JOHANNES KEPLER
UNIVERSITY LINZ

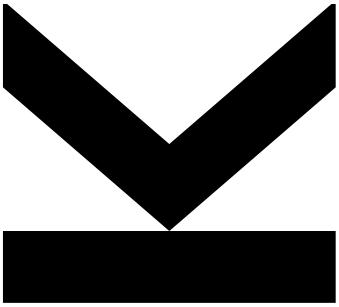
Submitted by
**Fabian Ahorner BSc.**

Submitted at
**Institute for Computational Perception**

Supervisor
**a.Univ.-Prof., Dr. Josef Scharinger**

Linz, December 2017

# Automatic detection of UIC train wagon numbers



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

**JOHANNES KEPLER**
**UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

# Abstract

The organization and identification of train wagons still heavily relies on humans reading their spray painted UIC numbers. A solution that can read these numbers automatically can be used in a variety of situations, such as damage identification and tracking. Although systems have been created for this task in the private sector, no public research or data is available. The first contribution of this thesis is a new UIC number dataset that was released to the public. The second contribution is a new system that is capable of locating and reading UIC numbers from low quality images. For this purpose, a modular framework, supporting an Extremal Regions and a FASText character region detector was implemented. Furthermore, the FASText method was extended for the use of color images. For grouping regions into lines, a new $\mathcal{O}(n^2 \log n)$ algorithm was designed for the UIC domain. The new method offers a significant performance increase over the established $\mathcal{O}(n^8)$ algorithm designed by Neumann & Matas. For decoding the lines into text, TesseractOCR and a Support Vector Machine are provided. Finally, an independent component was designed to group lines into UIC numbers, allowing a user to use the remaining pipeline for a different text detection system.

# Kurzfassung

Für die Organisation und Identifikation von Zugwaggons sind UIC-Nummern unerlässlich. Die zum Teil aufgesprühten Ziffern müssen vom Personal mühsam von jedem Wagon abgelesen werden. Ein System, welches die Nummern automatisch liest, kann in einer Vielzahl von Anwendungen eingesetzt werden. Beispiele wären die automatisierte Verfolgung von Wagons über lange Strecken und die automatische Zuordnung von erkannten Schäden. Obwohl es dafür schon Systeme auf dem kommerziellen Markt gibt, wurde noch keine wissenschaftliche Arbeit zu diesem Thema veröffentlicht. Der erste Beitrag dieser Masterarbeit ist ein neuer UIC-Nummern Datensatz, welcher damit für zukünftige Projekte öffentlich zur Verfügung gestellt wird. Der zweite Beitrag ist ein System, das zur automatischen Lokalisierung und Identifikation von UIC-Nummern bei Bildern mit niedriger Qualität eingesetzt werden kann. Zu diesem Zweck, wurde ein Framework mit einem Extremal Regions und FASText Zeichen Detector implementiert. Auẞerdem wurde der FASText Detector für Farbbilder erweitert. Zur Gruppierung von Zeichen in Zeilen wurde ein neuer $\mathcal{O}(n^2 \log n)$ Algorithmus entworfen. Dieser reduziert die Laufzeit des gesamten Systems - im Vergleich zu dem von Neumann & Matas entworfenen $\mathcal{O}(n^8)$ Algorithmus - dramatisch. Zur Zuweisung von Ziffern zu extrahierten Bildregionen wurden TesseractOCR und eine Support Vector Maschine verwendet. Abschlieẞend wurde eine unabhängige Komponente zur Gruppierung von Zeilen in UIC-Nummern entworfen. Dies erlaubt eine rasche Wiederverwendung des restlichen Systems für ein anderes Bilderkennungssystems.

# Contents

**Abstract** **i**

**Kurzfassung** **iii**

**1 Introduction** **1**
1.1 UIC Numbers . . . 1
1.1.1 Check digit . . . 2
1.1.2 Dataset . . . 3
1.2 Related Work . . . 6
1.2.1 Automatic number plate recognition (ANPR) . . . 6
1.2.2 ICDAR . . . 7
1.2.3 Text detection in complex fields . . . 9

**2 Theory** **11**
2.1 Character Detector . . . 12
2.1.1 Stroke Width Transform . . . 12
2.1.2 Maximally Stable Extremal Regions . . . 13
2.1.3 Extremal Regions . . . 13
2.1.4 FASText . . . 16
2.2 Line Detector . . . 18
2.2.1 NM Line detector . . . 18
2.2.2 Hough line transform . . . 20
2.3 Optical Character Recognition . . . 20

**3 System** **21**
3.1 Setup . . . 21
3.1.1 Installation of TesseractOCR . . . 21
3.1.2 Installation of OpenCV . . . 22
3.2 Pipeline . . . 22
3.3 Evaluation methods . . . 25
3.4 Character Detectors . . . 27
3.4.1 Extremal Region Detector . . . 27
3.4.2 FASText Detector . . . 33
3.4.3 Scale . . . 40
3.4.4 Comparison . . . 41
3.5 Line Detector . . . 43
3.5.1 Dealing with spray painted digits . . . 43
3.5.2 Neumann and Matas . . . 44
3.5.3 Custom line detector . . . 46
3.5.4 Comparison . . . 54
3.6 Optical Character Recognition . . . 55
3.6.1 TesseractOCR . . . 56

3.6.2 Support Vector Machine . . . 58
3.6.3 TesseractOCR + Support Vector Machine (TS) . . . 59
3.7 UIC Detector . . . 60
3.7.1 Grouping . . . 60
3.7.2 Scoring . . . 61
3.7.3 OCR . . . 62
3.7.4 Boosting valid UICs . . . 63
3.8 Further optimizations . . . 63
3.8.1 White list . . . 63
3.8.2 Two stage pipeline . . . 63

**4 Results** **65**
4.1 Character detector . . . 65
4.2 Optical character recognition . . . 67
4.3 Two stage pipeline . . . 68
4.4 Comparison . . . 70

**5 User interfaces** **73**
5.1 UIC Annotation tool . . . 73
5.2 UIC detector CLI . . . 74
5.2.1 Single image mode . . . 75
5.2.2 Dataset mode . . . 75

**6 Conclusion** **79**
6.1 Further work . . . 80

**Bibliography** **81**

**List of figures** **83**

**List of tables** **85**

**Curriculum Vitae** **87**

**Statutory Declaration** **88**

# Chapter 1

# Introduction

UIC wagon numbers are an international industry agreement for the identification of freight trains wagons. The numbering system is described in a leaflet by the international union of railways (short UIC for Union internationale des chemins de fer) [urlc]. The identification numbers are used to identify coaches over different countries and railway companies. It consists of 12 digits and is read manually by the train operators. An automatic system could allow faster tracking and handling of coaches. This thesis is testing the feasibility of a system that extracts these 12 digits from images of train wagons. The pictures used to extract the numbers were recorded with hand held devices and were mixed quality. Additionally, the condition of the mostly spray painted identification added a further challenge to the task.
An optical character recognition system was designed to provide a solution to this problem. Standard OCR systems often expect simple black and white images with the background clearly distinguishable from the characters. As a result these systems often depend on heavy preprocessing and are mainly used to detect text from scanned sheets of paper. The field of text detection in complex scenes focuses on more challenging conditions and also deals with the localization of text in images of everyday scenes. Because of the complexity of the input images of the UIC system and the unknown location of the identification number, techniques from the field of text detection in complex scenes were chosen. Section 1.1 provides more detail about UIC numbers and the used dataset. Section 1.2 talks about related work. Chapter 2 provides additional information about the problem domain and explains algorithms and concepts that were used from existing research to build a UIC detector. Chapter 3 talks about the specific implementation of the previously introduced concepts. Furthermore, chapter 3 discusses their extension and the introduction of new algorithms to create a UIC detector. Chapter 4 goes through the results achieved with said UIC detector and chapter 5 describes the final program and the tool created for the dataset annotation process. Finally, chapter 6 summarizes this thesis and provides suggestions of further work in this area.

## 1.1 UIC Numbers

Figure 1.1 shows a typical UIC number. It consists of 12 digits where the first 2 are on the first line, the next two are on the second line and the remaining eight are on the third line. Additionally, the first two lines have text next to them that provides a readable format of the first two lines. The first two digits describe the vehicle type and provide information about the interoperability of coaches. The second part is the country code. Next to it, the country and the owner of the wagon can be found in text form. This was only the country code till 2006 but was changed afterwards [urlc]. This resulted in some UIC numbers being painted over resulting in a different background color as


Figure 1.1: Structure of UIC numbers

seen in figure 1.3i. The next four numbers represent some additional type information about the wagon. Digit 9-11 are a serial number that can be used to uniquely identify each coach. Finally the last is a self check digit and can be used to validate the number. This is especially useful because the coaches can get damaged or spray painted over with graffiti and it allows the validation of the number. The algorithm used to calculate the check digit is described in section 1.1.1.

There are several properties of UIC numbers that can be helpful for their detection. This information is not only useful for grouping lines into a UIC number, but can also be helpful to optimize previous stages:

- The text on train wagons is always horizontal. If a stationary system is used, or the pictures are taken with a hand held device, it can be assumed that text is recorded horizontally. This can simplify the task because only one text direction has to be checked.

- It is only necessary to recognize digits. Traditional text detection methods have to be able to deal with different baselines and character heights. For example, the characters "y", "a" and "L" have a different baseline and height. The characters "0-9" all have the same size and height. This means that it can be assumed that all characters in UIC numbers have a similar height and baseline.

- The first two lines of the UIC number also contain text to the right of the two digits. This allows the detector to search for lines of text and not just for a group of 2 digits.

- The Text in the UIC block could be used to validate the first 4 digits. For example, "A-ÖBB" translates to 81.

- The check digit can be used to validate the result.

### 1.1.1 Check digit

Every UIC number can be validated using its check digit by the Luhn algorithm [Luh60]. The algorithm, also known as as mod 10 algorithm, is widely used for number validation and can detect any error that effects one digit. The most prominent use of this method are Credit card numbers.
A number can be validated with the algorithm shown in listing 1.1:

| Resolution | No. |
|---|---|
| 21 MP | 9 |
| 16 MP | 88 |
| 15 MP | 35 |
| 10 MP | 2 |
| 6 MP | 2 |
| 5 MP | 151 |
| 4 MP | 2 |
| 2 MP | 1 |
| 1 MP | 2 |

Table 1.1: Input image resolutions

```
bool luhn(int[] digits, int n){
  int sum = 0;
  for (int i = 0; i < n; i++) {
    // Start with the check digit
    int value = uic[n-i-1];
    // Double every other digit
    if ( i%2 == 1 )
      value *= 2;
    // Subtract 9 if the value is bigger than 9
    if (value > 9)
      value -= 9;
    // Sum up all value
    sum += value;
  }
  // The sum must be divisible by 10
  return sum % 10 == 0;
}
```

Listing 1.1: Validate number with Luhn algorithm

Although the check digit could be used to improve the result in cases where one digit has a very low confidence value, it would significantly increase the chance of validating a invalid number.

### 1.1.2 Dataset

A dataset containing UIC images was not publicly available, therefore the entire dataset used in this thesis was collected by the author. It was recorded with mobile devices and contains 292 images with 144 unique UIC numbers. The dataset was published and can be downloaded at [urlh]. Most images were recorded with a "Moto X Play" Android phone with (21MP f/2.0) and a compact camera with 5MP and optical zoom. The optical zoom allowed taking pictures of wagons in a further distance without having to walk over the train tracks.

Most images recorded with the phone were taken with 15 and 16MP whereas the compact cameras pictures have a resolution of 5MP. The exact distribution of the image sizes can be seen in table 1.1.

All pictures were annotated with the corresponding UIC number, its bounding box and the wagon's bounding box. This allows to test different scenarios by either using the original image as an input, an image of the coach or only the UIC number. Additionally, every image was given a subjective quality measure in the range of 1-5 whereas 1 is the best quality and 5 is hardly readable.
To simplify the annotation process a simple Java tool was written. Instructions on how


Figure 1.2: Examples with different ratios of the UIC area and the train area

to utilize this tool are stated in section 5.1. The final dataset contains one folder per image and UIC number. Its name consists of its UIC number and an index to distinguish between folders with the same number (eg. *"218124700764_4"* is the fourth image with the same UIC number). Each folder then contains the original image (*"original.jpg"*) and a text file with the additional information (*"data.txt"*). The syntax of the data file can be seen in listing 1.

```
2592.0          // Image width
1944.0          // Image height
2016-09-05T17:32:53         // Recording date and time
DSC-W580        // Camera model
613.0,594.0,1975.0,731.0    // Bounding box of the train wagon (x,y
    ,width,height)
668.0,1040.0,236.0,150.0    // Bounding box of the UIC number (x,y,
    width,height)
315453778591    // UIC number
4               // Subjective quality measure (1-5)
```

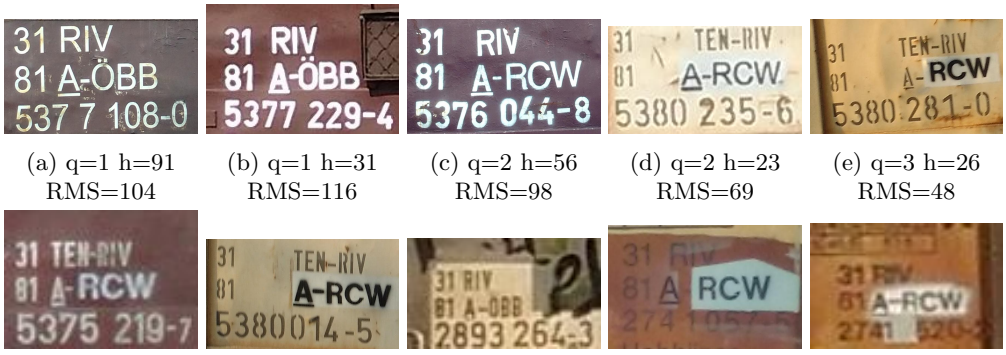
Listing 1: Example of a *"data.txt"* file

Figure 1.2 shows examples of train wagons with a UIC number. The images highlight that the UIC numbers only makes up a small area of the input image. Even though the resolution of the input image might be high, the UIC number might only be a small subsection of it. This in turn can result in a poor resolution available for the digit recognition.

Figure 1.3 shows examples of UIC numbers. Each image is labeled with its subjective quality rating, the smallest digit height and the Root Mean Square value. The RMS is calculated for the area in the digit bounding boxes of a UIC number and can be used as an indicator for the contrast of the digits. It is calculated by using the intensity values and size of the image (Equation 1.1). $I_{xy}$ is the intensity value of the pixel at position $(x, y)$. $W$ is the width of the image and $H$ is its height. For the application to color images, the luminance channel is used.

$$RMS = \sqrt{\frac{1}{WH} * \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} (I_{xy} - \overline{I})^2} \tag{1.1}$$

$$\overline{I} = \frac{1}{WH} * \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} I_{xy} \tag{1.2}$$




(a) q=1 h=91 RMS=104 (b) q=1 h=31 RMS=116 (c) q=2 h=56 RMS=98 (d) q=2 h=23 RMS=69 (e) q=3 h=26 RMS=48

(f) q=3 h=12px RMS=66 (g) q=4 h=77 RMS=58 (h) q=4 h=17 RMS=99 (i) q=5 h=61 RMS=21 (j) q=5 h=9 RMS=46

Figure 1.3: Examples of UIC images with their quality (q=1-5) and height

The combination of the height and the RMS value, offers a good indicator of the UIC quality. However, the examples in figure 1.3 show that these values are not sufficient to fully describe the quality of the images. Properties that can prevent a detection are:

- Low contrast
- Low resolution
- Rust can result in low contrast for black digits on brown background or damages to the digits (Figure 1.3c and 1.3i)
- The font size can vary within on line, as can be seen with the check digit in figure 1.3f
- Some numbers were spray painted with a template resulting in lines through digits (See figure 1.3d and section 3.5.1)

Figure 1.4 shows the distribution of the RMS and minimal height for each UIC entry with their subjective quality ranking. Thereby, the minimal digit height in pixels is a good indicator for the image resolution. The RMS value represents the contrast of the digits, the higher the better. It was calculated for the area inside the digits bounding boxes.

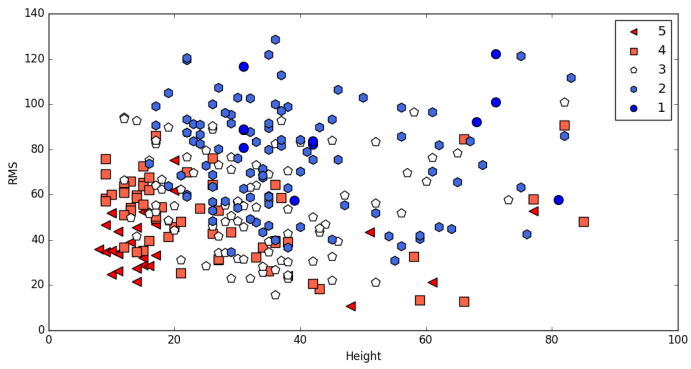



Figure 1.4: Subjective quality rankings of UIC number in dataset

## 1.2 Related Work

No previously published work in the filed of UIC number detection was found, but several commercial systems are available. Little information about these systems is publicly available. Due to a lack of performance metrics for these systems they cannot be directly compared to each other. To extend the publicly available data, all companies were contacted, but only some of them responded.

- **NumberCheck** is a solution by the company Advanced Security Engineering (ASE)
  The company claims that their system can reach a recognition rate of up to 96% with speeds up to 80km/h, but they admit that they can not detect the numbers if they are in bad condition such as being covered in snow or having a low contrast. They can achieve this accuracy by using stationary image sensors at both sides of the train track. This allows them to record multiple images of both sides and both versions of the UIC number [urld]. Additionally, their custom NumberCheck system comes with LED lighting and wheel sensors that can be used to determine the speed and direction of the train [urle]. Unfortunately, no information about the resolution of their input images or their processing speed was provided.

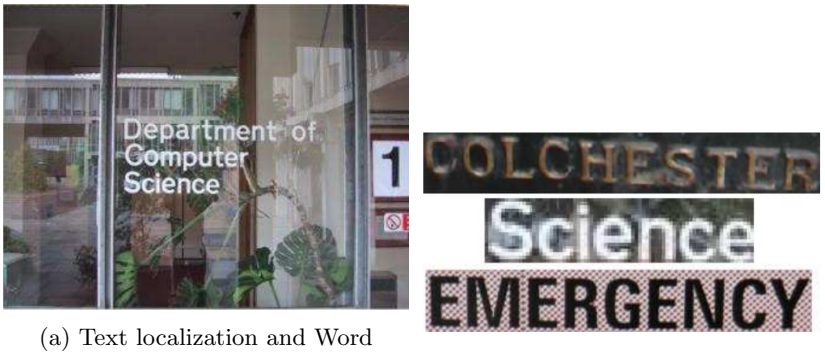
- **ARVIS** (Synonym for "Automatic rail vehicle identification and uic codes recognition system") was a project by multiple companies from the Czech Republic, Switzerland and Slovakia running from March 2012 till January 2016. Its goal was to research and develop a full system for the automatic detection of UIC numbers. Although little information could be found on the project site [urla], more details could be obtained by contacting the listed main contact Peter Honec. According to him, their final product uses a line scan camera with a resolution of approximately one pixel per 3x3mm and a height of 3000mm. With this and an illumination unit, they are capable of detecting UIC numbers at speeds up to 160km/h. In ideal conditions, their system can have an accuracy of more than 98.5% which is the same as their license plate recognitions system. In real world conditions, such as high speeds, night, rain, fog, dirt and rusty wagon numbers they reach an accuracy of 95% with passenger wagons. As for transport wagons, the topic of this thesis, the information given was that it is lower. Asked about the processing speed of the system, the answer was that ARVIS can process a whole train with approximately 30 wagons in a few seconds.

- **CARMEN UIC** by ARH is a software solutions that can be integrated in existing systems and has low system requirements. According to their website, it only requires a 1GHz processor and 512MB of RAM [urlb]. According to a ARH sales manager, they do not have any reliable information about the accuracy due to the many factors it depends on, but they use the same algorithm as for their automatic number plate recognition product with an accuracy of 98.5%.

- **Trex-Wagon** is offered by Nestor Technologies. It can be used with one or two cameras (one for each side) and visible or infrared lighting [urlg]. Unfortunately, no information about its performance could be found and an attempt to contact them remained unanswered.

### 1.2.1 Automatic number plate recognition (ANPR)

Although the above mentioned field of automatic number plate recognition (ANPR) seems very similar at the first glance, it has some crucial differences that do not make it directly transferable to the domain of UIC number detection.
ANPR focuses on the identification of vehicles by reading the information available on




(a) Text localization and Word recognition

(b) Character recognition

Figure 1.5: Example input images from the ICDAR 2003 challenges [LPS+03]

their number plates. The extracted information can be used in a number of applications such as automatic traffic surveillance, toll or parking systems. License plates are more strictly regulated as UIC numbers, a reliable detection system must be capable of dealing with a variety of different number plate standards and different vehicle types or countries. These inconsistencies include different aspect ratios, background images and character groupings. The detection is further aggravated by unstable lighting and weather conditions, and erosion or partial coverage (by dirt or other substance) of the plates.
On the contrary, the rectangular shape of the plate and its reflective properties, especially with the combination of infrared light, can be used to create very reliable sytems with accuracies of 99% and higher [DISB13].

### 1.2.2 ICDAR

The International Conference on Document Analysis and Recognition (ICDAR) was first held in 1991 and has been repeated every two years since. It does not only allow information exchange in the field of document analysis and recognition, but it also provides a platform for the comparison of different text detection algorithms in more complex scenes as documents. Since 2003 competitions in robust reading are held and are extensively used in the literature to compare different approaches [PHL09, EOW10, CTS+11, NM11a, NM12, GBYB12, NM13, BNM15, NM15]. The competitions provide a dataset and clearly defined measures such as precision, recall and f-measure for the evaluation. The sub-problems that are of interest for this work are:

1. **Text localization**
   The goal of this competition is to find the bounding boxes of all words in an image. As an example, a successful system should find a bounding box for "Department", "of", "Computer" and "Science" in image 1.5a. Thereby, it is not necessary for the system to detect each character individually as long as enough characters were found to extract the outline of the word. Lucas et al. defined the following evaluation framework with the focus on being easy to understand and compute [LPS+03]. Because it is also used for the evaluation of different stages of the UIC detector in this work, a short summary is provided in this section.
   A match of target rectangle $r_t$ and estimated rectangle $r_e$ is calculated by dividing the area of the intersection of the two rectangles by the area of the smallest rectangle containing both of them (Equation 1.3).

$$m(r_t, r_e) = \frac{r_t \cap r_e}{r_t \cup r_e} \tag{1.3}$$

The resulting value is 1 for completely overlapping rectangles and 0 for rectangles that are not overlapping at all. To compute the match value for one rectangle in a set of rectangles (e.g. the ground truth), the best match value is selected (Equation 1.5).

$$m_{\max}(r, R) = \max m(r, r') \mid r' \in R \tag{1.4}$$

With the help of this function, the precision $p'$, recall $r'$ and f-measure $f'$ of the target rectangles $T$ and estimated rectangles $E$ can be calculated as follows:

$$p' = \frac{\sum_{r_e \in E} m_{\max}(r_e, T)}{|E|} \tag{1.5}$$

$$r' = \frac{\sum_{r_t \in T} m_{\max}(r_t, E)}{|T|} \tag{1.6}$$

$$f' = \frac{2p'r'}{p' + r'} \tag{1.7}$$

$$F2 = (1 + 2^2)\frac{p'r'}{2^2p' + r'} \tag{1.8}$$

The recall describes how many of the target rectangles were found, weighing each found rectangle by its overlapping area. The precision corresponds to the quality of all estimated rectangles. The equally weighted combination of both is the f-measure. $F2$ weights the recall more than the precision.
Although this method was used widely until ICDAR 2011, the original paper already noted that it does not perform well with over or under segmentation [LPS+03]. Thereby, detection algorithms that split words into several components or merged several words into one are harshly penalized.
As a result of these weaknesses, the framework was replaced by the proposed method of Wolf and Jolion for the ICDAR 2011 competition and has been used since then [WJ06, SSD11]. They assessed multiple evaluation systems and proposed a more complex method that uses object count/area graphs. Additionally, they published the openly available software DetEval for the evaluation of object detection algorithms.

2. **Character recognition** The target of this challenge is to read words from cropped real world images as seen in figure 1.5b. Thereby the text does not need to be located anymore but no detailed information about the shape of each character is available. To compare different systems, the edit distance is used. Therefore, characters either have to be extracted from the background or the input images are directly classified. This problem falls into the field of Optical Character Recognition (OCR). A successful character recognition system can be combined with a text localization system to create a full text detection system.

3. **Word recognition** This challenge deals with end to end text recognition. It can be seen as a combination of the text localization and character recognition task but in reality many text localization systems provide more information than just the location of potential text areas. For example, the outlines of each character are known from an earlier stage and can be passed directly to an OCR system without preprocessing.

Because the success of end to end systems is strongly dependent on the initial localization stage, most research is done in this area. By providing a clearly defined evaluation framework and fully annotated dataset, the ICDAR competitions provide a universal benchmark for the research community. To address criticism of the framework, the dataset and evaluation method has been updated multiple times. The initial dataset from 2003 [LPS+03] was extended in 2011 and the ground truth was annotated more

carefully [SSD11]. Additionally, the evaluation method was changed to the framework suggested by Wolf and Jolion [WJ06]. The dataset was extended further in 2013 to a total of 462 images, of which 229 are used as the training set and 233 were used as the test set [KSU+13]. Additionally, the annotations were changed from bounding boxes to pixel level annotations. Table 1.2 lists a small selection of results for the datasets and highlights the progress made over the last decade in the field of text localization.

| Method | Dataset | Prec. | Reca. | f |
|---|---|---|---|---|
| Pan et al. [PHL09] | 2003 | 67 | 71 | 69 |
| Epstein et al. [EOW10] | 2003 | 73 | 60 | 66 |
| Chen et al. [CTS+11] | 2003 | 73 | 60 | 66 |
| Neumann et al. [NM11a] | 2003 | 59 | 55 | 57 |
| Neumann et al. [NM11a] | 2011 | 68.93 | 52.54 | 59.63 |
| Gonzlez et al. [GBYB12] | 2011 | 72.67 | 56.00 | 63.25 |
| Neumann et al. [NM12] | 2011 | 64.7 | 73.1 | 68.7 |
| Neumann et al. [NM13] | 2011 | 67.5 | 85.4 | 75.4 |
| Yin et al. [YYHH14] | 2011 | 68.26 | 86.29 | 76.22 |
| Neumann et al. [NM15] | 2013 | 72.4 | 81.8 | 77.1 |
| Busta et al. [BNM15] | 2013 | 69.3 | 84.0 | 76.8 |

Table 1.2: Text localization results of ICDAR competitions

### 1.2.3 Text detection in complex fields

Due to the lack of a distinctive property of UIC numbers, like the rectangular shape of number plates, it makes sense to look at the filed of text detection in complex scenes (TDCS) for useful techniques. Thereby, the focus lies on the direct detection of unstructured text. TDCS deals with the whole process of converting images of handwritten or printed text into digital strings of text. Input images can range from scanned pure text documents to an image of a food market with handwritten price tags at arbitrary locations. Although these systems are more complex because they are optimized to detect unstructured text, the underlying techniques can be used to find lines of text which in turn can be used to extract the UIC numbers.
Most of the approaches in this field can be broken apart into character detection, line detection and optical character recognition. The first stage extracts possible character candidates from the image. The next stage filters these candidates and tries to group them into lines and words. The last stage converts the lines into machine readable strings. The main focus in research lies on the efficient character detection.

In 2010, Epstein, Ofek & Wexler proposed stroke with transform (SWT) as a method to distinguish characters from background [EOW10]. It is based on the observation that characters tend to have a constant stroke width. Their algorithm computes the stroke width for each pixel of the components extracted from the gradient domain of the input image. They then compute the variance of the stroke width for each component. If the stroke width is too inconsistent, the component is rejected.
Chen et al. use an edge enhanced version of Maximally Stable Extremal Regions (MSER) and an adopted version of SWT for text detection [CTS+11]. They use MSER, originally developed by Matas, Chum, Urban & Pajdla, as a method for finding corresponding components in two stereo images, to find character candidates [MCUP04]. The extracted candidates are then filtered according to their stroke width variance and grouped in lines and words. Their final system archives a very similar performance to Epstein et al.
On the contrary to region based systems, Coates et al. use a convolution neural network and a big training dataset to calculate the probability of being part of a text segment

for each pixel [CCC+11]. Originally focusing on MSER as well, Neumann and Matas compute features for all detected components and use a classifier to filter out non characters [NM11a]. Although the performance of their approach was behind older methods such as SWT, a very similar approach with a more sophisticated selection of features by Gonzlez, Bergasa, Yebes & Bronte could deliver better results [GBYB12].
One year later, Neumann and Matas improve their method by computing the features during the Extremal Region (ER) detection stage and ignoring the criteria of stability. Instead, they use the computed features directly with a classifier to sort out non characters. In a second stage, they use more complex features to remove even more non characters before grouping them into lines [NM12]. Furthermore, they focused on combining characters extracted from different color channels and scale of the same image in [NM13].
Yin, Yin, Huang and Hao improve MSER by using an effective pruning algorithm [YYHH14]. Using MSERs as an initial starting point as well, Neumann and Matas use a method inspired by the SWT to extend faulty regions to improve the quality of the extracted characters [NM15].

# Chapter 2

# Theory

For the design of the UIC detector, inspiration was drawn from the field of Text detection in complex scenes (TDCS). This field can be split into the fields of **Character detection**, **Line detection** and **Optical Character Recognition**. The goal of this chapter is to explain methods created by other researchers that are applied to the domain of UIC number detection. Several properties of UIC numbers influenced the selection of methods from previous research. The following enumeration provides a brief summary of these factors:

1. **Character Detector** (Section 2.1)
   The character detector's purpose is to find all digits in a UIC number. The main difference to traditional TDCS is that it is more important to find all digits. In TDCS, a missed character in a word might not change the outcome of the found bounding boxes or can even be corrected by using a dictionary in a later stage, but a missed digit in a UIC number results in a complete failure of the detector on the input image. Additionally, the properties of UIC numbers provide more information (See section 1.1) that can be used in later stages to filter unwanted character candidates. Therefore, the character detector should focus more on recall rather than precision.

2. **Line Detector** (Section 2.2)
   It groups potential characters into line candidates. In the literature, line detectors can be classified into either horizontal line detectors, that can only detect horizontal lines, or more complex methods that aim to be rotation invariant. As discussed in section 1.1, it can be assumed that UIC numbers are horizontally aligned. Therefore, a horizontal text detector is sufficient. Another point to consider is that the number of characters that should be grouped from the character detector is higher due to the focus on recall. As a result, the line detector should have a low runtime complexity to be able to handle a high number of input regions.

3. **Optical Character Recognition** (Section 2.3)
   This stage classifies each region as a character. Traditional OCR has to recognize the whole alphabet A-Z, both uppercase and lowercase, in addition to digits 0-9. Different systems are even capable of detecting more complex alphabets in different languages, such as Chinese, but for the domain of this thesis it is only necessary to recognize digits. On the contrary, OCR systems often use dictionaries to improve their recognition rate, which is not possible for UIC numbers unless the UIC numbers are known and only have to be assigned to a specific train.

Note that the order of these steps is not fixed. A system could for example decide to group lines into UIC numbers before performing OCR, this way characters with a low

confidence can be filtered before grouping them into lines.
In section 2.1, the used character detectors and the underlying principles are discussed. Section 2.2 explains the two used line detectors and section 2.3 offers an overview over the used OCR solution.

## 2.1 Character Detector

Two different character detection methods were tested for the UIC detector. Both of them were published by Neumann and Matas, but the most recent one also included Busta as an author. This decision is based on the extensive research done by them and the high number of other publications building on top of their methods. Their performance in the ICDAR competition can be seen in table 1.2.

- Neumann and Matas 2012 [NM12]
  The first method, from here on referred to as ER-Detector, was developed by Neumann and Matas in 2012 and was included in the OpenCV contribution library [NM11a, NM12]. Not only does this have the benefit of not having to implement the solution manually, but it also provides a well tested framework. Their method achieved a precision of 64.7%, a recall of 73.1% and an f-measure of 68.7% on the ICDAR 2011 dataset.

- Busta, Neumann and Matas 2015 [BNM15]
  The newer of the two methods, called FASText, was published by Busta, Neumann and Matas in 2015. Although another method by Neumann and Matas achieved a slightly higher f-measure of 77.1% compared to 76.8% on the ICDAR 2013 dataset, FASText has a higher recall rate (81.8% vs 84.0%) [NM15]. This is advantageous because it is more important to detect all of the UIC digits correctly to extract a valid UIC number. Another reason why this method was chosen is that the original publication only focuses on the detection of text in a gray scale image, ignoring any color information. Therefore, the hope was that extending the algorithm to a more complex color model would improve the result further.

Both of the used approaches use different concepts. Whereas [NM12] focuses on the detection of robust connected components as Extremal Regions, [BNM15] focuses on the extraction of characters based on their stroke. As a result, both methods have their own strengths and weaknesses in the field of UIC detection. This section provides detailed information about the underlying principles of both approaches and highlights all changes made to improve their results in the domain of UIC detection.

### 2.1.1 Stroke Width Transform

Stroke Width Transform (SWT) is a method proposed by Epstein, Ofek and Wexler in 2010 to find text in complex scenes [EOW10]. Their method is based on the observation that most characters in natural text are made out of strokes of the same thickness. Therefore it focuses on finding connected components with a stable stroke width.
Their method uses a Canny edge detector and tries to find an opposite edge pixel for each edge pixel [Can86]. This is done by following the direction of its gradient until another edge pixel is hit. If the found pixel has a gradient in the opposite direction, it is considered to lie on the same line, and all pixels in between are assigned to the distance of the two opposing pixels. This distance metric is used as an approximation of the line width. The resulting output of the SWT is an approximation of the thickness of the corresponding line (Initially all pixels are set to infinity).
The next step uses the output of the SWT to group neighboring pixels with similar width values into connected components (CCs). Finally, the variance of stroke width of

each CC is calculated. If a CC has a low variance, then it is considered as a character. If the variance is too high, the aspect value exceeds a certain threshold or a component contains too many other components within its bounding box (e.g. border around text), it is discarded.
Finally, Epstein et al. further filters characters by grouping them into words and lines to discard components with similar stroke width properties as characters (e.g. lamp posts). The original algorithm was applied on the whole image which resulted in a slow system. This issue was addressed by Chen et al. by only applying the algorithm on previously extracted MSERs (Section 2.1.2) which significantly reduced the execution time, while retaining the same result in the ICDAR 2003 competition (See table 1.2) [CTS+11]. Although this algorithm is not directly used in this work, the concept of finding components with a stable stroke width has inspired multiple other algorithms such as FASText.

### 2.1.2 Maximally Stable Extremal Regions

Figure 2.1: Different threshold values applied to a gray scale image

Maximally stable extremal regions (MSER) were originally developed by Matas, Chum, Urban & Pajdla [MCUP04] to find matching regions in two stereo images, but were also successfully used for character detection [NM11a, CTS+11, YYHH14]. MSERs are extracted by thresholding an image at every possible value, extracting connected components from the resulting monochrome images and selecting the regions that do not change over a certain number of thresholding values. By thresholding the image at the lowest possible value, it results in a complete black image. By increasing the value, brighter areas start to appear as white regions. Increasing the threshold further lets the regions grow and merges them with neighboring blobs (Figure 2.1). The regions that stay stable over a defined range of thresholding values are extracted as MSERs. The resulting regions were then used by Matas et al. to find corresponding regions in a stereoscopic image. By sorting all pixels in the image in buckets of 0 to 255 according to their intensity value at the beginning of the process, the algorithm can be implemented in $\mathcal{O}(n * log(log(n)))$.

Although these regions are scale independent to some extent, Mikolajczyk et al. showed that MSER are sensitive to blur [MTS+05]. This not only hinders the detection in non optimal conditions, but also means that a down sampled version of the input image with more defined edges can yield more MSERs than the original image.
Chen et al. used MSER the first time for character detection and addressed the blur issue by combining it with Canny edges [CTS+11]. For the resulting edge enhanced regions, the stroke width transform was used to filter non characters.

### 2.1.3 Extremal Regions

Although Neumann and Matas originally used MSERs for text detection [NM11a], they discarded the "Maximally Stable" criteria and instead started focusing on Extremal Regions (ERs) directly [NM12]. Extremal regions are a super set of MSERs. They include all connected components extracted by thresholding the image at different values and

are not filtered by a stability criteria. Because keeping all ER is not an option due to their heigh number, a different filtering method was required.
In their original approach, they used a classifier to filter the extracted MSERs according to multiple computed features, like the aspect ratio and the background color consistency. The full list of these features can be found in table 2.1. Although they also used MSERs for the initial detection, they did not use any edge enhancing techniques to counteract the weakness of MSER with blurred images like Chen et al. [CTS+11]. This resulted in a lower recall rate of only 55% compared to 60% by Chen et al..

| | |
|---|---|
| Aspect ratio | Relative segment height |
| Compactness | Number of holes |
| Convex hull area to surface ratio | Character color consistency |
| Background color consistency | Skeleton length to perimeter ratio |

Table 2.1: Features used to filter MSERs in [NM11a]

To improve their result, Neumann and Matas decided to ignore the stability criteria and directly compute a score for each ER during the extraction process. Because the computation of their previously used features would be to expensive for all ERs, they proposed the use of features that can be computed in $\mathcal{O}(1)$ during the extraction process. To do this, their algorithm starts by initializing every pixel as a component and merging neighboring components during the thresholding operation. For the merging operation "$\bigoplus$", the metrics of the new component must be computable in $\mathcal{O}(1)$ using the information of its sub components.

- **Area** $a$
  The area can easily kept track of by initializing every pixel with 1 and adding the areas of both subcomponents ($\bigoplus = +$).

- **Bounding Box** $(x_{min}, y_{min}, x_{max}, y_{max})$
  Is initialized with the pixels location $(x, y)$ as $(x, y, x{+}1, y{+}1)$ and uses (min, min, max, max) for each element in the bounding box tuple to merge two components.

- **Perimeter** $p$
  $p$ can be computed by checking how much the perimeter is going to change by adding one pixel. By knowing the number of direct neighbors $n$ inside the component that should be merged with a pixel, the perimeter change can be calculated with $4-2n$. The perimeter of two neighboring components can be calculated with an addition.

- **Euler number** $\eta$
  $\eta$ is the difference between the number of connected components and the number of holes in an image. Because the number of connected components of the ER is one, it can be used to compute the number of holes. Neumann and Matas use an adopted version of an algorithm developed by Gray in 1971 [Gra71]. The Gray algorithm counts the number of occurrences of 2x2 pixel patterns in an image. The three different patterns $Q_1, Q_2$ and $Q_3$ including all of their rotations are shown in equation 2.1-2.3. By counting the number of occurrences of each pattern $C_1, C_2$ and $C_3$ in the image, the Euler number can be calculated with equation 2.4.
  To improve the performance during the thresholding algorithm, each component keeps track of $C_1, C_2$ and $C_3$. If a pixel gets added, the difference for each value is calculated. This can be done in constant time because each pixel has only 4 2x2 neighborhoods around it. When two regions are merged, $C_1, C_2$ and $C_3$ are simply

summed up.

$$Q_1 = \left\{ \blacksquare\blacksquare\blacksquare\square, \blacksquare\blacksquare\square\blacksquare, \square\blacksquare\blacksquare\blacksquare, \blacksquare\square\blacksquare\blacksquare \right\} \tag{2.1}$$

$$Q_2 = \left\{ \square\square\square\blacksquare, \square\square\blacksquare\blacksquare, \square\blacksquare\square\square, \blacksquare\square\square\square \right\} \tag{2.2}$$

$$Q_3 = \left\{ \blacksquare\square\blacksquare\square, \square\blacksquare\square\blacksquare \right\} \tag{2.3}$$

$$\eta = \frac{C_1 - C_2 + 2C_3}{4} \tag{2.4}$$

- **Horizontal crossings** $c$
  $c$ keeps track of the number of crossings of each Y value in a component. Crossings can be counted by analyzing how often a component is entered and left along a line with a fixed Y value. If a double sided queue is used for the implementation, the complexity is $\mathcal{O}(1)$. When a pixel is added, the corresponding Y value can be altered depending on the pixel's horizontal neighbors. Two components can be merged by an element wise addition of all horizontal crossings.

With the help of these metrics, a real AdaBoost [SS99] classifier was trained. With the classifier, the probability of being a character could be calculated and used to select the components with the highest probability as Extremal Regions. Thereby, the history of each component during the thresholding phase was used to only select the ones with a locally maximal probability. Additionally, the difference of probability between the local maxima and the local minima had to be above a certain threshold. Finally, each Extremal Region needed to have at least a minimum probability of being a character. This process can be seen as performing sanity checks for the character candidates where characters with too extreme properties are discarded. Instead of using separate filters for all features, the classifier can combine them and discard components with unusual combinations of features. The features used for the classifier are:

- **Aspect ratio:** $(x_{max} - x_{min})/(y_{max} - y_{min})$
  The aspect ratio can be computed using the bounding box of a connected component. It is a very simple feature, but it allows to discard a high number of components. Although the aspect ratios of 'm' and 'I' are very different, it can be used to filter out regions with significantly higher or lower values.

- **Compactness:** $\sqrt{a}/p$
  The compactness can be computed with the area and the perimeter. Components with a very low compactness can be rejected.

- **Number of holes:** $1 - \eta$
  The number of holes can be computed with the Euler number. An undistorted character should have a maximum of 2 holes ("8"/"B").

- **Number of horizontal crossings:** $\text{median}(c_{1/6h}, c_{3/6h}, c_{5/6h})$
  Thereby, only three values of the horizontal crossings list are used to allow a computation in $\mathcal{O}(1)$. One value from the first 1/6 of the height, one from the middle and the last one from 5/6.

In addition to these features, more complex metrics are computed for the remaining Extremal Regions. The combination of these is then used with a second AdaBoost classifier to reject ER with a low probability:

- **Hole area ratio:** $a_h/a$
  Is computed by dividing the area of holes $a_h$ by the character area $a$. Distortions can lead to holes in characters. If these holes are small in comparison to the area of the whole region they should not influence the scoring function.

- **Convex hull ratio:** $a_c/a$
  Is determined by dividing the area of the convex hull by the character candidate area. Can be seen as a more complex version of the compactness.

- **Number of outer boundary inflexion points:** $n_i$
  Is the number of changes from a convex to a concave angle around the region border. Characters have a limited number of inflexion points. A very high number therefore indicates a non character.

| Classifier 1 | Classifier 2 |
|---|---|
| Aspect ratio | Hole area ratio |
| Compactness | Convex hull ratio |
| Number of holes | Number of outer boundary inflexion points |
| Number of horizontal crossings | + Features from Classifier 1 |

Table 2.2: Features used to filter Extremal Regions [NM12]

The full list of features used in both classifiers can be seen in table 2.2. Even though all used features are scale invariant, they are not invariant to rotation. To counteract this issue, rotated versions of characters were used during the training of the classifier.

The process described so far can only detect characters in gray scale images. The extremal regions found in the intensity channel only accounted for 85.6% of the ground truth. Thereby, the algorithm was applied to the original and the inverted image to find dark characters on a white background. To apply it to color images as well, Neumann and Matas performed all described steps separately for the intensity, hue and saturation channel. Additionally, they used an intensity gradient representation of the intensity channel. A combination of all channels improved their recall rate to 94.8%. Having to invert each channel as well, results in 8 iterations of the algorithm. As a side effect of the multiple iterations, characters that are visible in multiple channels might be detected more than once. With the argument that the characters in one line of text tend to have the same color, Neumann and Matas decided to group characters into lines for each color channel separately, before combining all lines.

The Extremal Regions method was chosen as a character detector for UIC detection because of the simplicity of their implementation, and because of their adoption in the OpenCV library. This not only shows its usability in real world scenarios, but also provides a well tested implementation.

### 2.1.4 FASText

Inspired by the principle of the Stroke Width Transform, Busta, Neumann and Matas adapted the FAST feature detector to fire on stroke endpoints and bends [BNM15]. These text key points are then used to create connected components of characters which can be filtered by features calculated during the clustering process. The disadvantage of this approach is its scale dependency, which requires the use of image pyramids to detect characters with different stroke widths. Additionally, they only describe the application of this method for gray scale images and ignore potential information that color channels can provide. The algorithm can be segmented in three stages:

- **Keypoint extraction**
  For the extraction of keypoint from an image the 5x5 neighborhood of each pixel is checked. Thereby, the intensity $I$ of the center pixel $p_c$ is compared with the pixels $P_c$ in the surrounding circle of radius 2 (See figure 2.2). Each of the 12 $P_c$ pixels

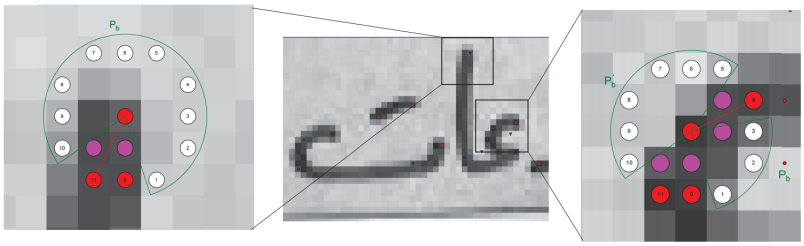


Figure 2.2: Extraction of Stroke End/Bend Keypoint (SEK/SBK) [BNM15]

is classified as either similar, brighter or darker. A pixel is similar if its intensity value lies within a predefined margin $m$. Using $I(p)$ as the intensity value of pixel $p$ and $P_c$ as the set of pixels in the outer circle, the sets of darker $P_d$, similar $P_s$ and brighter pixels $P_b$ can be defined as follows:

$$P_d = \{\ p \in P_c \mid I(p) \leqslant I(p_c) - m\} \tag{2.5}$$
$$P_s = \{\ p \in P_c \mid I(p_c) - m < I(p) < I(p_c) + m\} \tag{2.6}$$
$$P_b = \{\ p \in P_c \mid I(p) \geqslant I(p_c) + m\} \tag{2.7}$$
$$P_o = P_d \cup P_b \tag{2.8}$$

Note that the set $P_o$ contains the set of brighter and darker pixels.
If the circle contains brighter and darker pixels, or there are no similar pixels, the center pixel is rejected as a keypoint. Otherwise, the pixels are grouped in continuous circle segments.
If there is only one continuous segment with a maximum of 3 pixels, the center pixel is a Stroke End Keypoint (SEK). By rejecting similar segments of more than 3 pixels, corners that are part of a bigger area and not a stroke can be filtered out.
If there are two segments of similar intensity and both only contain a maximum of 3 pixels, it is considered as a stroke point. To reduce the number of keypoints, only bends are selected. This can efficiently be done by checking if any two opposite pixels have a similar intensity value as the center. If so, it can be discarded for being on a straight line. Otherwise, it is a Stroke Bend Keypoint (SBK).
Finally, it is checked if the outer segments of the SEKs and SBKs are connected to the center pixel. Each of the segments found in the earlier stage has to be connected to the middle pixel in its 3x3 neighborhood.
To further limit the number of keypoints, Busta et al. use non-max suppression to only use the keypoint with the highest contrast in a $3x3$ neighborhood. The contrast of a keypoint can be determined by calculating the maximum intensity difference of the center pixels with the outer circle pixels.
Additionally, the number of keypoints per image was limited to 4000 by splitting the image in cells of uniform size and only selecting a fixed amount of keypoints per cell. Again, the keypoints with the highest contrast are selected.

- **Region segmentation**
  In this step, all previously extracted keypoints are used to extract the stroke of each character. Thereby, a threshold value is extracted directly from the keypoint. This threshold value is then used by a flood-fill algorithm with the center pixel as its seed to create a connected component representing the stroke of a character. The threshold used by Busta et al. was calculated differently depending on it being a dark stroke on bright background or vice versa ($P_b \ldots$ Background pixels, $I_c \ldots$ Center intensity):

$$t = \begin{cases} \max(I_x) + 1 | I_x \in P_b & \forall I_x \in P_b : I_x < I_c \\ \min(I_x) - 1 | I_x \in P_b & \forall I_x \in P_b : I_x > I_c \end{cases} \tag{2.9}$$

- **Filtering**
  In the last stage, the number of extracted regions is further reduced by filtering non character regions. To do this in an efficient manner, the method previously used by Neumann and Matas in [NM12] was applied again. For this, they calculate the same features used for their first classifier during the segmentation stage (Table 2.2). Due to the $\mathcal{O}(1)$ complexity of this calculation, the impact on the performance is marginal.
  As an additional feature, they used the Character Stroke Area (CSA) ratio developed by Neumann and Matas in 2015 [NM15]. It is based on the observation that characters can be drawn by painting a set of strokes with the same stroke width. Therefore, the width of the stroke multiplied with the length of the stroke is the CSA. The ratio of the CSA divided by the area of the region is a good indicator for the region being a character or not. The CSA could be calculated efficiently by using the SEK of a region and following the stroke in its direction. Finally, a classifier was used to distinguish between text and non text characters.

This algorithm achieved state of the art performance when the work on this thesis was started. Additionally, it offered the potential of extending the method to color images which could improve the performance of the algorithm even further. These reasons led to the decision of using the FASText detector as an additional character detector.

## 2.2 Line Detector

In this stage, character regions are grouped into lines. The complexity of the problem can be calculated as follows: A line can or can not include every possible character, which can be visualized as a binary number with one bit per character. Therefore, the number of possible character combinations in one line are $2^n$. Furthermore, the result returned by the line detector could include every possible line. With $2^n$ possible different lines, the number of different line combinations that can be returned is $2^{2^n}$. This shows that an exhaustive search of every possible combination is not feasible and more elegant algorithms are required ($2^{2^4} = 65536$).
Both character detectors used in this thesis also describe a method for line detection. Neumann and Matas [NM12] reference one of their previously developed methods [NM11b] based on an exhaustive search. Their algorithm was also implemented in OpenCV and could be integrated in the designed UIC pipeline. More information about this line detector can be found in section 2.2.1.
The second method used by Busta et al. is based on the Hough transform [BNM15]. Although briefly mentioned in this section, the method was not used for UIC detection due to the requirement of detecting groups of just two characters. More information can be found in section 2.2.2.
Finally, a new character detector was designed that could be optimized for UIC detection. This newly developed system is described in section 3.5.3.

### 2.2.1 NM Line detector

This method was developed by Neumann and Matas in 2011 and is available in the OpenCV contribution library [NM11b]. Although, it does not support arbitrary line directions, it does support different baselines of characters that have to be considered for mixed case text. The line detector was originally developed for the use with a MSER character detector, but was also successfully used in combination with Extremal Regions [NM12]. Its strategy is to find initial character pairs within the previously extracted regions. These pairs are then combined to form triplets. The triplets can then be used to form baseline estimations (Figure 2.3). Finally, lines with similar baselines

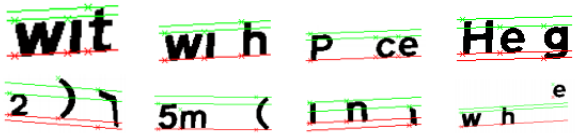


Figure 2.3: Text line estimates for triplets. Top four are valid triplets bottom four are invalid [NM11b]

are grouped into line segments. The following enumeration provides more details about the inner workings of the exhaustive search algorithm:

- **Finding pairs:**
  In this $\mathcal{O}(n^2)$ operation every previously extracted character has to be compared with every other character. Two characters are considered a pair if they have a similar height, shape and are sufficiently close to each other. These conditions previously used by Epstein et al. and Neumann and Matas were extended for the use of MSERs [EOW10, NM11a]. Thereby, characters that have a parent-child relationship are rejected as pairs (eg. Character 1 was extracted by increasing the threshold for Character 2). The result of this stage are $\mathcal{O}(n^2)$ pairs.

- **Finding triplets:**
  In this stage, line estimates are predicted. Although, it is possible to estimate the top and bottom line of a text line by just looking at two characters for upper case characters (eg. 'AB'), it is not possible to find the top and bottom line for every combination of uppercase and lowercase characters. The two characters 'Ay' for example would result in a wrong prediction, if the bottom left corners are used for the estimation of the bottom line and the top left corners are used for the estimation of the top line.
  One text line requires the algorithm to keep track of four different base lines. Two top lines for words like 'tale' and two bottom lines for words like 'you'. With this rule, triplets (two pairs that share one character), can be rejected if the text line would require more than two top or bottom lines. Examples of this can be seen in figure 2.3. Note that the algorithm tolerates a certain error that was determined through experiments.
  Each pair is potentially compared to every other pair which results again in an $\mathcal{O}(m^2)$ complexity, with $m$ being the number of pairs. The resulting runtime complexity and number of triplets of both stages is $\mathcal{O}(n^{2^2}) = \mathcal{O}(n^4)$.

- **Finding lines:**
  Finally, triplets are grouped into lines. This is done by merging triplets if at least one of their top and one of their bottom lines match. Again, an experimentally determined tolerance is allowed. To reduce the runtime, not every possible combination is checked. Instead, merged regions could not be part of any other lines. The resulting algorithm must potentially check every triple with every other, which results in a final complexity of $\mathcal{O}(n^{2^{2^2}}) = \mathcal{O}(n^8)$. Note that the number of actual triplets is significantly lower due to the closeness and similarity criteria. Nevertheless, it shows that the algorithm is not suitable for a high number of characters.

- **Selecting lines:**
  Finally, if two valid sequences share a same character, only the longer sequence is kept and returned as a line.

### 2.2.2 Hough line transform

The method proposed by Busta et al. is capable of detecting text lines in arbitrary text direction [BNM15].
First, they define neighbors as two characters that are similar size and have their centers close to each other. More precisely:

$$|| \ c(r_i) - c(r_j) \ || < \alpha \sqrt{max(\ A(r_i), A(r_j)\ )} \tag{2.10}$$

$$max(\frac{A(r_i)}{A(r_j)}, \frac{A(r_j)}{A(r_i)}) < \beta \tag{2.11}$$

Thereby $r_i$ and $r_j$ are extracted regions, $c(r)$ is the center and $A(r)$ the convex hull area of region $r$. $\alpha = 4$ and $\beta = 10$ are parameters chosen experimentally. Note that these parameters must be tolerant enough that "A" and "a" pass the test of similar size.
The next step is inspired by the Hough transform, where each valid pair votes for its text direction. The voting is done by transforming the direction of the two neighbors into the polar system $\rho = x \ sin(\Theta) + y \ cos(\Theta)$. The resulting parameter space $(\rho, \Theta)$ is quantized and the local maximums of all votes are selected as text lines. The characters on the lines can then easily be found by selecting the characters which cross the line within a certain tolerance. Note that the tolerance had to be high enough that the centroids of "aAa" all lie on the same line. To detect multiple lines, the line with the highest maximum is chosen first and all characters lying on it are removed from the remaining lines. The remaining lines are processed in decreasing order.

This approach works well if lines contain a high number of characters and all characters within a line are very close to each other. Unfortunately, this is not the case in the domain of UIC numbers, where the first two lines only contain two digits. Although the first and second line also contain a textual representation of the digits, the distance between both can be too high to classify them as neighbors (See figure 1.3).

## 2.3 Optical Character Recognition

To reduce the amount of work necessary to develop a UIC detection system, it was decided to use the open source framework Tesseract OCR for the optical character recognition stage of the pipeline. Tesseract was originally developed by HP between 1984 and 1994 as closed source. HP released the source code of Tesseract in 2005 and development continued in 2006 by Google [Smi07].
Tesseract is optimized to detect text from documents and its accuracy can be improved significantly by preprocessing input images. As a system for scanning documents, it supports up to 100 languages and can be trained to detect languages in custom domains. More importantly for UIC detection, it can be configured to recognize single characters and to limit its predictions to digits.
The library provides a command line interface and C++ library. Additionally, an API to interact with Tesseract is provided in the OpenCV library.
Tesseract OCR was used as a black box during this work and the main focus was laid on the localization and preprocessing of text and especially UIC numbers. More information about Tesseract can be found at [urlf].

# Chapter 3

# System

This chapter describes the methods used for the implemented UIC detector. Thereby, the inner workings of the used algorithms and its parameters are explained. It starts by describing how the environment required by the project has to be set up in section 3.1. Then, an overview of the pipeline and the building blocks of the program is given in section 3.2. Section 3.4 discusses different methods to measure the performance of pipeline. Section 3.4 explains the implementation details of the two character detectors. The next section (3.5) discusses the used line detectors and section 3.6 talks about the integration of TesseractOCR as a text detector. Subsequently, section 3.7 deals with the grouping of lines into UIC numbers. Finally, section 3.8 discusses additional measures to enhance the final result.

## 3.1 Setup

The program was developed in C++ using OpenCV and Tesseract. The operating system used was Ubuntu 16.04 LTS. This section provides instructions on how so set up the development environment on an Ubuntu system to run the created UIC detector program.

Although the current OpenCV library does not provide any tools for text detection, an extra module for this purpose can be found in the OpenCV_contrib repository. This repository is intended for the development of new features and modules that are not stable or popular enough to be integrated in the central OpenCV repository yet. The standard OpenCV library can easily be installed on Ubuntu using its APT package manager, but the standard package does not include the extra modules. To install a version with the text module, the source code needs to be compiled manually.
The text module not only provides algorithms that can be used for text detection in complex scenes, but also includes a wrapper around the TesseractOCR library. To use this functionality in OpenCV, the TesseractOCR library needs to be installed during the configuration and compilation of OpenCV.

### 3.1.1 Installation of TesseractOCR

Again, TesseractOCR can be installed with the package manager or compiled from source. Since no modifications to the base library are required, the installation with the package manager is sufficient. Note that Leptonica is required to convert images into TesseractOCR's internal image format within C++. Additionally, a command line interface for testing and training data generation of TesseractOCR can be installed with the package *tesseract-ocr*. All necessary packages can be installed with the following

command shown in listing 3.1. *liblept5* and *libtesseract3* include the binaries for Leptonica and Tesseract, whereas *libleptonica-dev* and *libtesseract-dev* include the header files.

Listing 3.1: TesseractOCR installation

```
1 sudo apt-get install liblept5 libleptonica-dev libtesseract3
    libtesseract-dev tesseract-ocr
```

### 3.1.2 Installation of OpenCV

The necessary instructions to compile OpenCV from source can be found in listing 3.2. During this process multiple other packages are required (Line 1-4). OpenCV and the contribution library are hosted on Github. The version control system *git* can be used to get a local copy of the source code (Line 5-6). To use the same version of OpenCV that was used during development, checkout *v3.2.0* in both repositories (line 7-8). Before compilation, a *Makefile* has to be created. This is done by using *CMake*. *CMake* checks the current system configuration and the provided options and creates a build file for the used system. To tell *CMake* to compile the contribution library as well, the location of the repository must be passed to it as a parameter (Line 10). Once the configuration is complete, it can be compiled using *make* (Line 11). Finally the library is installed into the system at line 11.

Listing 3.2: OpenCV compilation

```
1  # packages for compilation
2  sudo apt-get install build-essential
3  # required packages
4  sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec
       -dev libavformat-dev libswscale-dev
5  git clone https://github.com/Itseez/opencv.git
6  git clone https://github.com/Itseez/opencv_contrib.git
7  cd opencv_contrib; git chekcout 3.2.0
8  cd ../opencv; git chekcout 3.2.0
9  mkdir build; cd build
10 cmake -D OPENCV_EXTRA_MODULES_PATH="../../opencv_contrib/modules/"
        BUID_DOCS BUILD_EXAMLES CMAKE_BUILD_TYPE=RELEASE -D
       CMAKE_INSTALL_PREFIX=/usr/local ..
11 make -j
12 sudo make install
```

## 3.2 Pipeline

To make the created code reusable, a pipeline was designed that allows swapping out components of the UIC detector. This can be used to quickly implement and test different algorithms for different stages. Additionally, the created components can be used for different applications by keeping the text detection stages and replacing only the UIC stage. For this purpose, the following architecture was used.
Every stage in the pipeline is an “ImageProcess” that takes an input image and returns a result of a defined type. For example, the UIC detector takes an input image and returns possible UIC numbers and their location. Additionally, several helper classes were created that allow the processing of the image in individual color channels or scales. The class hierarchy of the pipeline can be seen in figure 3.1. The purpose of each task is as follows:

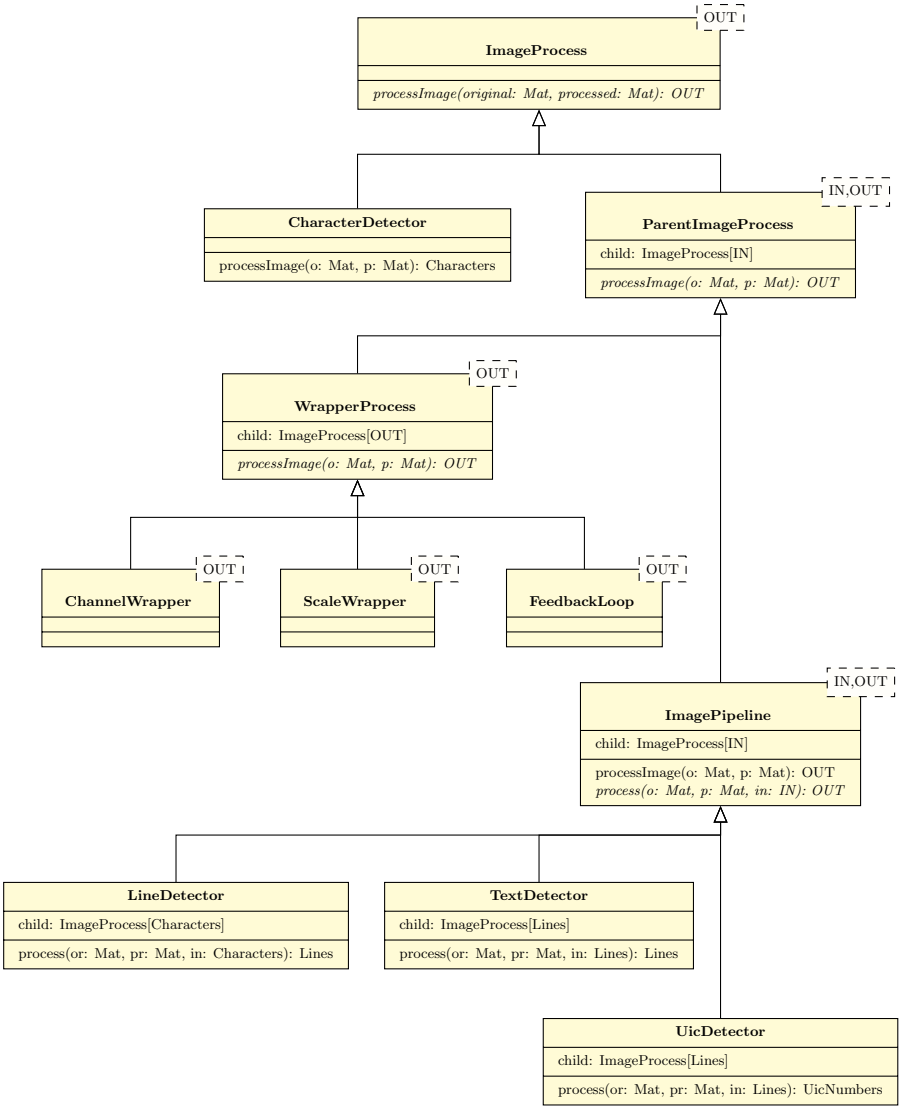



Figure 3.1: The key classes used for the UIC detector pipeline

- **Image Process**
  It receives an input image that is processed to create a result of a defined type. In addition to the input image that could be preprocessed by another task, the original image is also passed through. This allows subsequent stages to access the original information about the image and share preprocessing steps. As an example, a character detector can process only the red channel of an image to find characters, but use the original image to calculate the mean color of each character.

- **Parent Image Process**
  This is one of the main building blocks used to create complex pipelines. It uses a child ImageProcess with its own output type. This allows the parent process to use the child to compute an intermediate result, which can then be processed to produce its own result.

- **Wrapper Image Process**
  This is a parent image process with the same output type as its child process. This can be used to preprocess an input image or to filter/enhance the output of the child process.

- **Channel Wrapper**
  This wrapper process splits the input image into several channels and passes them individually to a child process. The results of the child process are then combined and returned as its own result. This can be done for any output type. For example, to extract characters from each channel individually and then pass all of them to a line detector. Thereby, the line detector can form lines out of characters that were detected in different color channels.

- **Scale Wrapper**
  It creates an image pyramid out of the input image and passes each scaled version of the image to a child process. In addition to scaling the image, it also translates the results of the child process back to the original size of the image.

- **Image Pipeline**
  This class uses a child of any type and applies it directly to the input image. The result is then processed and converted into its own output type. This way the component only has to focus on its own task. For example, a line detector groups the characters extracted from its child into lines, without any knowledge of the method used by the character detector. As a result, different stages of the pipeline can easily be switched out.

- **Character Detector**
  A character detector is an image process that returns a list of character candidates. Two version of this stage were implemented. The first one uses Extremal Regions by Neumann and Matas described in section 2.1.3 [NM12]. The second one, uses an adopted version of the FASText algorithm by Busta et al. which is described in more detail in section 3.4.2 [BNM15].

- **Line Detector**
  This process uses the extracted characters from a character process and groups them into lines. Again, two algorithms were used. The first one was developed by Neumann & Matas and implemented in OpenCV [NM11b]. More information about it can be found in section 3.5.2. The second one is a custom implementation for the domain of UIC number detection and is described in more detail in section 3.5.3.

- **Line Process**
  This is an image process that takes an image and returns lines of characters. Note that this can be a line detector, a wrapped line detector or even a process that extracts characters and lines in one step. In particular, the full OpenCV implementation of the method by Neumann and Matas was implemented as a pure line process that uses Extremal Regions. It can not be configured to use a different line detector [NM11b, NM12].

- **Text Detector**
  This image pipeline uses the lines extracted by its child and assigns each character a machine readable representation using an OCR system. For this stage, TesseractOCR and a Support Vector Machine were used. More details about this stage can be found in section 3.6.

- **UIC Process**
  A UIC process creates a list of potential UIC numbers. This includes the bounding box, the individual lines and the number that has been extracted. The results are ordered by a score and may include multiple versions of the same number.

- **UIC Detector**
  The UIC detector uses the lines from its child process and groups them into UIC numbers. More information about this process can be found in section 3.7.

- **Feedback Loop**
  Uses the location of the best UIC result from one process and uses a second process to find a better version of the number at this location. This allows the use of one fast process to find the location detection and a second slower process, with a higher recall, to extract the UIC number.

## 3.3 Evaluation methods

To select appropriate parameter configurations, all algorithms were tested with the UIC dataset. To reduce the runtime and to prevent overfitting, only a subset with a maximum of 50% of the dataset was used.
For the comparison of two different algorithms or configurations, each of the methods must be evaluated. These evaluation methods are limited by available dataset.
To only evaluate the final UIC detector, the true UIC number would be sufficient. The evaluation could be improved by also using its location. To evaluate a line detector, the bounding box of the lines must be known. For the evaluation of the character detector, it is necessary to know the location of each character. In an ideal scenario, the set of pixels part of a character is known to evaluate the quality of the extracted regions. Due to the high effort required to annotate the dataset in such detail, only the following evaluation methods were used:

1. **Correct UICs**
   The ratio of the correctly read UIC numbers, is the most straight forward evaluation method. Its disadvantages are the lack of detail this metric provides and that it can only be applied to the whole pipeline at once.
   A method that correctly reads 11 digits of each UIC number and misses one would score 0%, and would therefore seem inferior to a system that correctly reads one UIC number but cannot find a number in all other cases.

2. **Edit distance**
   For a more detailed result, the Levenshtein edit distance for each UIC number

can be calculated [Lev66]. Again this method can only be used to evaluate the whole UIC pipeline but cannot be used to compare sub segments like a character detector. Additionally, two methods can not be compared directly. It is not defined if a method with an edit distance of 0 in 50% of the cases and 8 in the remaining 50% is better than a system that has a edit distance of 4 in a 100% of cases.

3. **Character accuracy** The original dataset was augmented by adding bounding boxes of digit groups. This was less time intensive than marking every character but can still be used to estimate the bounding box of all digits in the UIC number. This is not an accurate representation of the character bounding boxes and it does not include characters not part of the UIC number. It therefore cannot be used to calculate the accuracy of the character detector directly, but it can be used to compare two methods.
   The evaluation is done by comparing the bounding box of the extracted characters to the annotated bounding boxes. The set of estimated and actual bounding boxes can then be used to calculate the precision and recall of the algorithm. Thereby the evaluation method from the ICDAR 2003 competition is used (Equation 1.5). This method had the disadvantage that it penalized merged or split word bounding boxes. This led to the replacement of the evaluation method in later competitions. However, this problem does not affect its application to single characters. As a result, the simpler method from the ICDAR 2003 competition was used.
   Note that although it is theoretically possible to achieve a recall of 100%, the precision is limited because the dataset does not actually include all characters in an image. As a result, a character detector is very likely to also find characters that are not part of the UIC. To reduce this effect and to reduce the runtime of each test, the input image was limited to the UIC bounding box. Thereby, a big portion of the additional text that can be found on train wagons was removed from the process. Note that this does not include all non annotated text. It does still include additional information, such as the owner written in the second line of the UIC number.
   Another disadvantage is that this method does not penalize repeated hits. If a character is detected multiple times, it does not have a negative effect on the precision. Because of the later filtering stages, repeated hits are not considered a primary issue. Nevertheless, it has a direct impact on the runtime of the UIC detector because it increases the number of characters that have to be processed by later stages.

   This method can be used directly to evaluate a character detector. In addition to that, it is possible to use the same approach for a line and UIC detector. Thereby, the extracted lines/UICs are broken into their individual characters and can then be compared to the ground truth. Although this method does not consider the grouping of the characters, it can still show how many of the relevant characters are detected and if the found characters are relevant. This approach considers the line and UIC detector as an additional filter that improves the precision of the initial character detector. As a result, an additional stage can only improve the precision and should ideally not decrease the recall.

4. **OCR quality**
   One of the problems of the previous approach is that it only uses the bounding box of the characters for the evaluation. Even if a character detector finds the correct bounding boxes, it does not mean that the character can be identified correctly. Of course this strongly depends on the character detector being used, but the OCR detector depends on the provided input images. Because the used character detectors provide a preprocessed image to the OCR system, it also makes sense to

evaluate the quality of the provided input image. If the character segmentation would be known on a pixel level, this comparison would be trivial and could be used directly by using the method from ICDAR 2013 (See section 1.2.2). Unfortunately, this information is not available and a less ideal approach had to be used.

The proposed method assumes that the computed segmentations are of higher quality if they are easier to read. Furthermore, it is assumed that it is easier for an OCR system to label a character correctly if the input image is of higher quality. Of course this assumption is flawed, because it only means that the input images are more similar to the dataset used to train the system. However, because the used TesseractOCR was trained on a high number of different fonts and we are only interested in comparing one character detector to another, it can be used as a sufficient estimate. Again, note that this metric therefore rewards input images that can be processed well by TesseractOCR but not necessarily by other systems.

The final OCR quality metric is computed by comparing the prediction of the OCR system to the ground truth for every detected character. A detected character is defined as a character that has at least a 50% overlap with the bounding box of a ground truth character. The quality of the segmentations is then calculated by dividing the number of correctly classified characters by the number of detected characters.

Unless stated otherwise, all experiments are performed on gray scale images.

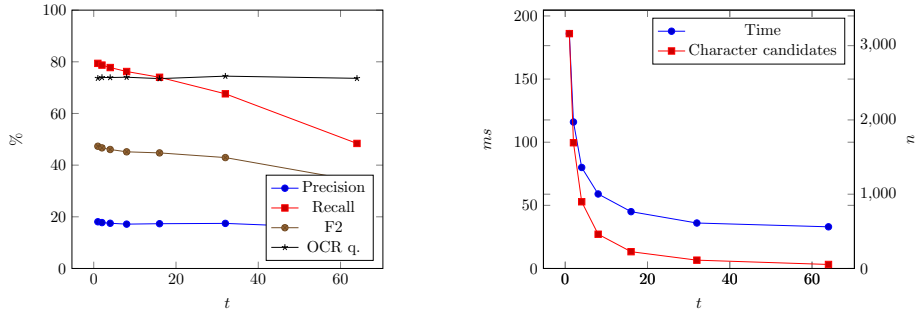
## 3.4 Character Detectors

A character detector returns a list of characters that can be used for further processing. Each character must have a bounding box and a way to draw its outline. Additionally, a score that can be used to rank them from likely to unlikely candidates is required. This value is necessary in case multiple overlapping characters could be part of the same line. Finally, each character must have information about its color that can be used to group characters with similar color.

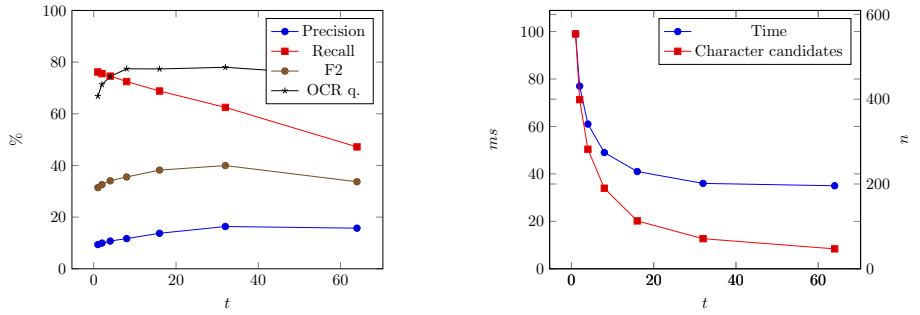
To increase the performance, each character detector could be configured to expect a certain number of characters in the input image. This information could be used to stop the search if the bounds of the individual characters are too big or if the image resolution is too small to extract the expected number of characters. For example, if the detector is configured to expect 5 lines with 2 characters each, it can be safely assumed that a single character will not be bigger than $\frac{1}{5*2}$ of the input image. Furthermore, if it is assumed that a character requires at least a size of $3 \times 5$ pixels, it is not possible to find any more characters if the size of the input image is below $(3*5)*(5*2) = 15 \times 10$. Because UIC numbers always have 3 lines with a maximum of 8 characters, the number of expected lines and characters was set to 8x3.
For the evaluation of character detectors, the recall and precision of the character accuracy and the OCR quality from section 3.3 were used.

### 3.4.1 Extremal Region Detector

The extremal region detector used in this project was developed by Neumann & Matas and is implemented in the OpenCV contribution library [NM12]. To find out more about the extraction of extremal regions see section 2.1.3.
The algorithm was not implemented manually. The implementation in OpenCV could be used directly and only a conversion to the character format used for the rest of the




(a) Without non max suppression




(b) With non max suppression ($p_d = 0$)

Figure 3.2: Effect of the threshold $t$ on the ER character detector performance. ($p_1 = 0$, $p_2 = 0$, gray scale)

pipeline was necessary.
The extraction of Extremal Regions was split into two functions in OpenCV. The first one extracted them from the input image and roughly filters them by features calculated in $\mathcal{O}(1)$ time. This process can be configured with the following parameters:

- **Classifier Model**
  This allows the use of a manually trained classifier model for the calculation of the character probability. Although a manually trained classifier could improve the performance, it would require the annotation of the digits in the dataset on a pixel basis. Furthermore, the features used with the classifier are very primitive and are unlikely to be significantly different for a different font. Additionally, the font for UIC numbers is not standardized and the used UIC dataset is too small. However, the OpenCV model was trained on multiple fonts making it usable for the UIC detector.

- **Threshold delta** $t$
  This is the step size of the thresholding operation. The smaller this value is, the more versions of each character are considered. If this value is too big, characters with a low contrast can stay undetected. On the contrary, the smaller the value is, the more false positives and duplicates are detected. Furthermore, it increases its own runtime and can increase the runtime of later stages if the number of extracted characters is too high. This value can be configured from outside of the character detector.
  Figure 3.2a shows the effect of different thresholding values on the ER detector. $p_1$ and $p_2$ were set to 0 to remove their influence on the experiment. As a result, all detected regions are returned unfiltered.
  Although a higher thresholding value does not have a strong influence on the precision and OCR quality, it does have a significant impact on the recall, the

runtime and the number of returned characters.
By increasing $t$, the chance of recognizing a character with a low contrast decreases. For example, if a character has an intensity of 18 and the background has an intensity of 10, it will only be detected certainly with a $t <= 8$. If $t > 8$, the chances of detecting the character reduce to $|I_b - I_c|/t$, where $I_b$ is the intensity of the background and $I_c$ is the intensity of the character.
The processing time increase is due to the higher number of thresholding operations that have to be performed on the input image. Additionally, it increases the number of Extremal Regions that have to be checked as characters. Because non max suppression was not used in this experiment, all extracted Extremal Regions are returned which can result in a very high number of characters that are detected multiple times. Note that this is not reflected in the precision metric. A repeated detection of a character is not penalized. Therefore, one single character that is detected many times in an image can result in a high precision value even though no other character was detected. Especially if the number of extracted character candidates is very high the precision should be considered with caution.
This initial recall cannot be improved by later stages but they can improve the precision and OCR quality. Because it is necessary to detect all 12 digits to correctly identify an UIC number, it is important to achieve a high recall. Although the recall values for $t = 1$ (79.40%) and $t = 2$ (78.72%) don't seem to differ a lot, it reduces the chance of detecting all 12 digits from 6.28% to 5.66%. However, the final recognition rate tends to be higher because the quality of the digits in an UIC number tends to be similar. As a result, the detector tends to either detect most digits or none.
To optimize a system towards recall, a threshold of $t = 1$ is recommended. Due to the slow speed increase for values of $t > 16$, it is recommended to use $t = 16$ for a speed optimized system. As a compromise between both goals use $t = 4$.

- **Minimal character area**
  The minimal area of an Extremal Region had to be passed as a percentage of the area of the input image. This was calculated by dividing the area of the smallest possible character (3x5px) by the area of the input image.

- **Maximal character area**
  Again, this parameter had to be passed relative to the size of the input image. Hereby, the expected number of lines and characters per line could be used. The maximal area can be calculated as a ratio with $\frac{1}{e_c \ e_l}$ where $e_c$ is the number of expected characters and $e_l$ is the number of expected lines.

- **Minimal probability** $p_1$
  Extremal Regions with a lower probability than $p_1$ are rejected. Note that this is not the final filtering stage. If this value is too high, it can lead to characters not being detected by the UIC detector at all. On the contrary, if this probability is too low, more complex features are calculated for each Extremal Region, which are then used to calculate a more accurate character probability. As a result, this value can be used to choose either a higher recall or speed. It can be configured externally.

  The influence of $p_1$ can be seen in figure 3.3. Again, $p_2$ was set to 0 to disable the last filtering stage. Additionally, $t$ was set to 1 and non max suppression was disabled. The chart shows that the change of $p_1$ only has a very small impact on the recall but increasing $p_1$ does improve the precision, OCR quality and processing time. Note that the decrease of processing time is because of the reduced number of extremal regions that are passed to the second filtering stage. Although this stage is disabled by setting $p_2 = 0$, the algorithm still calculates the features required

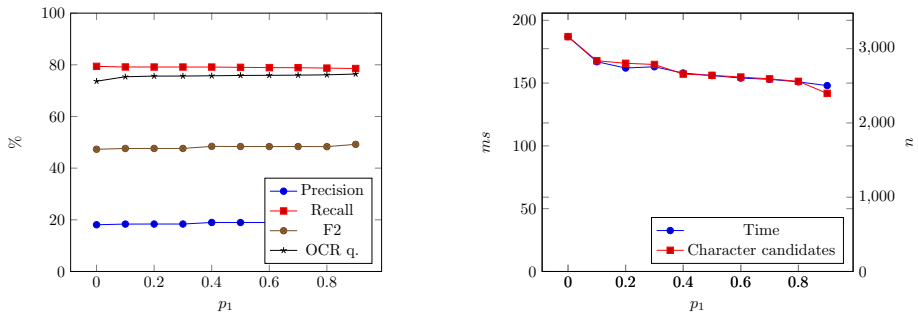



Figure 3.3: Effect of the minimal probability $p_1$ on the ER character detector performance. ($t = 1$, $p_2 = 0$, gray scale)

for the classifier. Due to the minor decrease of the recall and an improvement of all other metrics, it is recommended to set $p_1 = 0.9$ for all optimization goals.

- **Non Max Suppression**
  This option controls whether or not characters without a local maximal probability are kept. If enabled, it can increase the precision of the character detector, but can negatively effect the recall.
  Figure 3.2b shows the impact of the non max suppression on the response of different threshold values. The most obvious difference to figure 3.2a is the vastly reduced number of characters being extracted. This is because only characters whose character probability is a local maximum over multiple thresholding values are selected. Because of this much stricter filtering criteria, the recall drops with $t = 1$ from 79.40% to 76.18%.
  Due to a similar recall value and a strongly reduced number of extracted characters, one would expect the precision to be improved, on the contrary, it became worse as shown in the chart. This is because non max suppression strongly reduces duplicate characters by rejecting characters that were extracted from consecutive thresholding operations. As a result, false positives, that were extracted over a lower number of consecutive thresholding operations, take up a bigger proportion of the extracted characters. If $t$ is increased further, it reduces the number of false positives and the precision increases.
  By reducing the number of extracted characters, the speed of the algorithm is also improved. This is due to the lower number of Extremal Regions that have to be filtered in the later stages.
  Due to the lower recall, it is not recommended to use non max suppression for a system that should achieve a high recall. If the processing speed matters, non max suppression can improve the speed and reduce the number of extracted regions which in turn improves the processing speed of later stages.

- **Minimal Probability Difference** $p_d$
  This parameter controls the minimal probability difference required between a local maxima and minima to keep a character. It is only used if non max suppression is enabled. This controls how many different version of a Extremal Region are passed on to the next stage.
  It can be used to reduce the number of extracted characters, which in turn can improve the processing speed. The disadvantage of a higher $p_d$ is a lower recall.
  It can be seen in figure 3.4 that the parameter does not have a big impact on the character detector, but it does decrease the recall slightly. Therefore, it is not recommended to use this feature in the domain of UIC detection ($p_d = 0$).

The second function calculates more complex features for the remaining character

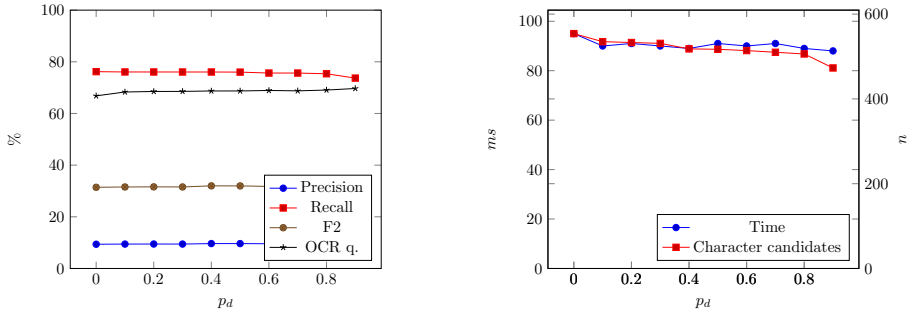



Figure 3.4: Effect of the min probability difference $p_d$ on the ER character detector performance. ($t = 0$, $p_1 = 0$, $p_2 = 0$, gray scale)

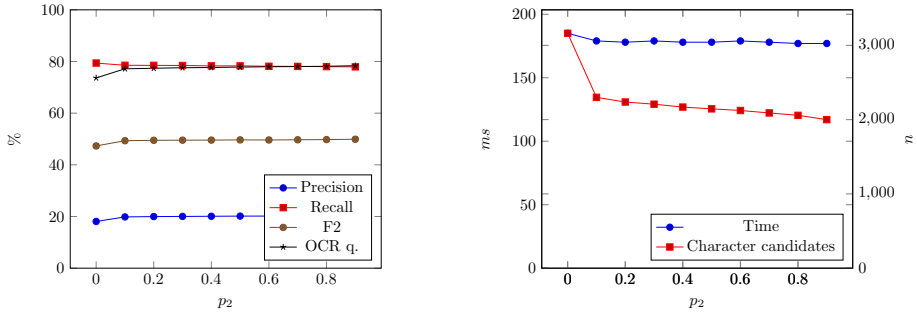



Figure 3.5: Effect of the minimal probability $p_2$ on the ER character detector performance. ($t = 1$, $p_1 = 0$, gray scale)

candidates and filters them further. The following parameters are available:

- **Classifier Model**
  A classifier model needs to be passed to assign each character a probability. Although, the features are more complex in this stage and therefore more distinctive, it was still decided to use the provided model instead of a custom trained model.

- **Minimal probability** $p_2$
  A lower minimal probability increases the number of returned characters by the character detector but also decreases the precision. This parameter should be used to control how many characters are passed to the next stage. To increase the recall a lower probability is desirable, but the following stages like the line detector must be capable of handling a high number of inputs. Additionally, this probability is passed on to the later stages to make more informed decisions.

  The effect of $p_2$ in combination with $p_1 = 0$ and $t = 1$ can be seen in figure 3.5. It shows that the increase of $p_2$ from 0 to 0.9 causes a decrease of the precision from 79.40% to 77.92%. Additionally, it shows that the processing time only decreases slightly if the filter threshold is increased. This is in contrary to the number of extracted character candidates that drops significantly for $p_2 \geq 0.1$. Because the extracted characters only have to be converted into a "CharacterCandidate" after this filtering stage, a tighter threshold only slightly improves the processing time. Although it does not have a strong impact on the speed of the character detector, the reduced number of character candidates does improve the speed of the remaining pipeline.
  Although the active second stage filter does decrease the recall, it is recommended to use at least a value of $p_2 \geq 0.1$ to speed up the following stages. For a system

| **Goal** | $t$ | $p_1$ | $p_2$ |
|---|---|---|---|
| Speed | 1 | 0.9 | 0.9 |
| Balance | 4 | 0.9 | 0.9 |
| Recall | 16 | 0.9 | 0.1 |

(a) Parameters

| **Goal** | **Precision** | **Recall** | **F2** | **Time** | **n** |
|---|---|---|---|---|---|
| Speed | 20.23% | 69.29% | 46.66% | 46 | 136 |
| Balance | 20.33% | 75.74% | 49.02% | 74 | 538 |
| Recall | 21.33% | 77.54% | 50.78% | 151 | 1832 |

(b) Performance (applied to the intensity channel only)

Table 3.1: The final three Extremal Region character detectors

that also tries to optimize speed, $p_2 = 0.9$ is recommended. Thereby, the recall is only reduced minimally but the number of extracted characters decreases further.

Finally, the remaining characters are converted into character candidates. These are then used for the next stages of the pipeline. For the creation of the character outline required for the OCR stage, a flood fill algorithm is used. The threshold at which the Extremal Region was extracted was used as a seed point from within the character outline. To save memory, this image is only generated if it is required in a later stage. This prevents the unnecessary effort of creating the outline for characters that are filtered by a later stage, such as characters candidates that are not grouped into a line.

Table 3.1 shows the parameters and results for three differently optimized systems. It can be seen that although the Recall System does have the highest recall with 77.54%, it is also significantly slower. It requires double the processing time and returns two times as many characters than the "Balanced System". Making things worse, the higher number of character candidates passed on to the next stage slows down the following steps even more. Depending on the line detector used, it can lead to optimal solutions not being found due to an oversized search space. Note that these results were calculated with the intensity channel and its inverse only.

#### 3.4.1.1 Color

Applying the algorithm to a color image requires the same operation to be performed on each individual color channel. This leads to a significant increase in processing time and extracted character candidates. Additionally, it increases the redundancy of the detected characters due to the visibility of some characters in multiple color channels. Neumann and Matas recommended the use of the HSI color space in combination with the gradient channel [NM12]. For convenience, the HSL space was used due to its availability in OpenCV. Due to the more specific text styles and colors used in UIC numbers, the experiments were repeated with the red $R$, green $G$, blue $B$, hue $H$, saturation $S$, luminance $L$ and gradient $D$ channel. For this test, the balanced parameters and 40% of the dataset were used. The results can be seen in table 3.2.
They show that the use of the RGB channels seems to be more applicable for the domain of UIC detection. Even though the gradient channel does perform weak, in combination with the RGB channels, it performs better than the RGB and the luminance channel. This suggests that the characters extracted from the gradient channel have a smaller overlap with the RGB channels and therefore result in more detected regions.
Although the combination of all channels detects the most characters, the RGB and gradient channel combined have a significantly lower runtime. Therefore, the RGB and

| Channels | Precision | Recall | F2 | Ocr q. | Time | n |
|---|---|---|---|---|---|---|
| R | 18.07% | 68.78% | 44.06% | 72.68% | 164 | 2001 |
| B | 21.11% | 78.29% | 50.78% | 76.54% | 170 | 2051 |
| G | 20.85% | 77.30% | 50.14% | 76.49% | 163 | 2002 |
| H | 7.81% | 37.24% | 21.24% | 31.55% | 63 | 577 |
| S | 9.81% | 50.60% | 27.63% | 63.84% | 129 | 1341 |
| L | 20.56% | 78.03% | 50.06% | 76.80% | 162 | 1936 |
| D | 11.92% | 52.21% | 31.16% | 58.83% | 159 | 1149 |
| RGB | 20.09% | 80.51% | 50.27% | 76.02% | 488 | 6055 |
| HSL | 15.94% | 79.46% | 44.22% | 73.71% | 344 | 3855 |
| D+RGB | 18.88% | 81.22% | 48.92% | 75.16% | 659 | 7205 |
| D+HSL | 15.09% | 80.19% | 43.05% | 72.23% | 517 | 5005 |
| D+HSL+RGB | 17.93% | 82.07% | 47.84% | 74.65% | 1005 | 11060 |
| RGB+L | 20.21% | 80.81% | 50.51% | 76.22% | 656 | 7992 |

Table 3.2: Combination of multiple color channels for the ER detector using the balanced parameters (G...Gray, I...Intensity, D...Gradient)

D channel are used for the final pipelines.

Applying the same algorithm to multiple channels can be done in two different ways:

1. **Direct combination**
   Hereby all extracted characters are merged into one list and then passed to the next stage. The advantages of this method are that all characters are available to the line detector and it can therefore make more informed decisions. Some characters might have a different color than their neighbors, but they can still be grouped into the same line. The disadvantage is the high number of characters that have to be processed by later stages at once.

2. **Seperate processing**
   The characters are kept separately and are combined at a later stage. For example, all characters from the red channel are passed to the line detector. The line detector groups them into lines and combines them with the lines found in the blue channel. Alternatively, the lines are still kept separate and only the resulting UIC numbers are combined. The advantage of this approach is the significantly reduced number of characters that have to be processed by the later stages at once. Additionally, text lines tend to have a consistent color and it therefore makes sense to only group characters which were detected in the same color channel.
   On the contrary, uneven illumination and shadows can alter a characters color. Furthermore, the background might be inconsistent, which can result in a text line being detected in the hue color channel and the last character being detected only in luminance channel. Therefore, the line will not include the last character even though it was detected in the previous stage.
   A further difficulty is that some UIC numbers use one color for the digits and a second color for the text in the first two lines. Additionally, the background color is not always consistent in case where the number was corrected at a later time or the metal became rusty (Figure 3.6).

### 3.4.2 FASText Detector

This algorithm developed by Busta, Neumann and Matas uses Stroke End and Bend Keypoints (SEK /SBP) for the extraction of character candidates [BNM15]. To find more information about the theory behind this algorithm see section 2.1.4.


Figure 3.6: UIC numbers with inconsistent background and text color

Although the source code for this project is publicly available, an improved version was created from scratch. Due to the lack of documentation, it would have required a lot of time to understand the code well enough to integrate it into the UIC pipeline. Additionally, rewriting the code allowed adding support for color images. Although all code was rewritten, some of the performance optimizations from the original code were adapted and extended with comments.
Furthermore, it has to be noted that, although the paper does not discuss the application of the algorithm to color images, the available code does include some incomplete extensions to process color images.

The algorithm was split into two sub problems. The first one deals with the extraction of Stroke End and Stroke Bend Keypoints (SEKs and SBKs). The second one uses these keypoints to extract the outlines of characters and filters unlikely candidates. The remaining regions are then converted into an appropriate format and passed to the rest of the UIC pipeline. Due to the characteristics of the algorithm, it had to be used with a scale pyramid. The scaling factor (1.6), was taken from the original paper although experiments with different scaling factors were conducted in section 3.4.3. Both of these stages were implemented with the support for single channel and color images. The application to single channel images is the same as described by Busta et al. (Section 2.1.4) and the application to color images is explained in section 3.4.2.4 [BNM15].

#### 3.4.2.1 Keypoint detector

The keypoint detector takes an input image and returns a list of Stroke End Keypoints and Stroke Bend Keypoints. Thereby, two parameters can be used to control the result. The **threshold** describes the maximal intensity difference a pixel is allowed to have to be considered as part of the same line. The second parameter is a flag that can be used to enable **non max suppression**. The remaining section explains the methods used for this implementation in detail.

The keypoint detector must scan every pixel. For this, the 5x5 neigbourhood, in particular its surrounding circle with a circumference of 12 pixels (See figure 2.2), had to be searched for segments with similar and different intensity. A pixel has a similar intensity if its intensity difference to the center pixel is smaller than the threshold.
Busta, Neumann and Matas suggest the use of a static threshold value that is passed to the keypoint detector. Choosing a high threshold leads to the detection of characters with a strong contrast but fails to detect low contrast characters. The threshold of 64 will detect characters with a stroke value of $I_s = 74$ and background value of $I_b = 4$, but fail to detect a character with $I_s = 32$ and $I_b = 4$. On the contrary, a low threshold like 4 will result in low contrast characters being detected, but can lead to a character with $I_s = 74$ and $I_b = 0$ to not being detected if the intensity of the center pixel is 68. To combat this issue, adaptive thresholding is introduced. Thereby, the average intensity

difference to the center pixel is calculated and then multiplied by the constant $a$:

$$t_a = \max(t, \frac{\sum_{n=1}^{12} |I_c - I_n|}{12 * a}) \tag{3.1}$$

$I_c$ is the intensity of the center pixel and $I_n$ the intensity of the 12 pixels on the circumference. Although this method is able to detect a wider range of characters, it does increase the runtime due to the extra step before the key point check. For an experimental comparison with the static threshold see section 3.4.2.3.

To speed up the process, the "is similar to the center" property is preloaded for each character in the circumference. At the same time, the keypoint is rejected if two opposite pixels are similar (Keypoint is on a straight line or it is no line at all) or some of the different pixels are brighter while some are darker (The stroke is expected to be only surrounded by either brighter or darker pixels). Finally, the number of segments needs to be counted. Thereby, a flag for the last state is used and whenever the state of a pixel changes from similar to different or the other way around, a "change" counter is increased. Furthermore, similar pixels are checked whether they are connected to the center and the size of a segment with similar pixels is limited to $\leq 3$. If it has more than 3 pixels, the stroke is too thick to be considered a line.
If the final "change" count is greater than 0 and smaller than 5, a keypoint is detected. Note that although the number of segments can only be an even number, the counter might only have detected 1 change if the order of similar $s$ and different $d$ pixels was *ssssssdddddd*. If the number of changes is 1 or 2, it is a Stroke End Keypoint and if the number of changes is 3 or 4, it is a Stroke Bend Keypoint.

To reduce the number of detected keypoints, Busta et al. suggested the use of non max suppression on a 3x3 neighborhood. To implement this feature, an array representing the last two lines was used to keep track of the already detected neighboring keypoints. The index of every detected keypoint was entered into the bottom line. If the entry to the left or one of the three entries above contained a keypoint, only the one with the highest contrast was kept. The contrast of a keypoint is defined as the intensity difference of the center pixel and the pixel with the most similar intensity value outside of the threshold.
Furthermore, Busta et al. suggested the use of a grid to limit the number of keypoints in a certain neighborhood. This was not implemented due to time constraints and the fact that it decreases the number of characters and therefore only improves the precision, but decreases the recall.

#### 3.4.2.2 Segmentation

In the next step, the extracted keypoints are used to generate character candidates. This is done by applying a flood fill algorithm to the image, with the values of the keypoint as a threshold. The method for the calculation of the threshold can be seen in equation 2.9. Unfortunately, the paper does not provide a lot of additional information about the thresholding step. The authors suggest applying a standard flood fill algorithm to every keypoint. Considering that one pixel can have multiple keypoints, the segmentation can result in multiple regions of the same character. One way to solve this is to ignore keypoints that are part of a previously extracted region. Due to the different neigborhoods of the source keypoints, the segmentation might be different for keypoints that are part of the same line. One keypoint might be on a section of a character with a high contrast and the segmentation therefore results only in a part of a character. If on the other hand, a keypoint is on a stroke section with low contrast, the segmented region might grow too big. An additional problem is keypoints that are part of lines that morph into uniform surfaces. This results in the flood fill algorithm having to segment

Figure 3.7: Digit with the largest possible area (15 × 30px, Stroke width = 3$px$)

a high number of pixels and wasting processing time on unimportant regions.
The final algorithm is a flood fill algorithm that stops prematurely if the width, height, or the area of a region exceed a defined limit. Given that the stroke width of a character has to be in between one and three pixels, the maximum width of a character was set to 15px and the maximum height was set to 30px. These values can be reduced even further if the minimum number of characters and lines in the image is known. As for UIC numbers, the minimum number of characters in a line is 8 and the number of lines is 3. If the width of the image is only 80px wide, it can be assumed that one character cannot be wider than 80px/8 = 10px. The maximal width $w_{\max}$, height $h_{\max}$ and area $a_{\max}$can therefore be computed as follows ($I_w \ldots$ image width, $I_h \ldots$ image height):

$$w_{\max} = \min(15, \frac{I_w}{8}) \tag{3.2}$$

$$h_{\max} = \min(30, \frac{I_h}{3}) \tag{3.3}$$

$$a_{\max} = \frac{w_{\max} \times h_{\max}}{2} \tag{3.4}$$

By using an eight as the digit with the longest possible stroke length and assuming the maximum stroke width of 3px (See figure 3.7), the maximal area can be computed: ($s_v \ldots$ stroke width vertical, $s_h \ldots$ stroke width horizontal)

$$s_v = \min(3, \text{ceil}(\frac{w_{\max}}{3})) \tag{3.5}$$

$$s_h = \min(3, \text{ceil}(\frac{h_{\max}}{5})) \tag{3.6}$$

$$a_{\max} = s_v \times h_{\max} \times 2 + s_h \times (w_{\max} - 2 \times s_v) \times 3 \tag{3.7}$$

Note that these performance improvements only have a small impact on UIC only images, but can significantly improve the performance for larger images.
Using every keypoint for this segmentation step results in a high number of false positive character detections. Therefore, Busta et al. suggest a further filtering stage. They use their previously developed algorithm from [NM12] to calculate features for each character candidate during the segmentation process with an $\mathcal{O}(n)$ time complexity. This is the same method that was used for the Extremal Region Detector and can be found in section 2.1.3.
The additional feature, using the Character Stroke Area proposed by Busta et al. (Section 2.1.3), was not implemented for the same reasons as stated for the grid filtering mechanism. Due to the removed CSA feature, the features calculated for a FASText region are identical to the features used for the ER character detector. This allows the reuse of its classifier and circumvents collecting a dataset and training a UIC specific classifier. Finally, all accepted characters were passed to the second stage character classifier to compute a more accurate character probability. Although the characters are not filtered by this probability, it can be used in later stages if the best character candidate out of several at the same position has to be picked.

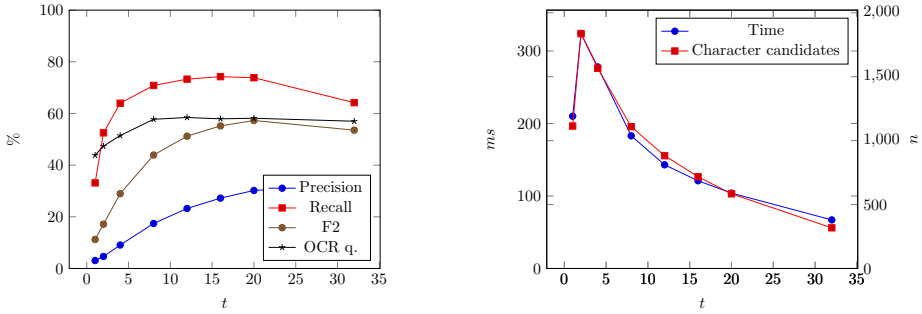



Figure 3.8: Effect of the threshold $t$ on the FASText detector. ($p_1 = 0$, $p = 0$, luminance channel, 20% of dataset)

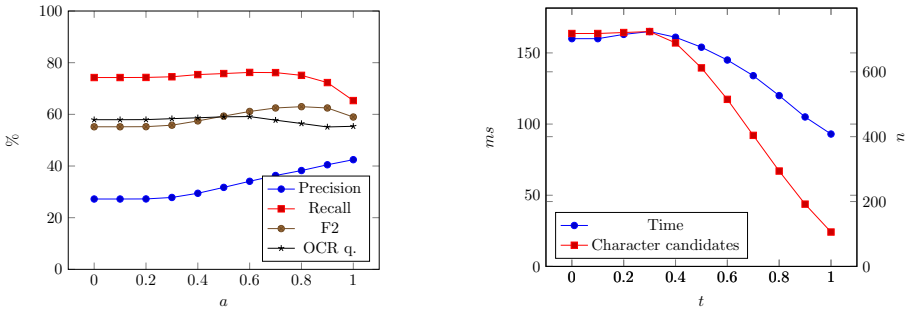



Figure 3.9: Effect of the adaptive threshold $a$ on the FASText character detector performance. ($t = 16$, $p = 0$, luminance channel, 20% of dataset)

#### 3.4.2.3 Experiments

Figure 3.8 shows different threshold values used for the FASText character detector. For this experiment, 20% of the dataset with only the luminance channel were used. The detector did not use adaptive thresholding or non max suppression. Additionally, the minimal probability for both classifier filters was set to 0. The data shows that the number of extracted characters decreases with higher thresholding values. Note that a threshold of one is an exception due to the too strong constraint. For values larger than one, the number of regions decreases. A too low threshold results in many keypoints being detected due to noise, which increases the keypoint count and therefore, the character count. Because they are only filtered by their size and not the character probability, the number of extracted character candidates stays high. This high number of detected keypoints requires an equally high number of segmentation steps and therefore, increases the runtime.
At the same time, a too low threshold value reduces the precision and recall due to intensity fluctuations within a character stroke. A low threshold value can therefore cause all neighbors of a stoke key point to be identified as different, which prevents the keypoint from being detected. An increase in the threshold value increases the precision and recall. If the threshold is increased too much, it prevents characters with a low contrast from being detected. As a result, the recall in the experiment decreases for too high threshold values. The recall is at its maximum with 74.25% at $t = 16$, but the precision increases further for higher thresholds. An explanation for this is that the keypoints that are still being detected need to have a high contrast which reduces the impact of noise.

The effect of adaptive thresholing can be seen in figure 3.9. The threshold used for

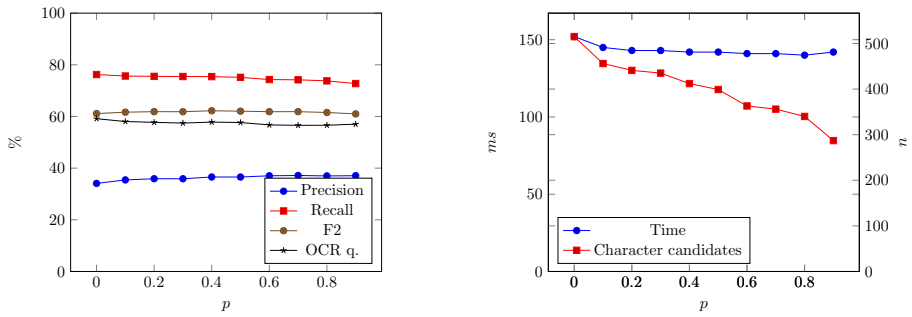



Figure 3.10: Effect of the minimal character probability $p$ on the ER character detector performance. ($t = 16$, $a = 0.6$, $p = 0$, luminance channel, 20% of dataset)

this test was $t = 16$, all other parameters were the same as the last test. $a$ controls the ratio of the average distance of a keypoints neighbors that is used for the threshold. To prevent this threshold from being too low, it is combined with the static threshold by using the bigger value of the two. $a = 0$ means that the adaptive threshold is always 0, which results in the static threshold being used instead. Consequently, the result is the same as setting the static threshold to $t = 16$ and disabling adaptive thresholding. Setting $a = 1$ on the other hand, means that neighbors need to have a higher intensity difference than the average neighbor. Using the adaptive threshold allows characters with a high contrast to be detected even if the intensity value of the stroke pixels varies more than the threshold. A character with an intensity of 220-255 on black background would not be detected with a static threshold of 16 but is detected with adaptive thresholding.
The recall is at its maximum with 76.23% at $a = 0.6$. Again, the precision increases further if $a$ is raised. Furthermore, the number of extracted characters and the processing time decreases due to the higher thresholding values being used.

The effect of the minimal probability can be seen in figure 3.10. Despite the reduced number of returned character candidates, the processing time stays the same. This is due to no further steps that have to be performed in the character detector. Nevertheless, the reduced number of characters will decrease the processing time of the complete UIC pipeline.
Although it is only a small change, the recall of the system decreases with a higher probability threshold and the precision increases. Therefore, it is recommended to use $p = 0$ for a purely recall optimized system and $p = 0.4$ (Highest $f^2$ measure) for a system optimized for speed. The results and parameters for both of the systems can be seen in table 3.3. Note that the models used for the probability prediction were generated for the Extremal Region detector. Although the same features were used for both, the regions returned by both methods differ. Therefore, the results for the FASText detector are inferior.
Additionally, both systems were tested with and without non max suppression. The results show that system 2 without a probability filter and non max suppression outperforms system 1 in terms of speed and recall.

#### 3.4.2.4 Color

Busta et al. only described the application of their algorithm to the intensity channel of an image. To improve the detection of low quality characters, the addition of color information is promising and was therefore investigated in more detail in this work. Two different methods were tested:

| Goal | $t$ | $a$ | $p$ |
|---|---|---|---|
| System 1 | 16 | 0.6 | 0.4 |
| System 2 | 16 | 0.6 | 0 |

(a) Parameters

| Goal | Precision | Recall | F2 | Time | n |
|---|---|---|---|---|---|
| System 1 | 34.77% | 73.43% | 60.07% | 151 | 355 |
| System 1 + NMS | 33.05% | 70.12% | 57.27% | 112 | 147 |
| System 2 | 34.09% | 76.23% | 61.12% | 150 | 515 |
| System 2 + NMS | 32.38% | 74.30% | 59.02% | 112 | 218 |

(b) Performance of characters with and without Non Max Suppression

Table 3.3: The final two FASText character detectors

| Channels | Precision | Recall | F2 | Ocr q. | Time | n |
|---|---|---|---|---|---|---|
| R | 29.64% | 70.13% | 55.08% | 58.06% | 147 | 517 |
| G | 35.37% | 76.88% | 62.27% | 58.59% | 142 | 522 |
| B | 33.74% | 75.23% | 60.38% | 59.06% | 147 | 500 |
| H | 0% | 17.00% | 0% | 20.47% | 91 | 167 |
| S | 14.50% | 46.32% | 32.20% | 48.87% | 139 | 457 |
| L | 34.09% | 76.23% | 61.12% | 59.14% | 146 | 515 |
| D | 23.62% | 67.83% | 49.35% | 47.10% | 195 | 488 |
| RGB | 33.06% | 79.08% | 61.86% | 58.73% | 433 | 1540 |
| HSL | 26.15% | 78.11% | 55.90% | 55.94% | 372 | 1140 |
| D+RGB | 31.15% | 82.31% | 61.96% | 57.66% | 631 | 2028 |
| D+HSL | 25.75% | 82.20% | 57.15% | 54.52% | 561 | 1629 |
| D+HSL+RGB | 29.45% | 83.12% | 60.92% | 57.04% | 995 | 3169 |
| RGB+L | 33.32% | 79.34% | 62.17% | 58.83% | 574 | 2055 |

Table 3.4: Combination of multiple color channels for the FASText detector using system 2 without NMS. (G...Gray, I...Intensity, D...Gradient)

- **Application to separate channels**
  This method uses the same algorithm for multiple color channels and combines the results. This is the same method used for the ER detector and therefore allows the reuse of the algorithm to separately process and combine color channels. Consequently, it suffers from the same disadvantages, a dramatical runtime increase and the detection of the same character in multiple channels. The result of the application of system 2 with non max suppression to different combinations of color channels can be seen in table 3.4. As for the ER detector, the combination of the RGB and gradient channel provides the best compromise between recall and processing time.

- **Direct application (FASText3)**
  This method tries to apply the FASText algorithm directly to all three channels of a color image (RGB). For this to work, some alterations to the original algorithm were necessary to handle three dimensional color information. The Euclidean distance was used to calculate the difference between two neighbors in the keypoint detection stage. This distance could then be used to decide if two pixels were part of the same stroke by comparing the value to a threshold.
  The original algorithm also rejected keypoints if not all pixels on the surrounding circle were either brighter or darker. Although reducing the color information to a brightness value seems like a promising method at the first glance, it can result

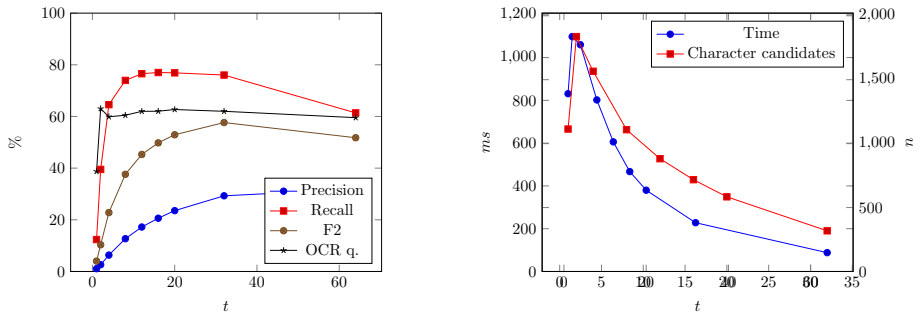



Figure 3.11: Effect of the threshold $t$ on the FASText character detector directly applied to RGB image. ($p = 0$, static threshold, 20% of dataset)

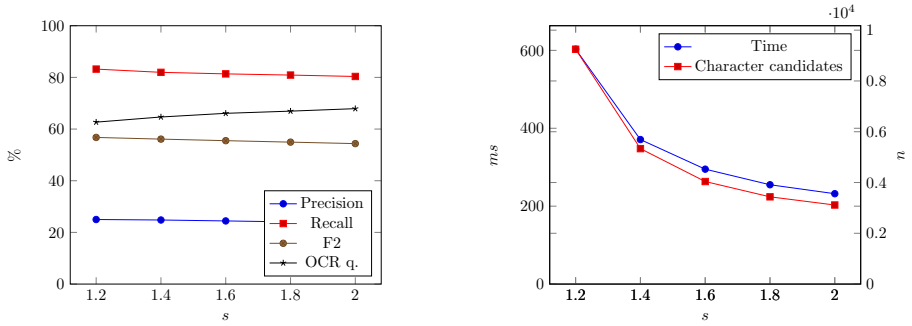
in the rejection of otherwise detectable characters. For example, a gray character on a brown background would result in its rejection if the backgrounds brightness fluctuates too. However, the gray character could be identified by its lack of color. Due to these reasons, the filter was disabled for color images. On the one hand, this results in a lower precision and speed due to the higher number of keypoints that have to be segmented in the following step. On the other hand, it improves the recall of the algorithm. Considering that the algorithm has almost the same runtime cost if it is applied to color or monochrome images it is still significantly faster than having to apply the algorithm separately to each color channel. Finally, the segmentation of the characters is done by using the Euclidean distance during the flood fill process.

Although most parameters can be taken over from the original algorithm, the optimal threshold had to be determined again to compensate the different distance computation. Having already shown the positive effect of adaptive thresholding, the adaptive thresholding parameter $a$ was kept at 0.6. The character probability filter was disabled. The results of the different static threshold values can be seen in figure 3.11. It shows that a static threshold of $t = 16$ yields the best result. By enabling adaptive thresholding, the recall could be improved to 78.55%. Additionally, it decreased the average runtime from 467ms to 395ms and the average number of characters from 1095 to 750.
As a result, it is recommended to use the same parameters as for system 2 for the direct application.

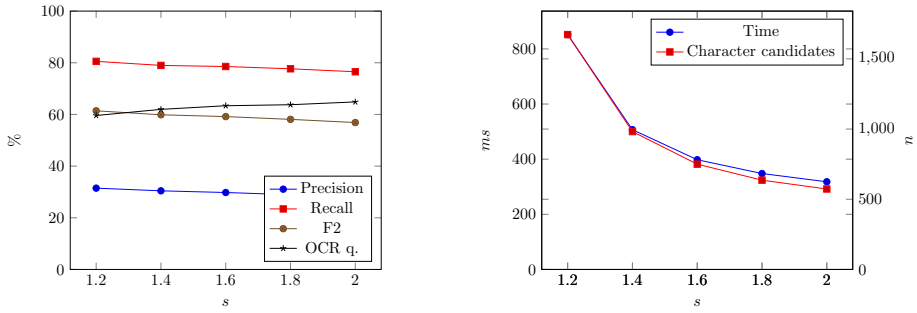
### 3.4.3 Scale

Although the ER character detector is able to detect characters without the use of a scale pyramid, Neumann and Matas showed that it can improve the result [NM13]. Thereby, the reduction of the image size acts as a Gaussian blur filter and can remove noise. This can result in some additional characters being detected that were previously hidden by noise. As an example, a gap in a character due to rust can be hidden by reducing the image size and results in the successful detection of the character.
Because the scaling capability already had to be added to the pipeline for the FASText detector, it could also be applied to the ER detector without any significant work overhead.

Figure 3.12a and 3.12b show the use of an image pyramid with different scaling factors for the FASText and ER detector. To reduce the runtime, only the brightness channel was used for the ER detector and FASText3 was used instead of its single channel variant.




(a) Single channel ER detector (Recall optimized system with luminance channel only)




(b) FASText detector with direct color application

Figure 3.12: Effect of different scale factors $s$ on character detectors (10% of dataset))

As expected, the recall grows with a reduction of the scaling factor. A scaling factor of two means the resolution was halved after every cycle. If it is lower than 2, it results in more images that have to be processed by the character detector. Note that a higher scaling factor cannot result in any previously detected characters to be lost but only in new characters being detected. Due to the resulting repeated application of the character detector to the same input image in different resolutions, it results in a significant increase of character candidates being detected for every additional scale level. This effect is even more significant for the ER detector due to the possibility of detecting the same character in most scale levels. On the contrary, the FASText detector can only detect characters whose stroke width is in between 1-3px. As a result, the same character can only be detected in a limited number of consecutive scale levels.
Eventually, $s = 1.6$ was chosen as the scale factor as it provides a good compromise between recall and runtime. This is the same value chosen by Busta et al. [BNM15]. Furthermore, the positive effect of a scale pyramid suggested in [NM13] could be confirmed for the Extremal Regions character detector. For a scale factor of 1.6, the recall could be improved from 77.54% without scaling to 81.36%. At the same time, the average number of returned character candidates and average runtime was increased from $n = 1832$ and $t = 151$ to $n = 4039$ and $t = 295$. The detected characters can be processed separately for each scale, which prevents lines with characters detected in different resolutions. Or alternatively, they can all be combined into a single list and passed to one line detector. This allows the mixture of characters from different scales but can result in a significantly higher processing time.

### 3.4.4 Comparison

This section summarizes all previously stated variants of character detectors and compares them directly to each other. Table 3.5 shows the results of a selection of character

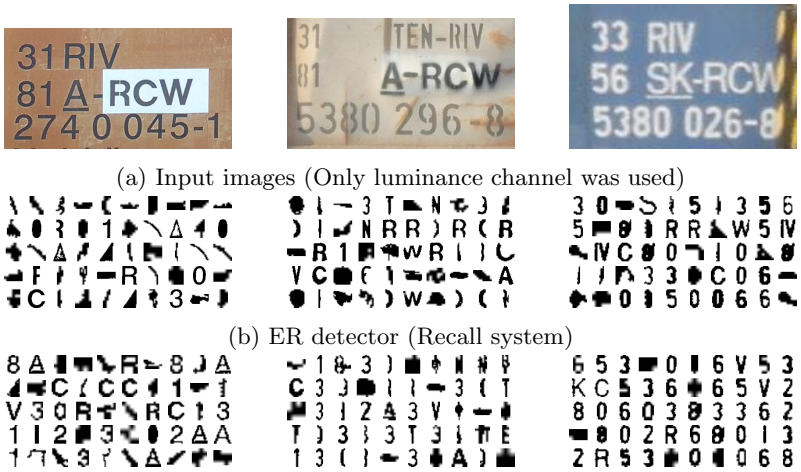
| System | Pre. | Rec. | F2 | Time | n |
|---|---|---|---|---|---|
| ER-Speed | 21.50% | 68.76% | 47.76% | 34 | 132 |
| ER-Speed+RGB | 20.73% | 74.62% | 49.09% | 100 | 406 |
| ER-Speed+RGB+Scale | 24.39% | 79.56% | 54.77% | 181 | 1014 |
| ER-Speed+NMS+RGB+Scale | 21.02% | 77.82% | 50.52% | 164 | 489 |
| ER-Balance+RGB | 21.45% | 78.54% | 51.25% | 176 | 1559 |
| ER-Balance+RGB+Scale | 25.40% | 83.09% | 57.14% | 343 | 3689 |
| ER-Balance+RGBD+Scale | 24.00% | 84.15% | 56.06% | 439 | 4295 |
| ER-Recall+RGBD+Scale | 23.12% | 85.75% | 55.62% | 1034 | 14020 |
| FASText3 | 31.46% | 78.11% | 60.24% | 350 | 672 |
| FASText+NMS+RGBD | 31.67% | 79.78% | 61.19% | 397 | 815 |
| FASText+RGBD | 32.40% | 81.14% | 62.37% | 564 | 1914 |

Table 3.5: A selection of character detector configurations (50% of dataset)

detectors being applied to 50% of the dataset. Thereby, the labels "Speed, Balance and Recall" refer to the Extremal Region detector system configurations from table 3.1. RGB and RGBD describe the used color channels. With "D" standing for the gradient channel. "Scale" is used when a scale pyramid was applied (With scaling factor 1.6). Note that all FASText detectors require the use of a scale pyramid and it was therefore omitted in the description. Finally, "NMS" refers to the use of Non Max Suppression. As explained in section 3.4.2, all FASText detectors use adaptive thresholding with $t = 16$.

Three versions of the "speed" ER detector were tested. Only the RGB channels were used as the additional Gradient channel only offers a small performance improvement. A scale pyramid could improve the recall by 4.94% while increasing the average runtime from 100ms to 181ms. Additionally, the number of detected characters increased from 406 to 1014. Although the recall was improved significantly, the higher number of extracted character candidates did not offer a good enough trade off for a system that is optimized towards speed. To reduce $n$, the system was also tested with Non Max Suppression, which still resulted in a 3.2% recall increase but only an increase of 64ms and 83 characters. This is an acceptable trade off and the system is therefore recommended for situations when the runtime is important.
The balanced system on the other hand, has a significantly higher runtime, but does have a better recall. This is caused by the more lenient filtering. It is notable that the "balanced" system without a scale pyramid performs worse in respect to recall and number of extracted characters, than the "speed" system with a scale pyramid. Therefore, it is not a promising candidate for a final system. If a scale pyramid is added, it boosts the recall significantly. When adding the gradient channel as well, it results in a similar recall than the best character detector tested. The recall is 84.15%, compared to the "recall" system with a scale pyramid and RBGD color channels with 85.75%. Additionally, the "recall" optimized system has a significantly longer average runtime and returns a higher number of character candidates. Note that the extracted characters can be processed individually for each channel and scale to reduce the number of characters the line detector has to process at once.

Comparing the ER detectors to the FASText detectors shows that although the processing time is longer for the FASText detectors, the number of extracted characters and the precision/F2 measure is of higher quality. Considering that several suggested stages for further filtering from the original paper were not implemented, the results show the potential of the method [BNM15]. Note that none of the left out features would have improved the recall, they would only have improved the precision. Nevertheless, the ER detector does seem to detect more characters in low quality conditions, which



(a) Input images (Only luminance channel was used)

(b) ER detector (Recall system)

(c) FASText detector (System 2)

Figure 3.13: Samples of character detector results (All characters were scaled to the same size)

makes it more applicable for the domain of UIC number detection.

Figure 3.13 show samples of the extracted characters with different systems. Thereby, only the luminance channel and no scale pyramid were used for the ER detectors. The characters extracted with the ER detector tend to be sharper and to have a higher resolution. This can be an advantage for the OCR software as it has more information to work with.
The FASText characters benefit from the focus on line keypoints. The extracted characters are more stroke like compared to the ER characters.

## 3.5 Line Detector

### 3.5.1 Dealing with spray painted digits

One issue specific to UIC numbers, that was not discussed in the previous section, is spray painted 0, 6, 8 and 9's. Most of the UIC numbers are spray painted by hand using templates for each digit. To allow for holes, the holes in the template must be connected to the rest of the template (See figure 3.14a). If these structural connectors are too big, they can lead to lines that separate the digits into 2 components as seen in figure 3.14b. This prevents the character detector from finding the digit as a whole. Instead, two unconnected regions are found. Although the text lines can still be detected, it does hinder the OCR software from identifying the digits if only half the digit is supplied. Even though a classifier could be trained to decode incomplete digits, it significantly increases the difficulty, especially when considering the similarity between half a 0 and a 1.
One method for dealing with this problem, is to reduce the resolution of the image to a point where the separating lines are not visible anymore and both components merge into one. Although this works up to a certain degree, if the separating lines are as thick as the stroke width, this method starts to fail. Additionally, it greatly reduces the resolution and therefore the quality of regions, which in turn makes the later OCR detection more difficult.

In addition to using a scale pyramid, regions that were identified as part of the same character were merged together. This was done when adding characters to the final text


(a) Spray template


(b) UIC number with “0”, “6” and “8” consisting of two regions

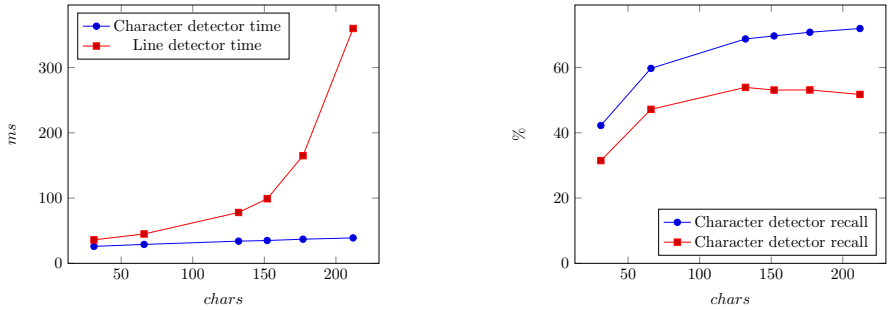
Figure 3.14: Spray painted UIC numbers

line object. Thereby, only the region to the left has to be checked, instead of having to check every possible character combination. The disadvantage of this method is that one part of a character might not have been added to the line. For example, the smaller part of a 6 might not be added because its height is too small to be added to the same line.
To merge two regions into one character, both of their aspect ratios have to be higher than 3 (must be slim). Furthermore, the aspect ratio of the bounding box, containing both characters, must not exceed 1.5. Both of these checks can be done without any significant performance impact. If both of the conditions are fulfilled, it is checked if the two regions are facing each other. For this, the “direction” of the top and bottom line of the region images is computed. A line is facing left if it has more region pixels on the left and facing right if more of the pixels are on the right. If both sides contain the same amount of pixels, it is not assigned a direction. If either the top or the bottom line of the two characters are facing each other, the two regions are merged.
Note that this part of the line detector can actually improve the recall of the line detector if the same number of valid characters are found and the corrected characters have a better match with the ground truth.

### 3.5.2 Neumann and Matas

This line detection method was developed by Neumann and Matas and was explained in detail in section 2.2.1 [NM11b]. It is from here on referred to as NM line detector. Unfortunately, this method is not optimized for the detection of UIC numbers and cannot detect character groups with only 2 digits. Additionally, the high runtime complexity of the algorithm dramatically limits the number of character candidates that it can process at once. Therefore, the characters had to be filtered more strictly before they were passed to the line detector, resulting in a lower recall.
Figure 3.15a shows the relation of the processing time and the number of characters that had to be processed. To alter the number of characters, the threshold of the Extremal Region detector was set to different values, which resulted in more character candidates being generated and passed to the line detector. To reduce the runtime of the character detector, only the luminance channel was used. Comparing the number of input characters with the runtime, clearly shows the runtime increase for higher $n$'s'. Although the runtime impact of the line detector is insignificant for low numbers of characters, it increases drastically for a higher number. For 132 characters, the runtime of the line detector almost exceeds the runtime of the character detector and rises even further, with more characters. For 265 characters, the average runtime per image is 1120ms, which renders the algorithm unusable, especially if you consider that the images used to run these tests only contain the UIC number. For images that picture a full train wagon, the number of extracted characters would be significantly higher.

Due to the significant runtime disadvantage, the line detector can only be used in




(a) Average processing time

(b) Average recall

Figure 3.15: Behavior of Neumann and Matas line detector depending on the number of input characters (50% of dataset)

combination with character detectors that return a low number of results. Furthermore, the use of multiple color channels and scale levels should be avoided on the character detection level. Instead, the line detector can process each channel and scale individually and combine the resulting lines at the end of the process. The disadvantage of this method is a higher number of duplicated lines, detected in different scales and color channels.
Table 3.6 shows the results of the line detector with the speed optimized ER detector. For the test with multiple color channels and scale levels, the lines were processed separately. By comparing the results of the NM line detector with the results of the character detector, it can be observed that the runtime increased significantly. On average, the line detection stage takes at least the same amount of time as the character detector. The average runtime of the speed optimized system, using the RGB channels and a scale pyramid, increased from $181ms$ to $575ms$. This problem occurs even though the speed optimized system extracts the lowest number of character candidates of all the tested character detectors. For this reason, the balanced and recall optimized system were not tested with the NM line detector. The processing time of the balanced optimized system with only the luminance channel and without a scale pyramid takes multiple seconds per image.

A further disadvantage of the algorithm can be seen in figure 3.15b. The recall of the line detector initially increases with the number of characters fed into it, but it starts to decrease again if the number of input characters becomes too high. The algorithm only allows one character to be part of one line grouping. As a result, more characters can result in the wrong lines being rejected if they share a character. This in turn decreases the recall.

The results using different pipeline configurations can be seen in table 3.6. The table only includes results of the speed optimized ER detector because the runtime of the other systems is infeasible. Comparing the results of the line detector to the character detector, shows that the additional stage decreases the recall as it filters more characters. At the same time, the average number of characters falls dramatically. By processing multiple color channels and resolutions with the NM line detector, the recall difference to the pure character detector falls from 13.65% to 6.39%. This is a result of the availability of multiple versions of characters, which increases the chances of the line detector finding a character in at least one of the color channels or resolutions.
Table 3.6 also shows that a scale pyramid is more effective than using the R, G and B channel instead of luminance channel. Not only does the use of the scale pyramid result in a faster processing time, it also has a higher recall and precision. As previously noted, the NM detector struggles with a too high number of input characters. If the number

| System | Pre. | Rec. | F2 | Time | n |
|---|---|---|---|---|---|
| ER-Speed | 21.50% | 68.76% | 47.76% | 34 | 132 |
| ER-Speed+RGB+Scale | 24.39% | 79.56% | 54.77% | 181 | 1014 |
| ER-Speed+NM | 32.50% | 55.11% | 48.38% | 76 | 21 |
| ER-Speed+NM+Scale | 33.30% | 67.06% | 55.75% | 181 | 72 |
| ER-Speed+NM+RGB | 31.82% | 65.32% | 53.96% | 234 | 61 |
| ER-Speed+NM+RGB+Scale | 32.69% | 73.17% | 58.64% | 558 | 209 |

Table 3.6: Results of the NM line detector

of characters in one channel is too high, it will likely be too high in other channels as well. Using smaller scale levels on the other hand, also reduces the number of found characters, which explains the better performance of the scale pyramid.

### 3.5.3 Custom line detector

The goal of the line detector is to filter invalid characters and group valid characters into lines. Since most line detectors only detect text segments with 3 or more characters, they are not suitable to detect UIC numbers. Although the first four digits are accompanied by additional text, the gap between them can be too high to be recognized by traditional methods (e.g. figure1.3e). On the other hand, the assumption from section 1.1, that all digits are the same height, simplifies the task and can improve the accuracy compared to more complex systems that consider upper and lower case letters. Additionally, the custom system for the detection of UIC lines only has to be able to process vertical lines. With these properties in mind, the following three stage system was designed:

1. **Create a line candidate for every character candidate**
   Every charter is converted into a line candidate and inserted into two sorted lists along the y-axis. The first set sorts the lines by the top position of the associated character (maximum Y) and the second one according to their bottom position (minimum Y). Characters with the same Y value were both kept and ordered by their order of creation. The standard C++ implementation provides a insert operation with a complexity of $\mathcal{O}(\log n)$. This results in an average time complexity of $\mathcal{O}(n \log n)$.

2. **Character candidates are added to all overlapping and matching line candidates**
   This can efficiently be done by starting with the topmost line candidate from the top sorted list and adding it to a hash set of active lines. Then, the Y-axis is scanned from top to bottom for the beginning and end of characters. Thereby, the next line is selected either from the top or bottom sorted list, depending on which one returns the higher Y value.

   - If it is the start of a character (top position), it is added to the active lines.
   - If it is the end of a character, it is removed from the active lines. This means that the character is overlapping with all currently active lines. Therefore, it is passed to all active lines and all characters from the active lines are passed to the line that is being removed. A line candidate accepts a character, if it satisfies all of the following conditions:
     - The added character is to the left of the reference character. The reference character is the character candidate that was passed during the creation of the line candidate.

– It has a similar size to the reference character ($h_r \ldots$ height of reference character, $v_h \ldots$ custom value):

$$\frac{\min(h_r, h)}{\max(h_r, h)} > v_h \tag{3.8}$$

– Its central Y position is similar to the position of the reference character($h_r \ldots$ height of reference character, $h_r \ldots$ Y position of reference character, $v_y \ldots$ custom value):

$$|y_r - y| < h_r / v_y \tag{3.9}$$

The final algorithm of this stage can be seen in listing 2. Assuming that all

```
auto itTopSorted = topSorted.begin();
auto itBottomSorted = bottomSorted.begin();

unordered_set<Ptr<LineCandidate>> activeLines;
double topDistance = (*itTopSorted)->characterCandidate->getTop();
double bottomDistance = (*itBottomSorted)->characterCandidate->
    getBottom();
while (itBottomSorted != bottomSorted.end()) {
    if (itTopSorted != topSorted.end() && topDistance <
        bottomDistance) {
        activeLines.insert(*itTopSorted);
        itTopSorted++;
        if (itTopSorted != topSorted.end())
            topDistance = (*itTopSorted)->characterCandidate->
                getTop();
    } else {
        PLineCandidate bottom = *itBottomSorted;
        activeLines.erase(bottom);
        for (PLineCandidate c:activeLines) {
            c->addLetterIfMatching(bottom->characterCandidate);
            bottom->addLetterIfMatching(c->characterCandidate);
        }
        itBottomSorted++;
        if (itBottomSorted != bottomSorted.end())
            bottomDistance = (*itBottomSorted)->characterCandidate
                ->getBottom();
    }
}
```

Listing 2: Algorithm to group vertically overlapping characters

characters are overlapping and of similar size, this would result in $n^2$ insertion operations. Because the characters within a line candidate are also sorted by their X position, the complexity is $\mathcal{O}(n^2 \log n)$

3. **Overlapping and lonely characters are removed from each line**
In this stage, the recursive algorithm uses the horizontally sorted characters of each line candidate to group them into separate segments. Note that these segments do not represent words. By adding all matching horizontal overlapping characters in the earlier stage, uncorrelated lines or non characters with big gaps in between can both be part of the same line candidate. The bigger the width of the original input image, the more segments can be in every line candidate.

An additional problem with the resulting line candidates from the earlier stages, is that they can contain the same character multiple times (e.g. from different scales or color channels). For this purpose, a function was required that compares the quality of two character candidates and uses the better one for the final text line. It works as follows (Listing 3):

- The left most character is picked from the beginning of the horizontally sorted list of characters. It is not clear that this character will be kept and it has to be checked if the character has a neighbor that makes it part of a segment. Because it is the first character, it can only have a neighbor on its right. The recursive function ("findNextNeighbor") is called with the first character as the currently observed one and the position in the list to query the next character. Additionally, the predecessor of the current character is passed. Because we are currently trying to find a neighbor for the left most character, null is passed as its previous character.
- The "findNextNeighbor" function takes the next character from the list as a potential next character.
  - If the potential next character is overlapping with the current one, only one of them is kept. The decision is made depending on which one of the two is more compatible with the previous character. If there is no last character, the one with the higher confidence value is kept. The winning character is then used as the new current character for another recursive call of "findNextNeighbor" with the preceding character staying the same. Note that the function will automatically pick the next character as the new potential character. The return value of this call is passed directly to the caller.
    Two characters are considered to be overlapping if $i/m > v_o$. Thereby, $i$ is the area of the intersection of both characters and $m$ is the area of the smallest bounding box containing both characters. If this value is 0, the two characters are not overlapping at all. If the value is 1, one of the characters is completely enclosed by the other one. The threshold $v_o$ is necessary to allow a small overlap that can occur for very close blurry characters (e.g. low resolution characters from the scale pyramid). In case the two characters are overlapping, several properties were checked to choose the more likely candidate. First, it is checked if the height of the new character $h$ is significantly more similar to the reference character $h_r$ than the old character:
    $$s_h = \frac{\min(h_r, h)}{\max(h_r, h)} \tag{3.10}$$
    If the height similarity differs by more than 0.1, the more similar character is chosen. Next the color difference to the reference character is computed. Thereby, the norm of the mean color in LAB space is used. If the difference is more than 10, the more similar character is selected. If both the height and the color of both characters are too similar, the character with higher confidence value returned by the character detector classifier is selected.
  - If the current character is not overlapping with any others, it is not clear yet if it has a close neighbor. To find out, the character of the next segment is computed with another recursive call of "findNextNeighbor". This time, the current character is used as the preceding one and the potential character is used as the current one. By knowing the next character as well as the last, it can be determined if the potential character has a close neighbor. If there is a character to the left and it is close

to the potential character or to the right and it is close, it is assumed that it is part of a segment and therefore, a valid part of the line. To determine if two characters are close, the following check is performed:

$$|y_l - w_l - y_r| < h_r * v_d \tag{3.11}$$

Thereby, $y_l$ and $w_l$ are the center Y position and the width of the left character, $y_r$ is the center Y position of the right character and $h_r$ is the height of the lines reference character.
If the character has a neighbor, it is added to the beginning of a list of valid characters and is returned from the function as a neighbor. If there is neither a left nor a right neighbor, the current one is ignored and the calculated next neighbor is returned. Note that it can also be null if there is no valid right segment. Note that this forces each segment to contain at least two characters.

If there is no next character in the list, it still has to be checked if the current character has a neighbor. Because there is no right character it has to be checked if the previous character is close. If there is no previous character or it is too far away, the current character is not added and null is returned, otherwise the current character is added to the beginning of the valid character list and returned.

Again, assuming that all characters are overlapping and similar size, the number of lines is $n$ and each line would contain up to $n$ characters. The resulting complexity of this stage is $\mathcal{O}(n^2)$, which leaves us with the combined complexity of the first and second stage of $\mathcal{O}(n^2 \log n + n^2) = \mathcal{O}(n^2 \log n)$.

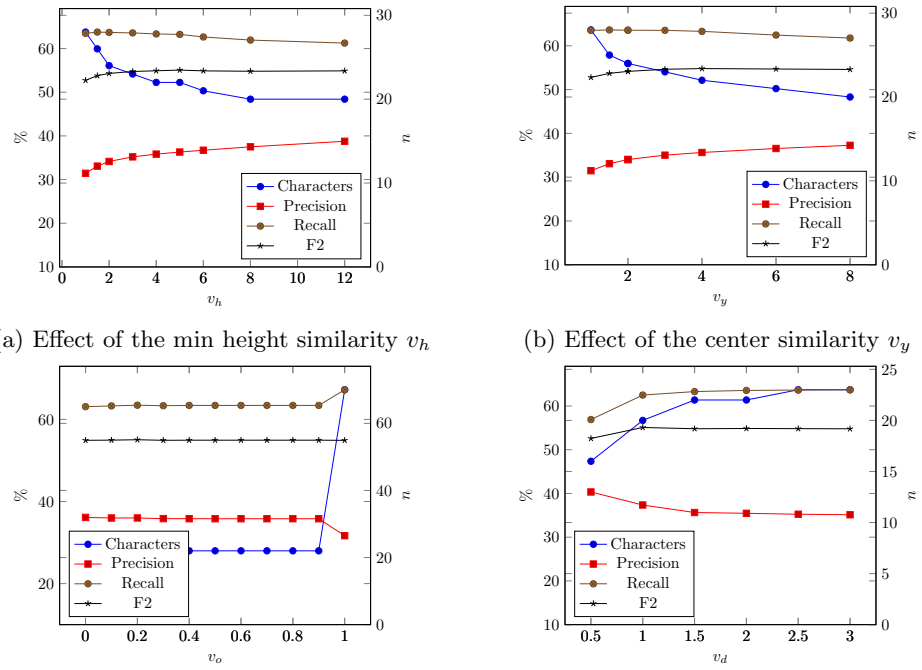
4. **Line candidates with the most characters are selected as text lines and split into words**
Finally, the line candidates with the most remaining characters are selected and split into final lines. Even if text segments are far apart, as for example in the first two lines of UIC numbers, they can still be detected due to the consistent character style.
The line candidates cannot be used directly because they may contain uncorrelated text segments with a significant distance in between them. This can happen if text shares the same baseline and properties. The initial grouping increases the likelihood of its detection, even if the separate segments are short. To split a line candidate in its final form, the previous character closeness criteria from equation 3.11 is used with a more relaxed threshold, to allow small spaces in between words. Note that the threshold of 3 was chosen to allow for a space between the digits and the description in the first two lines of the UIC number.

$$|y_l - w_l - y_r| < h_r * 3 \tag{3.12}$$

Finally, every character can only be part of one text line, otherwise the line candidate of each character would result in one final line. Therefore, each used character is blacklisted and line candidates that contain an already used character are rejected. Processing the line candidates with the highest numbers of characters first also ensures that the line candidate with the left most reference character is chosen, because it has the highest likelihood of containing all further characters. If two lines have the same length, the first characters "character probability" is compared. If the probability is the same as well, the next character is compared until a difference is found or a random line is selected.
Selecting the longest lines, requires the lines to be sorted, which can be done in $\mathcal{O}(n \log n)$ and does not change the complexity of the final algorithm ($\mathcal{O}(n^2 \log n +$

```
1  const PCharacterCandidate *
2  LineCandidate::findNextNeighbour(const PCharacterCandidate *last,
       const PCharacterCandidate &current,
3                                   set<cv::Ptr<cv::text::
                                        CharacterCandidate>, cv::text
                                        ::SortCharByTransformedLeft>::
                                        iterator &itLeft) {
4      if (itLeft == charsLeftSorted.end()) {
5          if (last && areClose(*last, current)) {
6              characters.push_front(current);
7              return &current;
8          } else {
9              return nullptr;
10         }
11     }
12
13     const PCharacterCandidate &possibleNext = *itLeft++;
14
15     bool areSeparated = getOverlap(current, possibleNext) < 0.3;
16     if (areSeparated) {
17         const PCharacterCandidate *actualNext = findNextNeighbour(&
               current, possibleNext, itLeft);
18         if (last && areClose(*last, current) || actualNext &&
               areClose(current, *actualNext)) {
19             // Current has a neighbour. We can add it.
20             characters.push_front(current);
21             return &current;
22         } else {
23             // Current has no close neighbour
24             return actualNext;
25         }
26     } else if (hasBetterMatch(last, current, possibleNext)) {
27         // Replace current with this character
28         return findNextNeighbour(last, possibleNext, itLeft);
29     } else {
30         // Try next character
31         return findNextNeighbour(last, current, itLeft);
32     }
33 }
```

Listing 3: Algorithm to filter characters from line candidates




(a) Effect of the min height similarity $v_h$

(b) Effect of the center similarity $v_y$

(c) Effect of the max overlap $v_o$

(d) Effect of the max distance $v_d$

Figure 3.16: Parameters of the custom line detector. Tests run with 50% of dataset and $v_h = 0.8, v_y = 4, v_o = 0.3$ and $v_d = 1.5$ as defaults. The speed optimized ER character detector with the luminance channel only was used

$nlogn) = \mathcal{O}(n^2 \log n)$). Splitting the remaining line candidates into final text lines can be done in $\mathcal{O}(n^2)$ which again does not change the final complexity of the algorithm.

Figure 3.16 shows the influence of the line detectors parameters. For the tests, the speed optimized ER character detector was used with the luminance channel only. Due to its low processing time, 50% of the dataset were used to create the charts. As a default, the parameters were set to $v_h = 0.8, v_y = 4, v_o = 0.3$ and $v_d = 1.5$. The results can be interpreted as follows:

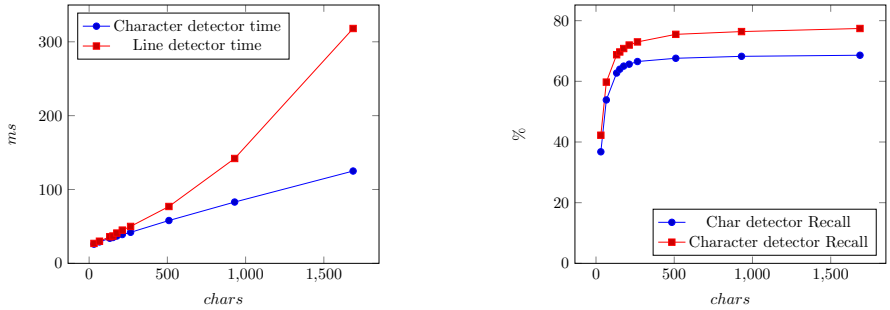
- **Minimal height similarity** $v_h = 0.8$ (Figure 3.16a)
  This parameter defines how similar the height of a character has to be to the reference character to be added to a line. A value of 0.5 results in most characters being accepted due to their height, but is also means that the final decision of the character being added to the line is dependent on the color and center similarity. As a result, the number of characters is high due to a higher chance of non characters being grouped, which in turn reduces the precision. At the same time, the recall is slightly lower, which can be explained due to the higher shared number of characters in between lines, which can lead to more lines being rejected.
  If $v_h$ is increased, the recall reaches its maximum at $v_h = 0.8$ with $p = 63.73\%$ and the same time the number of returned characters reaches its minimum. Thereby, the most characters are being rejected due to not having any neighbors.
  If the value is increased further, only characters with exactly the same height are grouped as a line, thereby reducing the chance of non characters finding a neighbor. As a result, the precision goes up further but the recall falls to its lowest value. For $v_h = 1$, this also means that characters with a higher resolution are less likely to be grouped together because they don't have exactly the same height. Although

intuitively one would expect the number of returned characters to go down as well, the contrary is the case. The strict height requirements lead to shorter lines, reducing the chance of two lines containing the same character, leading to the character detector returning multiple versions of the same character (e.g. from different threshold levels). Longer lines would have a higher probability of sharing at least one version of one character, but the shorter lines can result in the same line being detected twice with all different versions of the same characters.
For the final detector, $v_h = 0.8$ was chosen, as it has the highest recall and a low number of returned characters.

- **Minimal center similarity** $v_y = 3$ (Figure 3.16b)
  The minimal center similarity is defined as a portion of the reference characters height. The value is then used as a threshold for the vertical distance between the center of two characters. The chart shows that the number of characters and the recall decline with a smaller distance ($d = h_r/v_y$). Note that if $v_y$ is smaller than 2, the center point can lie outside of the vertical region overlapping with the reference character. As a result, characters that are above or below the line of the reference character might be included. This might also include characters which otherwise would not be detected, because they do not have a close neighbor. Although this can increase the recall, it is not a desired behavior of the line detector. Additionally, the less strict filtering allows a higher number of false positives to be accepted which is reflected in an increased recall for higher values of $v_y$. A disadvantage of only accepting characters within a small vertical distance, is that the algorithm becomes more sensitive to a small rotation of the image. If the lines are rotated by a small amount, characters further away from the reference character have a higher vertical offset, which can prevent them from being detected if the threshold is too strict. As a result, $v_y$ was chosen more leniently to accept characters to have a vertical distance of up to 1/3 of the height of the reference character.

- **Maximum overlap** $v_o = 0.2$ (Figure 3.16c)
  The chart shows that the recall, precision and number of characters is not impacted significantly by the allowed overlap. Only allowing the characters to be fully overlapping has a significant impact on the recall and the number of characters. By allowing fully overlapping characters within a line, fewer characters are filtered from the line, which increases the number of characters. In particular, the unfiltered lines can also contain multiple versions of the same character (e.g. different threshold values). Even though a repeated detection of the same character would not increase the recall, if they have the exact same bounding box, it can increase the recall if some of the detected regions include part of the character. The effect shows that the algorithm does not always pick the correct overlapping character if there are multiple ones available. Nevertheless keeping all overlapping characters is not a satisfying solution.
  Although allowing overlapping characters results only in an insignificant recall increase, the runtime cost is negligible. As a result, the value with the highest recall (except $v_o = 1$) $v_o = 0.2$ was chosen.

- **Maximum distance** $v_d = 2$ (Figure 3.16d)
  $v_d$ controls the maximal distance two characters can have to be classified as neighbors. Thereby, the distance is defined as $h_r * v_d$. Although a line with all characters being detected, should not have gaps bigger than the height of the reference character, a higher tolerance allows the line detector to still detect a character even if the character detector did not find its direct neighbor. Furthermore, the gap between the digits and the descriptive information in the first two lines of UIC numbers can be multiples of the characters height (See figure 3.19). Therefore, it




(a) Average processing time

(b) Average recall

Figure 3.17: Behavior of Neumann and Matas line detector depending on the number of input characters (50% of dataset)

is desirable for the domain of UIC detection to detect lines with bigger than usual gaps even if it increases the number of false positives. To reflect this, $v_d$ was set to 2.

The results of the final line detector with the chosen parameters can be seen in table 3.7 under "ER-Speed - Custom".

The complexity difference to the NM line detector can be seen in figure 3.17a. Again, an ER character detector with the luminance channel $p_1 = p_2 = 0.9$ and varying threshold was used to alter the number of characters passed to the line detector. The graph shows that the runtime increase of the custom line detector is significantly less than the increase of the NM method. This enables this algorithm to be used with character detectors that produce many candidates. In figure 3.17a, the line detector was used with up to 1687 input characters, increasing the runtime of the whole pipeline from $125ms$ to $318ms$.

The recall of the character and line detector can be seen in figure 3.17b. Although the recall improvements stagnate with the recall improvements of the character detector, the process of selecting the best fitting characters out of multiple overlapping ones does pay off and the recall improves with the availability of more characters.

As shown in table 3.7, the recall difference between the character detector and the line detector could be reduced to 4.75% for the speed optimized system using only the luminance channel, and to 3.54% with the RGB channels and a scale pyramid. In addition, the precision increase that is expected due to the stricter filtering of characters is higher than the increase of the NM detector.

Furthermore, the lower complexity of the algorithm results in shorter processing times, which allows the application of the line detector to higher number of input characters. This allows us to compare the results for the speed, balance and recall optimized systems (Table 3.7). It can be seen that the speed system is the only one that benefits more from the use of the RGB color channels than the use of a scale pyramid. This indicates that the additional characters detected by the more recall optimized systems are overlapping for different color channels. The scale pyramid on the contrary, can help to find new characters that were not detected in a color channel of the original resolution.

Due to the lower runtime complexity of the algorithm, it is also possible to apply the line detector to all characters from different scales or color channels at once. Therefore, the line detector does not process the different versions of the input image individually and merges the lines at the end, but merges all characters and then creates one set of lines. For the application of the RGB color channel, this increases the number of characters passed to the line detector by a factor of 6 ($3 \times 2$ due to the required inversion

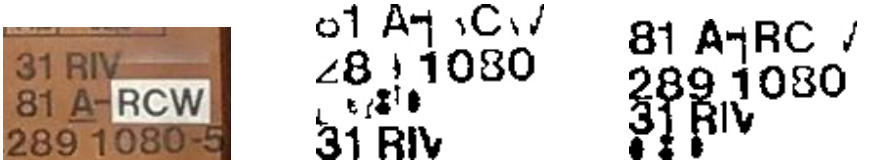
| System | Pre. | Rec. | F2 | Time | n |
|---|---|---|---|---|---|
| ER-Speed | 21.50% | 68.76% | 47.76% | 34 | 132 |
| ER-Speed + RGB + Scale | 24.39% | 79.56% | 54.77% | 181 | 1014 |
| ER-Speed + Custom | 35.07% | 64.01% | 54.95% | 37 | 23 |
| ER-Speed + Custom + Scale | 37.82% | 70.28% | 59.98% | 66 | 72 |
| ER-Speed + Custom + RGB | 34.16% | 70.92% | 58.36% | 105 | 70 |
| ER-Speed + Custom + RGB + Scale | 36.87% | 76.02% | 62.70% | 194 | 213 |
| ER-Speed + Custom + RGBD + Scale | 35.68% | 76.68% | 62.35% | 254 | 243 |
| ER-Speed + Scale + Custom | 35.41% | 64.55% | 55.43% | 50 | 23 |
| ER-Speed + RGB + Custom | 22.49% | 68.09% | 48.45% | 113 | 55 |
| ER-Balanced + Custom | 24.17% | 68.64% | 50.18% | 75 | 51 |
| ER-Balanced + Custom + Scale | 29.54% | 74.16% | 56.96% | 154 | 128 |
| ER-Balanced + Custom + RGB | 22.46% | 73.57% | 50.56% | 223 | 156 |
| ER-Balanced + Custom + RGB + Scale | 28.21% | 78.39% | 57.82% | 463 | 389 |
| ER-Balanced + Custom + RGBD + Scale | 26.73% | 79.62% | 57.05% | 584 | 479 |
| ER-Recall + Custom | 15.18% | 69.51% | 40.51% | 332 | 129 |
| ER-Recall + Custom + Scale | 21.11% | 75.43% | 49.80% | 663 | 249 |
| ER-Recall + Custom + RGB | 14.32% | 74.63% | 40.51% | 1049 | 407 |
| ER-Recall + Custom + RGB + Scale | 19.97% | 79.29% | 49.74% | 2096 | 780 |
| ER-Recall + Custom + RGBD + Scale | 19.13% | 81.51% | 49.34% | 2396 | 1009 |

Table 3.7: Results of the custom line detector

of the ER detector). Table 3.7 shows the results of the line detector being applied after the image was processed for all channels or scales. The number of characters returned from the line detector is similar to the number of characters returned by each individual line detection process before. Because the lines don't have to be merged with lines from other channels or resolutions, the final number of characters is lower. Note that a lower number of characters does not automatically mean that the precision is increased if only duplicates are eliminated. The opposite is the case. Both the recall and precision are lower if all characters are processed together. Although the selection of better characters within lines can help to improve the result, it also prevents the duplication of a line in the final result. By preferring certain characters, it is more likely that multiple line candidates contain the same character. The algorithm prevents lines with the same character from being added. Although this is a desired behavior, if the algorithm does not select the best line, the performance is decreased. By keeping lines from multiple versions more similar, characters are automatically grouped into the same lines and the final result contains multiple versions, which can be filtered in a later stage. If one's goal is to find lines and not UIC numbers, one might choose the combined line detection approach as it reduces the number of duplicates. An alternative would be to design a more sophisticated method of picking the better line.

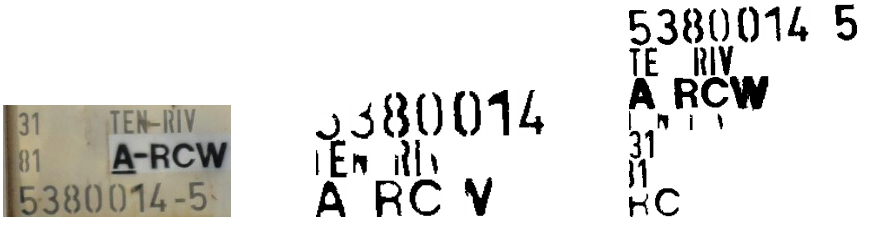
### 3.5.4 Comparison

Figure 3.18 - 3.20 show examples of lines extracted from the luminance channel using the speed optimized ER character detector. The results show the 5 longest lines found



(a) Input image (b) NM Line detector (c) Custom Line detector

Figure 3.18: Custom line detector picks higher quality characters



(a) Input image (b) NM Line detector (c) Custom Line detector

Figure 3.19: Custom line detector finds 2 digit groups

(a) Input image (b) NM Line detector (c) Custom Line detector

Figure 3.20: Custom line detector groups characters despite distance

by each detector.
Even though both detectors find all three lines in figure 3.18, the custom line detector selects higher quality characters. The characters returned by the NM detector can hardly be identified and are unlikely to be decoded by an OCR software. An additional advantage of the custom line detector is that it also detects groups of 2 characters. Despite a higher number of false positives that can be generated because of this rule, it is necessary for UIC numbers as shown in figure 3.19. Thereby, the big gap between the first two lines prevents a detection of the first 4 digits with the NM detector. Even though the custom line detector does not group the digits and the additional information as a line (desired behaviour), it detects the first four digits as groups of two. Finally, the new method is more tolerant towards gaps. Although this can lead to more false positives, figure 3.20 shows that it can help in case some of the characters were not found by the character detector.

Using a speed optimized ER detector with RGB color channels and a scale pyramid, the average recall (76.02% vs. 73.17%), precision (36.87% vs. 32.69%) and processing time (194ms vs. 558ms) all favor the custom line detector. As a result, it is used from this point onwards instead of the NM detector.
Note that these results only apply to the domain of UIC number detection. The custom line detector will perform worse if it is used to detect mixed case lines.

## 3.6 Optical Character Recognition

The purpose of this stage is to decode a character region into the corresponding digit. Although this could be done before the line detection stage, applying the OCR detector

to every region would result in a significant runtime increase. The benefit of this method would be that the line detector could use the confidence or lack of detection to pick the correct character.
Instead, the OCR detection is performed on lines as this reduces the number of characters that have to be processed. Furthermore, the OCR detection can even be performed after the UIC detection stage (See section 3.7).
However, a UIC number can be processed as an array of lines which means that the OCR stage can be designed independently of its location in the pipeline.

### 3.6.1 TesseractOCR

Due to time constraints, no specific OCR solution was created for the domain of UIC number detection, but the open source software TesseractOCR was used. For how to set up TesseractOCR, see section 3.1.1. Although OpenCV provides an interface for TesseractOCR, it was necessary to make some alterations to the implementation. The interface provides a wrapper around the TesseractOCR C++ library and allows the use of OpenCV data structures. Unfortunately, the complete feature set of the library was not implemented. Specifically, functionality to configure the adaptive classifier behavior was missing and had to be added.
Traditionally, TesseractOCR is used to process one or more lines at once and uses a dictionary to improve the recognition rate. It then creates a result consisting of a list of transcribed strings, bounding boxes and confidence intervals. Unfortunately, the results are returned per word/line and not per character.
For the application on UIC numbers, the detection was performed on each digit individually. This disabled the dictionary, which is a desired behavior as a UIC number does not follow a language specification. Furthermore, the library provides an interface to white list certain characters. This was used to only allow the characters “0-9” to be recognized. This effectively prevents errors like identifying a zero as an “O”. Note that this prevents the additional information on the first two lines to be correctly identified. If this information is required as well, it is recommended that this is done in a separate stage. In addition to using dictionaries, TesseractOCR can be trained to recognize new fonts. Regrettably, UIC numbers do not use a standardized font and can vary between different countries or companies. Therefore, the default model which was trained on a high number of fonts was chosen. If a bigger dataset was available, a custom model for UIC numbers could be trained.

TesseractOCR was originally intended to be used for the detection of printed text. As a result, it expects characters with a high contrast and quality. Although TesseractOCR is also capable of detecting lines, the original input images without significant preprocessing would not yield good results. Due to the nature of the character detectors used in the earlier stages, a black and white image can easily be created from the region information of a single character. These images are the ideal input for the OCR software.
Per default, TesseractOCR adapts its classification parameters after processing an image, assuming that further images will follow the same characteristics. If they do, the recognition rate can improve after multiple applications. If the image contains many false positives, it can lead to the creation of a false model, which can worsen the result. Furthermore, it makes the results for each image dependent on the images processed before. Processing an image a second time might lead to a different outcome than the first time. Although TesseractOCR provides a method to clear the model cache or completely disable adaptive thresholding, it is not implemented in OpenCV and had to be added manually.
Being able to clear the cache left the following options:

- Not clearing the cache between UIC numbers
  This can lead to inconsistent results if an image is processed as part of a dataset or individually.
- Clearing the cache for every UIC image
  This makes the processing deterministic but a high number of false positives can still lead to worse performance.
- Clearing the cache for each line
  If a character was found in two different color channels and has exactly the same shape, it might be correctly identified in one line but not in another.
- Disabling the cache
  This is leads to the same behavior as clearing the cache before processing each character. This way processing a character will always lead to the same result but the advantages of the adaptive thresholding are lost as well. Nevertheless, this method returned the the most consistent results and was chosen as it makes the evaluation of the results easier.

A new TesseractOCR model using neural networks and an LSTM is currently in development and available through beta channels. Nevertheless, it is an experimental feature and as it is not part of this thesis, the traditional Tesseract model was used in this work.

To evaluate an OCR software, character regions overlapping with ground truth characters were used. If the region of a detected character is overlapping with the bounding box of a ground truth character by more than 50%, it is expected to be decoded as the ground truth digit. To calculate the overlap ratio the following formula was used:

$$\frac{r_a \cap r_b}{r_a \cup r_b} > 50\% \tag{3.13}$$

Assuming that the ground truth character bounding boxes are not overlapping themselves, a character candidate can only be assigned to at most one ground truth character. All overlapping characters are used to calculate the accuracy of the OCR software. Thereby, the number of characters with the correctly identified digit is divided by the total number of characters overlapping with the ground truth bounding boxes. Note that the character candidates that are tested might not be identifiable as the correct digit. This can happen if the region only consists of a part of the character outline. This can reduce the maximal achievable accuracy. Furthermore, the result cannot be used to judge the accuracy of the final UIC detector. The tested character candidates might contain duplicates of the same digit. If at least one of the duplicates is identified correctly, it is still possible to identify the correct UIC number. On the other hand, a 100% accuracy only means that the found characters were identified correctly but they might only be a subset of all UIC digits.
Nevertheless, the accuracy allows the comparison of two OCR systems. Ideally the systems should be capable of identifying corrupted characters due to spray painting templates as described in section 3.5.1.

Because only the shape of the characters is important for the OCR process, all tests were only conducted with the luminance channel. It assumes that the characters extracted from the luminance channel will have a similar shape to the characters extracted from red, green or blue channel. The same argument is not valid about the scale pyramid. The shape of lower resolution characters can be very different from the original high resolution version. The OCR stage might have a high accuracy for high resolution characters, but can fail to identify lower resolution ones. By using a scaled pyramid for the tests, it is ensured that both types are weighted. The tests were run on 50% of the

| System | Rec. | Time | OCR |
|---|---|---|---|
| Speed+Tesseract | 64.01% | 104 | 80.26% |
| Speed+SVM | 64.01% | 40 | 92.75% |
| Speed+TS | 64.01% | 105 | 93.79% |
| Speed+Scale+Tesseract | 70.28% | 241 | 70.56% |
| Speed+Scale+SVM | 70.28% | 75 | 88.68% |
| Recall+Tesseract | 69.51% | 559 | 73.84% |
| Recall+SVM | 69.51% | 349 | 87.65% |
| Recall+TS | 69.51% | 543 | 89.54% |
| Recall+Scale+Tesseract | 75.43% | 1107 | 62.25% |
| Recall+Scale+SVM | 75.43% | 687 | 83.59% |
| Recall+Scale+TS | 75.43% | 1093 | 83.55% |

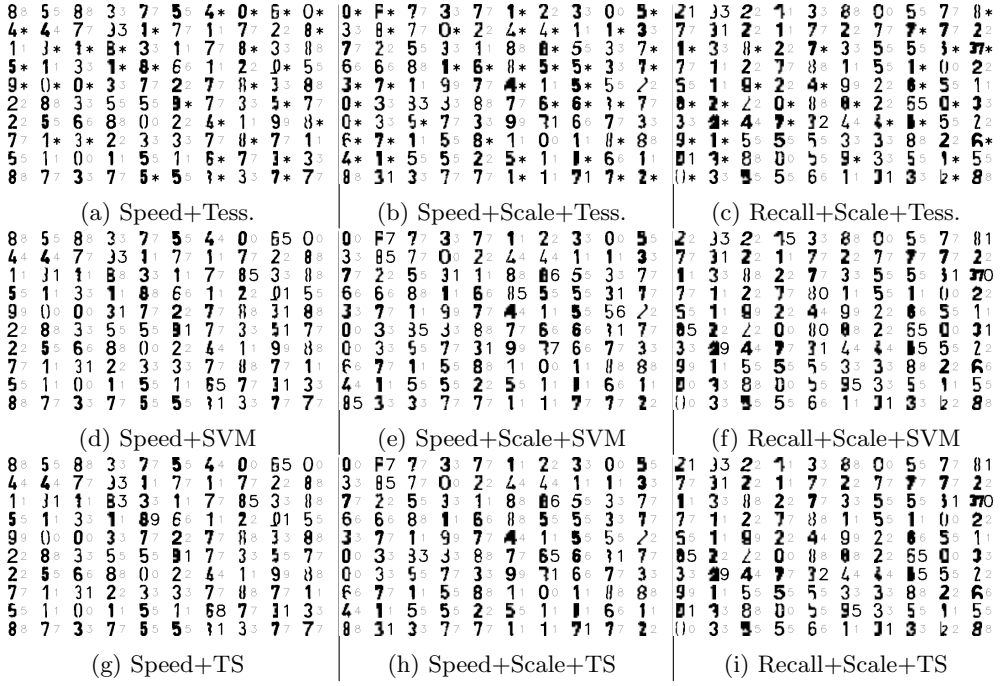
Table 3.8: OCR results (50% of dataset)

dataset.
A speed and a recall optimized ER character detector were used to test the performance on a system with a high and a low precision. By applying harsher filtering to the speed optimized system, the quality of the extracted characters is increased, which in turn simplifies the OCR task. By adding a scale pyramid or using the recall optimized system, more low quality digits are detected as well, which will decrease the performance of the OCR software.
Table 3.8 shows the results of both character detectors with and without a scale pyramid. As expected, the use of scale pyramids reduces the performance of the OCR detector. Furthermore, the filter system of the speed optimized system seems to effectively remove regions that are not a character. TesseractOCR is better at decoding the regions from the speed optimized system than the regions from the recall optimized system. Figure 3.21 shows an overview of characters processed with TesseractOCR and their assigned label. Note that although 3.21a does not use a scale pyramid, it still contains low resolution characters. This is due to UIC numbers with a low original resolution. Therefore, it is desirable for all pipelines to correctly recognize low resolution characters. A "*" is displayed instead of a digit when the decoder could not identify the character or its confidence is too low. This behavior is not helpful for the detection of multiple digits because a non-identified character cannot be corrected at a later stage (E.g. by using a dictionary). For the number detection, it is desirable that the algorithm always returns the character with the highest probability, even if has a low confidence. The results show that the digit recognition sometimes does not return a result for high quality characters and has a low success rate with low resolution characters. Due to the main objective of TesseractOCR lying in the recognition of words from a specific language, it was expected that the result could be improved by training a model purely for digit recognition.

### 3.6.2 Support Vector Machine

Although not originally planned, it was decided to train a custom digit classifier using OpenCVs support vector machine implementation. Thereby, the goal was to compare the results of a simpler method to TesseractOCR. Little time was spent optimizing the model or improving the dataset as the goal of this work is not a digit classifier but to investigate the feasibility of detecting UIC numbers. The results reported in this chapter can likely be improved further and are only meant as a proof of concept.

Because the digit wise annotation of the UIC numbers was only performed at a later stage, an artificial dataset was used to train a classifier for digit detection. To obtain a digit dataset, the installed system fonts were used (Including different font weights like



(a) Speed+Tess. (b) Speed+Scale+Tess. (c) Recall+Scale+Tess.

(d) Speed+SVM (e) Speed+Scale+SVM (f) Recall+Scale+SVM

(g) Speed+TS (h) Speed+Scale+TS (i) Recall+Scale+TS

Figure 3.21: Examples of ORC classifications

"Bold" and "Italic"). The fonts were then utilized to create images with the digits 0-9. Because some fonts were using the same digit style, duplicate images were removed. This resulted in a total of 3232 images. All images were then scaled to a size of 16x16. Using a low resolution was helpful to avoid overfitting and trained the model to recognize low resolution UIC numbers. To further increase the dataset size and force the model to recognize thick strokes that can be the result of using a scale pyramid, each character was duplicated and dilated by one pixel. For the training, a Support Vector Machine with a histogram intersection kernel and 10 fold cross validation was used. The resulting classifier could then be used to identify digits from the corresponding character regions.

The results using this method can be seen in table 3.8 and image 3.21. It can be seen that optimizing the model for digit recognition leads to superior results than using the default configuration of TesseractOCR. Although this could be improved by training TesseractOCR to recognize digits only, it would include a significant overhead due to its optimization for word recognition. The main advantage of the SVM approach is that it forces the model to pick one digit. As a result, even if the confidence is low, the classifier still has a chance to pick the right digit. This shows the advantage of a custom model for the digit recognition part. Furthermore, the average processing time is significantly less than the time required by TesseractOCR.

### 3.6.3 TesseractOCR + Support Vector Machine (TS)

Comparing the results in figure 3.21, shows that TesseractOCR's main problem is that it does not always return a result for a region. Nevertheless, the digits that were detected are mostly correct.
A disadvantage of the SVM method is that OpenCV does not expose the confidence of a prediction in contrary to TesseractOCR. Although this has no influence on the OCR result, it can be useful for the UIC detection stage.
By combining both approaches, TesseractOCR can be used to make an initial prediction.

If the result is not a digit, the SVM is used to return a digit and the confidence of the result is set to 1%. This has the advantage that every region is assigned a digit and a confidence score.
The results can be seen in table 3.8 and image 3.21 and outperform both individual methods. Furthermore, the performance impact of the additional SVM process is insignificant as it is only used when TesseractOCR fails to return a digit.

## 3.7 UIC Detector

The last step is to find a set of three lines containing the UIC number and to extract the correct sequence of digits. This process is made more complicated by missing or duplicated digits.
The designed algorithm can be split into the following two sub problems. The first problem is to find valid combinations of lines (Section 3.7.1). If a grouping of three lines was found, the UIC number can be extracted by using the first two digits of the first two lines and the first 8 digits of the last line. Due to the possibility of detecting multiple valid groupings, they have to be scored to select the most likely one (Section 3.7.2). Furthermore, the algorithm is designed to process lines with decoded characters and to perform the OCR internally to reduce its processing time (See section 3.7.3). Section 3.7.4 discusses the use of the UIC validation function to further improve the accuracy.

### 3.7.1 Grouping

For three lines to be accepted as a UIC number, they have to be within a certain area and have a similar height. Thereby, each line is considered as the top most line and used to find the second and third line. All lines are ordered from top to bottom. This way, only lines below the current line must be tested. Due to duplication of lines from different channels or scales, there might be multiple possible groupings for each line. With this in mind, a recursive algorithm was designed. Its goal was to find lines underneath it that can be part of the same grouping.
The algorithm receives the first line and a horizontal search area. This area has to include the start of every additional line and is used to reduce the number of lines that are processed. This area is estimated by using the average character width and the total length of the initial line. If the first line includes the first two digits and additional information, it is expected that all characters of the below lines are in a similar horizontal region. If the first line only contains the digits, the bottom lines might not be overlapping horizontally at all if the first digits of the lines below were not found. In the worst case, the first line only contains characters from the additional information and the lines below start to the left of the first line. To allow for these scenarios, $x_{min}$ and $x_{max}$ were calculated as follows:
Firstly, the number of characters that could be in the first line is computed by dividing the line width $l_w$ by average character width $\overline{c}_w$. To allow spacing between the characters, the character width is multiplied by 1.5. This is used instead of the number of regions in the line, to account for undetected characters. Assuming that the maximum number of digits in a line is 8, the number of missing characters can be calculated with $n = 8 - c_n$. Finally, this is converted to a width that should be used for the search $w = 1.5 * c_w * \max(2, n)$. To allow for errors, this is set to at least two characters. This results in the following estimated search area:

$$w = 1.5 * c_w * \max(2, 8 - \frac{l_w}{1.5 * \overline{c}_w}) \tag{3.14}$$

$$x_{min} = l_x - w \tag{3.15}$$

$$x_{max} = l_x + l_w + w \tag{3.16}$$

The recursive algorithm adds the current line to a list of potential UIC lines. If the list has a length of three, the current lines are used to extract and score a UIC number. The number with the highest score so far is returned as a result. If multiple numbers have the same score, all of them are kept.
If the list has less than three lines, the remaining ones have to be found. For this purpose, the original search area is restricted vertically with the assumption that the next line has to start below the current line and that they cannot be too far apart:

$$y_{min} = l_y + l_h \tag{3.17}$$
$$y_{max} = l_y + 2 * l_h \tag{3.18}$$

Using the previously top to bottom sorted list, and the index of the current line, it is easy to only check lines that are below the current one. Furthermore, lines that start below the search area can be ignored.
If a line's top left corner is in the search area and it has a similar height, it is used as a potential match. Combined with the previously found lines, it is passed to another call of the recursive algorithm. The line has a similar height if the following conditions are met ($h$ is the height of the current line and $\overline{c}_h$ is the average character height in the first line):

$$0.5 * \overline{c}_h < h < 1.5 * \overline{c}_h \tag{3.19}$$

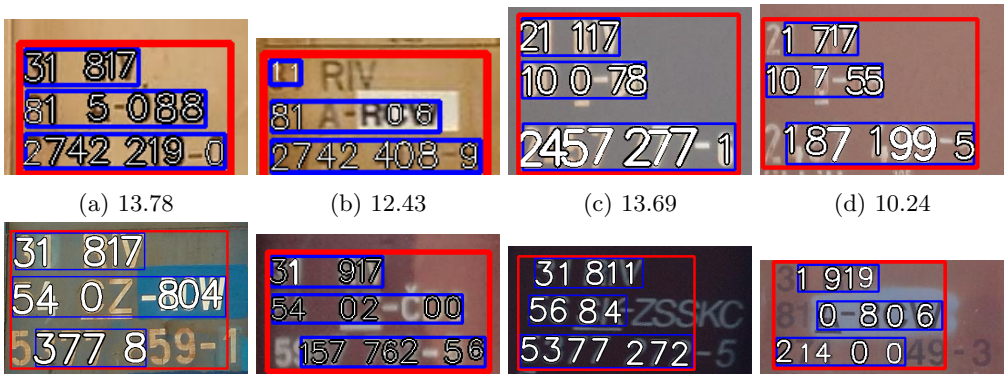
### 3.7.2 Scoring

The basic idea of the scoring function is to use the UIC number with the highest OCR confidence for its digits. To support results with missing digits, the concept has to be extended. Especially, digits missing at the beginning of a line can be hard to detect. In the worst case they could lead to the additional information in the first two lines to be interpreted as digits instead. To reduce the runtime, it is also desirable to score UIC numbers before the OCR step was performed. For this reason, the score was based on an estimated number of found characters. If the characters were decoded by an OCR process, the score can be refined by using the confidence of the predictions. The exact score can be calculated as follows:

1. First, the left most position $x_{min}$ of all three lines is calculated. Ideally all three lines start at the same X position, but if characters at the beginning of a line are missing, they might not match up. In this case, $x_{min}$ can be used to estimate how many digits are missing at the beginning of a line.

2. Using $x_{min}$ and the maximum length of the line $n_{max}$, separate values for each line can be calculated. The maximum number of UIC digits that can be expected considering a line shift to the right $n_s$ is calculated. This is the number of regions found in the line but cannot exceed $n_{max}$. Thereby, the X position of the line $l_x$ and the average character width $\overline{c}_w$ of the line are used. $n_s$ combined with the number of regions in the line can be used to calculate the expected number of regions part of the UIC number.
   If the digits in the line have a prediction and a confidence, the average confidence of its characters $\overline{p_c}$ can be calculated. If a character is not a digit, its confidence is set to 0. It is expected that the confidence is provided as a value between 0 and 1. In addition to that, the number of digits $n$ can be refined by subtracting the number of regions that were not classified as a digit $n_e$(e.g. by TesseractOCR):

$$n_s = \max(0, n_{max} - \max(0, [\frac{l_x - x_{min}}{1.5 * \overline{c}_w}])) \tag{3.20}$$
$$n = \min(n_l, n_s) - n_e \tag{3.21}$$
$$\tag{3.22}$$



(a) 13.78 (b) 12.43 (c) 13.69 (d) 10.24

(e) 8.47 (f) 11.72 (g) 10.79 (h) 7.29

Figure 3.22: Examples of extracted UIC numbers and their score

3. Having calculated the estimated number of characters $n$ and the average confidence $\overline{p}_c$ for each line, the total score $s$ can be calculated as follows:

$$s = n_1 + n_2 + n_3 + \frac{\overline{p}_{c,1}/2 + \overline{p}_{c,2}/2 + \overline{p}_{c,3}/8}{12} \tag{3.23}$$

If no OCR was performed yet, $\overline{p}_c$ is set to 0. The score can be interpreted as the estimated number of characters, plus the confidence of the digit predictions as a decimal value.

For examples of extracted UIC numbers and their score value, see figure 3.22. Figure 3.22d shows that the algorithm correctly identifies the missing first digit in the first two lines and does not count the "7" as the second digit of the first line. Figure 3.22e shows that the number of missing leading characters only has an impact on the score if the line contains too many characters (Figure 3.22f). Examples of the score not corresponding to the true number of missing digits, can be seen in image 3.22g and 3.22h. The horizontal shift of the lines in figure 3.22g causes the algorithm to believe that the first line is missing the first digit. Nevertheless, the final UIC extracted UIC number correctly identifies the first 11 digits. The algorithm always picks the first two digits of the first two lines and the first 8 of the last line, assuming that returning an incorrect or a missing digit leads to the same edit distance. Finally, the second line in figure 3.22h shows that the count of missing leading characters can fail if the average character width of the line is not a good representation of the digit width. In this particular case, the bigger width of the letters in the additional information causes the algorithm to believe that only one digit is missing at the beginning of the second line.

### 3.7.3 OCR

Because OCR has a significant performance impact, its use should be avoided on lines which are not part of the UIC number. To optimize the performance of the whole UIC pipeline, the UIC detection stage was designed to only perform OCR on a subset of lines. Thereby, all potential line groupings are initially scored without the digit confidence part. Normally, only the grouping with the highest score would be kept. Not having an OCR result available means that the score is an integer representing the estimated number of digits available. This makes it likely for multiple groupings to have the same score. In this scenario, all of the groupings have to be kept. After scoring each line, optical character recognition is performed on all lines part of the highest scoring line groupings. The disadvantage of this approach is that a line grouping that contains 12 "A"s might be assigned a higher score than a grouping that contains 11 digits, leading to only the "A"

grouping being recognized. The advantage of this approach is that it greatly reduces the numbers of lines requiring OCR. Finally, the groupings are re-scored using the new predictions and confidence values and the highest scoring grouping is returned as a UIC number (For a comparison see figure 4.2).

### 3.7.4 Boosting valid UICs

A possibility for further optimization is the use of a UIC numbers check digit, previously discussed in section 1.1.1. Thereby, the score of a line grouping can be boosted by one if the extracted UIC number is complete and valid. In the scenario that two groupings contain the same number of digits but different numbers were extracted, this method can increase the likelihood of selecting the right number. For an example see figure 3.22a. The disadvantage of this method is that it also boosts results that are incorrect but by chance, contain a sequence of digits that passes the validation. This can lead to an unusually high number of false positives that cannot be rejected by using the check digit (Figure 3.22c).
Using the speed optimized ER detector with a scale pyramid and the luminance channel only, the ratio of correctly identified UIC numbers could be increased from 36.99% to 46.23% by bossing the score (50% of dataset). At the same time, the number of false positives increased from 2.40% to 9.59%.
Ultimately, the decision lies with the user, depending on his weighting of recall and precision.

## 3.8 Further optimizations

### 3.8.1 White list

In some cases, a list of valid UIC numbers might be known in advance. For example, it might be known that one train contains wagons with defined UIC numbers, or all UIC numbers of trains arriving within a certain time frame are known. In these cases, the list of numbers can be used to increase the accuracy of the detector by correcting mistakes of the detector. For this purpose, the edit distance of an extracted number is calculated for every white listed number. The number with the lowest edit distance is selected. If multiple numbers have the same edit distance, the extracted number is returned.
Note that this method should not be combined with the score boosting described in section 3.7.4. Furthermore, both of these methods prevent the validation of the final result with the check digit. They increase the chance of returning a valid UIC number even if some of the digits were identified incorrectly.
The white list could be used more extensively by using the confidence of the digits to compare two numbers with the same edit distance. But by correcting more of the extracted numbers, the likelihood increases that an incorrect number was chosen from the white list.

### 3.8.2 Two stage pipeline

The components of the pipeline were parameterized to optimize either processing time or recall. For example, the speed optimized ER detector is capable of localizing a UIC number in a short period of time, but has a low chance of finding each individual digit. On the contrary, the recall optimized system with multiple color channels and a scale pyramid, can take multiple seconds to detect a UIC number, even in small images. Its advantage is that it finds a bigger portion of the digits, which can reduce the edit distance. With the advantages of both systems in mind, the following method was designed:

The two stage pipeline requires two UIC detectors. One fast, and one with a high recall. The fast UIC detector is used to find the location of the UIC number and the slow detector restricts its search area to the found location. Especially for input images with only a small region containing the UIC numbers, this can speed up the process significantly. In detail the algorithm works as follows:

1. The fast UIC detector is used to extract an initial guess of the UIC number. If its result is empty, the slow detector has to process the whole input image. If the returned UIC number is valid, it can be returned directly without a second step, otherwise the input image for the recall optimized detector has to be prepared for the second run.

2. Even if a UIC number was detected in the first run, its bounding box might not include all digits. Furthermore, the character detector requires a margin around characters to find them. For this reason, a new bounding box was generated for each line. Each new bounding box starts at the left most X position of all three lines minus a margin. It is assumed that at least one of the lines contains the first character. The margin is calculated by dividing the height of the lines bounding box by 8 ($m = h/8$). The height of the new bounding box is set to the original height plus two times the margin. The length is set depending on the number of digits in the line $n$ (2,2,8). To allow spacing in between characters, the length is set to $h_l * n + 2 * m$. Using the bounding boxes of the three lines, a new image is generated and the content of the lines is copied. The area not part of the lines is set to black. The size of this new image is set to the minimal area containing all three lines. This image is then used by the recall optimized UIC detector to find the remaining digits.

3. To reduce the processing time further, the character detector of the second stage detector can be wrapped in a filter that only accepts characters that have a similar height to the previously extracted lines. In particular, the characters were rejected if $h_c < 0.7 * h_l$ or $h_c > 1.5 * h_l$. ($h_c \ldots$ Character height, $h_l \ldots$ Line height). This reduces the number of characters that have to be processed by the line detector and the OCR stage.

# Chapter 4

# Results

Combining all components from the pipeline described in section 3, a UIC detector can be created. Two compare multiple UIC detector configurations, they were tested with the remaining 50% of the dataset. Even though the optimizations of the component were limited, they could have caused overfitting to the previously used data. Bear in mind that even though the remaining 50% have not been used in any previous runs, the complete dataset contains multiple versions of the same UIC number. Thus, the training and test set could include the same UIC number, recorded in different conditions. All results were generated with the custom line detector and OCR was only performed on UIC numbers with the most characters unless stated otherwise. The combined TS OCR detector was used.
For the comparison of UIC detectors, the following values were used:

- **Processing time** Average time it takes to process a single image.
- **Found** The percentage of UIC numbers that are overlapping at least 50% with the bounding box of the ground truth.
- **Accuracy** The percentage of UIC numbers that were extracted correctly (All 12 digits).
- **Score** Combines the result of the edit distance for every image into one value. Can be understood as the area of a bar chart showing the cumulated edit distance. $e_i$ is the number of results with an edit distance of $i$. Therefore, $e_0$ is the number of correctly identified UIC numbers and $e_{12}$ is the number of results that incorrectly identified every single digit. With the number of entries not returning a result $n_e$, the score $s$ can be calculated as follows:

$$s = \sum_{i=0}^{12} \frac{\sum_{j=0}^{i} e_j}{n_e + \sum_{m=0}^{12} e_m} \tag{4.1}$$

The results of different UIC detector configurations can be seen in table 4.1.

## 4.1 Character detector

Table 4.1 shows the results of different speed-, balance- and recall-optimized systems. Even though the recall optimized character detector performed the best during the character detection test (Section 3.4.1), it can be seen that the highest score is achieved by the balanced system and the highest accuracy by the speed optimized system. The higher recall of the character detector does result in a higher ratio of found UIC numbers. At the same time, their lower quality reduced the performance of the OCR detector,

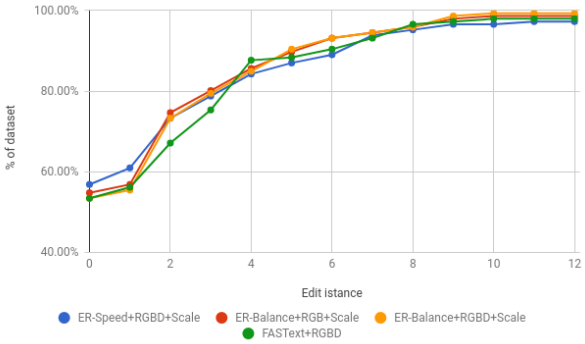



Figure 4.1: Edit distance of different UIC detectors (50% of dataset, UIC images)

which results in a lower accuracy. Furthermore, the runtime of the pipelines using the recall system vastly exceed the runtime of the remaining systems. As a result, it is not recommended to use the recall character detector in a final system.

The fastest system (83ms) is the speed optimized character detector without the use of color channels. Unfortunately, it also performs the worst in all other categories, achieving an accuracy of only 28.77%. Nevertheless, the ratio of found UIC numbers is 82.88%, which makes it a good candidate for the use in a two stage UIC detector. The ratio of found characters can be increased further by adding a scale pyramid. Note that the scale pyramid is faster and results in more found UIC numbers than the use of the RGB color space, but the higher resolution of the regions extracted from the RGB channels simplifies the OCR detection and results in a higher accuracy.
Furthermore, the stricter filtering of the speed optimized system increases the quality of the extracted regions. As a result, the combination with the RGB, gradient channel and scale pyramid results in the highest accuracy of any tested UIC detector. Its processing time (605ms) is significantly less than systems with a similar accuracy (e.g. “ER-Recall+RGBD+Scale” with 2806ms and 55.48% accuracy).
Nevertheless, the stricter filtering also prevents hard to read characters from being found, which reduces its accuracy. The highest ratio of found UIC numbers (99.32%) is achieved by “ER-Balance+RGBD+Scale+TS” and “ER-Recall+RGBD+Scale+TS” but the balanced system is significantly faster and has a higher accuracy. Although the systems find the most UIC numbers, the balanced system without the gradient channel has a slightly higher score, with 86.09% vs 85.93%. The difference in score can be explained by looking at the edit distance of the best performing systems in figure 4.1. The balanced system without the gradient channel starts with a higher accuracy and is only overtaken by the gradient system at an edit distance of 5.
Although the speed optimized system starts of the best, the other systems find more UIC numbers with edit distances of more than 2, which results in a lower score.

In comparison to the Extremal Regions detector, the FASText detector performs significantly worse in terms of speed, and detected UIC numbers. The lower quality of the extracted regions makes it more difficult for the OCR detector to extract the correct digits. This in turn, reduces the chance of correctly extracting all 12 digits, which results in a low accuracy. Nevertheless, the found digits result in a high score and ratio of found UIC numbers.
Initially, the high score and ratio of found UIC numbers make the direct color channel application and non max suppression (”FASText3-NMS”) a good candidate for the first stage of a two stage UIC detection system. Applying the detector to images of the train wagons, reveals that the processing time grows faster for larger images than with the

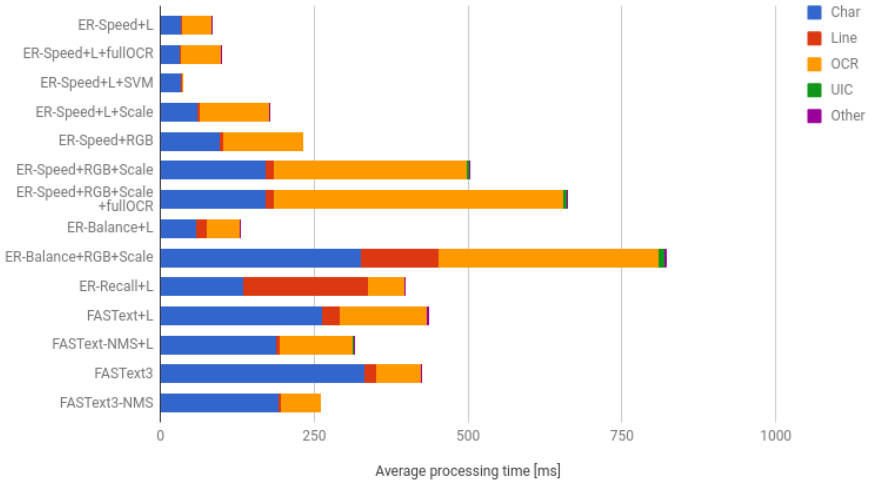



Figure 4.2: Processing time of UIC detectors (50% of dataset, UIC images)

use of an ER detector.

## 4.2 Optical character recognition

By looking at the processing time distribution of the UIC detectors in figure 4.2, it becomes clear that it takes up a significant portion of the total processing time. At the same time it can be seen that the time is reduced by performing the OCR step only for the best line groupings. See “ER-Speed+L” and “ER-Speed+GRB+Scale” with and without “fullOCR”. Table 4.1 also shows that this optimization step has no impact on the returned results. Nevertheless, the time difference is not as significant as expected. Using the UIC images instead of images of the train wagon causes most of the extracted lines to be part of the UIC number. The high duplication due to different channels and scales causes a high number of line groupings with the same number of found digits. This results with the UIC detector performing OCR for all of them. The performance increase is significantly higher when used on images picturing a whole train wagon (See figure 4.3).

One might be tempted to use the SVM, or Tesseract only classifier instead of the combined method. As previously stated, the runtime improvement of using Tesseract only is insignificant and does not outweigh the loss in accuracy. Although the accuracy and score loss of the SVM detector prevents it from being used as a stand alone pipeline, the speed increase could make it a good replacement for finding the initial location of the UIC number. As seen in figure 4.3, this drastically decreases the runtime of the OCR stage for UIC images. Unfortunately, when used on images of a train wagon, it has a less dramatic impact on the total runtime of the UIC detector. Due to the bigger images, the character detector becomes the slowest part of the pipeline. In comparison, the OCR processing time difference is minimal. The result is only a 5.45% total time decrease for the “ER-Speed+L” UIC detector and a 3.64% decrease for “ER-Speed+L+Scale” compared to the combined character detector.

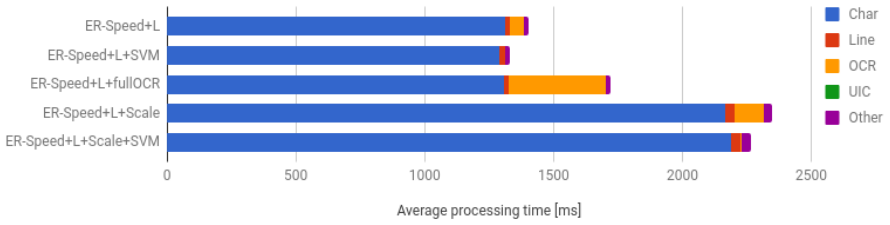



Figure 4.3: Processing time of UIC detectors (50% of dataset, Train images)

## 4.3 Two stage pipeline

Looking at the processing time of the "ER-Speed+RGBD+Scale" detector for train images, it becomes clear that an average of 9333ms is too long for a practical system. To reduce the runtime, a two stage system was used. Thereby, the location of the UIC number is found by a fast system, and the final digit extraction is performed by a second, more accurate, but slower system. If the first system does not find a UIC number, the slow system has to process the whole image, which increases the average processing time. On the contrary, if the first system already finds a valid UIC number, the second step can be skipped, reducing the average time.
Systems that are fast and perform well at finding UIC number locations are:

- **ER-Speed+L** This is the fastest tested detector. The average time to process a UIC image is 1391ms with combined OCR and 1319ms with a SVM classifier. It finds 72.6% of UIC numbers.
- **ER-Speed+L+Scale** Although this system is slower (2333/2251ms), it finds more UIC numbers which can reduce the number of times the slow system has to process the whole image.

Although the use of the SVM classifier reduces the processing time, it also decreases the accuracy, making it less likely that the second stage can be skipped.
For the second stage classifier, the following three systems were used:

- **ER-Speed+RGB+Scale** Provides the best compromise between accuracy and speed.
- **ER-Speed+RGBD+Scale** Achieved the highest accuracy for UIC images.
- **ER-Balance+RGB+Scale** Achieved the highest score, but it is also the slowest of the second stage systems.

For the second stage system, the combined OCR stage was used. The results for the different combinations can be seen in table 4.1.

Comparing the first stage classifier with combined OCR and a SVM classifier shows that the time saved by not having to use the second stage detector is not enough to cancel out the additional time required for TesseractOCR. The same phenomenon can be seen when using a scale pyramid for the first stage detector. It does not impact the accuracy or the score of the results but it increases the runtime.

The best combination of UIC detectors is "ER-Speed+L+SVM and ER-Balance+RGB+Scale". Although "1936ms" is still a long time for processing a single image, it is significantly faster than using a single UIC detector with a similar accuracy. Figure 4.4 shows examples of UIC numbers extracted with the detector.

(a) 318127400592 (2) (b) 318153753302 (0) (c) 318153802646 (0) (d) 318127400543 (9)

(e) 318153771080 (0) (f) 318128910177 (10) (g) 335653800300 (2) (h) 318153760505 (0)

(i) 318153801077 (0) (j) 218024572776 (2) (k) 378053770939 (0) (l) 318153803909 (2)

(m) 318153765181 (0) (n) 318127420608 (4) (o) 318153802588 (1) (p) 378053757191 (0)

(q) 318153752411 (8) (r) 318127416044 (9) (s) 318153803693 (3) (t) 318153801101 (0)

(u) 318153772294 (0) (v) 318127412381 (9) (w) 318153802653 (10) (x) 378053757175 (0)

Figure 4.4: Examples of UIC numbers and their edit distance extracted from train images with "ER-Speed+L+SVM and ER-Balance+RGB+Scale"

## 4.4 Comparison

To compare the own results to the results reported for the "ARVIS" system mentioned in section 1.2, only a subset of the own dataset can be used. Peter Honec reported that their system used input images with 3x3mm per pixel. Assuming that a UIC digit is approximately 100mm height, every digit has a height of 33px. Comparing this to the new UIC dataset shows that the quality of the used dataset is significantly lower. The used dataset only contains 45.2% of images with a digit height of at least 33px. To compare the results of the two systems, images with a too small resolution were filtered out and images with a higher resolution were scaled down. The optimized dataset contains 132 images. Applying the "ER-Speed+L+SVM and ER-Balance+RGB+Scale" system to it results in a 64.39% accuracy with a score of 79.43%. 82.58% of the extracted UIC numbers were overlapping with at least 50% with the ground truth location. The average processing time is 1978ms. Although this result is significantly below the 96% accuracy reported about "ARVIS", no illumination unit was used to record the images. Furthermore, the image quality of the stationary unit is likely to be higher than the quality of the own dataset.

| System | Time | Found | Score | Accuracy |
|---|---|---|---|---|
| **UIC image:** | | | | |
| ER-Speed+L | 83ms | 82.88% | 69.07% | 28.77% |
| ER-Speed+L+SVM | 37ms | 82.88% | 67.91% | 24.66% |
| ER-Speed+L+fullOCR | 99ms | 82.88% | 69.07% | 28.77% |
| ER-Speed+L+Scale | 177ms | 90.41% | 77.66% | 43.84% |
| ER-Speed+RGB | 230ms | 89.04% | 77.82% | 47.95% |
| ER-Speed+RGB+SVM | 210ms | 97.26% | 80.03% | 42.47% |
| ER-Speed+RGB+Scale | 500ms | 97.26% | 84.83% | 55.48% |
| ER-Speed+RGB+Scale+SVM | 210ms | 97.26% | 80.03% | 42.47% |
| ER-Speed+RGB+Scale+fullOCR | 658ms | 97.26% | 84.83% | 55.48% |
| ER-Speed+RGBD+Scale | 605ms | 97.26% | 85.14% | 56.85% |
| ER-Balance+L | 129ms | 86.99% | 73.76% | 32.88% |
| ER-Balance+L+Scale | 286ms | 95.89% | 82.67% | 48.63% |
| ER-Balance+RGB+Scale | 817ms | 98.63% | 86.09% | 54.79% |
| ER-Balance+RGBD+Scale | 983ms | 99.32% | 85.93% | 53.42% |
| ER-Recall+L | 395ms | 92.47% | 75.34% | 32.19% |
| ER-Recall+L+Scale | 772ms | 97.95% | 81.24% | 47.26% |
| ER-Recall+RGBD+Scale | 2806ms | 99.32% | 83.93% | 55.48% |
| FASText+L | 432ms | 96.58% | 81.88% | 46.58% |
| FASText+RGBD | 1932ms | 97.95% | 84.56% | 53.42% |
| FASText-NMS+L | 313ms | 91.78% | 78.19% | 39.04% |
| FASText-NMS+RGBD | 1329ms | 95.89% | 83.3% | 52.05% |
| FASText3 | 422ms | 95.89% | 83.09% | 41.1% |
| FASText3-NMS | 259ms | 93.15% | 80.51% | 36.3% |
| **Train image:** | | | | |
| ER-Speed+L | 1391ms | 72.6% | 65.07% | 32.88% |
| ER-Speed+L+SVM | 1319ms | 72.6% | 64.07% | 28.08% |
| ER-Speed+L+fullOCR | 1708ms | 72.6% | 65.07% | 32.88% |
| ER-Speed+L+Scale | 2333ms | 80.14% | 71.07% | 39.04% |
| ER-Speed+L+Scale+SVM | 2251ms | 80.14% | 67.49% | 35.62% |
| ER-Speed+RGBD+Scale | 9333ms | 89.04% | 80.08% | 50.0% |
| ER-Balance+RGB+Scale | 9211ms | 88.36% | 77.61% | 49.32% |
| FASText+L | 3103ms | 87.67% | 73.34% | 34.25% |
| FASText3+SVM | 5404ms | 84.93% | 65.7% | 25.34% |
| ER-Speed+L and ER-Speed+RGB+Scale | 1810ms | 71.23% | 64.91% | 43.15% |
| ER-Speed+L+SVM and ER-Speed+RGB+Scale | 1777ms | 71.23% | 64.7% | 43.84% |
| ER-Speed+L+SVM and ER-Speed+RGBD | 1906ms | 72.6% | 65.75% | 43.84% |
| ER-Speed+L+SVM and ER-Balance+RGB+Scale | 1936ms | 76.03% | 68.76% | 45.21% |
| ER-Speed+L+Scale and ER-Speed+RGB+Scale | 2632ms | 71.23% | 64.91% | 43.15% |
| ER-Speed+L+Scale+SVM and ER-Speed+RGB+Scale | 2449ms | 71.23% | 64.7% | 43.84% |
| ER-Speed+L+Scale+SVM and ER-Speed+RGBD+Scale | 2556ms | 72.6% | 65.75% | 43.84% |
| ER-Speed+L+Scale+SVM and ER-Balance+RGB+Scale | 2636ms | 76.03% | 68.76% | 45.21% |

Table 4.1: Results of UIC detectors (50% of dataset, UIC images)

# Chapter 5

# User interfaces

## 5.1 UIC Annotation tool

This utility tool was developed to annotate images with their corresponding UIC numbers and to export the annotated information as a dataset in a readable format to the UIC detection program. The program was developed in Java. The annotation tool can be seen in figure 5.1. To annotate raw images they have to be moved into the "to_import" project directory. Once opened, the program will scan the directory and display the first entry in the top left panel of the displayed window. To create multiple cropped versions of the same image with the UIC number taking up more and more space, the user has to mark the area of the train wagon, the UIC number and all digit groups. Thereby marking the train wagon shows the marked area in the bottom left panel which allows a easier selection of the UIC number. Thereby the number does not have to be selected perfectly as its exact bounding box can be calculated from the combined bounding box of the digit group. Next the image of the UIC number is shown in the top right panel where the user can mark every individual digit group. It was decided to only mark digit groups to reduce the time required for the annotation process in contrast to annotating every digit separately. Knowing the number of digits in each line, the groups can automatically be split into individual digits.
Once the UIC number rectangle was selected, the zoomed in version can also be used to identify the digits and write them into the first text field in the bottom left panel. Finally the user can enter a subjective quality measure in between 1-5 into the second text field (1 corresponding to a high quality). Once the annotation is completed, the user can save the information by pressing enter. To avoid annotation mistakes while

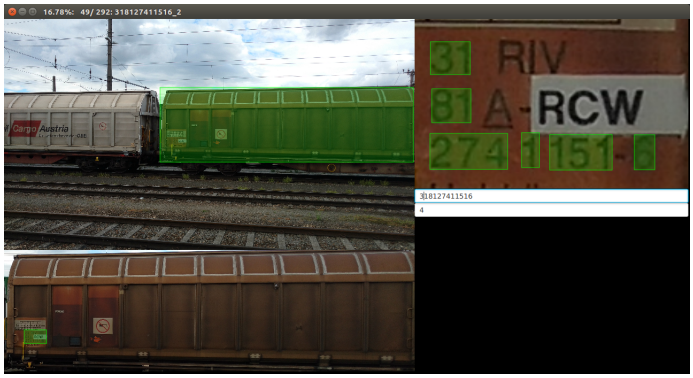



Figure 5.1: GUI of the UIC annotation tool

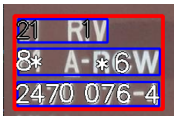
typing the UIC number, the program uses the check digit to verify the number. If the information is correct the program shows the next image to import, otherwise it keeps showing the same image. If all images from the "to_import" folder were annotated, the program shows the complete dataset with the previous annotations. The left and right key can be used to navigate through the images and alterations can be saved with the enter key.

## 5.2 UIC detector CLI

The finished program provides a command line interface that can be used to process a single image or to test a UIC detector on a specified dataset. Thereby, the program provides a choice of different character detectors, which can be configured by passing a selection of color channels that should be used for the detection. Both functions can be further configured with the following options:

- "–**whitelist arg**" or "**-w arg**" Can be used to pass a list of comma separated uic numbers that will be used during the detection process.
- "–**detector arg**" or "**-d arg**" With this option a specific UIC detector can be selected.
- "–**channels arg**" or "**-c arg**" The argument passed defines the color channels that the detector should use. Note this option has no effect on a FASText3 detector as it does not use color channels. The channels can be passed as keys without a delimiter (e.g.: "-c RGBD" for red green blue and gradient channel). The possible channel are:
  - **R** ... Red channel
  - **G** ... Green channel
  - **B** ... Blue channel
  - **H** ... Hue channel
  - **S** ... Saturation channel
  - **L** ... Luminance channel
  - **D** ... Gradient channel
- "–**ui**" or "**-u**" Enables the graphical user interface. Both, the single and the dataset mode offer a GUI that allows further analysis of the returned results. For this purpose the following images are shown after processing an input (Figure 5.2):
  - **UIC number** (Figure 5.2b)
    This image shows the bounding box of the UIC number as a red rectangle and the bounding boxes of each line as blue rectangles. The detected digits are overlaid directly over the bounding box of the corresponding character which allows a user to identify missed characters.
  - **UIC outline** (Figure 5.2c)
    Figure 5.2c shows the outline of all characters part of the detected UIC number. This can be helpful to see the quality of the returned characters.
  - **UIC characters** (Figure 5.2d)
    The last image that is shown to the user, shows the outlines of each detected character but adds a gap in between them. This can be useful to identify characters that are linked to each other, which can prevent the OCR software to identify them correctly.

(a) Original image shown with the use of an annotated dataset



(b) The extracted UIC number

(c) Outline of all digits in the extracted UIC number

(d) All characters part of the extracted UIC number

Figure 5.2: Optional GUI of the command line interface

### 5.2.1 Single image mode

If the program was started with the path of a "JPG" or "PNG" it will process the image and return the detected UIC number on the command line. This function can be used in a production system where only the result is of interest. If the GUI is enabled, three windows are opened at the end of the detection process, allowing a in depth analysis of the result (For examples see figure 5.2). In addition to the previously mentioned parameters, this mode also supports the following options.

- "–**print-rect**" or "**-r**" If enabled the program will return the detected UIC number and its bounding box. The bounding box is returned as a comma separated list ("x,y,width,height"). This can be helpful if an image of only the UIC number should be saved as a reference or should to presented to a human to identify missing digits.
- "–**print-timings**" or "**-t**" With the print timing flag the program returns the runtime of each pipeline component.

### 5.2.2 Dataset mode

If the program is started with a path to folder, the program reads the dataset in that folder and tests the passed UIC detector with it. The dataset needs to be provided in the format described in section 1.1.2. In addition to the parameters the dataset mode shares with the single image mode, it also allows the user to define which part of the image should be used during the detection process. The area can be specified with the "–**region** arg" option. The possible selections are:

- **original** . . . Full size image
- **train** . . . Cropped version, containing the train wagon
- **uic** . . . Selection roughly containing the UIC number
- **digits** . . . Contains only the bounding box of the UIC number

When the dataset is evaluated the current process is shown on the command line. An example of this can be seen on line 1-7 in listing 4 (Line 1-7). The first value shows the process as a percent value, the second value is the index of the current entry. Then the UIC number and the overlap of the extracted bounding box with the ground truth are listed. The final value for each entry is the edit distance of the extracted UIC number. If no number was found at all, $-1$ is shown instead.
Once the process is completed, the result is summarized as seen on lines 9-40 in listing 4. The shown information is:

```
0.00% :   0 218024572776_1 -> Overlap: 96.51% Edit Distance: 3
0.34% :   1 218024572776_2 -> Overlap: 95.23% Edit Distance: 1
0.68% :   2 218024574905_1 -> Overlap: 98.97% Edit Distance: 7
...
98.97% : 289 378053771747_2 -> Overlap: 99.10% Edit Distance: 0
99.32% : 290 378053771887_1 -> Overlap: 96.66% Edit Distance: 2
99.66% : 291 838027450652_1 -> Overlap: 0.00% Edit Distance: -1

Detector: FasTextDetector1C -> CustomLineDetector >> Channels ->
    SimpleUicDetector >> Scale Pyramid
Total Time: 61440
Mean Time: 210
48105    FasTextDetector1C
10883    TesseractSingleCharClassifier
 1829    FasTextDetector1C -> CustomLineDetector
   16    FasTextDetector1C -> CustomLineDetector >> Channels ->
       SimpleUicDetector

   Overlap          Cnt           %
      <30%           31      10.62%
      >30%            3       1.03%
      >50%          258      88.36%

EditDistance           Cnt.          %     Cumulated
         0             40      13.70%      13.70%
         1             35      11.99%      25.68%
         2             52      17.81%      43.49%
         3             45      15.41%      58.90%
         4             26       8.90%      67.81%
         5             22       7.53%      75.34%
         6              7       2.40%      77.74%
         7              8       2.74%      80.48%
         8             10       3.42%      83.90%
         9             11       3.77%      87.67%
        10              4       1.37%      89.04%
        11              1       0.34%      89.38%
        12              0       0.00%      89.38%
         -             31          -%      10.62%
Area:     67.89%
CriticalErrors:          1            0.34%
Avg (overlapping only): 3.12644
Avg (all): 4.06849
```

Listing 4: Result of evaluating a dataset on the command line

1. The used UIC detector.
2. The total time needed to process the whole dataset in milliseconds.
3. The average time one entry in the dataset required to be processed in milliseconds.
4. The total times required by the subcomponents of the UIC detector in milliseconds. Note that all these values might not sum up to the total time previously stated, because not all stages such as spitting the image into color channels are tracked.
5. Line 17-20 show how many UIC bounding boxes were overlapping less than 30%, 30-50% and more than 50% with the ground truth.
6. Line 22-36 show the distribution of the edit distances of the results. The first value shows the edit distance. The next value, shows the number of entries with that result. The penultimate value shows the same value as a percentage and the last value shows the cumulated percentage value of entries with the same or less edits. The last column shows the number of entries where no UIC number was found.
7. The area is the area of the cumulated line chart of the edit distances divided by the maximal possible area of this chart (See equation 4.1).
8. Critical Errors shows the number of UICs that were detected incorrectly, but have a correct check digit.
9. Line 39 shows the average edit distance of all UIC numbers, with a bounding box that was overlapping at least 30% with the ground truth.
10. The last line shows the average edit distance counting not detected UIC numbers as an edit distance of 13.

If the application was started with the GUI option, the dataset is not directly evaluated, but instead the dataset is loaded and can be browsed by the user. The program can be controlled using the following keys:

- **“A”** and **“D”** allow the selection of an UIC image. The images are sorted by their UIC number. The current image is shown as in figure 5.2a. The blue boxes represent the digit bounds that were calculated using the bounding boxes of the digit groups. The command line also shows the index of the current entry and its key. The key consists of the UIC number and an index if one UIC number exists multiple times.
- **“W”** and **“S”** allow the selection of an UIC detector. The current detector is shown on the command line by printing all of its components.
- **“O”**... Original, **“T”**... Train, **“U”**... Uic and **“I”** ... dIgits, can be used to set the region that is passed to the UIC detector.
- **Space** is used to evaluate the current entry. Thereby, the result is shown as described in section 5.2. In addition to that, the command line prints the run time of the major pipeline segments and the 5 best UIC results. Each result includes the detected number, its edit distance, the overlap of its bounding box with the ground truth and its score.
- **“L”** enables and disables the use of a white list. Thereby all UIC numbers from the dataset are used. The white list is only used for evaluating the whole dataset, not a single entry.

- **"E"** evaluates the dataset with the current UIC detector, region mode and optional white list. The result is shown on the command line and is in the same format as previously described in this section (See listing 4).
- **"C"** calculates and shows the height in pixels of the smallest digit and the standard deviation (RMS) of all digits (Only using the area inside the digit groups). These values can be used to judge the quality of an image.
- Any other key closes the program.

# Chapter 6

# Conclusion

Although standard text detection systems can achieve near perfect results, they heavily rely on high quality input images and dictionaries to correct mistakes. However, the domain of UIC number detection provides neither of these. Recognizing a UIC number requires the correct identification of every single digit which allows no room for mistakes. In addition, UIC numbers are not standardized, which results in many different fonts, sizes and colors. The bad condition of some of the numbers can make them hard to read even for humans. In addition to the difficulties of the domain, the with hand held devices created dataset contains a high amount of low quality images.
Two character detection algorithms were tested. The FASText detector was extended to support color images [BNM15]. Although, the method was more novel, its slow processing time compared to the well established Extremal Regions detector, prevented its adaptation into the final system [NM12]. However, the scale pyramid implemented originally for the FASText detector turned out to also boost the performance of the ER detector [NM13].
Furthermore, a custom line detector was implemented that was optimized for UIC number detection and works significantly faster than comparable methods. Allowing its application to a higher number of characters. Finally, the lines were grouped into potential UIC numbers and scored with a new algorithm.
All of these building blocks were implemented as separate components, allowing the simple addition of new improved components or its adaptation to a different domain.
Testing the UIC detector, shows that the character detector finds most digits in high quality images and only fails once the resolution or contrast are too low. Once all digits were found, they are successfully being grouped into lines and UIC numbers. The most critical point of failure is the Optical Character Recognition. The low accuracy of the OCR, and the low quality of the input images and some UIC numbers account for most failures of the system.
Final tests revealed a low success rate on the custom UIC dataset preventing its use in a fully automated system. Especially compared to results reported from the industry, the accuracy of the custom system can not compete [urla]. However, the comparison might be unfair, as the quality of the used input images and potentially used aids such as white lists were not reported.
Even though, rapid progress was made at reading text from high quality images, text detection in complex scenes remains a challenging task. The results show, that UIC number detection is a challenging problem in this domain and further research and test with different methods are necessary, to improve its accuracy. The designed framework provides a good starting point for this endeavor.

## 6.1 Further work

The whole pipeline was implemented without any parallel processing. Especially the digit detection in different color channels and scales is easy to parallelize and can significantly improve the processing time.
Furthermore, the OCR stage can be improved by replacing TesseractOCR with a faster system such as a SVM with a confidence score. Additionally, the classifier can be trained on a UIC specific dataset and using incomplete digits during training can further improve its accuracy.
Another weakness of the system is that it can not identify digits that were not found by the character detector. To counteract this issue more lenient filtering rules were used to obtain the most character candidates. An alternative method would be to use a different system for the digit identification once their location was found. A possible approach is to use a deep convolutional sequence as described by He et al. [HHQ+16].
If the system is used with video streams, the location of the UIC number can be detected with a fast system and a Kalman filter can be used to quickly find its location in subsequent frames. Finally, the quality of each frame can be calculated and the highest scoring one can be used to extract the UIC number with a slower but highly accurate system.

# Bibliography


[BNM15] M. Busta, L. Neumann, and J. Matas. Fastext: Efficient unconstrained scene text detector. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1206–1214, Dec 2015.

[Can86] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, Nov 1986.

[CCC+11] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, T. Wang, D. J. Wu, and A. Y. Ng. Text detection and character recognition in scene images with unsupervised feature learning. In *2011 International Conference on Document Analysis and Recognition*, pages 440–445, Sept 2011.

[CTS+11] H. Chen, S. S. Tsai, G. Schroth, D. M. Chen, R. Grzeszczuk, and B. Girod. Robust text detection in natural images with edge-enhanced maximally stable extremal regions. In *2011 18th IEEE International Conference on Image Processing*, pages 2609–2612, Sept 2011.

[DISB13] S. Du, M. Ibrahim, M. Shehata, and W. Badawy. Automatic license plate recognition (alpr): A state-of-the-art review. *IEEE Transactions on Circuits and Systems for Video Technology*, 23(2):311–325, Feb 2013.

[EOW10] B. Epshtein, E. Ofek, and Y. Wexler. Detecting text in natural scenes with stroke width transform. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2963–2970, June 2010.

[GBYB12] . Gonzalez, L. M. Bergasa, J. J. Yebes, and S. Bronte. Text location in complex images. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 617–620, Nov 2012.

[Gra71] S. B. Gray. Local properties of binary images in two dimensions. *IEEE Transactions on Computers*, C-20(5):551–561, May 1971.

[HHQ+16] P. He, W. Huang, Y. Qiao, C. Loy, and X. Tang. Reading scene text in deep convolutional sequences. In *AAAI*, pages 3501–3508, 2016.

[KSU+13] D. Karatzas, F. Shafait, S. Uchida, M. Iwamura, L. G. i. Bigorda, S. R. Mestre, J. Mas, D. F. Mota, J. A. Almazn, and L. P. de las Heras. Icdar 2013 robust reading competition. In *2013 12th International Conference on Document Analysis and Recognition*, pages 1484–1493, Aug 2013.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[LPS+03] S. M. Lucas, A. Panaretos, L. Sosa, A. Tang, S. Wong, and R. Young. Icdar 2003 robust reading competitions. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 682–687, Aug 2003.

[Luh60] P. Luhn. Computer for verifying numbers, August 23 1960. US Patent 2,950,048.

[MCUP04] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing*, 22(10):761–767, 2004.

[MTS+05] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Van Gool. A comparison of affine region detectors. *International Journal of Computer Vision*, 65(1):43–72, 2005.

[NM11a] L. Neumann and J. Matas. *A Method for Text Localization and Recognition in Real-World Images*, pages 770–783. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[NM11b] L. Neumann and J. Matas. Text localization in real-world images using efficiently pruned exhaustive search. In *2011 International Conference on Document Analysis and Recognition*, pages 687–691, Sept 2011.

[NM12] L. Neumann and J. Matas. Real-time scene text localization and recognition. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3538–3545, June 2012.

[NM13] L. Neumann and J. Matas. On combining multiple segmentations in scene text recognition. In *2013 12th International Conference on Document Analysis and Recognition*, pages 523–527, Aug 2013.

[NM15] L. Neumann and J. Matas. Efficient scene text localization and recognition with local character refinement. In *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 746–750, Aug 2015.

[PHL09] Y. F. Pan, X. Hou, and C. L. Liu. Text localization in natural scene images based on conditional random field. In *2009 10th International Conference on Document Analysis and Recognition*, pages 6–10, July 2009.

[Smi07] R. Smith. An overview of the tesseract ocr engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 2, pages 629–633, Sept 2007.

[SS99] R. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[SSD11] A. Shahab, F. Shafait, and A. Dengel. Icdar 2011 robust reading competition challenge 2: Reading text in scene images. In *2011 International Conference on Document Analysis and Recognition*, pages 1491–1496, Sept 2011.

[urla] Automatic rail vehicle identification and uic codes recognition system (arvis). `http://www.eurekanetwork.org/project/id/6741`. Accessed: 2017-04-14.

[urlb] Carmen uic software by arh. `http://www.arh.hu/doc/arh_uicsoftware.pdf`. Accessed: 2017-04-14.

[urlc] Leaflet 419-2, analytical numbering of international freight trains. `http://www.uic.org/etf/codex/codex-detail.php?langue_fiche=E&codeFiche=419-2`. Accessed: 2017-05-1.

[urld] Numbercheck by ase. `http://www.ase-industry.de/N-check-Prospekt_ASE_engl.pdf`. Accessed: 2017-04-14.

[urle] Numbercheck-sensoric by ase. `http://www.ase-industry.de/wASE_en/downloads/documents/pdf/Numbercheck_Prospekt.pdf`. Accessed: 2017-04-14.

[urlf] Tesseract ocr. `https://github.com/tesseract-ocr/tesseract`. Accessed: 2017-07-16.

[urlg] Trex-wagon by nestor technologies. `http://www.nestor-tech.com/files/File/en/products/trex-wagon/trex_wagon_brochure_en.pdf`. Accessed: 2017-04-14.

[urlh] Uic number dataset. `https://drive.google.com/drive/folders/0B0WUQab4xB1XbGJYa3JmcElPZWM?usp=sharing`. Accessed: 2017-07-16.

[WJ06] C. Wolf and J. 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.

[YYHH14] X. C. Yin, X. Yin, K. Huang, and H. W. Hao. Robust text detection in natural scene images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(5):970–983, May 2014.

# List of Figures


1.1 Structure of UIC numbers . . . . . 2
1.2 Examples with different ratios of the UIC area and the train area . . . . . 4
1.3 Examples of UIC images with their quality (q=1-5) and height . . . . . 5
1.4 Subjective quality rankings of UIC number in dataset . . . . . 5
1.5 Example input images from the ICDAR 2003 challenges [LPS+03] . . . . 7

2.1 Different threshold values applied to a gray scale image . . . . . 13
2.2 Extraction of Stroke End/Bend Keypoint (SEK/SBK) [BNM15] . . . . . 17
2.3 Text line estimates for triplets. Top four are valid triplets bottom four are invalid [NM11b] . . . . . 19

3.1 The key classes used for the UIC detector pipeline . . . . . 23
3.2 Effect of the threshold $t$ on the ER character detector performance. ($p_1 = 0$, $p_2 = 0$, gray scale) . . . . . 28
3.3 Effect of the minimal probability $p_1$ on the ER character detector performance. ($t = 1$, $p_2 = 0$, gray scale) . . . . . 30
3.4 Effect of the min probability difference $p_d$ on the ER character detector performance. ($t = 0$, $p_1 = 0$, $p_2 = 0$, gray scale) . . . . . 31
3.5 Effect of the minimal probability $p_2$ on the ER character detector performance. ($t = 1$, $p_1 = 0$, gray scale) . . . . . 31
3.6 UIC numbers with inconsistent background and text color . . . . . 34
3.7 Digit with the largest possible area ($15 \times 30$px, Stroke width $= 3px$) . . . 36
3.8 Effect of the threshold $t$ on the FASText detector. ($p_1 = 0$, $p = 0$, luminance channel, 20% of dataset) . . . . . 37
3.9 Effect of the adaptive threshold $a$ on the FASText character detector performance. ($t = 16$, $p = 0$, luminance channel, 20% of dataset) . . . . . 37
3.10 Effect of the minimal character probability $p$ on the ER character detector performance. ($t = 16$, $a = 0.6$, $p = 0$, luminance channel, 20% of dataset) 38
3.11 Effect of the threshold $t$ on the FASText character detector directly applied to RGB image. ($p = 0$, static threshold, 20% of dataset) . . . . . 40
3.12 Effect of different scale factors $s$ on character detectors (10% of dataset)) 41
3.13 Samples of character detector results (All characters were scaled to the same size) . . . . . 43
3.14 Spray painted UIC numbers . . . . . 44
3.15 Behavior of Neumann and Matas line detector depending on the number of input characters (50% of dataset) . . . . . 45
3.16 Parameters of the custom line detector. Tests run with 50% of dataset and $v_h = 0.8, v_y = 4, v_o = 0.3$ and $v_d = 1.5$ as defaults. The speed optimized ER character detector with the luminance channel only was used . . . . . 51
3.17 Behavior of Neumann and Matas line detector depending on the number of input characters (50% of dataset) . . . . . 53

3.18 Custom line detector picks higher quality characters . . . . . 55
3.19 Custom line detector finds 2 digit groups . . . . . 55
3.20 Custom line detector groups characters despite distance . . . . . 55
3.21 Examples of ORC classifications . . . . . 59
3.22 Examples of extracted UIC numbers and their score . . . . . 62

4.1 Edit distance of different UIC detectors (50% of dataset, UIC images) . . 66
4.2 Processing time of UIC detectors (50% of dataset, UIC images) . . . . . 67
4.3 Processing time of UIC detectors (50% of dataset, Train images) . . . . . 68
4.4 Examples of UIC numbers and their edit distance extracted from train images with "ER-Speed+L+SVM and ER-Balance+RGB+Scale" . . . . . 69

5.1 GUI of the UIC annotation tool . . . . . 73
5.2 Optional GUI of the command line interface . . . . . 75

# List of Tables


1.1 Input image resolutions . . . . . 3
1.2 Text localization results of ICDAR competitions . . . . . 9

2.1 Features used to filter MSERs in [NM11a] . . . . . 14
2.2 Features used to filter Extremal Regions [NM12] . . . . . 16

3.1 The final three Extremal Region character detectors . . . . . 32
3.2 Combination of multiple color channels for the ER detector using the balanced parameters (G...Gray, I...Intensity, D...Gradient) . . . . . 33
3.3 The final two FASText character detectors . . . . . 39
3.4 Combination of multiple color channels for the FASText detector using system 2 without NMS. (G...Gray, I...Intensity, D...Gradient) . . . . . 39
3.5 A selection of character detector configurations (50% of dataset) . . . . . 42
3.6 Results of the NM line detector . . . . . 46
3.7 Results of the custom line detector . . . . . 54
3.8 OCR results (50% of dataset) . . . . . 58

4.1 Results of UIC detectors (50% of dataset, UIC images) . . . . . 71

# Fabian Ahorner

## Curriculum Vitae

93 Cardigan Road
Reading, RG1 5QW
Birthday: 14. July 1992
**+43 7498664537**
**f.ahorner@gmail.com**

## EXPERIENCE

**Qmee Ltd,** Reading (UK) — *Software Developer*

January 2017 - PRESENT

Mobile app and web development

**Runtastic,** Linz (Austria) — *Android Developer*

February 2013 - August 2014

Development of Android and Tizen apps

**Miteinander GmBH,** Linz (Austria) — *Support Worker*

August 2011 - April 2012

Caring for people with mental and physical disabilities as a civilian servant

**Alpenverein Enns,** Enns (Austria)— *Climbing instructor*

September 2009 - July 2016

Training and guiding groups. Climbing wall maintenance

## EDUCATION

**Johannes Kepler Universität,** Linz (Austria) — *Masters*

September 2015 - PRESENT

Master degree in Computer Science in the field of computational engineering.

**University of Reading,** Reading (UK) — *Bachelors*

September 2014 - April 2015

Study abroad

**Johannes Kepler Universität,** Linz (Austria) — *Bachelors*

September 2012 - July 2015

Bachelor degree in Computer Science

**HTL Steyr,** Linz (Austria) — *High School*

September 2006 - July 2011

Electronical engineering (Design of circuits, Signal processing) and programming (C, Java, Assembly)

# Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references. This printed thesis is identical with the electronic version submitted.

Fabian Ahorner, December 2017