKit failed to identify:

- 1238 out of 2832 Horizontal words
- 1686 out of 2618 Multi-Oriented words.
- 3174 out of 3752 Curved words.

Furthermore, we can calculate the normalised edit distance:

$$GlobalEDW = \frac{1238 + 1686 + 3174}{9202} \approx 66.27\%$$

$$HorizontalEDW = \frac{1238}{2832} \approx 43.71\%$$

$$MultiOrientedEDW = \frac{1686}{2618} \approx 64.40\%$$

$$CurvedEDW = \frac{3174}{3752} \approx 84.59\%$$

We can observe inferior results for curved and multi-oriented instances compared with horizontal ones from the normalised distance. Thus, it leads to a high global normalised word distance of about 66%. However, we can analyse each class separately. Figure 6.7 puts the word edit-distance in perspective. There is a good result for horizontal recognition since the error rate is approximately 44%. Note that the word distance is calculated comparing GT and MLKit strings. If they are different just by one character, the distance increases. Therefore, it reflects the accuracy of MLKit on recognising whole words in a scene detection context. However, to better analyse the pros and cons of MLKit, we need to go down a level and evaluate the character recognition.

6.3.1.2 Character Recognition

The character distance is calculated using the Levenshtein formula 4.1. After computation, our algorithm prints the distance to characters:

- 3923 out of 13109, for Horizontal characters.
- 6861 out of 11130, for Multi-Oriented characters.
- 13497 out of 16881, for Curve characters

Now we can calculate the normalised character edit distance:

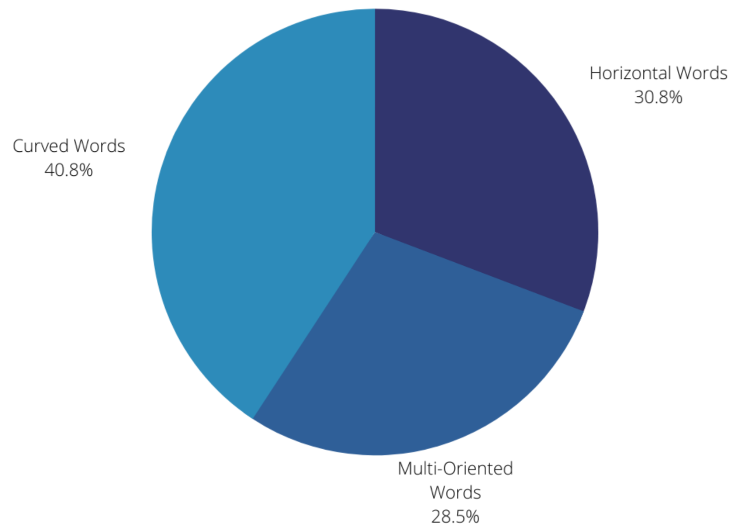
$$GlobalEDC = \frac{3923 + 6861 + 13497}{41120} \approx 59.05\%$$

$$HorizontalEDC = \frac{3923}{13109} \approx 29.93\%$$

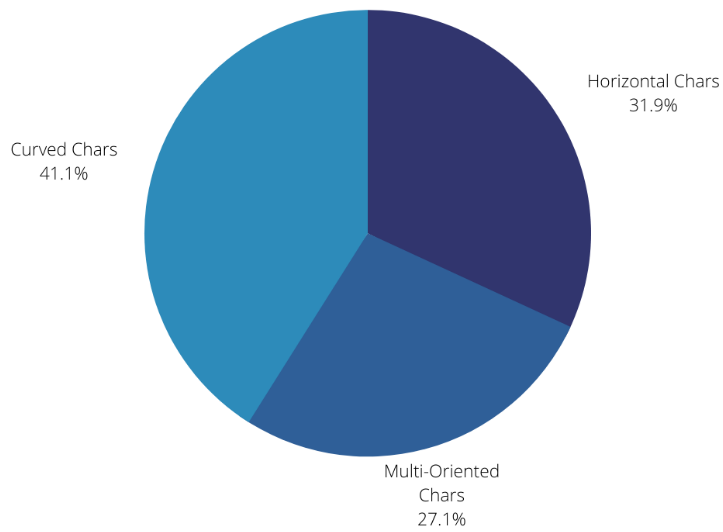
$$MultiOrientedEDC = \frac{6861}{11130} \approx 61.64\%$$

$$CurvedEDC = \frac{13497}{16881} \approx 79.95\%$$

The distance for characters shows, as expected, slightly better results across all the classes. Putting it in perspective 6.8 we can visualise a decreased error rate for horizontal characters compared to full recognised horizontal words. While multi-oriented and curved instances have a difference between EDW and EDC of 2.76% and 4.64%, respectively, the horizontal class has an impressive 13.78%, which means that it recognised more characters than other classes for incorrect words.



(a)



(b)

Figure 6.6: (a) Word distribution (b) Character distribution

.

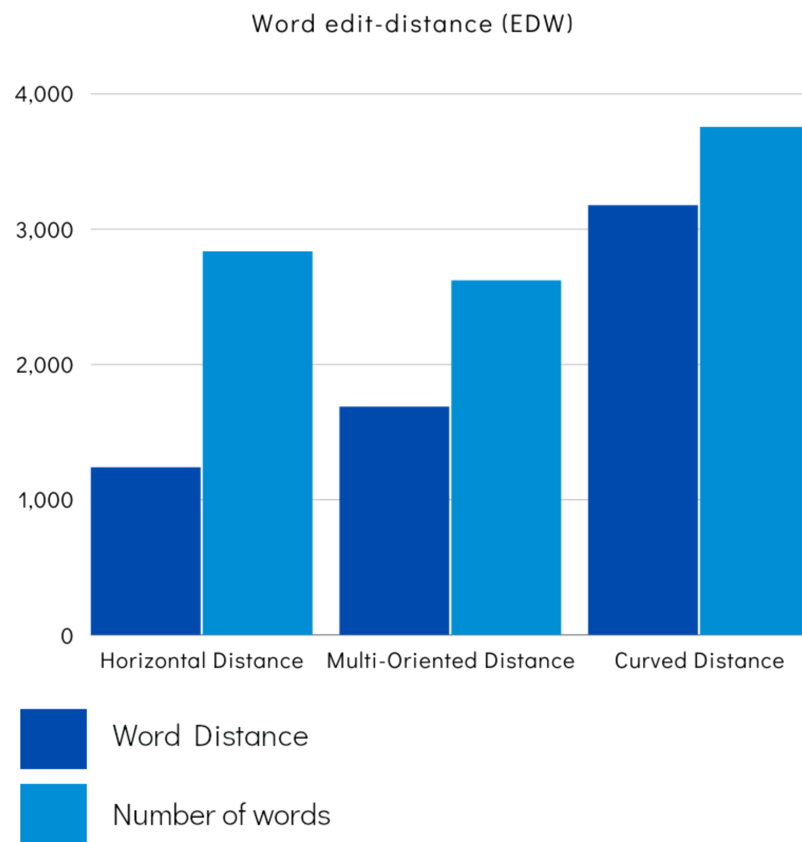


Figure 6.7: EDW distribution

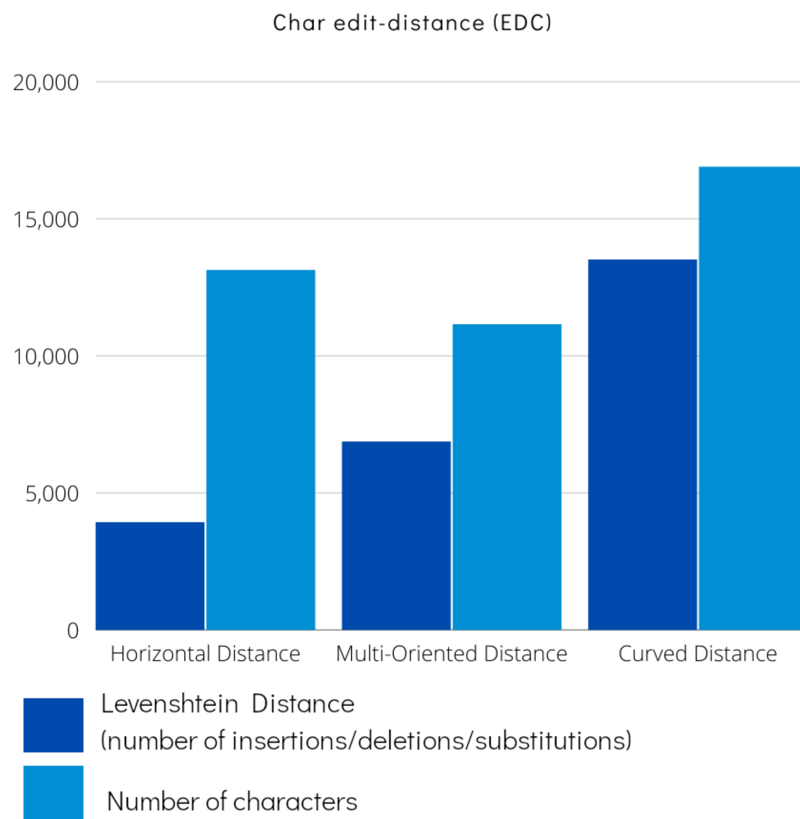


Figure 6.8: EDC distribution

Chapter 7

Conclusions and Future work

Text Recognition is a computer vision field that focuses on text extraction present in the imagery. Over the years, researchers have found ways of building systems capable of accurately detecting and recognising text. Detection and recognition can be treated as separate problems. However, they share the same final objective and depend on one another. Robust reading competitions such as ICDAR [22, 21, 29] saw competitors breaking state-of-the-art solutions with their submissions. Although these technologies target desktop environments, they can be migrated to mobile devices for real-time execution. However, we went with a native Android solution to implement on-device recognition, MLKit. By choosing this technology, it saved much time implementing the proposed requirements for the library 3.

MLKit is an excellent SDK for machine learning solution implementation. We built a library capable of recognising invoice payment methods, easing the developer's work when implementing applications requiring camera functionalities. Furthermore, we tested the accuracy of MLKit *TextRecognition* API with a challenging scene text dataset: Total-Text. The benchmark recognition framework for this dataset, submitted by Youngming et al [3] reports an impressive 78.7% of accuracy. Compared to our 33.73% of hit-rate for complete words, MLKit stays behind the state-of-the-art solutions. However, MLKit is not adapted for distorted or curved text instances. It proved efficient for frontal and horizontal text in natural images with a surprising 56.29% of hit rate for word instances.

Finally, we met all the requirements that ITSector proposed for this project. In future iterations of our library, we will continue to provide support functions and implement other camera use cases such as a barcode reader. Hopefully, Google will also improve their text recognition accuracy without sacrificing the excellent execution time, and together, we can continue to make applications enjoyable for users.

Even though this on-device accuracy is not as precise as other systems reported by the state-of-the-art, MLKit's results demonstrate other possible advantages, such as the ability to identify

an image within an average of 2 seconds correctly. Compared to other systems, it consumes less computational resources, taking less time to process information, which could also improve user experience and be a better fit for its commercial purposes and simple document scans.

References

[1] Eric Arazo, Diego Ortego, Paul Albert, Noel E O'Connor, and Kevin McGuinness. Pseudo-labeling and confirmation bias in deep semi-supervised learning. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[2] Luca Ardito, Riccardo Coppola, Giovanni Malnati, and Marco Torchiano. Effectiveness of kotlin vs. java in android app development tasks. *Information and Software Technology*, 127:106374, 2020.

[3] Youngmin Baek, Seung Shin, Jeonghun Baek, Sungrae Park, Junyeop Lee, Daehyun Nam, and Hwalsuk Lee. Character region attention for text spotting. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 504–521, Cham, 2020. Springer International Publishing.

[4] Tim Berners-Lee, Roy Fielding, Larry Masinter, et al. Uniform resource identifiers (uri): Generic syntax, 1998.

[5] David Berthelot, Nicholas Carlini, Ian J. Goodfellow, Nicolas Papernot, Avital Oliver, and Colin Raffel. Mixmatch: A holistic approach to semi-supervised learning. *CoRR*, abs/1905.02249, 2019.

[6] Jorgen Boegh. A new standard for quality requirements. *IEEE software*, 25(2):57, 2008.

[7] Subham Bose, Madhuleena Mukherjee, Aditi Kundu, and Madhurima Banerjee. A comparative study: java vs kotlin programming in android application development. *International Journal of Advanced Research in Computer Science*, 9(3):41, 2018.

[8] Xiaoxue Chen, Lianwen Jin, Yuanzhi Zhu, Canjie Luo, and Tianwei Wang. Text recognition in the wild: A survey. *ACM Computing Surveys (CSUR)*, 54(2):1–35, 2021.

[9] Zhanzhan Cheng, Fan Bai, Yunlu Xu, Gang Zheng, Shiliang Pu, and Shuigeng Zhou. Focusing attention: Towards accurate text recognition in natural images. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[10] Zhanzhan Cheng, Yangliu Xu, Fan Bai, Yi Niu, Shiliang Pu, and Shuigeng Zhou. Aon: Towards arbitrarily-oriented text recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5571–5579, 2018.

[11] Jeongwoo Choi, Yongmin Kim, Jinwoo Lee, and Jiman Hong. Dynamic code whitelist for efficient analysis of android code. In *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems*, pages 165–166, 2018.

[12] Chee Kheng Ch'ng, Chee Seng Chan, and Chenglin Liu. Total-text: Towards orientation robustness in scene text detection. *International Journal on Document Analysis and Recognition (IJDAR)*, 23:31–52, 2020.

[13] Antonio Clavelli, Dimosthenis Karatzas, and Josep Llados. A framework for the assessment of text extraction algorithms on complex colour images. In *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*, pages 19–26, 2010.

[14] Android Developers. What is Android? *Dosegljivo: http://www. academia. edu/download/30551848/andoid–tech. pdf*, 2011.

[15] Davies E.R. *Computer vision. Principles, algorithms, applications, learning*. Elsevier, Academic Press, Fifth edition, 2018.

[16] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[17] Afzal Godil, Afzal Godil, Patrick Grother, and Mei Ngan. *The Text Recognition Algorithm Independent Evaluation (TRAIT)*. US Department of Commerce, National Institute of Standards and Technology, 2017.

[18] Google. Camerax overview. `https://developer.android.com/training/camerax`. Accessed: 2021-10-30.

[19] Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Reading text in the wild with convolutional neural networks. *International Journal of Computer Vision*, 116(1):1–20, jan 2016.

[20] Yingying Jiang, Xiangyu Zhu, Xiaobing Wang, Shuli Yang, Wei Li, Hua Wang, Pei Fu, and Zhenbo Luo. R2CNN: rotational region CNN for orientation robust scene text detection. *arXiv preprint arXiv:1706.09579*, 2017.

[21] Dimosthenis Karatzas, Lluis Gomez-Bigorda, Anguelos Nicolaou, Suman Ghosh, Andrew Bagdanov, Masakazu Iwamura, Jiri Matas, Lukas Neumann, Vijay Ramaseshan Chandrasekhar, Shijian Lu, et al. ICDAR 2015 competition on robust reading. In *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1156–1160. IEEE, 2015.

[22] Dimosthenis Karatzas, Faisal Shafait, Seiichi Uchida, Masakazu Iwamura, Lluis Gomez i Bigorda, Sergi Robles Mestre, Joan Mas, David Fernandez Mota, Jon Almazan Almazan, and Lluis Pere De Las Heras. ICDAR 2013 robust reading competition. In *2013 12th International Conference on Document Analysis and Recognition*, pages 1484–1493. IEEE, 2013.

[23] Anand Koirala, Kerry B. Walsh, Zhenglin Wang, and Cheryl McCarthy. Deep learning – method overview and review of use for fruit detection and yield estimation. *Computers and Electronics in Agriculture*, 162:219–234, 2019.

[24] Jianye Liu and Jiankun Yu. Research on development of Android applications. In *2011 4th International Conference on Intelligent Networks and Intelligent Systems*, pages 69–72, 2011.

[25] Xiyan Liu, Gaofeng Meng, and Chunhong Pan. Scene text detection and recognition with advances in deep learning: a survey. *International Journal on Document Analysis and Recognition (IJDAR)*, 22(2):143–162, 2019.

[26] Tian Lou et al. A comparison of Android Native App Architecture MVC, MVP and MVVM. *Eindhoven University of Technology*, 2016.

[27] Ruth Malan, Dana Bredemeyer, et al. Functional requirements and use cases. *Bredemeyer Consulting*, 2001.

[28] Bruno Gois Mateus and Matias Martinez. An empirical study on quality of Android applications written in Kotlin language. *Empirical Software Engineering*, 24(6):3356–3393, 2019.

[29] Nibal Nayef, Fei Yin, Imen Bizid, Hyunsoo Choi, Yuan Feng, Dimosthenis Karatzas, Zhenbo Luo, Umapada Pal, Christophe Rigaud, Joseph Chazalon, et al. ICDAR2017 robust reading challenge on multi-lingual scene text detection and script identification-rrc-mlt. In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 1, pages 1454–1459. IEEE, 2017.

[30] Lukáš Neumann and Jiří Matas. Real-time scene text localization and recognition. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3538–3545. IEEE, 2012.

[31] Victor Oliveira, Leopoldo Teixeira, and Felipe Ebert. On the adoption of Kotlin on Android development: A triangulation study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 206–216. IEEE, 2020.

[32] Mohd Hafeez Osman and Mohd Firdaus Zaharin. Ambiguous software requirement specification detection: An automated approach. In *Proceedings of the 5th International Workshop on Requirements Engineering and Testing*, RET '18, page 33–40, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Davide Pasetto, Fabrizio Petrini, and Virat Agarwal. Tools for very fast regular expression matching. *Computer*, 43(3):50–58, 2010.

[34] Albrecht R Schmidt, Florian Waas, Martin L Kersten, Daniela Florescu, Ioana Manolescu, Michael J Carey, Ralph Busse, et al. The xml benchmark project. 2001.

[35] Ray Smith. An Overview of the Tesseract OCR Engine. In *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*, pages 629–633, Washington, DC, USA, 2007. IEEE Computer Society.

[36] Neil Smyth. *Android Studio 4.2 Development Essentials-Kotlin Edition: Developing Android Apps Using Android Studio 4.2, Kotlin and Android Jetpack*. Payload Media, 2021.

[37] Zhan Su, Byung-Ryul Ahn, Ki-Yol Eom, Min-Koo Kang, Jin-Pyung Kim, and Moon-Kyun Kim. Plagiarism detection using the Levenshtein distance and Smith-Waterman algorithm. In *2008 3rd International Conference on Innovative Computing Information and Control*, pages 569–569. IEEE, 2008.

[38] Wei Sun, Haohui Chen, and Wen Yu. The exploration and practice of MVVM pattern on Android platform. In *2016 4th International Conference on Machinery, Materials and Information Technology Applications*. Atlantis Press, 2017.

[39] Yipeng Sun, Chengquan Zhang, Zuming Huang, Jiaming Liu, Junyu Han, and Errui Ding. Textnet: Irregular text reading from images with an end-to-end trainable network. In *Asian Conference on Computer Vision*, pages 83–99. Springer, 2018.

[40] Ahmad P Tafti, Ahmadreza Baghaie, Mehdi Assefi, Hamid R Arabnia, Zeyun Yu, and Peggy Peissig. OCR as a service: an experimental evaluation of Google Docs OCR, Tesseract, ABBYY FineReader, and Transym. In *International Symposium on Visual Computing*, pages 735–746. Springer, 2016.

[41] Zhi Tian, Weilin Huang, Tong He, Pan He, and Yu Qiao. Detecting text in natural image with connectionist text proposal network. *CoRR*, abs/1609.03605, 2016.

[42] Andreas Veit, Tomas Matera, Lukas Neumann, Jiri Matas, and Serge Belongie. Coco-text: Dataset and benchmark for text detection and recognition in natural images. In *arXiv preprint arXiv:1601.07140*, 2016.

[43] Kai Wang, Boris Babenko, and Serge Belongie. End-to-end scene text recognition. In *2011 International conference on computer vision*, pages 1457–1464. IEEE, 2011.

[44] Christian Wolf and Jean-Michel Jolion. Object count/area graphs for the evaluation of object detection and segmentation algorithms. *International Journal of Document Analysis and Recognition (IJDAR)*, 8(4):280–296, 2006.

[45] Yongchao Xu, Yukang Wang, Wei Zhou, Yongpan Wang, Zhibo Yang, and Xiang Bai. Textfield: Learning a deep direction field for irregular scene text detection. *IEEE Transactions on Image Processing*, 28(11):5566–5579, 2019.

[46] Xiao Yang, Dafang He, Zihan Zhou, Daniel Kifer, and C Lee Giles. Learning to read irregular text with attention mechanisms. In *IJCAI*, volume 1, page 3, 2017.

[47] Cong Yao, Xiang Bai, Nong Sang, Xinyu Zhou, Shuchang Zhou, and Zhimin Cao. Scene text detection via holistic, multi-channel prediction. *arXiv preprint arXiv:1606.09002*, 2016.

[48] Zhuoyao Zhong, Lei Sun, and Qiang Huo. An anchor-free region proposal network for faster r-cnn-based text detection approaches. *International Journal on Document Analysis and Recognition (IJDAR)*, 22(3):315–327, 2019.

[49] Zhao Zhou, Hao Ye, Luhui Chen, and Yingbin Zheng. Detecting curve text with local segmentation network and curve connection. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 3374–3379. IEEE, 2020.

[50] Zhi-Hua Zhou. A brief introduction to weakly supervised learning. *National science review*, 5(1):44–53, 2018.