Fast Text Detection for Road Scenes

Matias Alejandro Valdenegro Toro

Publisher: Dean Prof. Dr. Wolfgang Heiden

University of Applied Sciences Bonn-Rhein-Sieg,
Department of Computer Science

Sankt Augustin, Germany

August 2015

Technical Report 05-2015

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

ISSN 1869-5272

Uniform Resource Name urn:nbn:de:hbz:1044-opus-15936
URN-Resolver at the German National Library http://nbn-resolving.de

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Fraunhofer
IAIS

Master Thesis

Fast Text Detection for Road Scenes

Author:
Matias Alejandro
Valdenegro Toro

Advisors:
Prof. Dr. Paul Plöger
Prof. Dr. Gerhard Kraetzschmar
Dr. Stefan Eickeler

Bonn-Rhein-Sieg University of Applied Sciences
Department of Computer Science

Fraunhofer Institute for Intelligent Analysis
and Information Systems

September 2014

Corrected June 2015

Abstract

Fast Text Detection for Road Scenes

by Matias Alejandro
Valdenegro Toro

Extraction of text information from visual sources is an important component of many modern applications, for example, extracting the text from traffic signs on a road scene in an autonomous vehicle. For natural images or road scenes this is a unsolved problem.

In this thesis the use of histogram of stroke widths (HSW) for character and non-character region classification is presented. Stroke widths are extracted using two methods. One is based on the Stroke Width Transform and another based on run lengths. The HSW is combined with two simple region features– aspect and occupancy ratios– and then a linear SVM is used as classifier. One advantage of our method over the state of the art is that it is script-independent and can also be used to verify detected text regions with the purpose of reducing false positives.

Our experiments on generated datasets of Latin, CJK, Hiragana and Katakana characters show that the HSW is able to correctly classify at least 90 % of the character regions, a similar figure is obtained for non-character regions. This performance is also obtained when training the HSW with one script and testing with a different one, and even when characters are rotated. On the English and Kannada portions of the Chars74K dataset we obtained over 95% correctly classified character regions.

The use of raycasting for text line grouping is also proposed. By combining it with our HSW-based character classifier, a text detector based on Maximally Stable Extremal Regions (MSER) was implemented. The text detector was evaluated on our own dataset of road scenes from the German Autobahn, where 65% precision, 72% recall with a f-score of 69% was obtained. Using the HSW as a text verifier increases precision while slightly reducing recall. Our HSW feature allows the building of a script-independent and low parameter count classifier for character and non-character regions.

Acknowledgements

Hope is what makes us strong. It is why we are here.
It is what we fight with when all else is lost.

希望は我々を強くする。私たちはここにいるから。
それこそが、すべてを失った時、我々と戦うのだ。

希望是什我。就是什我在里。是我一切，失去了斗

- Pandora, from God of War 3.

Many people have made my stay in Germany a beautiful experience. The Professors and Staff at Bonn-Rhein-Sieg University of Applied Sciences are part of such people and I am really grateful for the education and mentoring I received from them.

The people in the University's International Office always provided invaluable support to international students like me, specially when we needed to deal with the German bureaucracy. I really acknowledge the help I received from Lea Mund and Maria do Carmo Massoni, without their help it would have been harder to live in Bonn. Many thanks.

My friends Cristian Rodriguez, Juan Guerra, Karina Bunster, Jazmin Espinoza, Carolina Mora, Cecilia Burgos and Carolina Goyheneche always gave moral support from my home country of Chile, even when the distance was an impediment. I really appreciate the support from my best friend Polett Escanilla, she was always there when she was needed. Menaka Naazare gave me invaluable help in understanding the Kannada script. Akie Nakamori contributed with her Japanese knowledge, specifically about the Hiragana and Katakana scripts. Many thanks to Alexander Hagg, Teena Hassan and Eduardo Graells for proofreading my thesis.

Last but not least I would like to thank the mentoring I received from my advisors Prof. Dr. Paul Plöger and Prof. Dr. Gerhard Kraetzschmar. Dr. Stefan Eickeler deserves special thanks for giving me the opportunity to work for Fraunhofer IAIS.

Finally I would like to thank the authors of this thesis's Latex Template (Steven Gunn and Sunil Patel), which made it available under a Creative Commons License (CC BY-NC-SA 3.0) and was downloaded from http://www.latextemplates.com.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

MSER	Maximally Stable Extremal Regions
RL	Run Length
SWT	Stroke Width Transform
SVM	Support Vector Machine
TV	Text Verifier

Dedicado a la memoria de mis Abuelos Sergio y Paulina, quienes siempre me incentivaron a estudiar y a seguir mis sueños. Que descansen en paz.

Dedicated to the memory of my grandparents Sergio and Paulina, who always encouraged me to study and follow my dreams. May they rest in peace.

Chapter 1

Introduction

Writing is considered one of humanity's most important advancements, since it allows to permanently record information for later use. The human civilization has produced countless written text that lie on physical media, such as paper, stone, walls, etc.

The retrieval of such textual information from non-digital sources is a topic of much interest, and it covers many areas of knowledge, such as Computer Vision, Statistics, Pattern Recognition and Machine Learning. In general we take reading for granted, but computers in general have great difficulty in doing so.

One of the great difficulties with text detection and recognition is the large amount of variation in text color, size, orientation, texture, appearance, font, style and script. The assumptions that can be taken to ease the problem are little.

Many applications require extraction of text from visual sources, such as still image and video. Some examples are augmented reality, license plate recognition, blind person assistance and automated data entry into computer systems. In the context of autonomous systems and autonomous vehicles, the information stored in traffic signs and panels is relevant to the driving process, and legally required for driving in the same way as human drivers do.

Text information can be used in many ways by an autonomous vehicle. Speed limits can change due to traffic conditions and accidents, and temporary signs can be installed, such as when works are performed on a road. Thus there is a need for vision algorithms that can extract such information from a video camera installed in the vehicle.

Many algorithms for text detection and recognition exist [12] [13]. Text detection is the task of finding "whether" and "where" the text is located inside the image, and text recognition is the task of converting the text parts of an image to a digital text representation (such as character strings).

But recognition algorithms in general perform poorly in real-world scenes, mainly because the task itself is complex due to the big variation in text size, color, orientation, script and style, and because of many parts of natural scenes can be considered to be text. Also, many algorithms are designed only for documents and not for real-world scenes such as road scenes.

For example, a text detection and recognition algorithm by Neumann et al [14] could only correctly extract 32.9% of the text in natural scenes from the SVT dataset [3]. Newer methods that use advanced convolutional neural networks can correctly retrieve up 70% text from the same dataset, but are computationally expensive.

Script should also be considered. Only 37.1% of the world population uses the Latin script their day to day writing and reading needs, but almost all text detection methods operate only on Latin script text. This could be due to the prevalence of top Western universities in computer vision research.

There is always a tradeoff between detection and computational performance. Many algorithms that are fast do not detect all text and perform poorly, while algorithms that are slow are able to detect much more text correctly.

In this thesis we develop and introduce a new text detector method based on character classification using a histogram of stroke widths. This feature allows to classify and detect text without making assumptions on the script, and as we will show, it gives good performance, even for road scenes. A text grouping method is also introduced, which only requires one parameter. As it will be shown, computational and detection performance of both methods are adequate for the problem, and we believe they are a contribution to the state of the art.

Chapter 2

Related Work

There are vast amounts of information currently stored in textual form, and a number of different systems are designed with requirements of “reading” or retrieving this textual information from non-digital sources, such as printed documents, captured images or directly from video cameras attached to robots and autonomous vehicles. Then this information could be stored in other formats, such as digital information, from where processing by computers is easier.

For example, an autonomous vehicle could take advantage of reading text in traffic signs and panels for path planning, since some traffic panels above the road contain information about which lane will take the vehicle to its destination, as well as real-time information about the conditions in the road ahead. Some of such panels can be seen in Figure 2.1.

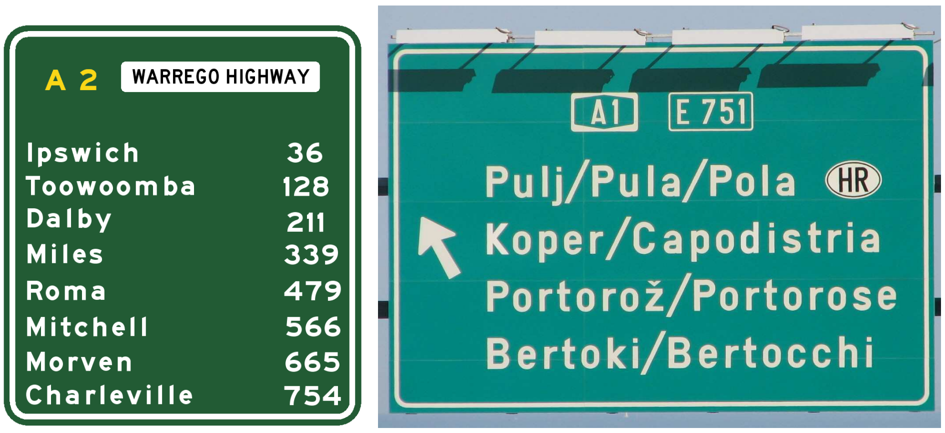
Reading this textual information from visual sources is called text recognition and usually requires 3 stages [15]:

Image Acquisition

An image of the target text is obtained. For a scene, a camera is usually used, while for documents an optical scanner is used. In this initial stage many situations can influence the quality of further stages, such as blur, camera focus, lighting and shadows.

Text Detection

Text is localized in the image and detected text regions are passed into the



(A) Australian Traffic Panel (B) Slovenian Traffic Panel

FIGURE 2.1: Some traffic panels from Australia and Slovenia. **Source**: Public Domain

$$\begin{tikzcd}
\boxed{\text{Image Acquisition}} \arrow[r] & \boxed{\text{Text Detection}} \arrow[r] & \boxed{\text{Text Recognition}} \arrow[r] & \text{Recognized Text}
\end{tikzcd}$$

FIGURE 2.2: Text Recognition Pipeline

next stage. The purpose of this stage is to reduce the size of the hypothesis space, since text recognition algorithms are slow when too many hypotheses must be tested, and recognition performance is not optimal since too many false positives will be produced. This stage is also called Text Localization.

Text Recognition

Text is recognized from patches of the original image and converted into a digital representation (usually strings of characters) by using a machine learning and/or pattern recognition algorithm. Algorithms that do recognition are called Optical Character Recognition (OCR). The quality of the located text greatly influences the results of the recognition stage.

The pipeline is shown in Figure 2.2. Many real-world applications require some form of text recognition [15], such as:

License plate recognition

Where cameras are placed on roads and license plate numbers of passing cars are required, with different purposes, such as vehicle and traffic flow control, law enforcement, and billing for tolls.

Automatic data entry

Documents are put through a scanning device and the information is presented to the user or used by a computer system. Examples of this are bank checks, tickets, invoices and any kind of textual information stored "on paper" that need to be read by a computer or be digitalized. Many banks use such systems to automatically process checks, as well as post offices for mail.

Book scanning and storage
A complete book is scanned and converted into digital format, in order to be made available to a wider audience by storing it in a digital form, which can be transmitted over the internet or safely stored in a permanent medium for future use. Project Gutenberg does this [1].

Historical documents
Since the medium where they are written degrades over time, digital storage of their contents is a must to keep cultural knowledge for future generations.

Technologies to assist blind and visually impaired persons
Most written content is not available in a form that is friendly to computer systems used by such persons. This kind of systems usually presents the information in another medium, such as reading the book in audio form with a speech synthesizer software, or outputting the text through a braille system.

Translation
An image or live camera feed is analyzed and words are recognized and translated into another language, in an augmented-reality fashion. There are mobile phone applications that already do this, such as Word Lens Translator [2].

Search
Search engines in general only search in digital text information, such as text documents and web pages, and ignore information that might be stored in images and/or video. The extraction of textual information from images could enable better search engines and would increase the amount of information available to search on.

Autonomous driving
Autonomous cars are currently being developed, as well as Advanced Traffic Assistance Systems (ADAS) [16]. Both systems either replace or aid the driver with the driving task, and also would require the use of textual information

[1] http://www.gutenberg.org/wiki/Main_Page

[2] https://play.google.com/store/apps/details?id=com.questvisual.wordlens.demo&hl=en

available in traffic signs and panels, as well as other text information in road scenes, such as the names of places, nearby stores and shops.

There is vast amount of literature about Text Detection and/or Localization, and many algorithms and variations of them have been developed, which is a indicator of the difficulty of this problem. A survey on the current state of the art in text detection can be found in Zhang et al. [12]:

Edge-based Methods

Edges from an edge detector algorithm are used along with image processing operations to extract text regions, usually by doing morphological operations, such as dilation to connect edges into complete boundaries of an object.

Texture-based Methods

This kind of methods uses a sliding window approach, by taking all possible windows from the image of a given size, and using texture features to discriminate text from non-text windows.

Region-based Methods

Text regions are extracted in a image with a region detection algorithm, and then are classified as text or non-text by a text and/or character classifier. Connected component methods also fall into this category, since they usually are designed detect regions of an image that have a common property, such as characters or text.

Stroke-based Methods

This kind of methods is a mix of the previous ones, but concentrating on the use of stroke information as a mean for discriminating text from non-text regions.

Text detection itself is a very hard problem due to the very high variability of characters and text regions [13] [12], such as:

Size

There is no defined size of characters in the image, as well as the number of characters in words and text lines.

Orientation

Text could appear in any orientation. While the most common orientation is horizontal, text also could be vertical, or in diagonal orientations, and even skewed due to the camera viewpoint.

Color

Text could have any color, and while usually text is designed to have a very strong contrast with its background, but this might not be able to be perceived from the camera viewpoint due to illumination conditions.

Texture

In general text always has a constant color, but it could also be textured, or have implicit edges, such as inscriptions on stone, where lighting is used to "see" the text.

Font and Appearance

Many different fonts exists, and graphics designers' creativity tends to create new ways and forms to present text and characters. Text could even be presented as shapes without a constant color or texture, such as the "M" formed by the McDonalds sign.

Script

The script from where characters are drawn can also vary. The most common assumption is the use of Latin or Roman characters, but many countries in Asia and the Middle East do not use such characters, and many algorithms fail under such circumstances.

2.1 Public Datasets

There are many datasets for evaluation of text detection and recognition. The primary dataset used for text detection is the ICDAR dataset, both in its 2003 [1] [3] and 2011 [2] [4] versions.
The ICDAR 2003 dataset contains color images of sizes between 1280x960 to 1600x1200 pixels. The training subset contains 258 images, and the testing subset contains 251 images. Some images from this dataset are shown in Figure 2.3.

[3]http://algoval.essex.ac.uk/icdar/Datasets.html
[4]http://robustreading.opendfki.de/wiki/SceneText

The ICDAR 2011 dataset also contains color images of sizes between 640x480 to 3888x2592 pixels, with 229 training images, and 255 testing images. Some images from this dataset are shown in Figure 2.4.

Both datasets primarily contain English text in real-world scenes, but also in general the text regions cover a considerable area of the image, so it is possible to say that this datasets are biased towards text that is "big" inside the image frame, and do not contain small pieces of text (such as in a road scene).

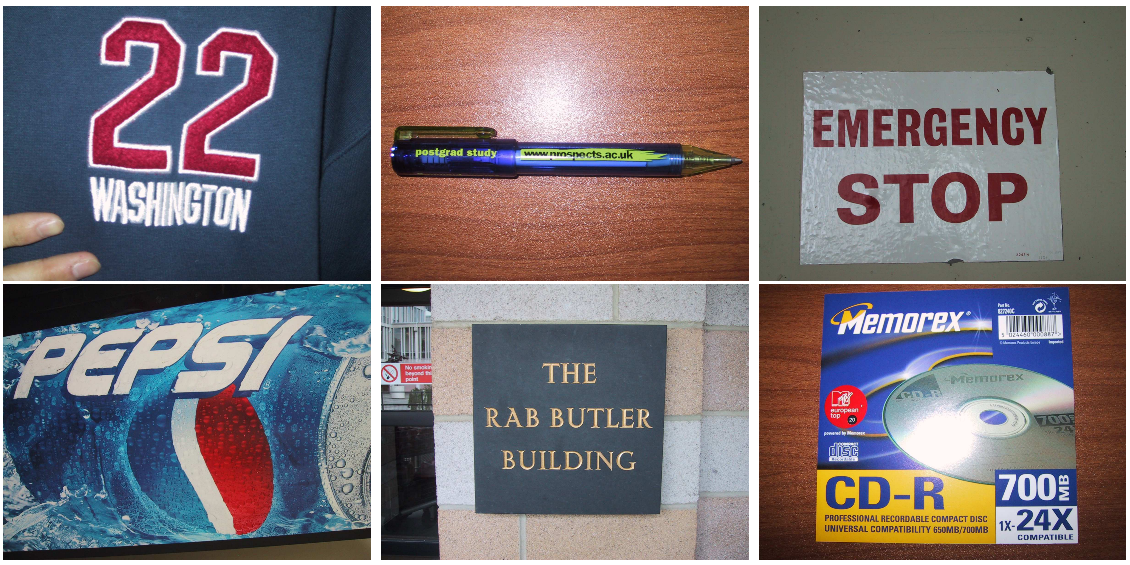


FIGURE 2.3: Some images from the ICDAR 2003 dataset [1]

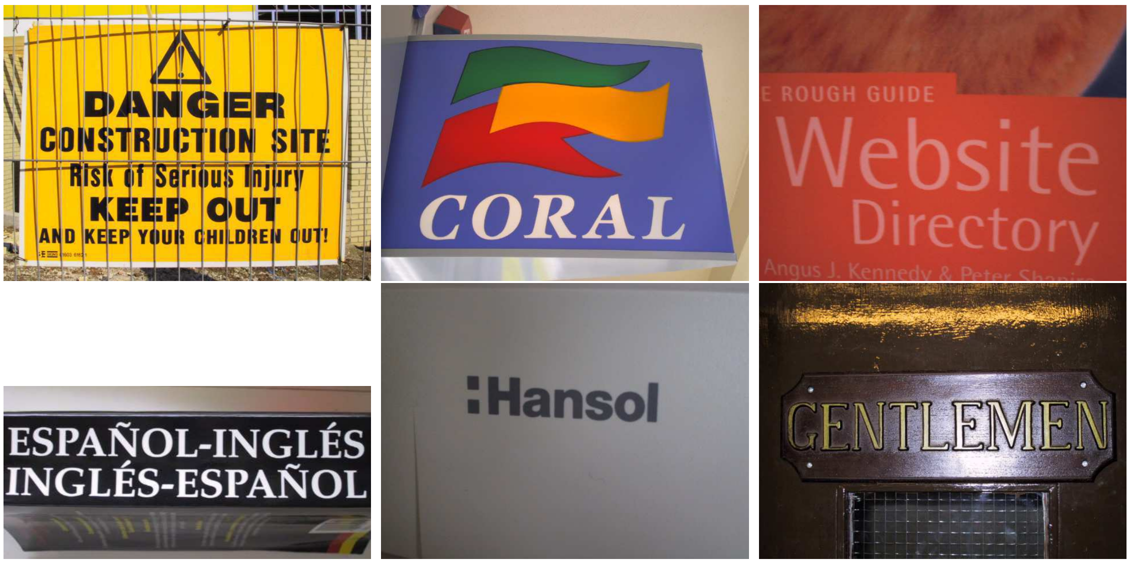


FIGURE 2.4: Some images from the ICDAR 2011 dataset [2]

Another dataset is the Street View Text (SVT) [3] [5], which is made from images taken from Google Street View, and contains 100 color images for training, and 249 color images for testing. Image sizes vary from 1024x768 to 1918x898 pixels, and in general the text regions cover smaller areas that in comparison with the ICDAR 2003 and 2011 datasets. Some images from this dataset can be seen in Figure 2.5.

The images in this dataset are from road scenes, but in general they contain "artifacts" due to the stitching done by Google to construct the Street View images. Google uses cars with several cameras attached to the roof, and image stitching is needed to produce a 360° panorama that is used for Google Street View. The dataset also exhibits a small amount of deformation due to lens distortion.

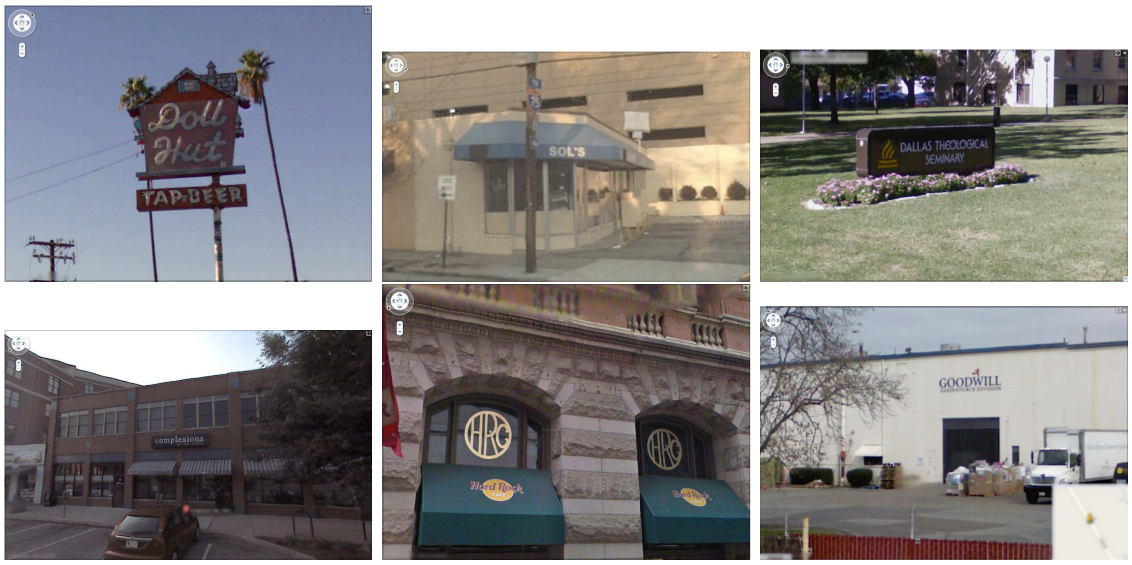


FIGURE 2.5: Some images from the SVT dataset [3]

The Chars74K dataset [4] [6] is a dataset normally used for character recognition that contains images of characters. This dataset is split into several sub-datasets, for both the English language (Latin/Roman characters) and the Kannada script (commonly used in East India). It contains 7705 characters from natural images, 3410 handwritten characters obtained from a Tablet PC, and 62992 characters generated from different fonts.

Generated font images are 128x128 pixels in size and only contain Latin characters. Kannada script is present on images of handwritten text captured with a Tablet PC.

[5]http://vision.ucsd.edu/~kai/svt/
[6]http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/

Some generated font numbers from this dataset can be seen in Figure 2.6, while some generated lowercase letters from a computer font can be seen in Figure 2.7.

The figures show the huge variation of fonts and combinations of roman/bold/italics in the dataset, which makes it well suited to train character classifiers and character recognizers. We mention this dataset because it was used in this work to evaluate our character classifier.

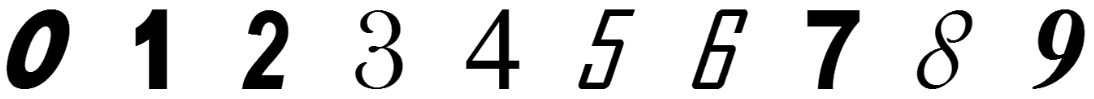


FIGURE 2.6: Some numbers from the Chars74K Dataset [4]

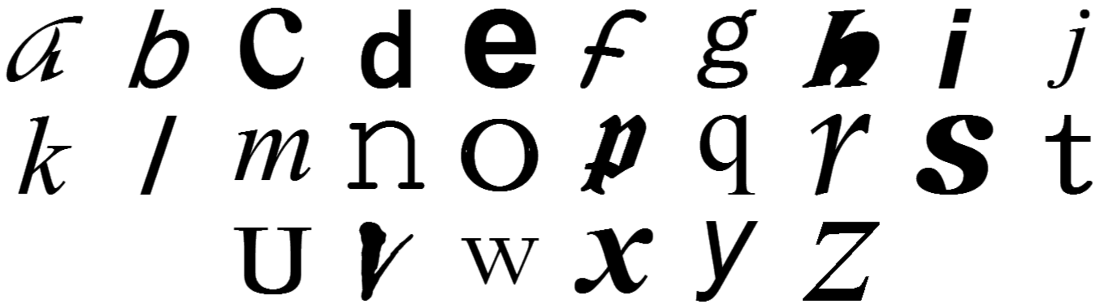


FIGURE 2.7: Some lowercase letters from the Chars74K Dataset [4]

2.1.1 Evaluation Metrics

The most common evaluation metrics for text detection are precision and recall [1]. Ground truth information is provided in the datasets, usually in the form of axis aligned bounding boxes around text (Rectangles). From this information, we denote T as the set of targets, the correct bounding boxes in the ground truth, and E as the set of estimates, the bounding boxes returned by the detector for a given input image. The set of correct estimates is denoted as c. Then precision p is computed as:

$$p = \frac{|c|}{|E|} \tag{2.1}$$

And recall is computed as:

$$r = \frac{|c|}{|T|} \tag{2.2}$$

A low precision means the detector is over-estimating the amount of text bounding boxes in the image, while a low recall means the detector is under-estimating the amount of text bounding boxes in the image. Both a high precision and recall are desired for a good text detector. One way to combine precision and recall into a single measure is the f-score, computed as:

$$f = \frac{2}{\frac{1}{p} + \frac{1}{r}} \tag{2.3}$$

Precision, recall and f-score are always in the $[0, 1]$ range. One remaining issue is to define when a rectangle estimate is correct or not, since rectangles from the ground truth and estimated rectangles will never exactly match, and some degree of flexibility must be taken into account. The ICDAR 2003 competition [1] uses the following evaluation protocol. Given 2 rectangles, the match between both rectangles A and B is defined as:

$$\text{match}(A, B) = \frac{\text{area}(A \bigcap B)}{\text{area}(A \bigcup B)} \tag{2.4}$$

Where $A \cup B$ represents the bounding box that contains both A and B. The match between 2 rectangles is a number between 0 and 1, where 0 means no match, and 1 is a perfect match. Then the best match for rectangle r from a set of ground truth rectangles R is:

$$\text{bestMatch}(r, R) = \max_{r_0 \in R} \text{match}(r, r_0) \tag{2.5}$$

Then precision and recall can be re-defined as:

$$p = \frac{1}{|E|} \sum_{r \in E} \text{bestMatch}(r, T) \qquad r = \frac{1}{|T|} \sum_{r \in T} \text{bestMatch}(r, E) \tag{2.6}$$

Some authors [17] instead consider a rectangle as correct if and only if the best match is bigger than some threshold α, where the most common value is $\alpha = 0.5$.

2.2 Edge-based Methods

Edge methods use the information available from the high contrast between a character and its background to extract edges, then group edges into candidate character regions. Candidate character regions are validated through heuristic rules, such as region height, number of holes and aspect ratio, and then grouped using a clustering algorithm to form text lines. This is very similar to Region-based methods, but in general edge information is unreliable due to image noise, motion blur, lighting conditions and the fact that information about the interior of the regions is completely ignored.

The use of heuristic rules is in general very unreliable, since they can provide good performance in one dataset, and completely fail to generalize to other datasets, and in general they are application dependent. In general these rules require parameter tuning which can be tedious and error-prone. Finally, edges can be disconnected due to various reasons, such as noise and lighting conditions, which can "split" the candidate text regions.

Smith and Kanade [18] used a 3x3 horizontal difference filter and thresholding to extract vertical edges from TV news video. After thresholding, smoothing is applied to remove small disconnected edges and to connect strong disconnected edges. Then clustering is applied to identify text lines and bounding boxes are computed.

Text regions are detected if they meet three heuristic constraints: Bounding box aspect ratio bigger than 0.75, occupancy ratio bigger than 0.45 and cluster size bigger than 70 pixels. The authors evaluated their algorithm in a small video dataset, containing approximately 20 text regions, and in average their algorithm detected 90% of the available text, with a false positive rate of 20%.

Sato et al. [19] extracted character candidates by using a bank of filters that detect horizontal, vertical, left and right diagonal edges, since characters usually have edges in these directions. The input image is first interpolated to sub-pixel precision, due to the low resolution their video frames (320×240), and the correlations between each filter in the bank and the interpolated image is computed. All filter outputs that are positive are accumulated into a character image, which is then thresholded to obtain a binary image.

Vertical and horizontal projections are then used to extract character candidates from the binary image. For Video OCR, their method correctly detected 89.8% of the characters in their dataset.

Chen at al. [20] use Canny edge detection to extract edges, which are grouped by means of morphological dilation. Heuristic rules are used to filter text from non-text region. A set of asymmetrical Gabor filters and a neural network are used to estimate the scale of text, and then scale is used to enhance the edges of text. Performance on their own dataset is 82.6% recognition for enhanced images, compared to 36.1% with non-enhanced images.

Liu and Samarabandu [21] used a multiscale edge detection approach, on which the input image is convolved with a bank of filters that detect edges in four orientations (the authors called this the Compass operator). Four edge orientation images are produced for each scale, in orientations of $0°$, $45°$, $90°$ and $135°$. Edge orientation images are then combined to produce a feature map that enhances text regions.

A 7x7 morphological dilation element is used to cluster text regions, and two simple heuristics are used to remove non-text regions: occupancy and aspect ratios, then bounding boxes are constructed. The authors evaluated the algorithm on their own dataset, on which they obtained 91.8% precision and 96.6% recall.

Neumann and Matas [22] also use a multiscale approach, but their purpose is to detect strokes instead of pure edges. For this they used a filter specially designed to detect strokes at different scales, which is equivalent to edge detection between two “ridge” orientations that form a stroke. After computing the stroke image, they threshold it to obtain a binary image, where a connected components algorithm is executed to extract candidate character regions.

For each candidate character region, clustering is performed by exhaustively evaluating all bounding boxes generated by the K-nearest neighboring regions. On the ICDAR 2011 dataset the authors obtained state of the art (at that time) performance of 66.4% recall and 79.3% precision, but their method is very expensive. Their Matlab implementation takes 35 seconds per frame, and a C++ implementation might not be fast enough for real-time performance.

2.3 Texture-based Methods

Texture-based methods use texture properties to discriminate text from non-text. First, a sliding window is used over the image, and for each window, texture features are extracted from the window, such as Wavelet transform [23], Local Binary Patterns (LBP) [24] [25] and Histogram of Oriented Gradients (HOG) [26] [27]. Then a machine learning algorithm classification algorithm is trained on the features and used to discriminate text from non-text windows.

Finally, intersecting detections are merged to produce the final text detections. This kind of methods, in general, is computationally very expensive due to the number of windows that have to be evaluated, but trade-offs can be made. Small windows are in general more precise but the number of windows to be evaluated grows very quickly, while the use of bigger windows reduces the number of windows to be evaluated, but precision drops.

Another issue with texture-based methods is scale. Since the sliding window has a fixed size, text that is smaller or bigger than the window might not be detected as such, so multiscale approaches are needed, by building a pyramid representation of the input image [28].

Viola and Jones [29] pioneered the use of weak classifiers for object detection by using AdaBoost to train a cascade of weak classifiers. Their work was focused on face detection, and they used Haar Wavelets [30] as features that are then discriminated by a weak classifier in the form of a step function on a weighted linear combination of the features x_i:

$$f(x) = \text{step}\left(\sum w_i x_i\right) \tag{2.7}$$

The boosted cascade of weak classifiers has been successfully used for object detection. This kind of object detectors can be trained for any kind of object, and it has been successfully used for text detection as well. For face detection, Viola and Jones obtained 93.9% correct classification rate on the MIT+CMU dataset. The evaluation of the Haar wavelets is very fast due to the fact that this wavelet can be quickly evaluated with the use of summed area tables [28].

Another type of texture feature is the Histogram of Oriented Gradients (HOG), initially developed by Dalal and Triggs [26]. Their work focused on human and pedestrian detection, but the technique has been used for other types of objects as well.

The HOG descriptor is computed over a sliding window, first contrast-normalizing the image, and then by computing the image gradient, and dividing the window into overlapping 6x6 cells, and for each cell a 9-bin histogram of the gradient magnitude is computed (from $0°$ to $180°$). Each histogram vote from a gradient element is weighted by the distance to the center of the cell, and by the gradient magnitude.

Then, for each 3x3 block of cells, all histograms are concatenated and normalized to create a block descriptor. All the block descriptors inside the window are concatenated to produce a final HOG descriptor for the window, which forms a very large feature vector. Finally a Linear SVM is trained and used to discriminate object from non-object. The HOG algorithm has also been successfully used for text detection [31].

Since the HOG feature vector is very large, using a Linear SVM has a very high probability of finding a separating plane [32], but the computation of the HOG feature vector is expensive, due to the required sliding window and multiscale approaches. GPU implementations of HOG exist [33] that achieve hard real-time performance.

Hanif et al. [34] used a boosted cascade of weak learners, in the same way as Viola-Jones [29] for text detections. The authors developed features based on the Mean Difference Feature (MDF), the Standard Deviation (SD) and the Histogram of Oriented Gradients (HOG), with 39 features in total (7 from MDF, 16 from SD and 16 from HOG). The features are extracted from each window, by splitting it into blocks. As a weak learner, they used linear discriminant analysis (LDA) and the log-likelihood ratio test (LRT).

They evaluated their cascade on the ICDAR 2003 dataset, and obtained 92.9% detection rate with a 780 LDA weak classifiers, and 94.9% detection rate with 780 LRT weak classifiers. The authors mentioned that their cascade is “fast”, with a computation time of 2 seconds for 640x480 images with 8 scales.

Minetto et al. [31] develop the T-HOG feature based on HOG for detection and recognition of single line text. By exploiting the fact that the distribution of gradients in top, medium and bottom regions of the window are not the same, the authors compute a HOG descriptor by using horizontal cells, setting the number of cell columns to 1. First, the sliding window extracted from the image is resized, keeping aspect ratio, to a constant height between 20 and 25 pixels, and the window is also contrast normalized.

For each cell the HOG descriptor is computed in a similar way that of Dalal and Triggs [26]. To avoid sharp cell boundaries, the authors weight the cell histogram votes with a Gaussian function instead of weighting blocks. Finally all cells histograms are concatenated and the final descriptor is normalized. Linear SVM is again used for text discrimination.

Minetto et al. [35] built a text detector and recognition system called SnooperText that uses the T-HOG feature descriptor for text classification. On the ICDAR 2005 dataset they obtained a precision of 74%, and a recall of 63%, with a better precision than the current state of the art, but with a lower recall. On the Street View Text dataset they obtained a precision of 36% and a recall of 54%, which is better than the current state of the art.

2.4 Region-based Methods

Region-based methods use a region detector (with a connected component algorithm) to detect regions of interest in the input image, which can be considered to be a bottom-up approach, since pixels are individually identified as belonging to a candidate text region, and from the aggregation of pixels, a candidate text region is constructed.

After candidate text regions are obtained, they have to be classified as text or non-text regions, especially if a generic region detector is used. For this heuristic rules and/or machine learning classification algorithms are used [12].

We should note that candidate text regions might be individual characters, or connected text regions, depending on the characteristics of the text in the image. For example, some forms of handwritten letters have connection strokes between each character, while normal computer font characters usually do not have this property.

A clustering or grouping algorithm is used to group candidate text regions into full text lines. Again for this purpose handmade heuristics or unsupervised clustering algorithms are used, such as spectral clustering [36].

The advantage of region-based methods is that in general they are fast and they are scale invariant since no sliding window or assumption about size is required.

For text detection, the most common and best performing region detector is the Maximally Stable Extremal Region (MSER) Detector [37], which detects connected regions with stable area under varying thresholds. In a sense, it takes a grayscale input image and does thresholding and connected component analysis at the same time. MSER can be computed in linear time [11], and in general text regions with constant color are almost always detected as MSER's.

MSER has also been extended to detect color regions [38], but generally only grayscale MSER is used for text detection. The MSER algorithm will be described in detail in Chapter 3.

Neumann and Matas [6] used the MSER detector on grayscale, red, green and blue channels of the input image to obtain candidate character regions, and then used an SVM classifier with a Radial Basis Function (RBF) kernel to discriminate between character and non-character regions. Their SVM classifier used 8 features, which are shown in Table 2.1 and was trained on real-world MSER regions extracted from images on Flickr [7].

Their character classifier obtained a 94.4% correct classification rate. Then they construct a graph of regions, and text line formation is done by means of finding a

[7]http://www.flickr.com

path through this graph that maximizes the probability of being text. The graph is constructed by sequentially extracting horizontal text lines from character regions.

Their method also performs text recognition by means of a 200-feature dimension vector that is generated by first converting each MSER region into a 35x35 pixel matrix, then blurring it with a Gaussian filter and subsampling it to a 5x5 matrix, and repeating this for 8 directions. An SVM classifier was trained with a RBF Kernel.

On the Chars74K dataset, the authors obtained 71.6% correctly recognized characters, 12.1% incorrectly recognized characters, and 16.3% characters that were not detected in the image (Giving a 83.7% correct character detection rate).

On the ICDAR 2003 dataset the authors obtained a precision of 59% and a recall of 55%. For individual characters, 79.9% of the total number of characters was correctly detected.

Aspect Ratio	Relative Height	Compactness
Number of Holes	Convex Hull Area to Surface Area Ratio	Color Consistency
Background Color Consistency	Skeleton Length to Perimeter Ratio	

Table 2.1: Features for Character Classification used by Neumann and Matas [6]

In [14], Neumann and Matas introduced an exhaustive search method over MSER regions to discriminate character from non-character regions. To do this, they prune the tree generated by the MSER algorithm to contain only regions with a high likelihood of being a character, by using a verification function, trained with a SVM classifier on a small set of features (similar to the ones in Table 2.2).

Their method obtained a precision of 65% and a recall of 64% on the ICDAR 2003 dataset, which outperforms the current state of the art when considering the f-score of 63%.

Neumann and Matas [7] then switched to use all possible Extremal Regions (ER) instead of only the maximally stable extremal ones [6] [14]. ERs were extracted from the input image in the RGB and HSI color space [28] projections, as well on the intensity gradient of the image computed with the maximum intensity difference method. The authors report that 94.8 % of all characters are detected as Extremal

Regions in at least one of the image projections.

Neumann and Matas also introduces the use of incrementally computable descriptors as features for a character classifier. This descriptors can be computed along with the ER evaluation by the MSER algorithm, and the authors provide 5 descriptors: area, bounding box, perimeter, Euler number η and horizontal crossings.

Their character classifier uses a probabilistic model trained on the features shown in Table 2.2. A Real AdaBoost decision tree classifier was used on a 2 stage classifier, while the second stage used an SVM classifier with RBF Kernel on three features: the area-to-hole ratio, the convex hull ratio and the number of outer boundary inflexion points.

This method was combined with the exhaustive search method of Neumann and Matas [14]. The authors evaluated on the ICDAR 2011 dataset as well as the SVT dataset. On the ICDAR 2011 dataset, they obtained 64.7% recall, with 73.1% precision, with a recall that is better than the current state of the art.

On the SVT dataset, the authors obtained a recall of 32.9%, and a precision of 19.1%. The authors note that the SVT dataset contains text watermarks in the image, and their method also detects such watermarks, which explains the low precision.

Aspect Ratio $\frac{w}{h}$	Compactness $\frac{\sqrt{\text{area}}}{\text{perimeter}}$
Number of Holes $1 - \eta$	Median Horizontal Crossings

Table 2.2: Features for Character Classification used by Neumann and Matas [7]

MSER regions are known to be sensitive to motion blur [39] [15]. Chen et al. used MSER regions enhanced with edge information to avoid this problem, as well as using the distance transform to obtain stroke width information, which is used for character classification and grouping. Their method obtained 73% precision and 60% recall on the ICDAR 2003 dataset, which is state of the art performance.

Multiple segmentations and a multiscale approach has also been used to improve the results of MSER-based text detectors. Neumann and Matas [40] used this approach to obtain 67.5% recall and 85.4% precision on the ICDAR 2011 dataset, which is

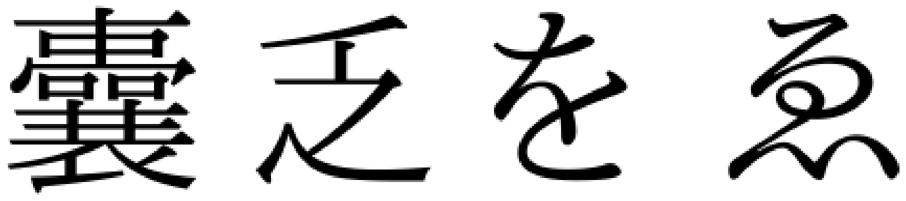


FIGURE 2.8: Example characters from different Asian scripts

the best from the state of the art, but this approach increases computation time considerably, to 3.1 seconds per frame.

2.5 Stroke-based Methods

Text, as drawn by humans, is drawn with pencils, fountain pens and other kinds of writing devices that produce text with an almost constant stroke width. Thus stroke width is a inherent property of text that can be used to detect text regions.

It should be mentioned that not all writing scripts have constant stroke width. In particular many scripts used in Asia are drawn using brushes, and this produces a varying but bounded stroke width, as can be see in Figure 2.8.

One of the first methods to successfully use stroke information to detect text is the Stroke Width Transform (SWT) [8]. This method uses the Canny edge detector [41] to extract edges from the image, then a stroke width image is computed by raycasting in the direction of the gradient for each detected edge pixel, until an appropriate opposite edge is found. This opposite edge must have a gradient angle opposite to that of the ray, plus a tolerance (The authors used $\frac{\pi}{6}$). The Euclidean distance between the starting and ending edges gives an estimation of the stroke width in pixels, and is written to the stroke width image for each ray pixel.

After computing the stroke width image, a connected components algorithm is executed over this image, which groups pixels that have a stroke width ratio less than 3.0. This way connected components represent characters. Character regions are filtered with a set of heuristic rules, shown in Figure 2.3. Then a set of hand tuned heuristic rules are used to group character regions into text lines with a minimum of three characters.

The grouping heuristics consider features such as: similar median stroke width, similar height, distance between letters and color similarity.

Aspect Ratio	Stroke Width $\frac{\sigma}{\mu}$ Ratio
CC Diameter to Median Stroke Width Ratio	Bounding Box Intersection
CC Height	

Table 2.3: Features for Character Classification used by Epshtein et al. [8]

This method has obtained good results on the ICDAR 2003 dataset, with precision of 73% and recall of 60% (with f-score of 66%). Computation time reported by the authors is 0.94 seconds per frame. But the SWT algorithm has many drawbacks, such as the high number of parameters, the use of hand-tuned heuristic rules, and since its a gradient-based algorithm, noise in the image can introduce small holes into the SWT image, which can produce disconnected regions.

In general the SWT algorithm was a milestone in text detection, and other authors have been trying to find alternate ways to compute a more reliable stroke width image. Chen at al. [20] used the distance transform to produce a stroke width image.

The SWT algorithm itself is not appropiate for text with arbitrary orientations, since the heuristic rules used by Epshtein et al. are biased for horizontal text. Yao et al. [42] created a text detector based on the SWT that is able to detect text at arbitrary orientations, by using features that are rotation invariant [17] , and estimation of the minor and major axes, as well as the orientation of the character regions by means of the Camshift algorithm [43].

The amount of features used for character classification and text line grouping is high, with 6 complex features for character classification, and 11 features for text line grouping. A modified HOG is used for character classification.

Since the ICDAR datasets do not contain oriented text, the authors developed their own dataset that contains a mixture of ICDAR 2003 and a proposed dataset with oriented text, the authors obtained 63% precision, 63% recall, while the SWT algorithm [8] obtained 25% precision and 25% recall, which is a clear improvement.

2.6 Other Methods

In general most text detection algorithms are trained and/or evaluated only on Latin characters, and have poor performance or fail if images contain characters from other scripts, such as Asian or Middle East scripts. Also many algorithms assume horizontal text or even text that lies in a line (collinear), which in the real-world does not always hold. Some examples of curved text can be seen in Figure 2.9.

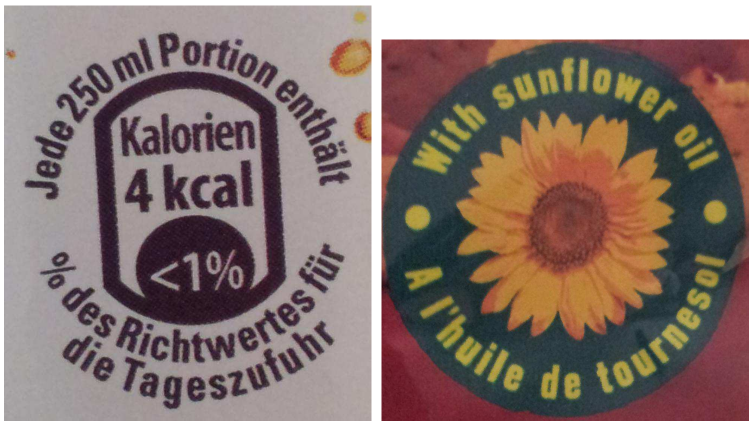

Jede 250 ml Portion enthält
Kalorien
4 kcal
<1%
% des Richtwertes für die Tageszufuhr
With sunflower oil
A l'huile de tournesol

FIGURE 2.9: Examples of Non-Collinear Text

Kasar and Ramakrishman [9] develop a multi-script and multi-oriented text detector, first by using color edge detection with a Canny edge detector on each color channel of the image, and then combining edges from each color channel into one image with the OR binary operator. Edges are then linked and candidate character regions are obtained from connected component analysis and the COCOCLUST color clustering algorithm [44]. Then 12 features are used to identify regions that contain characters. Features are shown in Table 2.4.

Aspect Ratio	Occupancy Ratio	Boundary Smoothness
Boundary Stability	Stroke Width Standard Deviation	Stroke Width to Height Ratio
Stroke Homogeneity	Gradient Density	Gradient Symmetry
Area Ratio	Gradient Angle Distribution	Convex Deficiency

TABLE 2.4: Features for Character Classification used by [9]

An SVM classifier with RBF kernel and a neural network are trained on such features over the ICDAR 2003 dataset. The authors evaluated their algorithm on their own dataset that contains multi-language scripts used in India as well as English, and also contains curved text. They report a precision of 80% and a recall of 86%, which is pretty high for a text detector, but this evaluation only considers per-pixel character detection, and the dataset used by the authors is not public and more conclusions or comparisons cannot be done.

To group characters into words, the authors used Delaunay triangulation and some heuristic rules. The full text detector was evaluated on the ICDAR 2003 dataset, and they obtained a precision of 63%, and a recall of 59%, which is similar to the best methods of the state of the art.

Gomes and Karatzas [45] developed a text detector with a different structure than other detectors presented in the literature. Their concern was to develop a text detector that can detect characters and text with any script, without taking previous assumptions about any specific script. To do this, their work is based on perceptual organization, which is the grouping of perceptually significant atomic objects, since this is in theory what humans use to recognize textual information.

The authors start by extracting MSER regions from the input image, and filtering character regions with simple rules based on region size, aspect ratio, stroke width variance and number of holes. Then characters are clustered by using perceptual organization, by means of a different set of features (geometry, mean Region color, boundary mean color, stroke width and mean gradient magnitude on the border). Group hypothesis are generated and evaluated with an evidence accumulation framework and only meaningful text clusters are output as detections.

To evaluate their approach, the authors used the KAIST dataset [46], which contains images of English and Korean text. On this dataset, they obtained a precision of 66% and a recall of 78%, which is higher than the recall of 60% presented by Lee at al in [46].

Wang et al. [47] used convolutional neural networks (CNN) for text detection and recognition (end-to-end). First they trained patches of character images from the ICDAR 2003 dataset, as well as generated character images, using unsupervised feature learning [48] to extract features, and then used a 4-layer CNN to do text detection with such features.

A CNN is a type of neural network used for image processing and object detection that is similar to do filtering with set of banks, but unlike standard image processing, the filters are "learned" by the network during the training process.

The authors obtained state of the art results, obtaining f-score of 76% on the ICDAR 2003 dataset, and f-score of 46% on the SVT dataset, which is better than the state of the art of [3]. For character and word detection, the authors also obtained 90% recall on the ICDAR 2003 dataset, and 70% recall on the SVT dataset.

In general CNN fall into the category of texture-based methods, and also have the performance problem due to the sliding window. CNN methods also have issues with training, due to the massive amount of parameters in the network, which requires a massive amount of training data, which also makes training times very large. Wang et al. [47] used GPUs for training, which shows the problem.

2.7 Discussion

Scene text detection is a not solved problem. This can be seen in the evaluation of algorithms under the SVT dataset, where precision and recall are usually low, less than 50%. This indicates the complexity of the problem. While most algorithms are evaluated under the ICDAR datasets, such datasets do not really represent text in natural scenes.

Edge-based methods are slow and the use of filters does not generalize to non-latin scripts. In general, filters in specific directions are used for edge detection, and this works well for Latin characters, since strokes are only horizontal, vertical or diagonal. But for other types of scripts, such as Kannada, CJK and Asian scripts, this assumption does not hold, since those scripts have strokes in almost any direction.

Texture-based methods are also generally slow, since the sliding window approach requires the evaluation of a very large amount of windows, thus making a real-time implementation impossible. GPU implementations of HOG can run in real-time [33], but this approach is very specific to a certain type of hardware, and does not generalize to the kind of devices that end users use, such as cellphones and tablet computers.

Most text detection algorithms make implicit or explicit assumptions about the script that will be detected [15]. Most algorithms can only detect Latin characters and they have not been trained or evaluated on other scripts. Since the minority of people in the world speak Latin-based languages (37.1% to be precise), there is still much work to do in order to be able to detect non-latin scripts.

In general, the number of features used by text detection algorithms is high, in order to provide a good generalization performance in the train and/or test datasets, but also such features make generalization to other kind of scripts very difficult. For example, a popular feature is the number of holes in a connected component region, since Latin letters have a small number of holes (A has one hole, B has two holes, and such), but Asian scripts have a much higher number of holes, as can be seen in Figure 2.8, so for this kind of feature, a Asian script region could be mistaken for a non-character region, and text detection would fail.

Multi-script [44] and multi-oriented [42] text detectors rely on a high number of features as well, which also makes character classification expensive, since more features are needed, and some features are expensive to compute, such as HOG-based features.

There is big fraction of the literature that relies on hand tuned heuristic rules to group text, and such rules in general do not generalize well to other datasets [15]. Tuning this rules by hand is very tedious, but grid search can also be used to automatically tune, but since the number of parameters is large, grid search becomes untractable.

The Stroke Width Transform [8] and stroke width information is a popular choice for current state of the art algorithms, but the SWT algorithm itself uses heuristic rules for character classification and text grouping. Since the SWT algorithm is based on Canny edge detector, missing and unlinked edges can create disconnected regions that are mistaken for non-text regions.

The use of a tolerance threshold to compare the opposite edge gradient angle is also tricky, since for some fonts there is no opposite edge with the appropiate angle. This can be seen in Figure 2.10, where two angle tolerance thresholds θ_t are presented, and the value of $\theta_t = \frac{\pi}{6}$ as recommended by Epshtein et al. [8] produces holes in the regions. This can be fixed by incrementing the threshold to $\theta_t = \frac{\pi}{2}$, where the regions no longer have holes, but this threshold is extremely permissive and might cause other problems.

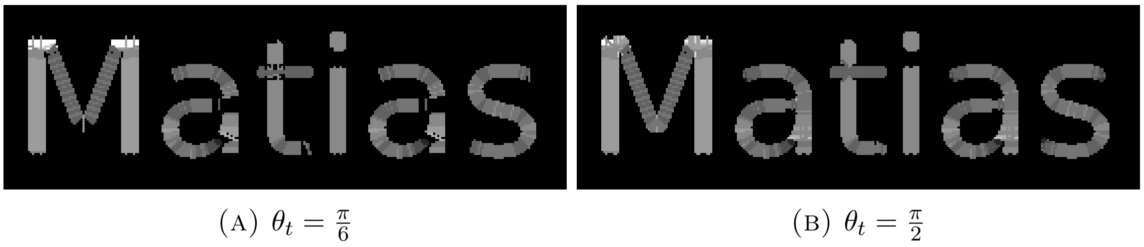

Matias
Matias
(A) $\theta_t = \frac{\pi}{6}$
(B) $\theta_t = \frac{\pi}{2}$

FIGURE 2.10: Holes in SWT character regions

This problem is very characteristic of joints between strokes. Similar issues can be seen in the joints at the top of the "M" letter, where the stroke width is higher, and there is a high variation of the stroke width in that region. Since the SWT and other algorithms use the stroke width variation as threshold for character classification, this can easily fail. This can also be seen with other letters in Figure 2.11.

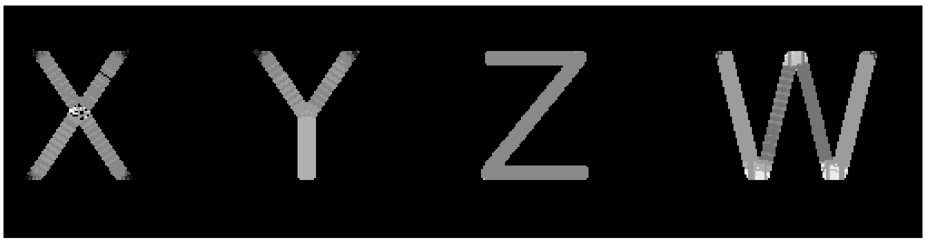


FIGURE 2.11: Stroke Width Transform of Letters X, Y, Z and W

Many text detection algorithms have not been evaluated with respect to computational performance. There is no comprehensive information in the literature about computational performance metrics about text detectors, and only some publications (Like [7] and [8]) provide computation times per frame, but in general such information is not available.

Chapter 3

Background

3.1 Writing Systems

Writing is part of day to day life, and practically almost all information is currently stored in some form of writing, whatever this storage medium is: a clay tablet, parchment, paper, or bits in a digital media. Writing can be defined as a system of storing information, usually coming or representing a given language [49] that humans or computers use.

Writing systems are considered to be one of the most important technological advancements in the history of mankind [49], and such technology is one of the cornerstones of modern society. Writing became a necessity from the needs of storing information, such as commercial transactions, contracts, and recording history in general.

From the need of storage also comes the need for retrieving the stored information. Autonomous systems and robots in general do not have high quality capabilities to read and write in most common writing systems, and much research is devoted to that topic [1].

Some components of a writing system are [50]:

1. A set of symbols usually called characters. The set of characters is called a script.

2. A set of rules that define the meaning of characters as well as their relations between each other.

3. One or more language that will be represented by the writing system.

4. A way to record the characters into a permanent medium.

The result of writing is usually called text. Many writing systems are currently in use around the world. A map of writing systems can be seen in Figure 3.1. While the most common writing system is Latin, this is not the majority as can be seen in the data available in Table 3.1. Only 37.1% of the human population uses the Latin script, and the rest uses different varieties of non-Latin scripts, such as Japanese Kanji, Chinese, Korean, Kannada, Cyrillic, etc.

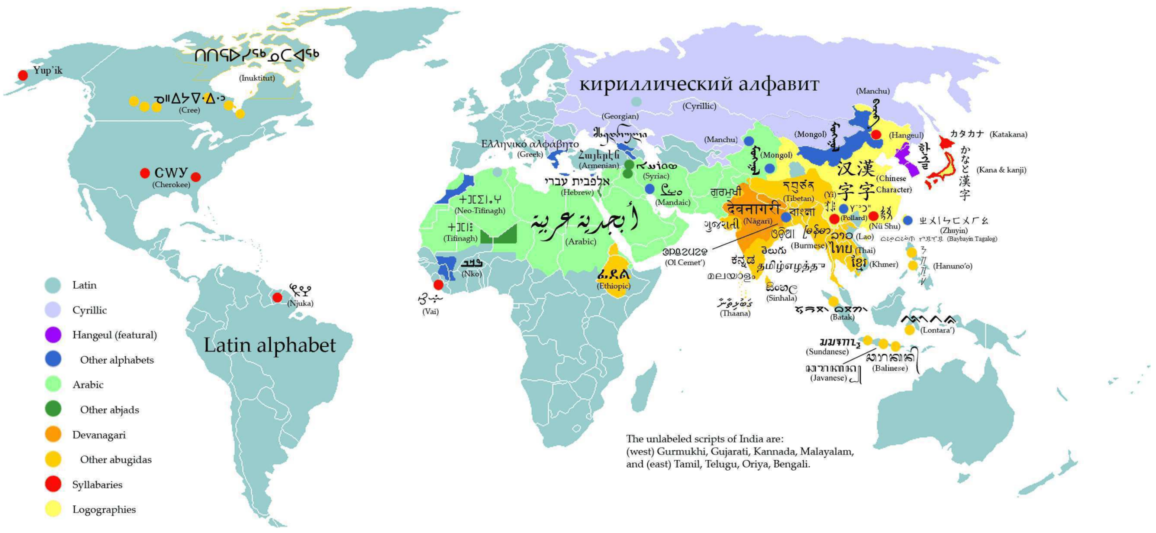


FIGURE 3.1: Writing Systems of the World. **Source**: Wikimedia Commons, File `WritingSystemsoftheWorld.png`, used under the GNU Free Documentation License (GFDL)

Script	Estimated number of Users	% of total number of users
Latin	2600 Million	37.1 %
Chinese	1300 Million	18.6 %
Indian	1200 Million	17.1 %
Arabic	1000 Million	14.3 %
Other	900 Million	12.9 %

TABLE 3.1: Estimated distribution of Script users in the World according to [10]

Many devices are also used to do the actual action of writing (recording characters into the permanent medium). This can range from very simple devices such as a quill and ink, to medium complex pens to highly advanced printers. The high variety of devices that are used for writing is one of the sources of variability that makes the text detection and recognition problem very hard. Other sources of variability between characters are computer fonts, artistic drawings and writing styles.

3.2 Scripts

Here we describe some of the scripts used in this thesis. This does not intend to be a in-depth description of such scripts, but we will only present relevant information for this thesis.

3.2.1 Latin

Latin is the most common script, used by many popular languages such as English, Spanish, Portuguese, Italian, Turkish, German and their variants. The basic Latin alphabet consists of 26 characters, which can have lowercase or uppercase variations, plus the 10 digits and some characters used for sentence control and expression, like the period, comma, parenthesis and other signs (Figure 3.2).

Some languages extend the basic Latin alphabet with additional characters, which usually are just variations of Latin characters. Examples of this are the Spanish tilde characters (á, é, í, ó, ú) and the German Umlaut characters (ä, ö, ü).

3.2.2 Kannada

Kannada is a script used in South India to write the Kannada language. It consists of 13 vowels, 2 vowel-consonants and 35 consonant characters. Characters used by this script can be seen in Figure 3.3. The interest for this script in this work is due to the availability of handwritten and real-world Kannada characters from the Chars74K Dataset [4].

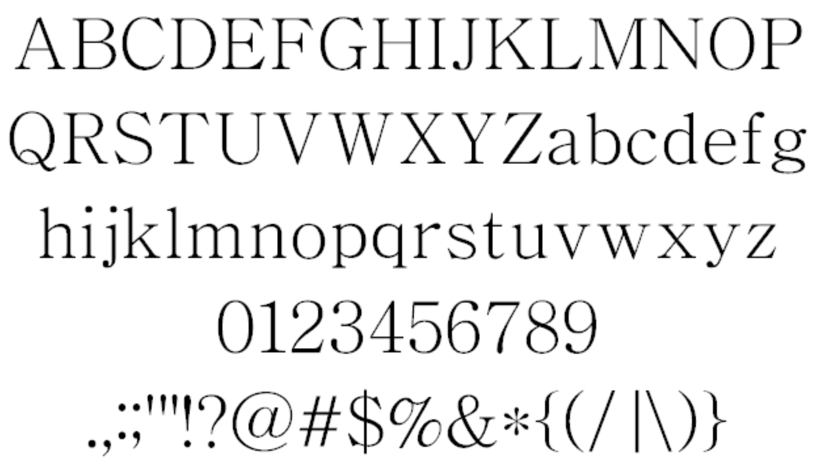


FIGURE 3.2: Characters from the Latin Script

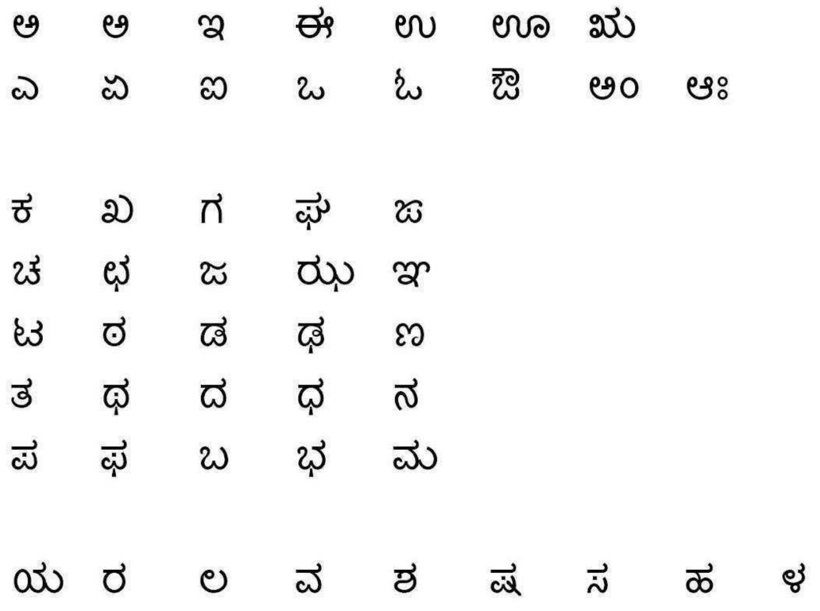


FIGURE 3.3: Characters from the Kannada Script

3.2.3 Chinese-Japanese-Korean (CJK)

CJK is a term that means Chinese, Japanese and Korean, and covers script used in all 3 languages. CJK itself is not a script but a collection of scripts. CJK mostly contains Chinese (Han) characters and their derivations into other Japanese and Korean characters. Some CJK characters can be seen in Figure 3.4.

CJK as a set of scripts contains approximately 75000 different characters that are encoded through Unicode [51]. Separate code point blocks are allocated for CJK characters. The CJK Unified Ideographs block contains 20941 characters in the range U+4E00 to U+9FCC, and the CJKUI Ext A block contains 6582 characters in the range U+3400 to U+4DB5, while 42711 characters are encoded in the CJKUI Ext B block in the range U+20000 to U+2A6D6 [51].

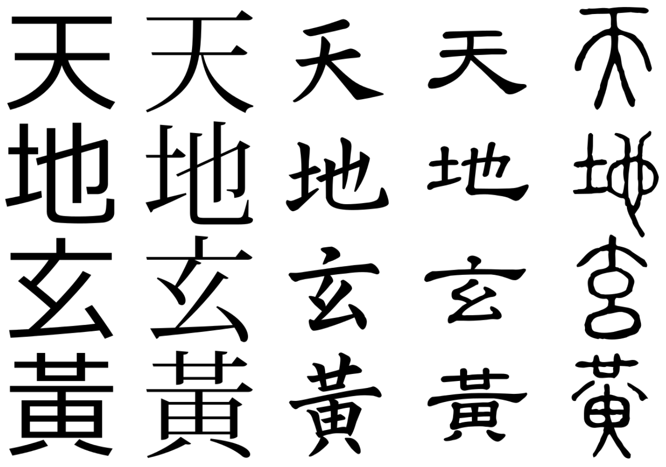


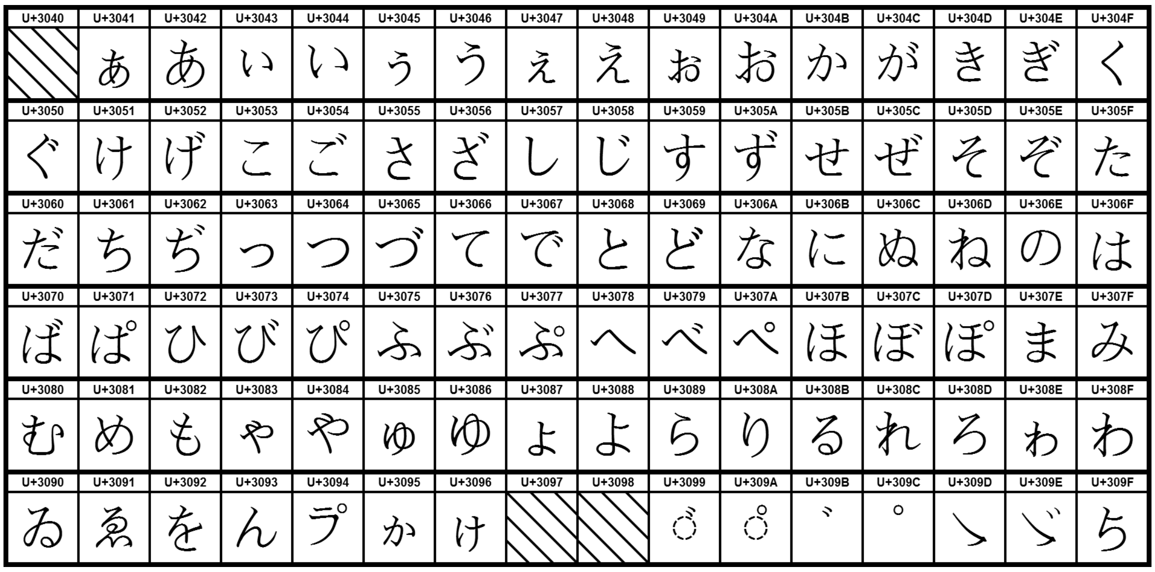
FIGURE 3.4: Some CJK characters. **Source**: Public Domain

3.2.4 Hiragana and Katakana

Hiragana and Katakana are Japanese scripts, part of the Japanese writing system, which also include the Kanji characters and the Latin script. The basic Hiragana character set contains 93 characters, while the basic Katakana character set contains 96 characters. Characters from both scripts can be seen in Figure 3.5.

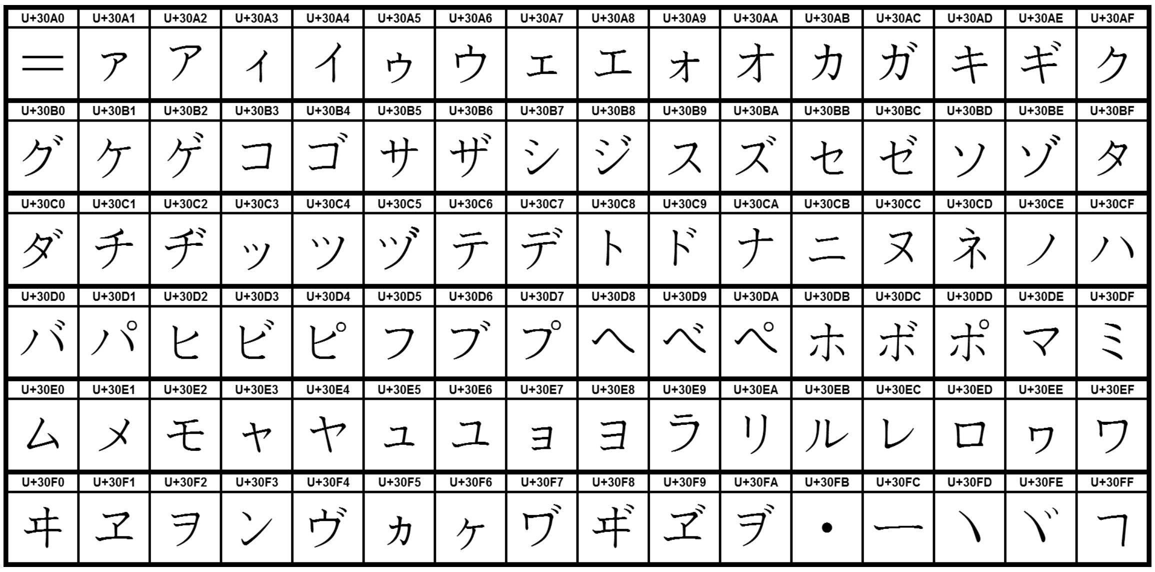
As with many scripts used by Asia, Hiragana and Katakana require several strokes to be drawn, and specific rules were designed to draw characters. Unlike CJK, the stroke width of Hiragana and Katakana is pretty constant with no big variations.

Hiragana and Katakana are usually encoded with Unicode [51]. Hiragana codepoints are in the range of U+3040 to U+309F, while Katakana codepoints are in the range U+30A0 to U+30FF.



U+3040	U+3041	U+3042	U+3043	U+3044	U+3045	U+3046	U+3047	U+3048	U+3049	U+304A	U+304B	U+304C	U+304D	U+304E	U+304F
▨	ぁ	あ	ぃ	い	ぅ	う	ぇ	え	ぉ	お	か	が	き	ぎ	く
U+3050	U+3051	U+3052	U+3053	U+3054	U+3055	U+3056	U+3057	U+3058	U+3059	U+305A	U+305B	U+305C	U+305D	U+305E	U+305F
ぐ	け	げ	こ	ご	さ	ざ	し	じ	す	ず	せ	ぜ	そ	ぞ	た
U+3060	U+3061	U+3062	U+3063	U+3064	U+3065	U+3066	U+3067	U+3068	U+3069	U+306A	U+306B	U+306C	U+306D	U+306E	U+306F
だ	ち	ぢ	っ	つ	づ	て	で	と	ど	な	に	ぬ	ね	の	は
U+3070	U+3071	U+3072	U+3073	U+3074	U+3075	U+3076	U+3077	U+3078	U+3079	U+307A	U+307B	U+307C	U+307D	U+307E	U+307F
ば	ぱ	ひ	び	ぴ	ふ	ぶ	ぷ	へ	べ	ぺ	ほ	ぼ	ぽ	ま	み
U+3080	U+3081	U+3082	U+3083	U+3084	U+3085	U+3086	U+3087	U+3088	U+3089	U+308A	U+308B	U+308C	U+308D	U+308E	U+308F
む	め	も	ゃ	や	ゅ	ゆ	ょ	よ	ら	り	る	れ	ろ	ゎ	わ
U+3090	U+3091	U+3092	U+3093	U+3094	U+3095	U+3096	U+3097	U+3098	U+3099	U+309A	U+309B	U+309C	U+309D	U+309E	U+309F
ゐ	ゑ	を	ん	ゔ	ゕ	ゖ	▨	▨	◌゙	◌゚	゛	゜	ゝ	ゞ	ゟ

(A) Hiragana



U+30A0	U+30A1	U+30A2	U+30A3	U+30A4	U+30A5	U+30A6	U+30A7	U+30A8	U+30A9	U+30AA	U+30AB	U+30AC	U+30AD	U+30AE	U+30AF
゠	ァ	ア	ィ	イ	ゥ	ウ	ェ	エ	ォ	オ	カ	ガ	キ	ギ	ク
U+30B0	U+30B1	U+30B2	U+30B3	U+30B4	U+30B5	U+30B6	U+30B7	U+30B8	U+30B9	U+30BA	U+30BB	U+30BC	U+30BD	U+30BE	U+30BF
グ	ケ	ゲ	コ	ゴ	サ	ザ	シ	ジ	ス	ズ	セ	ゼ	ソ	ゾ	タ
U+30C0	U+30C1	U+30C2	U+30C3	U+30C4	U+30C5	U+30C6	U+30C7	U+30C8	U+30C9	U+30CA	U+30CB	U+30CC	U+30CD	U+30CE	U+30CF
ダ	チ	ヂ	ッ	ツ	ヅ	テ	デ	ト	ド	ナ	ニ	ヌ	ネ	ノ	ハ
U+30D0	U+30D1	U+30D2	U+30D3	U+30D4	U+30D5	U+30D6	U+30D7	U+30D8	U+30D9	U+30DA	U+30DB	U+30DC	U+30DD	U+30DE	U+30DF
バ	パ	ヒ	ビ	ピ	フ	ブ	プ	ヘ	ベ	ペ	ホ	ボ	ポ	マ	ミ
U+30E0	U+30E1	U+30E2	U+30E3	U+30E4	U+30E5	U+30E6	U+30E7	U+30E8	U+30E9	U+30EA	U+30EB	U+30EC	U+30ED	U+30EE	U+30EF
ム	メ	モ	ャ	ヤ	ュ	ユ	ョ	ヨ	ラ	リ	ル	レ	ロ	ヮ	ワ
U+30F0	U+30F1	U+30F2	U+30F3	U+30F4	U+30F5	U+30F6	U+30F7	U+30F8	U+30F9	U+30FA	U+30FB	U+30FC	U+30FD	U+30FE	U+30FF
ヰ	ヱ	ヲ	ン	ヴ	ヵ	ヶ	ヷ	ヸ	ヹ	ヺ	・	ー	ヽ	ヾ	ヿ

(B) Katakana

FIGURE 3.5: Hiragana and Katakana Scripts along with their Unicode codepoints. **Source**: Wikimedia Commons, File `UCB_Hiragana.png` and `UCB_Katakana.png`, User Antonsusi, available under the Creative Commons Attribution 3.0 Germany license

3.3 Road Scenes

In the context of robotics and autonomous vehicles, road scenes play a key role since usually cameras and other visual sensors are installed into such vehicles with the purpose of perception for some required tasks, such as path planning, collision and obstacle avoidance, etc [52].

In general, road scenes have some special characteristics that make extracting information from them a challenge. Here we assume that the camera is pointed parallel to the direction of motion of the vehicle:

1. Some basic structure like the sky is the upper portion of the image, and the road in the lower portion. Usually the sides of a road contain signs that provide information to the driver (traffic signs and panels), or other kind of information such as advertising, buildings, etc. Traffic panels could also be located in the upper part of the image, as shown in Figure 3.6.
2. Text information in panels and signs is usually small in size when compared to the size of the complete image. While driving towards the sign, the text increases, but becomes blurry due to the movement of the vehicle.
3. The vehicle is in constant motion and this produces motion blur in some image frames.
4. There is no control over weather conditions, such as rain, snow, clouds and sunlight.
5. A special challenge that is not considered in the literature is the fact that road scenes also needed to be analyzed in low light conditions or in the night, since most visual sensors give no meaningful information under low light conditions.
6. Items that contain textual information such as traffic signs and panels naturally degrade over time.
7. Temporary occluders could be present in the scene, such as tree leaves and branches.

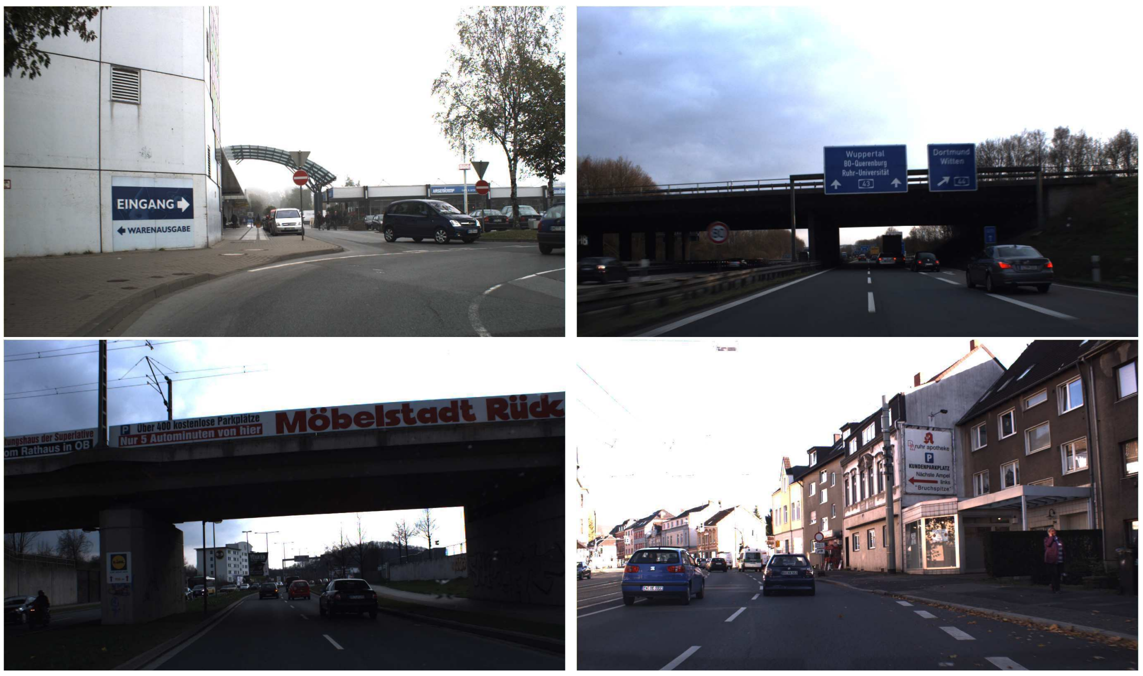


FIGURE 3.6: Some images of Road scenes, from the GTSDB [5]

For this thesis work, we are interested in retrieving the textual information stored in traffic signs and traffic panels, such as the ones seen in Figure 3.7. Here we note that traffic signs and panels do not necessarily use the Latin script.

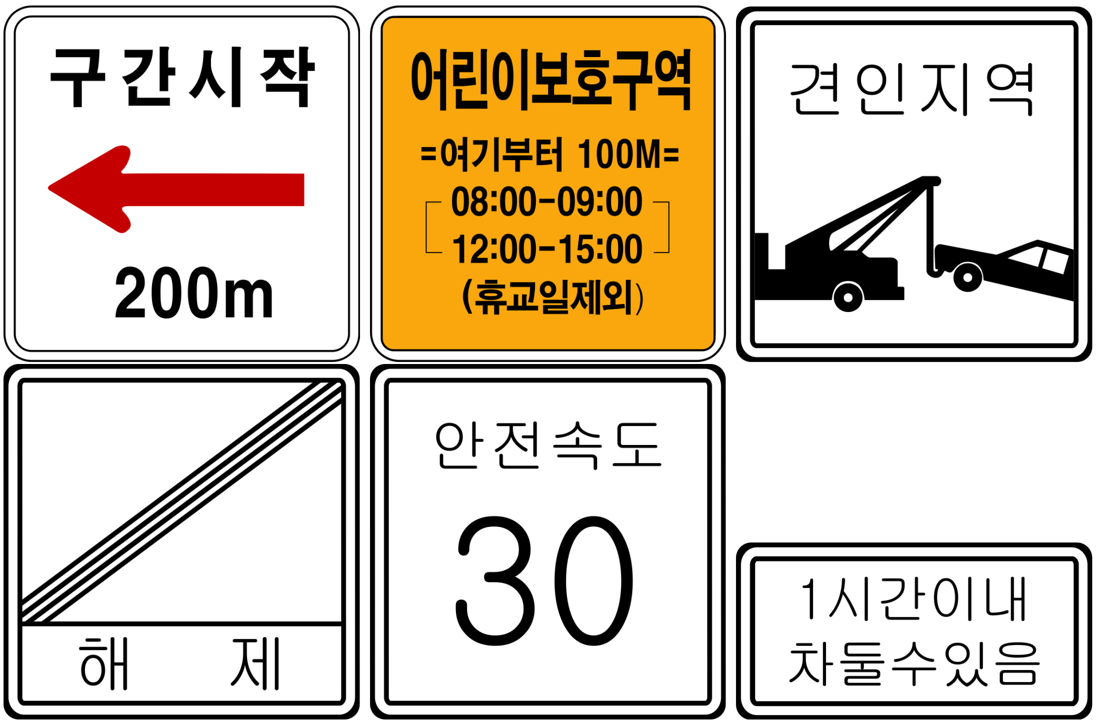


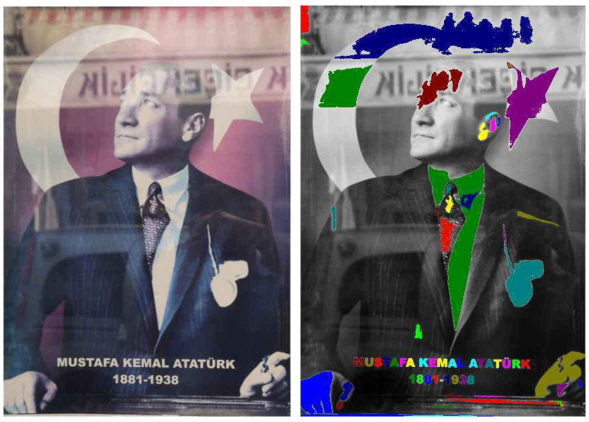
FIGURE 3.7: Korean Traffic Signs and Road Panels. **Source**: Public Domain

3.4 Maximally Stable Extremal Regions

Maximally Stable Extremal Regions (MSER) is a region detection algorithm originally proposed by Matas et al. [37]. The basic concept of this algorithm is that if we threshold an image I with Equation 3.1 with all possible threshold values t (usually from 0 to 255 for 8-bit images), and find the connected components for each binary image T, we will note that some connected regions do not change much, while other regions have huge variations. The regions that have small changes as we change the threshold are called stable regions.

$$T(\mathbf{p}, t) = \begin{cases} 1 & I(\mathbf{p}) > t \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

This process on example image in Figure 3.8a can be seen on Figure 3.9. In this "sequence of frames" we can see that some regions do not change, such as the text in the lower part, the star and the moon crest in the Turkish flag, while other regions have a much bigger variation.

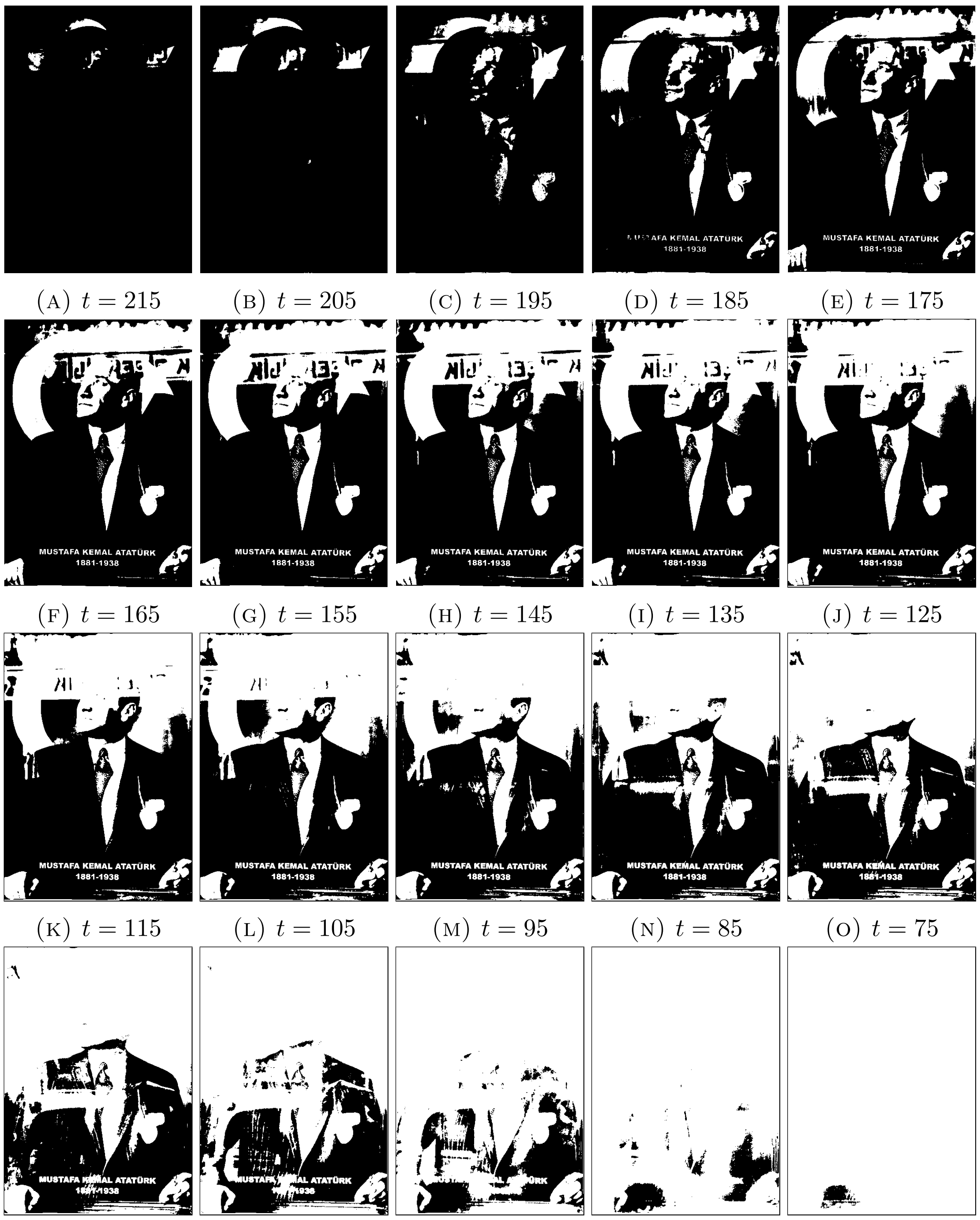


(A) Input Image (B) MSER Regions

FIGURE 3.8: Example MSER detection. In (B) each color represents a different detected maximally stable extremal region.

Initially when the threshold is the maximum value of $t = 255$, the image is completely black, and as the threshold value is decreased, some regions appear, and as we advance, such regions grow and merge with other regions, and when the threshold is the minimum value of $t = 0$, then the image is completely white, which means that the whole image is detected as one region.

The Maximally Stable Extremal Regions of Figure 3.8a are shown in Figure 3.8b. We can note that text was detected as several MSER's.



(A) $t = 215$ (B) $t = 205$ (C) $t = 195$ (D) $t = 185$ (E) $t = 175$

(F) $t = 165$ (G) $t = 155$ (H) $t = 145$ (I) $t = 135$ (J) $t = 125$

(K) $t = 115$ (L) $t = 105$ (M) $t = 95$ (N) $t = 85$ (O) $t = 75$

(P) $t = 65$ (Q) $t = 55$ (R) $t = 45$ (S) $t = 35$ (T) $t = 25$

FIGURE 3.9: Thresholding and connected component analysis of a example image. Borders added for clarity

The MSER algorithm defines an image as the function $I : D \subset \mathbb{N}^2 \rightarrow S$, where S is a totally ordered set, and for 8-bit grayscale images $S = \{0, 1, 2, 3, \cdots, 255\}$ [37]. An adjacency relation in the image I must be defined, which means that pixels in the image have neighbors. The most usual adjacency relations are 4-connectivity and

8-connectivity [28].

A region R on image I is a contiguous subset of D, where contiguity is defined by the adjacency relation. This is the basic definition of a connected component in the image. An Extremal Region (ER) is a region on image I where either all pixels in the boundary of the region are strictly lower or higher than the pixels values in the region. This defines two types of Extremal Regions:

Maximum Intensity Region

$$\forall r \in R \wedge b \in \partial R \quad I(r) > I(b)$$

Minimum Intensity Region

$$\forall r \in R \wedge b \in \partial R \quad I(r) < I(b)$$

Where ∂R denotes the boundary of region R. MSER regions are extracted by constructing a component tree [53], where the tree is built from the bottom-up. Each node of the tree represents one region in the image at a given threshold level, and the parent-child relation between nodes represents region growing when the threshold decreases. This also implies that all pixels that belong to a child region also are contained in the parent region. An example component tree from [53] can be seen in Figure 3.10.

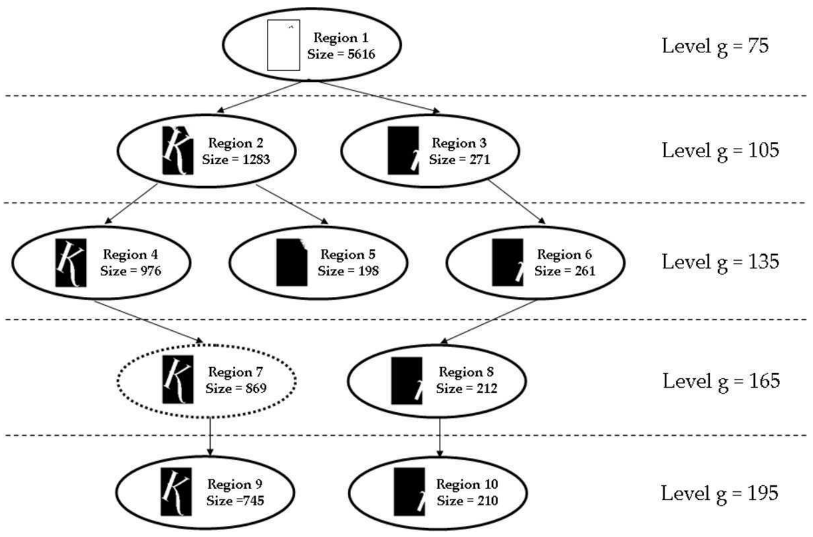


FIGURE 3.10: MSER Component Tree, extracted from [53]

A Maximally Stable Extremal Region is a region R_i^t where the function $\Psi(i)$ [53] has a local minimum:

$$\Psi(R_i^t) = \frac{|\text{area}(R_i^{t-\Delta}) - \text{area}(R_i^{t+\Delta})|}{\text{area}(R_i^t)} \tag{3.2}$$

Where R_i^t is the i-th region at threshold level t on the component tree. We should notice that $\Psi(i)$ also depends on the parameter Δ. This equation evaluates if a region is stable by computing the relative area difference of the region, when the threshold changes from $t - \Delta$ to $t + \Delta$. If this relative area change is a local minimum with respect to the threshold level t, then the region is deemed stable.

There is an algorithm that computes the component tree [11], and it is very efficient, with complexity $O(n \log m)$, where n is the number of pixels in the input image I and m is the number of gray levels. Since this number is usually constant (equal to 255), then this algorithm is linear in the number of pixels. The Linear MSER algorithm is presented in Algorithms 1 and 2.

After constructing the component tree, the tree is scanned and maximally stable regions are extracted. We start at the leaves of the tree and continue going up until we reach the root. Regions that minimize Equation 3.2 are selected as MSER's. Then filtering is performed. A region is output if and only if:

1. It is stable. A minimum stability can also be used.
2. Its size is bigger than the minimum size and smaller than the maximum size.
3. It is sufficiently different from its parent region.

Then the MSER detector has 4 parameters:

Margin Δ

Defines by how many threshold levels a region is considered stable. The most common value used in the literature is 5.

Minimum and maximum areas

Both are numbers in $[0, 1]$ that represent sizes relative to the complete area of the image. They allow to filter very small or very large regions, since in general the MSER output contains too many regions, and very small or very large regions are usually not interesting (such as single pixels and background).

Minimum difference from parent

This parameter sets a minimum threshold on the variation between a region and its parent. This is used to remove duplicate regions, since a region could be stable for several levels of threshold (more than the margin). This parameter is also called minimum diversity.

Algorithm 1 Linear Time MSER Computation [11]

```
Require: Input Image $I$, maximum grey value $t_{\max}$ (default to 255)
 1: function LINEARTIMEMSER($I$, $t_{\max}$)
 2:     Convert input image $I$ to grayscale image $G$.
 3:     Initialize boundary heap as a array of stacks indexed by gray level.
 4:     Initialize component stack.
 5:     curPixel ← (0, 0), curLevel ← $G(0, 0)$
 6:     while true do
 7:         for Neighbors p of curPixel that have not been explored do
 8:             if $G(\mathbf{p})$ < curLevel then
 9:                 Push p into boundary heap.
10:                 curPixel ← p, curLevel ← $G(\mathbf{p})$
11:             else
12:                 Push p into boundary heap.
13:         if curPixel has not been visited then
14:             Add curPixel to the top of the component stack. Here the region grows.
15:             Mark curPixel as visited.
16:         if Boundary heap is not empty then
17:             Pop boundary heap into curPixel.
18:             if $G(\mathbf{curPixel})$ > curLevel then
19:                 PROCESSSTACK ($G(\mathbf{curPixel})$, componentStack)
20:                 curLevel ← $G(\mathbf{curPixel})$
21:         else
22:             PROCESSSTACK($t_{\max}$, componentStack)
23:     return Component tree.
```

Algorithm 2 `processStack` method [11]

```
1: function processStack(newGreyLevel, Component stack)
2:     while newGreyLevel is bigger than the greylevel on top of the component stack do
3:         Pop component stack into a variable called top.
4:         Add top to the component tree.
5:         if newGreyLevel is smaller than the greylevel on top of the component stack then
6:             Set greylevel of top to newGreyLevel.
7:             Push top into the component stack.
8:         else
9:             Merge top and the component on top of the stack.
10:            Push the merged component into the component stack.
```

3.5 The Stroke Width Transform

The Stroke Width Transform (SWT) is a text detection algorithm originally proposed by Epshtein et al. [8]. The core of the SWT is to compute a stroke width image, of the same size as the input image, and where each pixel contains an integer denoting the width in pixels of the most likely stroke region the pixel belongs to, or zero in case the pixel does not belong to any stroke region.

To compute such image, Epshtein et al. use Canny edge detection along with raycasting. For each edge pixel, a ray is cast in the direction of the gradient until another edge pixel is found or the ray gets outside of the image.

If another edge pixel is found, then a comparison between the ray direction and the gradient orientation in the found edge is performed. If the ray is parallel to the edge gradient within a tolerance θ_t, then the stroke width is computed as the distance between the ray origin and the found edge. This can be seen in Figure 3.11. Then the minimum between what is already stored in the stroke width image and the computed stroke width is stored in the stroke width image for each pixel in the ray. The SWT algorithm can be seen in Algorithm 3.

After computing the stroke width image, Epshtein et al do a special kind of median filtering to avoid a problem they encountered in letters such as the “L”, where the stroke width was incorrect due to the long parts of the letter. Their median filter consists on that, for each successful ray, they compute the median stroke width m,

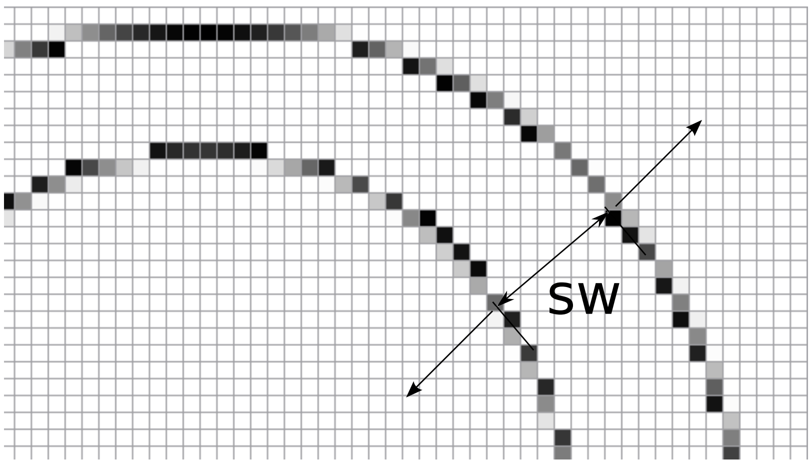


FIGURE 3.11: Gradient vectors and stroke width computation by raycasting

Algorithm 3 Stroke Width Transform

```
Require: Input image $I$, Gradient orientation tolerance $\theta_t$.
1: Do Canny Edge Detection on $I$ and obtain binary edge image $E$.
2: Compute Gradient orientation $\theta(\mathbf{p})$ image.
3: Create stroke width image $S$ with integer format and same size as $I$.
4: $R \leftarrow \emptyset$
5: for Edge position $\mathbf{e} \in E$ do
6:     Do raycast from edge position $\mathbf{e}$ with direction $\theta(\mathbf{e})$ until an edge is found or the ray exits the image.
7:     if Edge is found at position $\mathbf{p}$ and $|\theta(\mathbf{p}) - \theta(\mathbf{e})| < \theta_t$ then
8:         Compute stroke width $sw$ as distance between $\mathbf{e}$ and $\mathbf{p}$.
9:         For all pixels in the ray, set the stroke width in $\mathbf{s} \in S$ to $\min\{S(\mathbf{s}), sw\}$.
10:        $R \leftarrow R \cup \text{ray}$.
11: SWTMEDIANFILTER($S$, $R$)
12: return $S$
```

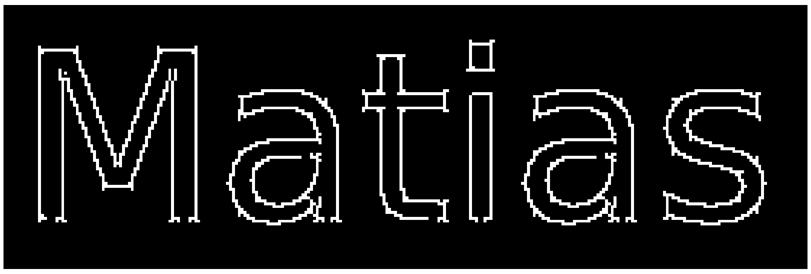
and for each pixel of the ray, if the stroke width is bigger than the median, they set the stroke width in that position to the median value. This median filter can be seen in Algorithm 4.

Algorithm 4 SWT Median Filter

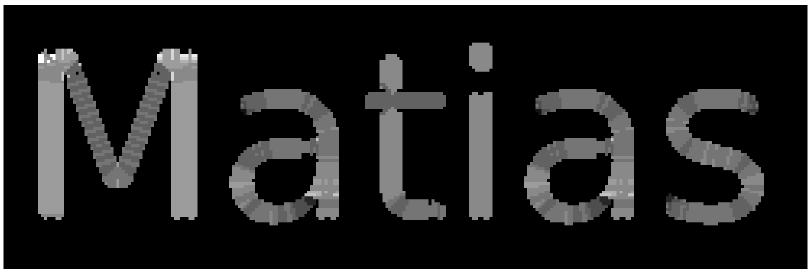
```
Require: Stroke width image $S$ and list of rays $R$.
1: function SWTMEDIANFILTER($S$, $R$)
2:     for Ray $r \in R$ do
3:         Compute median $m$ of the stroke width along the ray $r$ pixels.
4:         for Pixel $\mathbf{p} \in r$ do
5:             if $S(\mathbf{p}) > m$ then
6:                 $S(\mathbf{p}) \leftarrow m$
7:     return $S$
```

An example of the SWT results of a image with one word can be seen in Figure 3.12.

(A) Input Image



(B) Canny Edge Detection



(C) Stroke Width Image

FIGURE 3.12: SWT Example of the word “Matias”

3.6 Support Vector Machines

An Support Vector Machine (SVM) [54] is a machine learning classification algorithm that uses a maximum margin as a criteria to decide the best hyperplane that will separate the positive from the negative examples. SVM is usually preferred over other algorithms as a classifier due to its resistance to overfitting (due to the maximum margin) and the use of the kernel trick to obtain non-linear decision boundaries.

Overfitting is an undesired effect that happens when training a classifier, where the classifier learns the target function in the training data as well as learning the noise in the data, so the trained classifier fails to generalize the desired function when evaluated in sets of data different from the training set [30].

One way to avoid or minimize overfitting is to do cross-validation, which consists of splitting the dataset into two parts: a training set and a validation set. The SVM is trained with the training set, and then the validation set is used to evaluate the performance of the trained classifier, and overfitting would be clear if performance in the validation set is poor.

K-Fold cross validation is another technique used to split datasets for training [55]. But in this method, the dataset is split in k parts at random (with approximately the same sizes), and training proceeds in k rounds. In each round the classifier is trained into $k-1$ splits, and the remaining split is used for validation. This is repeated once for each split of the dataset.

3.6.1 Linear SVM

The most basic SVM is the Linear SVM, where the decision boundary is a hyperplane in the n-dimensional space $\mathbb{R}^n$. Given a set of training data T in the form $T = \{(\vec{\mathbf{x}}, y) \,|\, \vec{\mathbf{x}} \in \mathbb{R}^n\,,\, y \in \{-1, 1\}\}$, the equation of the hyperplane is:

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} - b = 0 \tag{3.3}$$

Where $\vec{\mathbf{w}}$ is the weight vector of the decision boundary, which is perpendicular to the hyperplane, and b is the offset of the hyperplane from the origin. Assuming the training data is linearly separable, then there exists a value of w that generates 2 hyperplanes where the area between the 2 hyperplanes does not contain any value in the training data. This can be seen in Figure 3.13. The equation of such hyperplanes are:

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} - b = 1 \qquad\qquad \vec{\mathbf{w}} \cdot \vec{\mathbf{x}} - b = -1 \tag{3.4}$$

The distance between both hyperplanes is denominated the “margin” of the classifier and its value is equal to $\frac{2}{||\vec{\mathbf{w}}||}$. Since we want to maximize the margin, then we need to minimize the norm of $\vec{\mathbf{w}}$. This can be formulated as an optimization problem, and the constraints are designed to avoid that training data points falling inside the margin area.

Then an SVM can be trained by solving the following optimization problem:

$$\min_{\vec{\mathbf{w}},b} \frac{1}{2}||\vec{\mathbf{w}}||^2 \qquad \text{subject to} \\ \forall i \quad y_i(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_i - b) \geq 1 \tag{3.5}$$

After training the values of $\vec{\mathbf{w}}$ and b are stored and can be used to classify new examples with the following equation:

$$f(\vec{\mathbf{x}}) = \begin{cases} 1 & \text{if } \vec{\mathbf{w}} \cdot \vec{\mathbf{x}} - b > 0 \\ -1 & \text{if } \vec{\mathbf{w}} \cdot \vec{\mathbf{x}} - b < 0 \end{cases} \tag{3.6}$$

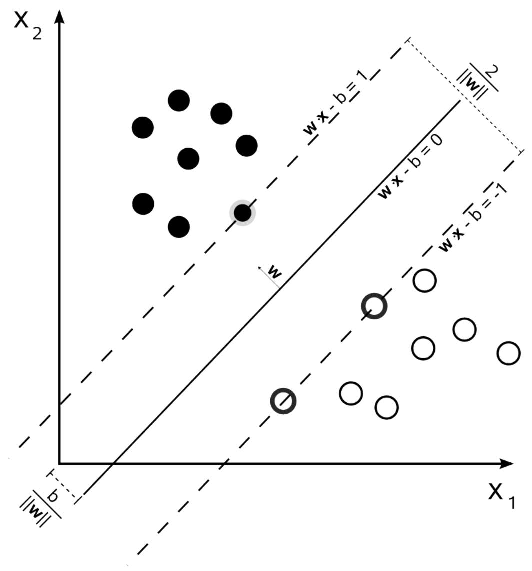


FIGURE 3.13: Geometry of a Support Vector Machine. **Source**: Public Domain

The training examples that line on the margin are called support vectors and give the name to the SVM. The importance of the support vectors is that only the support vectors are required to train the SVM, and if any non-support vector training example changes, the trained SVM will be the same (unless a new example falls inside the margin).

Then the number of support vectors is used as a measurement of the complexity of the trained classifier, since if less support vectors are needed, the trained SVM is simpler.

But this classic Linear SVM only works if the training data is linearly separable, and many real datasets are not linearly separable. But the SVM can be converted into a classification algorithm that also work for such cases.

3.6.2 Soft-Margin SVM

The Soft-Margin SVM is used in the case that the training data is not linearly separable and some examples will be misclassified [55]. For this, a small modification of the constraints is required:

$$y_i(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_i - b) \geq 1 - \xi_i \tag{3.7}$$

Where $\xi_i \geq 0$ for all i, and ξ_i represents the misclassification "error" of a given training example. Then the modified optimization problem used to train the Soft-Margin SVM is:

$$\begin{aligned} \min_{\vec{\mathbf{w}},b} \frac{1}{2}||\vec{\mathbf{w}}||^2 + C\sum_i \xi_i \qquad \text{subject to} \\ \forall i \quad y_i(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_i - b) \geq 1 - \xi_i \\ \xi_i \geq 0 \end{aligned} \tag{3.8}$$

Where $C > 0$ is a parameter that controls how much misclassification is tolerated when training the SVM. In some way it can be considered similar to a regularization coefficient used in other machine learning algorithms. A small value of C will make the SVM tolerate many misclassifications, while a big value of C will make the SVM strict, and no misclassifications will be tolerated. When $C \to \infty$ then the Soft-Margin SVM behaves like a linearly separable SVM [55].

Chapter 4

Proposed Approach

In this chapter we present our text detection approach.

4.1 Overview

After evaluating the literature, we selected a classic text detection pipeline with 4 stages:

Region Extraction
We used MSER to extract regions from the image, which represent characters. MSER is well known for text detection, since it behaves very well under noise, detected regions are invariant to affine transformations [56] and a tracking framework exists which can improve performance [53].

Character Filtering
Then we perform filtering to select regions that are characters and discard the rest. To do this we propose a new novel feature, called the Histogram of Stroke Widths (HSW) that has a very high classification rate and can discriminate between character and non-character regions very well.

Text Line Grouping
Then we group character regions into text lines via a novel raycasting method that we also propose.

Text Verification

For each text line produced in the previous stage, we verify that they are indeed a valid text region by means of the Histogram of Stroke Widths. Every valid text line is then the output of our detector.

A block diagram of our detector architecture is shown in Figure 4.1. For our detector we took the following assumptions:

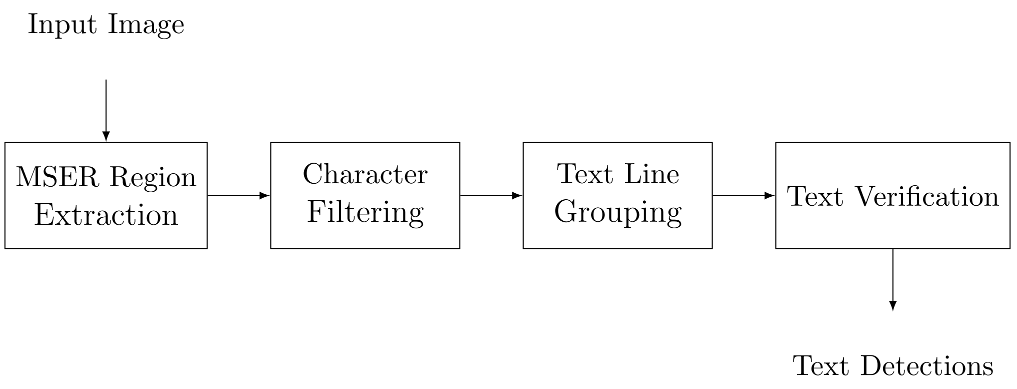


FIGURE 4.1: Text Detector Block Diagram

1. Individual characters can be detected as connected components from the MSER algorithm.
2. Characters can belong to any script. In particular we tested our classification algorithm with Latin, Kannada, CJK, Hiragana and Katakana Scripts, but the Histogram of Stroke Widths should work with any script.
3. Text lines formed by consecutive connected components form a line. This assumption is violated if the text is curved, but we believe our method has the potential to also consider curved text if the character orientation can be taken into account.

Two example results from our proposed text detection pipeline can be seen in Figures 4.2 and 4.3. For each figure, there are subfigures that show the resulting output for each stage.

4.2 Region Extraction

For region extraction we use Maximally Stable Extremal Regions over a grayscale projection of the input image. The advantages of such approach are:

Scale invariance
: Regions can be detected at any scale, and the detector allows to set a minimum and maximum region sizes, so we don't need to process really big or very small regions. Unlike other methods, MSER region detection does make assumptions about the size of the text.

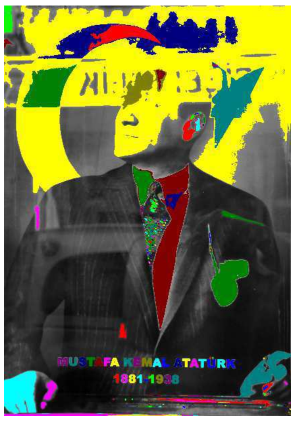
Affine transform invariance
: If a region is affine transformed, the transformed region will also be detected, and this transformation can be undone with the algorithm presented in [56], but we did not use this option.

Text as MSER
: Characters and Text is well known to be detectable as MSERs [57], and in general this performs very well.

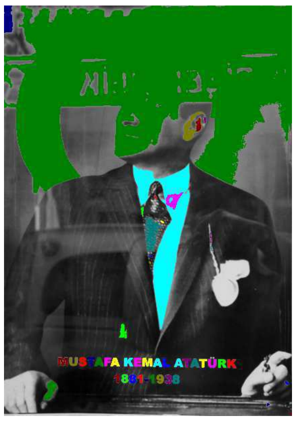
Low parameter count
: The MSER detector has only 4 parameters and their values generalize very well under many different images. The only parameters that need tuning are the minimum and maximum region sizes.

The MSER algorithm detects light regions over a dark background (this is called MSER+). To detect dark regions over light background, it is required to take image with inverted pixel values (Difference between maximum grayscale value and each pixel value) and to run the algorithm again on that inverted image (This is called MSER-) [53]. Then we can use both sets of regions for further processing in the pipeline.

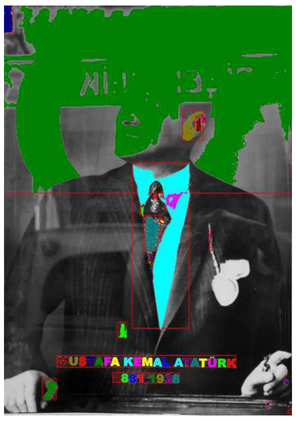
(A) Input Image



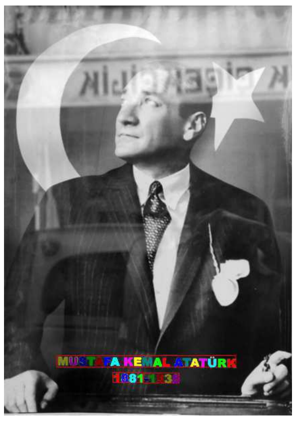
(B) MSER Regions (218 regions)



(C) Filtered Character Regions (48 regions)

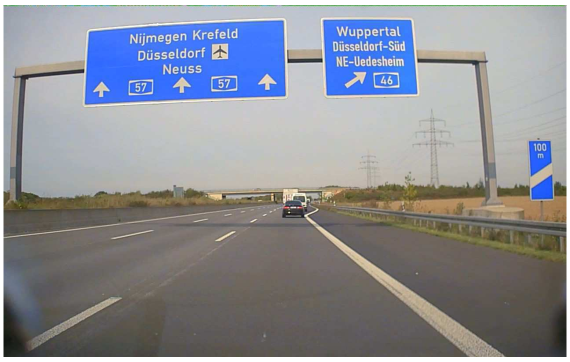


(D) Text Lines (23 lines)

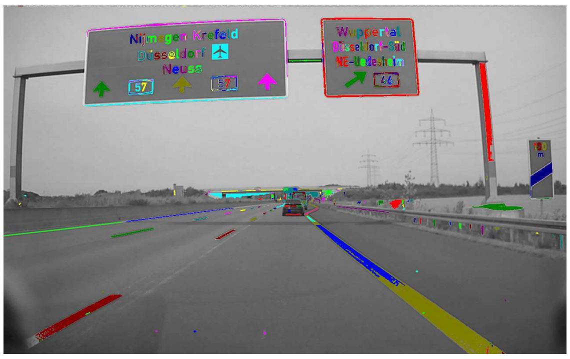


(E) Filtered Text Lines (3 lines)

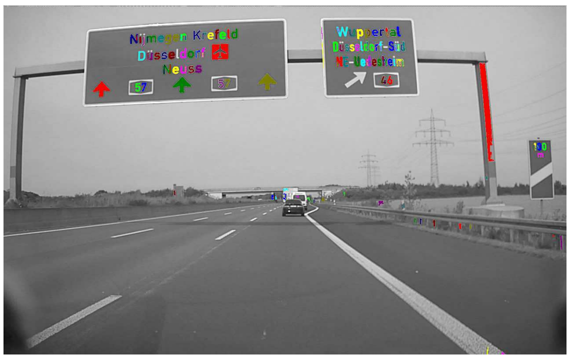
FIGURE 4.2: Text Detection Pipeline results, divided per stage, with a simple image



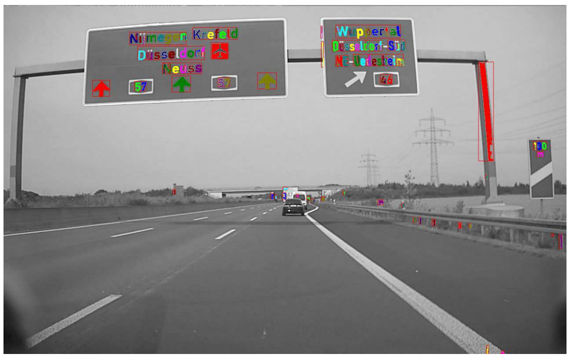
(A) Input Image



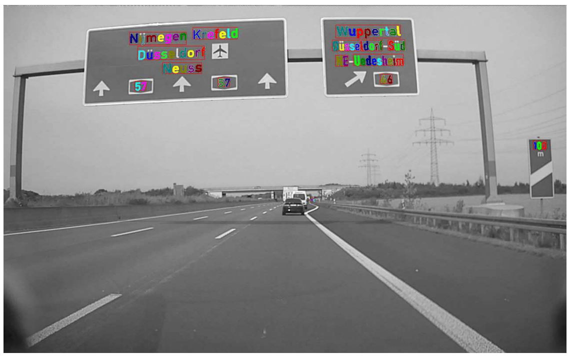
(B) MSER Regions (270 regions)



(C) Filtered Character Regions (138 regions)



(D) Text Lines (76 lines)



(E) Filtered Text Lines (14 lines)

FIGURE 4.3: Text Detection Pipeline results, divided per stage, with a road scene image

4.3 Character Filtering

To filter character regions from non-character regions, we developed a feature descriptor for text regions. We call this feature the Histogram of Stroke Widths, and as like the name says, we extract stroke widths from the region, and make a histogram of them.

4.3.1 Histogram of Stroke Widths

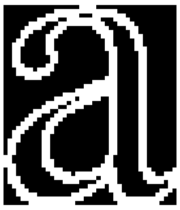
Since text and character regions have "almost" constant stroke widths [8], the histogram should have a very noticeable peak around the stroke width used to draw the character, with variations since the stroke width sometimes varies according to different fonts, different writing styles and on different characters.

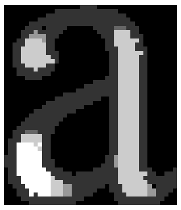
We noticed that the distribution of stroke widths between a character and a non-character region were different. In Figure 4.4 we present the histogram of stroke widths of three character regions, and in Figure 4.5 the same but for three non-character regions. We can see that the histograms are different, non-character regions accumulate more mass around the first bins, while character regions have a peak that is further to the right of the first bin.

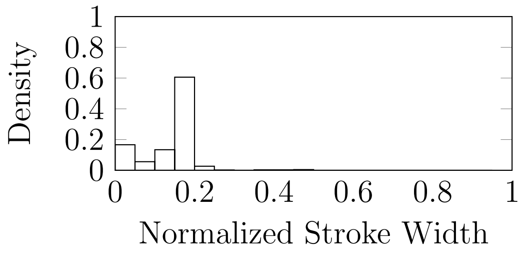
But we make no such assumptions on the difference between histograms, and we will let a classification algorithm be trained over the histogram of stroke widths feature to distinguish character from non-character regions. Fraunhofer IAIS was using a similar method of stroke width distributions for the same purpose, but comparing it with template histogram using Earth Mover's distance [58].

Since stroke width values change with font sizes, distance to the camera or size of the regions, we normalize stroke widths by the width of the bounding box of the connected component region. This provides scale invariance since stroke widths in general cannot be larger than the width of the connected component. After building the histogram, we normalize it by dividing each bin by the sum of all bins.

(A) Region

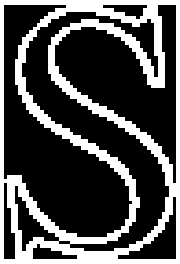

(B) Edges

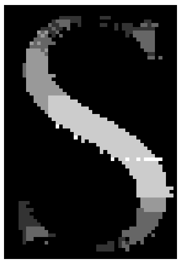

(C) SWT

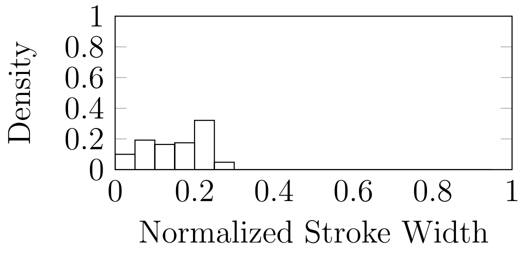


(D) Histogram

(E) Region

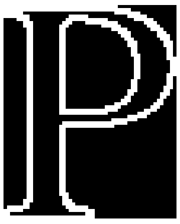

(F) Edges

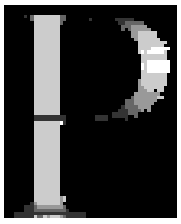

(G) SWT

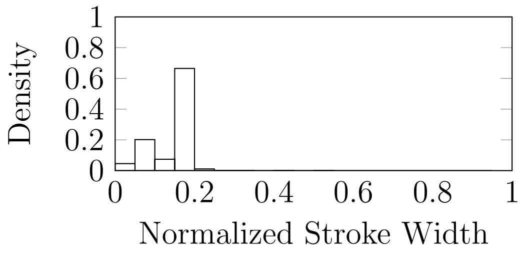


(H) Histogram

(I) Region


(J) Edges


(K) SWT



(L) Histogram

FIGURE 4.4: Histogram of Stroke Widths for 3 Character Regions (Computed with SWT and 20 histogram bins)

The histogram of stroke widths algorithm is shown in Algorithm 5. Before we can use this algorithm, we need a way to extract stroke widths from a connected component. We used two different methods: One based on the Stroke Width Transform [8] and another using run-lengths to approximate the stroke widths. Run lengths are explained in Section 4.3.3.

To classify characters, we compute the histogram of stroke widths, and then we train a Linear SVM with a soft-margin as classifier. The input features for the SVM classifier are the histogram of stroke widths, and two additional features:

Aspect Ratio

The aspect ratio r of a connected component region is the ratio of width to height of the bounding box containing the component:

$$r = \frac{w}{h} \tag{4.1}$$

Occupancy Ratio

The occupancy ratio o of a connected component region is the ratio of the number of pixels n in the region to the total area of the bounding box containing the component:

$$o = \frac{n}{wh} \tag{4.2}$$

The only parameter required to compute the HSW feature is the number of histogram bins, which should be greater than zero. We decided to use a Linear SVM because of previous results that used similar techniques, such as HoG [26]. We also experimented with different kernels, such as Gaussian and polynomial, but they were not superior to a Linear kernel.

Algorithm 5 Histogram of Stroke Widths Computation

```
Require: List of stroke widths $W$, width of the region $R_w$
1: and number of histogram bins $N$.
2: function HistogramOfStrokeWidths($W$, $R_w$, $N$)
3:     Create output histogram $H$ with $N$ bins.
4:     for $w \in W$ do
5:         Compute normalized stroke width $sw = \frac{w}{R_w}$
6:         Vote into histogram $H$ in bin number $sw \cdot N$.
7:     Normalize $H$ by dividing each bin by the sum of all bins.
8:     return $H$
```

4.3.2 SWT-based Stroke Width Computation

Given a connected component region, we convert this component to an image. Assuming the component is represented as a list of points (x, y) indicating the pixel coordinates that belong to the region, to convert this representation to an image, we first compute the bounding box of the region, construct a image of the same size as the bounding box, then for each point in the region, translate it by the upper-left corner and set its value in the image to 255. Then we compute the image gradient over the region image, by using the local difference operator, which is defined by the following convolution kernels:

$$L_h = \begin{pmatrix} 1 & -1 \end{pmatrix} \qquad L_v = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \tag{4.3}$$

We decided to use the local differences kernel instead of the more common Sobel kernels because most detected regions are small, and the Sobel kernel incorporates smoothing, which affects the stroke width computations. Local differences works much better than Sobel for this case.

After computing the image we do a simple edge detection algorithm. Since the region image is completely noise-free [1], we can obtain the edges by taking the gradient magnitude image and thresholding it with $t = 1$. This procedure works because the region image was created by our algorithm and is just a new representation of the region.

Once we have the edges and the gradient orientation, we can use the Stroke Width Transform to obtain the stroke widths. We made a small modification to the SWT algorithm.

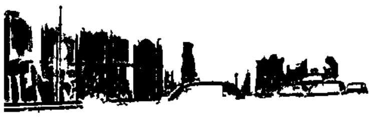
To avoid "holes" in the regions obtained from SWT and missing rays due to noise, we removed the condition that gradient orientation at ray stop positions must be roughly parallel to the ray direction. Our ray stops when it hits an edge or the ray exits the image. We can remove this constraint because the region image is completely noise free (since it is generated from a region and not extracted from an image patch).

Our SWT-based stroke width extraction algorithm is presented in Algorithm 6. The stroke widths on Figures 4.4 and 4.5 were computed with our SWT algorithm.

[1] We say it is noise free because it is just another representation of the region, and in this case the image has two possible values: 0 or 255

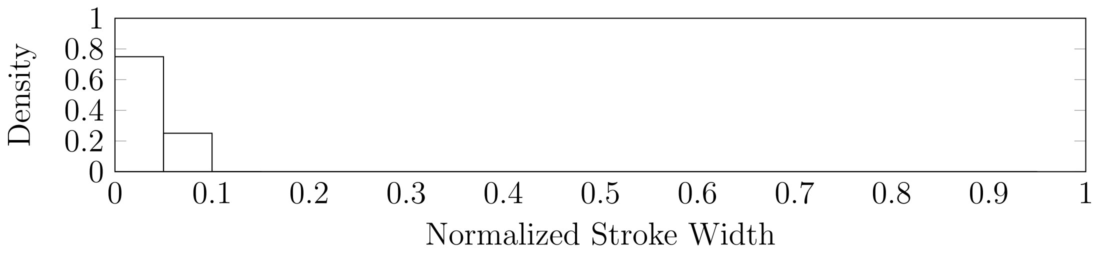
Algorithm 6 SWT-based stroke width extraction

```
Require: Region $R$ as a list of points (Pixel positions).
 1: Convert $R$ to a grayscale image $I$.
 2: Do edge detection by convolving $I$ with the local differences kernel $L_h$ and $L_v$
    (Equation 4.3).
 3: Compute gradient magnitude $M(\mathbf{p})$ and gradient orientation $\theta(\mathbf{p})$ images.
 4: Threshold $M > 1$ to obtain a binary edge image $E$.
 5: Create stroke width image $S$ with integer format and same size as $I$.
 6: $R \leftarrow \emptyset$
 7: for Edge position $\mathbf{e} \in E$ do
 8:     Do raycast from edge position $\mathbf{e}$ with direction $\theta(\mathbf{e})$ until an edge is found or
    the ray gets outside of the image.
 9:     if Edge is found at position $\mathbf{p}$ then
10:         Compute stroke width $sw$ as distance between $\mathbf{e}$ and $\mathbf{p}$.
11:         For all pixels in the ray, set the stroke width in $\mathbf{s} \in S$ to $\min\{S(\mathbf{s}), sw\}$.
12:         $R \leftarrow R \cup \text{ray}$.
13: swtMedianFilter($S$, $R$)
14: $W \leftarrow \emptyset$
15: for Pixel positions $\mathbf{r} \in R$ do
16:     $W \leftarrow S(\mathbf{r}) \cup W$.
17: return $W$
```

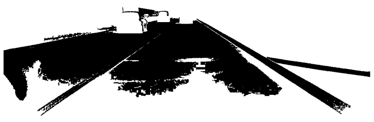

(a) Region

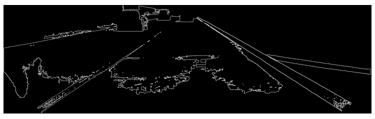
(b) Edges

(c) SWT

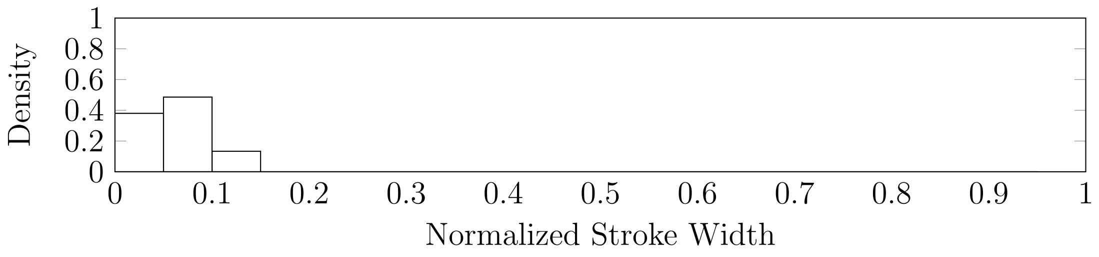


(d) Histogram

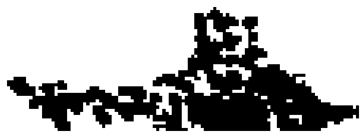

(e) Region

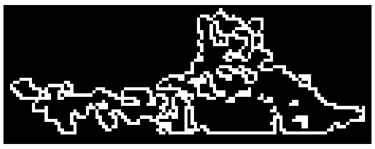

(f) Edges

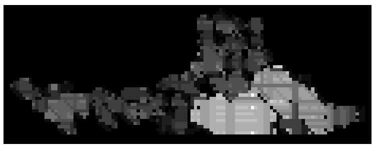
(g) SWT

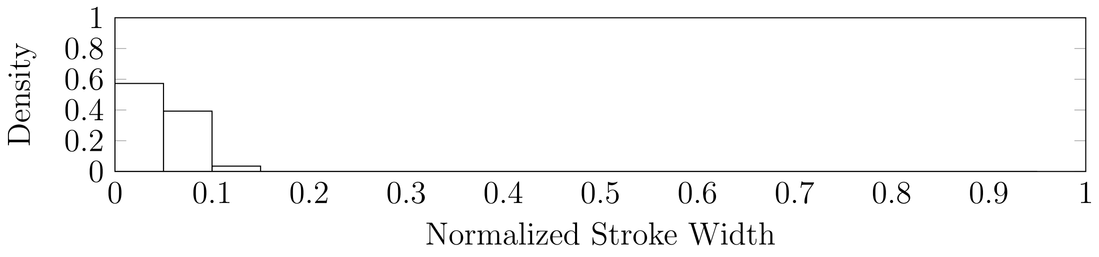


(h) Histogram


(i) Region


(j) Edges


(k) SWT



(l) Histogram

Figure 4.5: Histogram of Stroke Widths for 3 Non-Character Regions (Computed with SWT and 20 histogram bins)

4.3.3 Run-Length Stroke Width Computation

We also developed a different way to estimate stroke widths by means of run lengths. Run-Length Encoding (RLE) is a method for data/image compression [28], where a binary image is compressed by storing consecutive elements with the same value as a single value, instead of the origin run of elements. A example of RLE compression can be seen in Figure 4.6. For stroke width extraction we use run lengths without line wrapping.

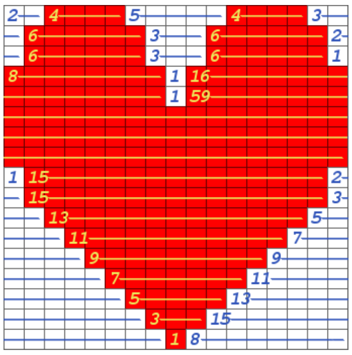


FIGURE 4.6: Run-Length Encoding Image Compression **Source**: Wikimedia Commons, File `RunLengthEncoding.png`, used under the GNU Free Documentation License (GFDL)

For example, the binary string “00001110100000001111” can be encoded as 4 zeros, then 3 ones, then 1 zero, then 1 one, then 6 zeros, and finally 4 ones. The same concept can be used to estimate the stroke widths of a region, by converting the region to a binary image (same as SWT stroke widths), and computing the run lengths for each scanline of the image, but only for pixels that have the value of “true”. Pixels with a value of “false” correspond to the background and are not relevant for stroke width computation.

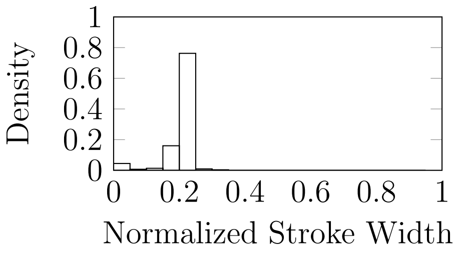
Using the run lengths for stroke width extraction will overestimate many stroke widths, but our experiments show (in Section 5) that this is also a viable approach. Our stroke extraction algorithm is presented in Algorithm 7.

A comparison between SWT and Run-Length stroke width extraction is shown in Figure 4.7. In that figure we can see that both histograms look very similar, with differences in long regions where the run lengths based approach overestimates the stroke width.

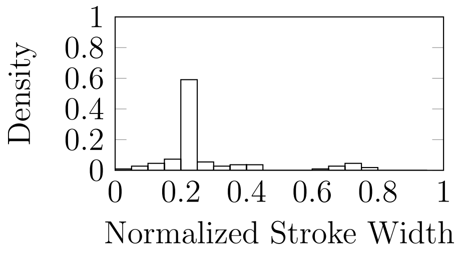
Algorithm 7 Run Lengths based stroke width extraction

```
Require: Region $R$ as a list of points (Pixel positions).
 1: Convert $R$ to a binary image $I$.
 2: strokes $\leftarrow \emptyset$
 3: for $y = 0$ to I.height do
 4:     $is \leftarrow$ false,    start $\leftarrow 0$
 5:     for $x = 0$ to I.width do
 6:         if $I(x, y) =$ true and $\neg is$ then
 7:             $is \leftarrow$ true,    start $\leftarrow x$
 8:         if $I(x, y) =$ false and $is$ then
 9:             $is \leftarrow$ false
10:             $sw \leftarrow x - start$
11:             strokes $\leftarrow$ strokes $\cup\ sw$
12: return strokes
```

(A) Region

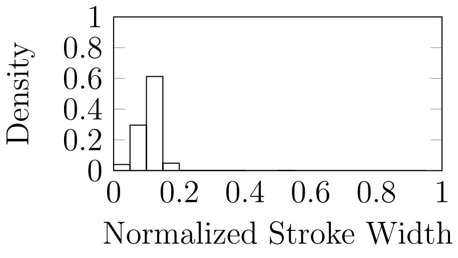


(B) SWT Histogram

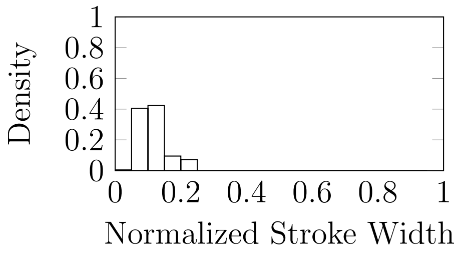


(C) Run-Lengths Histogram

(D) Region

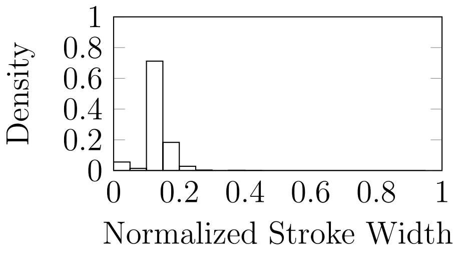


(E) SWT Histogram

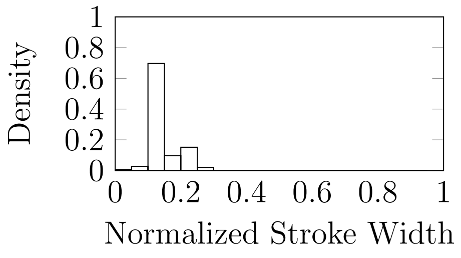


(F) Run-Lengths Histogram

(G) Region



(H) SWT Histogram



(I) Run-Lengths Histogram

FIGURE 4.7: Comparison of SWT and Run-Lengths stroke extraction (20 histogram bins)

4.4 Text Line Grouping

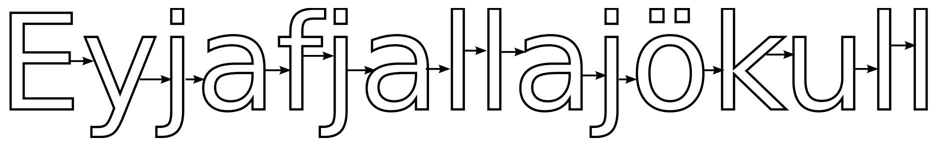


FIGURE 4.8: Raycast-based Text Line Grouping

After filtering character regions, we need to group them into text lines. For this we propose a method that uses raycasting from one region to find the next region in the text line. This method is similar to the one presented in [59]. The basic idea can be seen in Figure 4.8.

Given a list of character regions, first we convert it to a labeled image, which is a integer image where a value of zero means background or no label, and a positive value ≥ 1 indicates the index of the region in the list that pixel belongs to.

Now from the previous label image we proceed to do raycasting. For each character region, we take the middle point of the right side of the bounding box, and cast a horizontal ray until we find the next region, or the ray exits the image.

To avoid false positives and regions that are too far to be valid text, we introduce a distance threshold. If the distance between the current c and the next region n is bigger than this threshold, we stop the raycast process and proceed to the next region in the list. To maintain scale invariance, we test the following value v against the distance threshold.

$$v = \frac{\text{distance}(c, n)}{\min(c.width, n.width)} \tag{4.4}$$

The distance threshold can be obtained with cross-validation by maximizing precision and recall. After this process we obtain many text lines, and some lines are duplicates or contain subparts of a text line. To reduce the number of false or duplicated text lines, we merge text lines that have at least one region in common using a union-find merging algorithm.

After merging we consider the text lines that have at least two regions and produce the bounding boxes of each of such text lines and return them as text detections.

The complete text grouping algorithm is presented in Algorithms 8 and 9.

Our grouping method takes the assumption that the text line is horizontal or nearly-horizontal. But this is not a strict requirement since the method could be adapted text with any orientation. The only required information for oriented text is the orientation of the character region to select the direction of the raycast, but this adds complexity to our simple algorithm.

Algorithm 8 Raycast-based Text Line Grouping

```
Require: List of Character Regions $R$, Distance threshold $d_t$.
 1: Convert $R$ to a label image $I$.
 2: $tl \leftarrow \emptyset$
 3: for $r \in R$ do
 4:     $lineRegions \leftarrow \emptyset$
 5:     $curRegion \leftarrow r$
 6:     $curLabel \leftarrow$ label of region $r$.
 7:     while $curLabel \neq 0$ do
 8:         $rayLabel \leftarrow \textsc{raycast}(curRegion, I)$
 9:         if $rayLabel = 0$ then
10:             break
11:         $candRegion \leftarrow R(rayLabel)$
12:         if distance$(candRegion, curRegion) > d_t \min(candRegion.width, curRegion.width)$ then
13:             break
14:         $curRegion \leftarrow candRegion$
15:         $curLabel \leftarrow rayLabel$
16:         $lineRegions \leftarrow lineRegions \cup curRegion$
17:     $tl \leftarrow tl \cup lineRegions$
18: Merge text lines with common regions in $tl$.
19: return $lt$
```

Algorithm 9 Raycasting Method

```
Require: Starting region $R$ and label image labels.
 1: function RAYCAST($R$, labels)
 2:     $r = (x, y) \leftarrow$ middle point of the right side of $R$'s bounding box.
 3:     if $r$ is not a valid coordinate then
 4:         return 0
 5:     while $r$ is a valid coordinate and labels$(r) = 0$ do
 6:         $r.x \leftarrow r.x + 1$
 7:     if $r$ is not a valid coordinate then
 8:         return 0
 9:     else
10:         return labels$(r)$
```

4.5 Text Verification

Finally the last stage is text verification. This stage is designed to remove the many false positives that are generated in the previous stages. Reasons for false positives could be regions that look like text but are not, such as light poles, road markings, etc. Also our text line grouping algorithm will produce some false positives if the character classification stage did not remove non-character regions.

For text verification we also use the histogram of stroke widths, but instead of classifying a region, we classify the set of regions in the text line.

To classify a text line, we compute the histogram of stroke widths for each region in the text line, and take the average of all histograms, and then renormalize the histogram so its sum is 1.0. We also use two additional features, but their definition is slightly different. A Linear SVM classifier is trained on these features plus the histogram of stroke widths:

Aspect Ratio The aspect ratio r of a list of connected component regions is the ratio of width to height of the bounding box containing all the components in the list:

$$r = \frac{w}{h} \tag{4.5}$$

Occupancy Ratio The occupancy ratio o of a list of connected component regions is the ratio of the sum of number of pixels $\sum_i n_i$ across all regions to the total area of the bounding box containing all components in the list:

$$o = \frac{1}{wh} \sum_i n_i \tag{4.6}$$

Chapter 5

Experimental Evaluation

In this chapter we present the experimental evaluation of our text detector. All tests were run on a laptop with a Intel Core i5-3317U CPU, with a normal frequency of 1.7 GHz and a TurboBoost frequency of 2.4 GHz. The laptop has 10 GB of RAM. The implementation was done in C++, with the GCC 4.9.1 compiler, with `-O2` optimizations enabled

5.1 Datasets

We used 3 kinds of datasets to evaluate our detector. Fraunhofer IAIS provided video frames of a camera mounted on a vehicle that took video footage from the German Autobahn, which was used to produce a small dataset of 95 images.

To train the character classifier and compare its performance with different scripts, we used the Chars74K Dataset [4] English and Kannada images, as well as five small datasets generated from computer fonts.

5.1.1 Chars74K Dataset

We used the Chars74K Dataset because it contains images from font-generated Latin characters, as well as handwritten Kannada characters in a format that is very easy to use. We used two subsets of this dataset:

English

The Chars74K dataset contains 62992 images of Latin characters generated from a very big selection of computer fonts, with image size of 128x128 pixels. There are 1016 image samples for each character (with a total of 62 different characters). A example of some characters in this dataset was previously shown in Figures 2.6 and 2.7.

Kannada

The Chars74K also contains Kannada script characters, from natural images as well as handwritten images. We selected to use the handwritten character images. In this dataset we have 25 image samples for each of the 51 basic Kannada characters, for a total of 1275 image samples. Image sizes are 1200x900 pixels. Some samples of this dataset are shown in Figure 5.1.

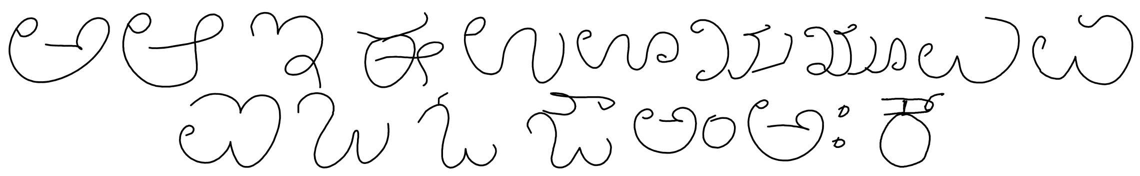


FIGURE 5.1: Some Handwritten Kannada characters from the Chars74K Dataset [4]

5.1.2 Fraunhofer Traffic Panel Dataset

Since there are no public datasets of images from the perspective of an autonomous vehicle with a front facing camera, we decided to create our own. Fraunhofer researchers used a camera on a car and produced many 720p resolution videos of Autobahn driving. We named this dataset the Fraunhofer Traffic Panel Dataset (FTPD).

We took the videos and selected frames that contained traffic panels, and we obtained 95 image samples, split into 40 training and 55 test images. Some examples from the training set can be seen in Figure 5.2. Image sizes are 1280x800 pixels.

Since the frames come from MP4-compressed video, the images itself are very noisy, and the MPEG compression artifacts can be seen if the images are zoomed in. This can be seen in Figure 5.3. Rectangles were used to label text regions in each image. A example of such labels can be seen in Figure 5.4. In general the text of this dataset is horizontal, with some cases where the text is slighly rotated.

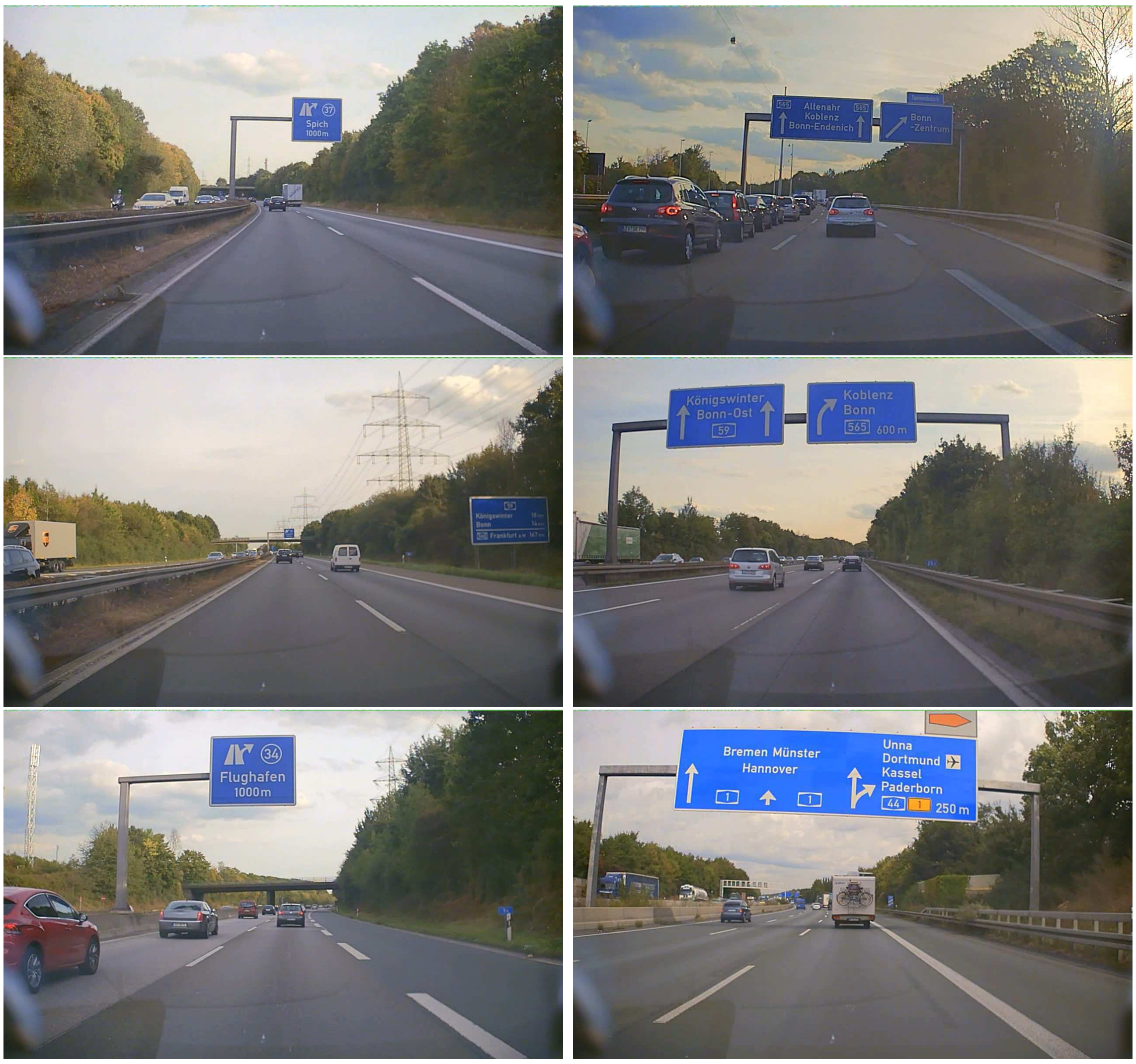


FIGURE 5.2: Some images from the Fraunhofer Traffic Panel Dataset

Parameter	**Value**	**Parameter**	**Value**
Margin Δ	10	Minimum Diversity	0.75
Minimum Area	20 px	Maximum Area	30% of total image area

TABLE 5.1: MSER Parameters for character and non-character region extraction

In order to train the character classifier we extracted character and non-character regions from this dataset. To extract we used a MSER detector configured with the parameters shown in Table 5.1, and used the labeled bounding boxes to decide if a region is a character (positive example) or a non-character (negative example) to train a Linear SVM classifier. A sample of such regions are presented in Figures 5.5 and 5.6. From this dataset we extracted 1280 positive examples and 9072 negative examples.

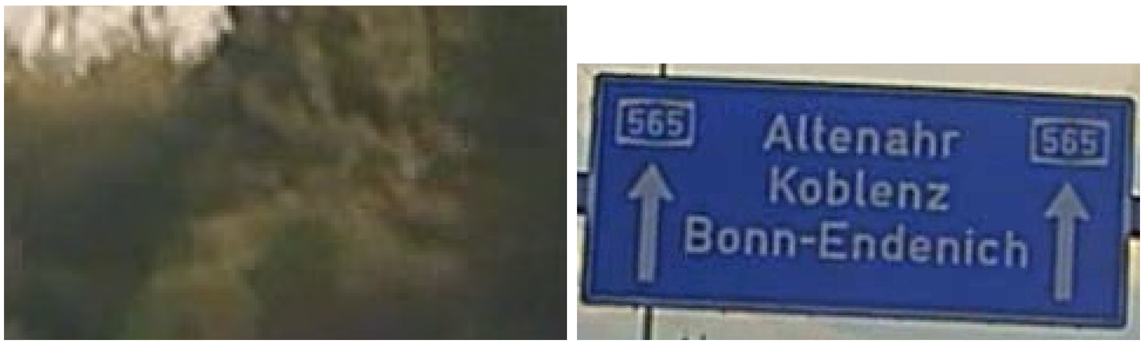


FIGURE 5.3: MPEG Compression artifacts on images from the Fraunhofer Dataset

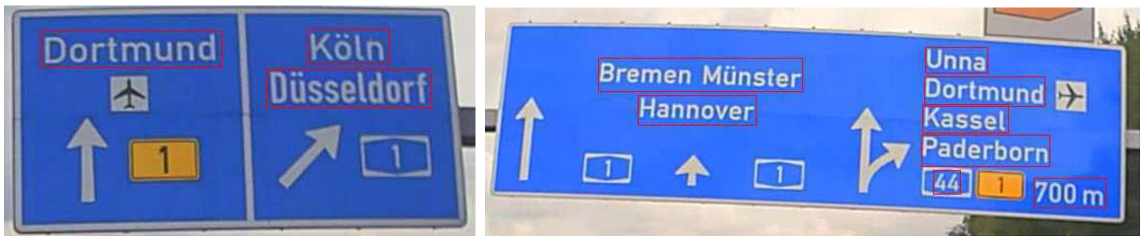


FIGURE 5.4: Labeled text regions from the Fraunhofer Dataset

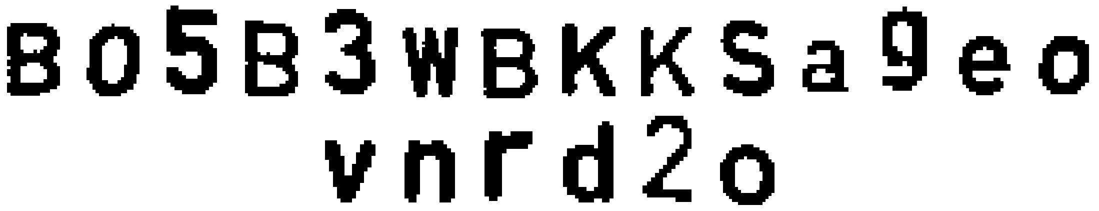


FIGURE 5.5: Extracted MSER character regions from the FTPD

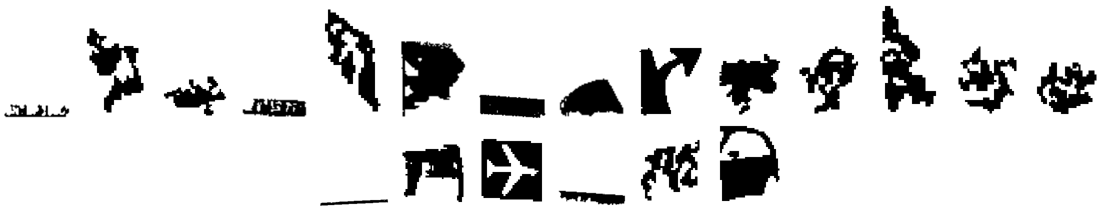


FIGURE 5.6: Extracted MSER non-character regions from the FTPD

5.1.3 Character Datasets

Since most public datasets contain only text with Latin characters, we decided to generate some datasets with non-latin characters from computer fonts. The purpose was to evaluate the generalization power of the character classifier. The scripts we selected were CJK, Hiragana, Katakana and Latin (for control purposes).

To generate Asian script characters, we used the IPA Fonts [1] set provided by the

[1]http://ipafont.ipa.go.jp/#en

Information-Technology Promotion Agency of Japan , which are available under the "IPA Font License Agreement v1.0" License [2].

From the IPA Fonts set we used the IPAGothic, IPAMincho, IPAPGothic and IPAP-Mincho fonts to generate Asian script characters, as following:

CJK

We generated 2257 CJK characters in the Unicode codepoint range U+4e00 to U+56d0, with a total of 81252 image samples.

Hiragana

We generated 93 Hiragana characters in the Unicode codepoint range U+3040 to U+309F, with a total of 3348 image samples

Katakana

We generated 96 Katakana characters in the Unicode codepoint range U+30A0 to U+30FF, with a total of 3456 image samples

Latin

We generated 62 Latin characters (0-9, a-z, A-Z) with a total of 8370 image samples.

To generate each character image, we painted a black character on a 110x110 pixel image, with a white background, and varied the size and properties of the font. Font sizes (in points) were drawn from $s \in \{18, 20, 22, 24, 26, 28, 36, 48, 72\}$. To generate Latin characters, we used the Arial, Comic Sans MS, Fontin, FreeSans and FreeSerif fonts, and for each character and size combination we added 3 versions of the character, with Roman, Bold and Italic styles.

Example characters from the generated datasets are presented in Figures 5.7, 5.8, 5.9 and 5.10.

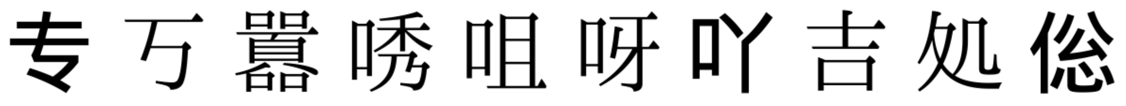


Figure 5.7: Some generated CJK characters

[2]http://opensource.org/licenses/IPA

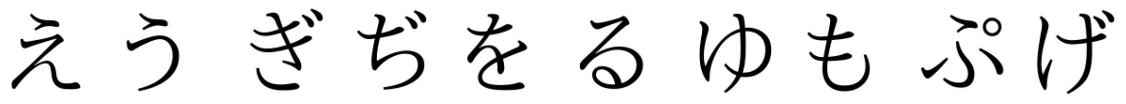


FIGURE 5.8: Some generated Hiragana characters

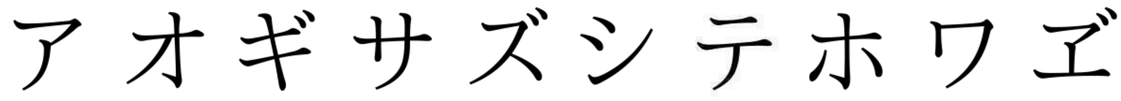


FIGURE 5.9: Some generated Katakana characters

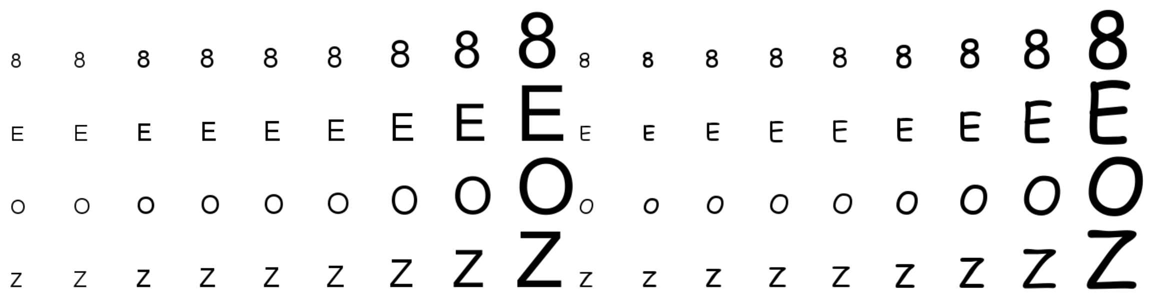


FIGURE 5.10: Some generated Latin characters with different font sizes

5.2 Character Classifier per Script

5.2.1 Experimental Setup

With the purpose of evaluating the generalization power of the character classifier, we designed several synthetic experiments that consisted in training the character classifier with a given set of positive examples, drawn from a script, and a set of negative examples, drawn from the Fraunhofer Traffic Panel Dataset.

To set the values of the SVM misclassification penalty C and the number of histogram buckets N, we used grid search, with a 10-fold cross validation. Each point (C, N) where $C \in \{0.01, 0.1, 1, 10, 100, 1000\}$, $N \in \{10, 20, 30, \cdots, 190, 200\}$ was used to train the classifier and the one with biggest correct classification rate was selected.

Each script dataset was split into training and validation subsets, with the training subset being used for k-fold cross validation and the validation dataset used for final classifier selection.

To compare the generalization performance of the character classifier, we trained it with different scripts. We build a "confusion" matrix, where we trained a character classifier with a given script, and tested it with another script. This way if the character classifier really learned the distribution of stroke widths, it should give a good classification performance over a different script.

We evaluated 6 scripts: English (from the English subset of the Chars74K dataset), Handwritten Kannada (also from the Chars74K dataset), CJK, Hiragana, Katakana and Latin.

5.2.2 Results and Analysis

Results of this experiment can be seen in Table 5.2 for a character classifier using SWT strokes, and Table 5.3 for a character classifier using Run Length strokes.

Train/Validate	**English**	**Kannada**	**CJK**	**Hiragana**	**Katakana**	**Latin**
English	99%	96%	99%	97%	89%	95%
Kannada	97%	99%	99%	98%	93%	86%
CJK	1%	2%	98%	99%	94%	96%
Hiragana	1%	1%	98%	98%	94%	96%
Katakana	1%	1%	98%	99%	98%	97%
Latin	2%	2%	98%	99%	94%	96%

(A) Character

English	**Kannada**	**CJK**	**Hiragana**	**Katakana**	**Latin**
94%	94%	100%	100%	98%	100%

(B) Non-Character

TABLE 5.2: Character Classifier - Correct classification rates with SWT

Train/Validate	**English**	**Kannada**	**CJK**	**Hiragana**	**Katakana**	**Latin**
English	96%	84%	97%	98%	90%	96%
Kannada	96%	100%	98%	98%	90%	90%
CJK	2%	1%	98%	99%	95%	96%
Hiragana	2%	1%	98%	99%	94%	96%
Katakana	2%	1%	98%	99%	95%	96%
Latin	6%	1%	98%	99%	96%	99%

(A) Character

English	**Kannada**	**CJK**	**Hiragana**	**Katakana**	**Latin**
93%	100%	100%	99%	99%	99%

(B) Non-Character

TABLE 5.3: Character Classifier - Correct classification rates with Run Lengths

For SWT, we can see that the generalization performance is excellent. For example, training a classifier with Latin English characters also generalizes very well to detect Kannada, CJK, Hiragana and Katakana characters, with most classification rates over 90%, where the Katakana script has the lowest classification rate of 89%.

Correct classification rates for non-character are also very high, with the English charaters having the lowest correct classification rates.

For run lengths we also see a very good generalization performance, with all character classification rates over 90%, except for English vs Kannada with a 84% classification rate. This could be explained because the strokes in the Handwritten Kannada dataset are very thin, and the run length classifier could have problem capturing their distribution.

Comparing SWT versus RL we see that there is no clear winner, classification rates for character are sometimes higher for SWT and sometimes higher for RL. But for non-character RL is slightly better since it has a 100% correct classification rate for non-character regions, and for other datasets they are very similar, with minimum differences.

There are several cases where classification fails and that happens when we train with one of the generated datasets (CJK, Hiragana, Katakana and Latin) and test with the non-generated datasets (English and Kannada). This should not be surprising due to the big difference in size between such datasets, and there is no way we could capture the whole variation of font size and style of the non-generated datasets in a relatively small dataset such as the ones we generated.

We can conclude that the Histogram of Stroke Widths can learn the stroke width distributions of several scripts and generalize very well with other scripts, which make it a very powerful feature for character and text classification.

5.3 Text Detector on Road Scenes

5.3.1 Character Classifier Training

To evaluate the whole text detection pipeline, first we trained a character classifier on the character and non-character regions extracted from the FTPD. The performance of the trained classifier represented as a confusion matrix is shown in Table 5.4. Please note that we abbreviated "character" as "char" in that table.

	Char	**Non-Char**
Char	88%	12 %
Non-Char	12 %	88%

(A) SWT

	Char	**Non-Char**
Char	92%	8 %
Non-Char	18 %	82%

(B) Run Lengths

TABLE 5.4: Character Classifier performance while trained on the Fraunhofer Dataset

For both SWT and RL the character classification performance is good, with RL being slightly better (4% difference), but with a lower correct classification rate for non-character regions. Results from the grid search process for C and N are available on Table A.1 in the Appendix.

5.3.2 Text Verifier Training

The second stage that requires training is the text verifier. To train it we took samples from the FTPD, but instead of taking single characters, we took complete text regions inside the labeled bounding boxes. To obtain negative examples, we ran the raycasting algorithm over the detected MSER regions and selected any false positive that was completely outside the text bouding boxes.

Both kinds of regions can be seen in Figures 5.11 and 5.12. From this dataset we extracted 189 positive examples and 403 negative examples.

Results from the text verifier training are shown on Table 5.5 and the complete results from the grid search process for C and N are available on Table A.3 in the appendix. We can see that classification performance is very high, with 97% of text regions correctly classified, but there is no significant difference between using SWT

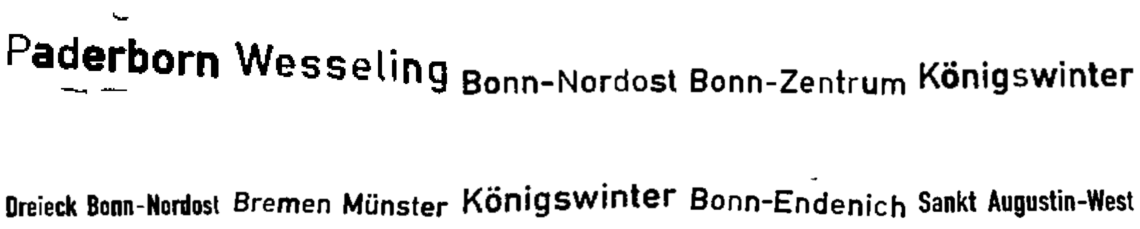


FIGURE 5.11: Extracted MSER text regions from the FTPD

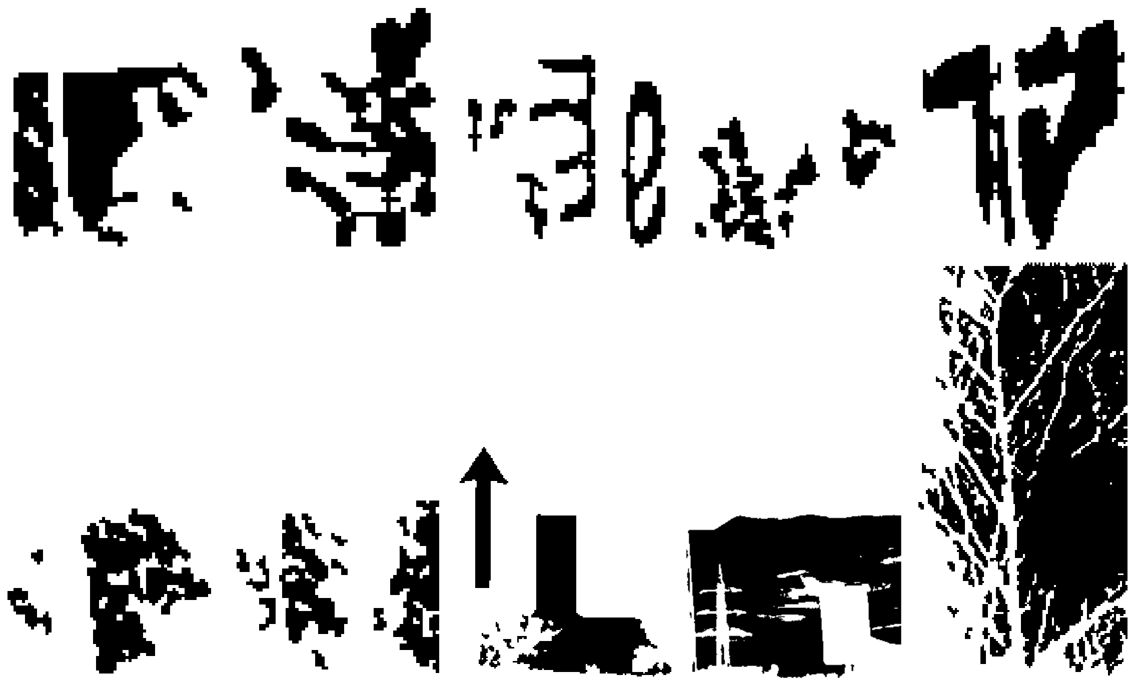


FIGURE 5.12: Extracted MSER non-text regions from the FTPD

or RL. Run lengths seem to be slightly worse since they confuse 3 times more non-text regions as text, and have a 2% lower classification rate for non-text regions.

	Text	Non-Text
Text	97%	3 %
Non-Text	1 %	99%

(A) SWT

	Text	Non-Text
Text	97%	3 %
Non-Text	3 %	97%

(B) Run Lengths

TABLE 5.5: Text Verifier performance while trained on the Fraunhofer Dataset

5.3.3 Experimental Setup

Now we can proceed to evaluate the whole text detection pipeline over the FTPD. As mentioned before, we trained the character classifier and text verifier on the training portion of the FTPD, and ran the whole pipeline on the test portion of the dataset.

Parameter	**Value**
Margin Δ	10
Minimum Diversity	0.75
Minimum Area	0.001% of total image area
Maximum Area	30% of total image area

Table 5.6: MSER Parameters for our Text Detector evaluation

The parameters used for testing are shown in Table 5.6. The value of the distance threshold was set to $d_t = 1.5$.

We consider a correct text detection of the detected rectangle matches the labeled rectangle in the ground truth in at least 70%, and then we compute precision and recall, as well as obtain computation times for each stage. The text verifier (TV) was optionally enabled or disabled to allow for comparisons of its effect in the detection performance.

Enabling or disabling the text verifier gives four possible configurations:

SWT-TV Off
: Stroke Width Transform used for stroke width extraction, with the text verifier disabled.

SWT-TV On
: Stroke Width Transform used for stroke width extraction, with the text verifier enabled.

RL-TV Off
: Run Lengths used for stroke width extraction, with the text verifier disabled.

RL-TV On
: Run Lengths used for stroke width extraction, with the text verifier enabled.

5.3.4 Results and Analysis

Text Detection performance is shown in Table 5.7. We can see that recall of the RL text detector is better than the SWT text detector, with a best recall of 76%. But for precision, the best performance is obtained with the SWT text detector.

Enabling the text verifier has the effect of increasing precision by a considerable amount, up to 24%, but it also has the effect of decreasing recall slightly. The observed decrease in recall was 7% for SWT and 4% for RL.

The best performing text detector is the Run Lengths one with an enabled text verifier, with f-score of 69%. The same configuration with SWT stroke widths has a slightly lower f-score of 61%.

	SWT-TV Off	**SWT-TV On**	**RL-TV Off**	**RL-TV On**
Precision	40%	**62** %	41%	65%
Recall	68%	61%	**76**%	72%
F-Score	50%	61%	53%	**69**%

Table 5.7: Text Detector performance on the Fraunhofer Dataset

We also obtained computational performance data, which is available in Table 5.8. In this table we can see that in general all detector configurations take roughly 1 second to process a frame, and clearly the most expensive part is the extraction of Maximally Stable Extremal Regions. The Run Lengths detector is clearly faster, since the character classification stage is much faster (35 times faster to be precise) than the SWT stroke widths.

For the SWT text detector, Roughly 2% of the time is spent on image projection (Converting to grayscale), 73% on MSER extraction, 24% on character classification, 0.4% on text line grouping, and 0.6% verifying text. For the RL text detector, 95% of the time is spent on the MSER extraction and only 5% on the rest of the pipeline.

Enabling the text verifier has a very good impact on detection performance while having a very small computational cost. Since Run Lengths have the best detection performance and the lowest computational cost, it is clearly the best choice for a text detector based on MSER.

A comparison of our different detector configurations with the SWT algorithm is presented in Table 5.9 over the FTPD. We can see that the SWT has terrible performance, with a f-score of only 28%, and a very high computation time.

	SWT-TV Off	**SWT-TV On**	**RL-TV Off**	**RL-TV On**
Image Projection	23 ± 4 ms	23 ± 4 ms	22 ± 7 ms	22 ± 7 ms
MSER	844 ± 131 ms	844 ± 131 ms	859 ± 143 ms	859 ± 143 ms
Character Classifier	281 ± 136 ms	281 ± 136 ms	8 ± 3 ms	8 ± 3 ms
Text Line Grouping	6 ± 2 ms	6 ± 2 ms	7 ± 4 ms	7 ± 4 ms
Text Verifier	0 ± 0 ms	8 ± 6 ms	0 ± 0 ms	1 ± 1 ms
Total	1154 ± 188 ms	1162 ± 189 ms	896 ± 143 ms	896 ± 143 ms

Table 5.8: Text Detector computation time while on the Fraunhofer Dataset

We believe this is due to the very noisy nature of the images in our dataset, since they are real-world road scenes, with blur, and characters are very small in size when compared with the size of the image frame. Also the artifacts from MPEG compression play a big role here, affecting the gradient computations, which are known to be very sensitive to noise [60].

	Precision	**Recall**	**F-Score**	**Time**
MSER-SWT	40%	68%	50%	1.2 ± 0.2 s
MSER-SWT-TV	**62**%	61%	61%	1.2 ± 0.2 s
MSER-RL	41%	**76**%	53%	0.9 ± 0.1 s
MSER-RL-TV	65%	72%	**69**%	0.9 ± 0.1 s
SWT [8]	21%	40%	28%	2.5 ± 0.2 s

Table 5.9: Comparison between SWT and our Text Detector on the Fraunhofer Dataset

5.4 Distance Threshold Sensitivity

In order to evaluate the sensitivity of our text detectors with respect to the distance threshold parameter d_t, we obtained an ROC curve, which can be seen in Figure 5.13.

The ROC curve again shows that the superior detector configuration is Run Length strokes with an enabled text verifier, by a wide margin. The optimal value of the distance threshold d_t was selected by performing grid search over the range $[0.0, 2.0]$ and finding the value of d_t that maximized the F-Score.

The difference between SWT curves is smaller than the difference between RL curves. This might indicate that tuning a precise value of d_t is more critical for the Run Lengths detector than for the Stroke Width one.

It should be noted that the ROC cuve in Figure 5.13 represents a "Convex hull" of the Precision-Recall data obtained from testing the detector algorithm, according to recommendations from [61]. Data points that fall inside the curve area were skipped.

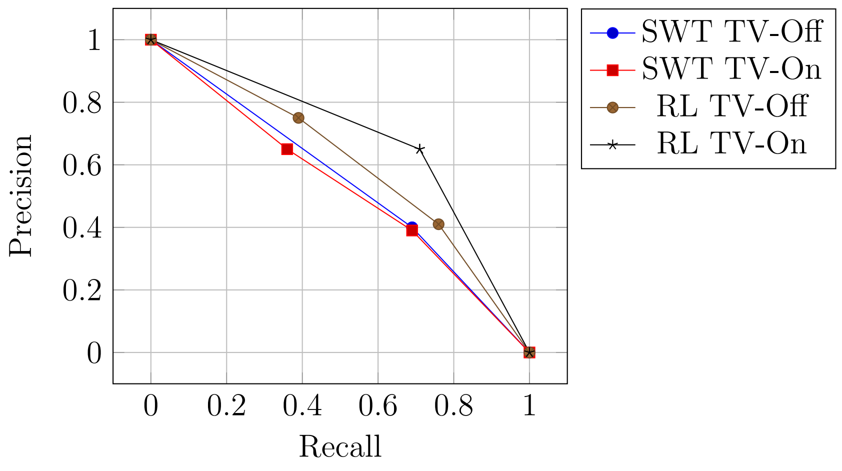


FIGURE 5.13: ROC curve with respect to the distance threshold d_t

5.5 Character Classifier per Character

In order to perform a better comparison of SWT and run length stroke width extraction, we performed additional experiments aimed at seeing what is the difference between both methods.

5.5.1 Experimental Setup

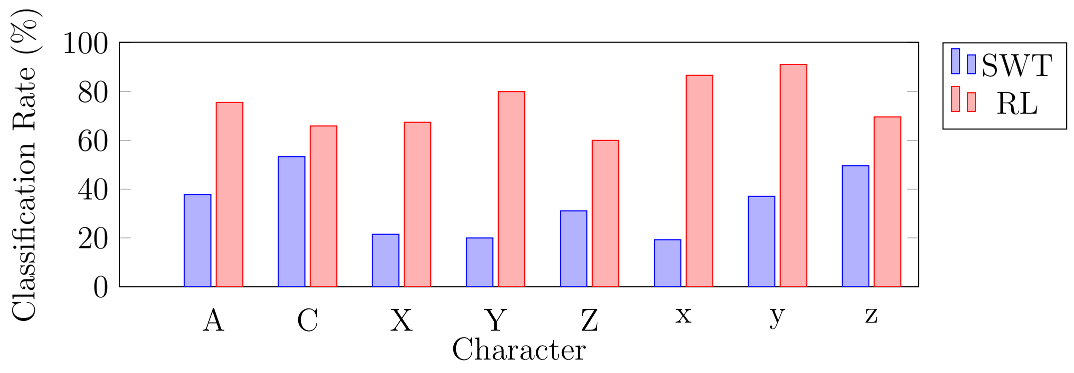
We suspected that the correct classification rate is not the same across different characters. To evaluate this hypothesis, we tested the character classified trained on the FTPD with the Latin script dataset, but evaluated separately for each character. We tested numbers from 0 to 9, lowercase characters (a to z) and uppercase characters (A to Z).

5.5.2 Results and Analysis

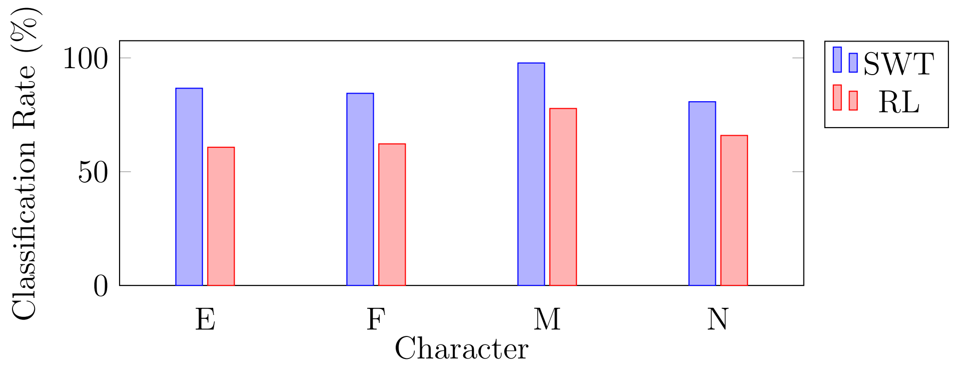
Results of this experiment are shown on Figures 5.15 and 5.16. For SWT the average correct classification rate is 77%, and for Run Lengths it is 90%.

Comparing both plots, we can see that the SWT has issues classifying the characters “X”, “Y” and “Z”, where they have the lowest correct classification rates among all characters. We believe this is due to discretization effects of the gradient, and this affects the gradient orientation. The same effect can be seen in Figure 2.11, where there are smaller stroke widths around the section of the letter where 2 strokes meet. The SWT seems to have issues in these cases. An individual comparison of selected characters is shown in Figure 5.14.

Run Lengths seem to be a better choice, since their classification rate is bigger by a wide margin, and does not have any particular problematic characters, but still some characters have lower detection rates than the rest, such as the “N”, “T” and “Z”/“z” characters.



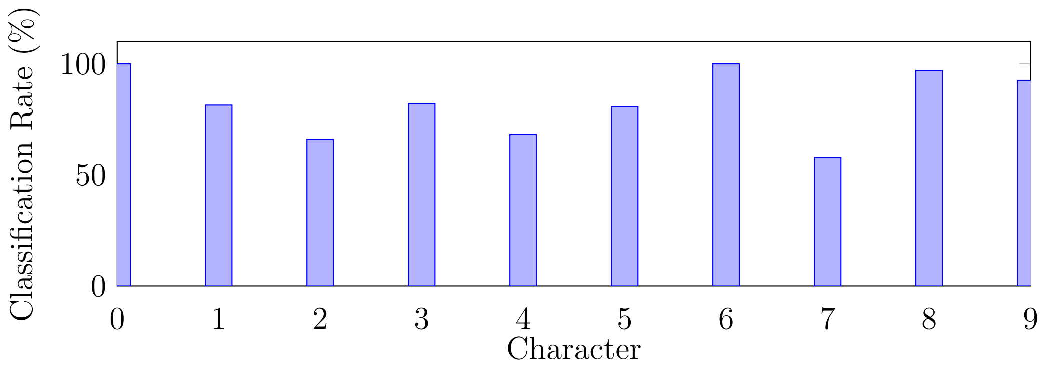
(A) Characters where Run Lengths are superior



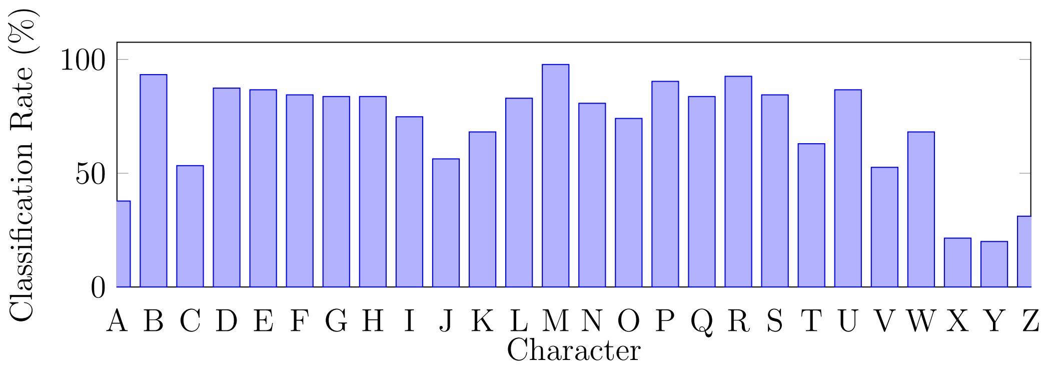
(B) Characters where Stroke Width Transform is superior

FIGURE 5.14: Comparison of SWT and RL performance over selected characters

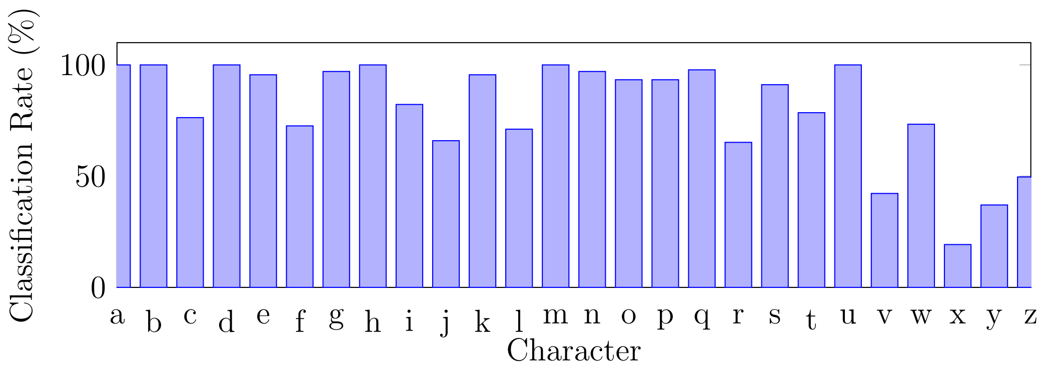
We note that problematic characters for the Run Lengths all have long horizontal strokes, for which the stroke width will be a overestimation, which causes misclassifications. Comparison between SWT and RL still needs more research, since results can change depending on the dataset or fonts used to draw text.



(A) Numbers

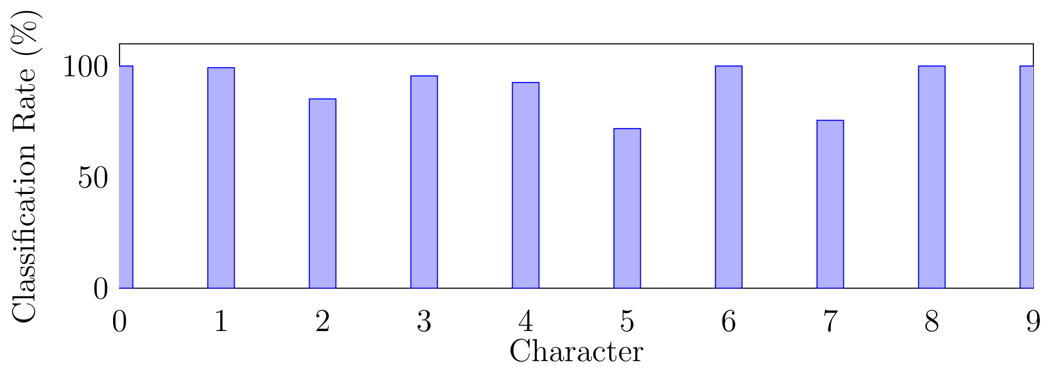


(B) Uppercase Letters

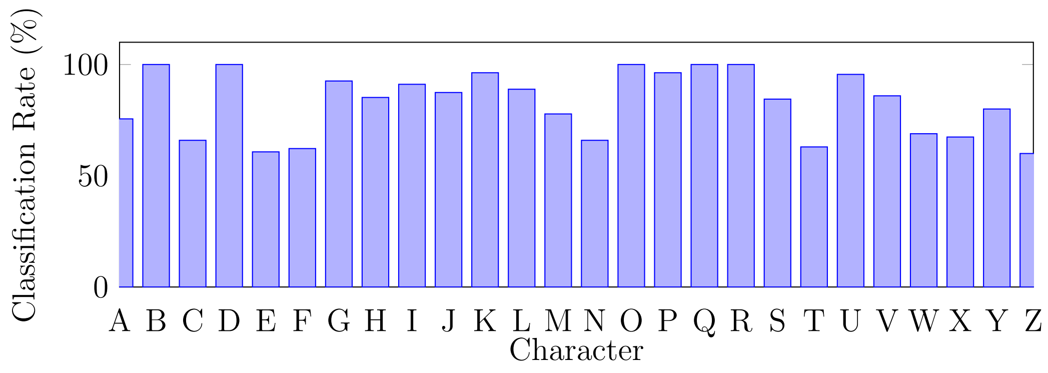


(C) Lowercase Letters

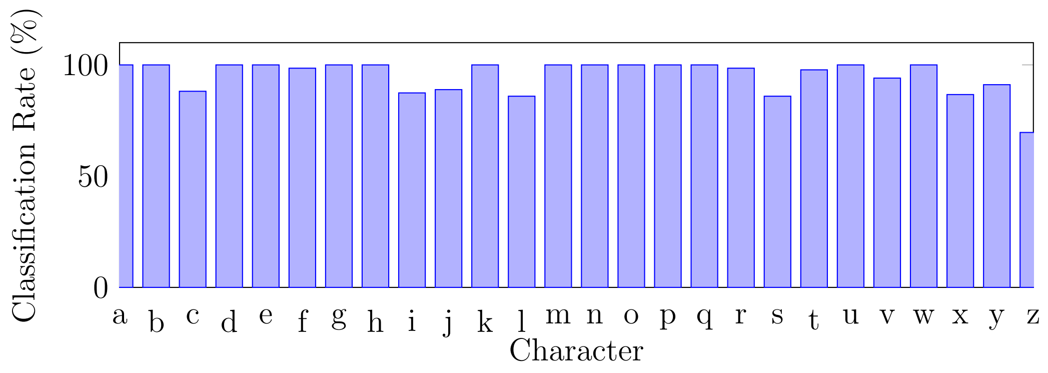
FIGURE 5.15: SWT Character Classifier Performance per Character (77% average classification rate)



(A) Numbers



(B) Uppercase Letters



(C) Lowercase Letters

Figure 5.16: RL Character Classifier Performance per Character (90% average classification rate)

5.6 Rotation Invariance

In theory the SWT stroke widths are rotation invariant (up to the effects of discretization), while the Run Length stroke widths are not, since a rotated character would produce different run lengths, which will change the distribution of stroke widths. We designed this experiment to evaluate the performance of the trained character classifiers when the characters are rotated.

5.6.1 Experimental Setup

We took the images from the already generated character datasets and rotated them (around the Z axis) to produce rotated versions, and evaluated the performance of the different character classifiers versus the new examples. Each character image sample was rotated by an angle of $\theta \in \{0, 5, 10, 15, \cdots, 360\}$ degrees.

We tested each classifier with the corresponding rotated characters of the same script. We did not perform cross-script evaluation of rotated characters. Only the Latin, CJK, Hiragana and Katakana scripts were evaluated.

5.6.2 Results and Analysis

Plots of the relation between classifier performance and rotation angle θ are shown in Figure 5.17 for SWT strokes, and in Figure 5.18 for Run Length strokes.

From Figure 5.17 we can see that performance is not affected by rotated characters, except for the Latin script, where at $\theta = 90$ and $\theta = 270$ the classifier fails, with a zero classification rate. We believe this happens because the classifier was not trained on rotated characters, and while in theory the SWT is rotation invariant, the stroke width distributions do change slightly, which can make the classifier fail. This can be seen in Figure 5.19.

One issue that must be pointed out is normalization. Our histogram is normalized by the width of the region, and while characters are rotated, the stroke widths from SWT do not change much, but the width of the region changes. To be truly rotation

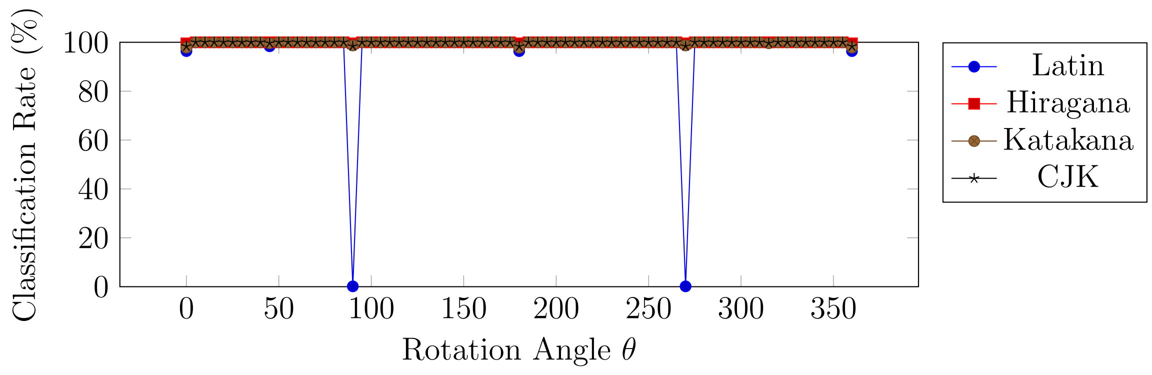


FIGURE 5.17: SWT Character Classifier Performance by Rotation Angle θ

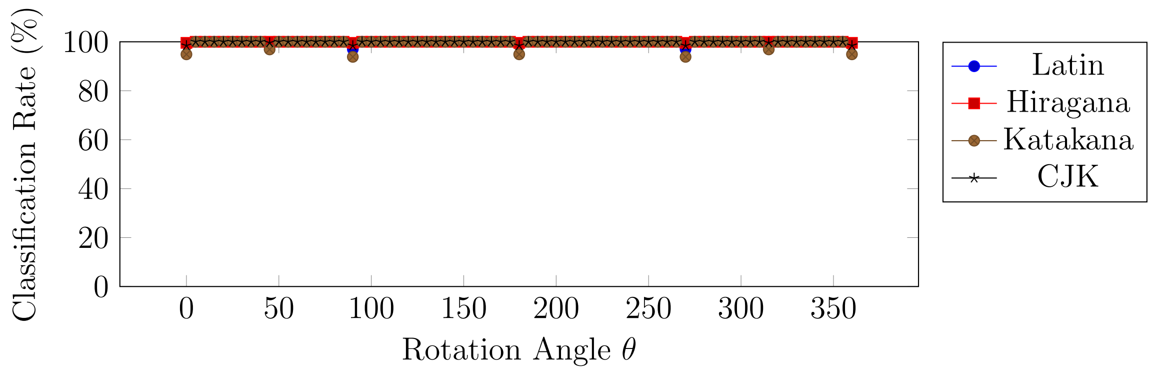
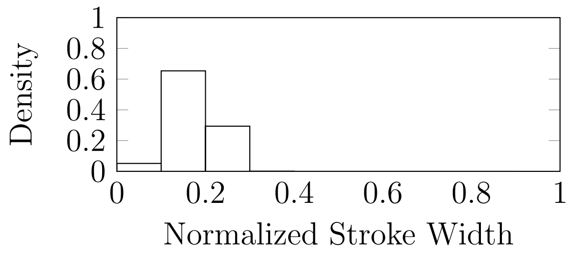


FIGURE 5.18: RL Character Classifier Performance by Rotation Angle θ

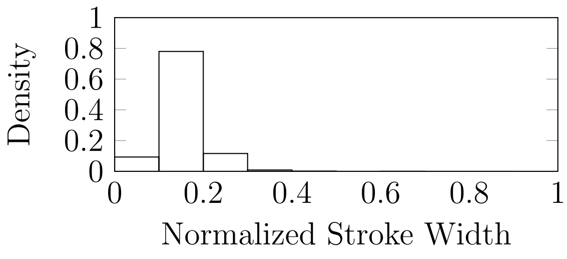
invariant, the histogram should be normalized by the minimum value between width and height [17]. We believe this is the reason why SWT stroke widths fail at $\theta = 90$ and $\theta = 270$.

For Run Lengths in Figure 5.18, detection performance is the same as we rotate the characters, with very small drops for discrete angle values such as $\theta = 45, 90, 270$, and 215. Even that in theory the Run Lengths are not rotation invariant, character classifier performance does not change with rotated characters. We can even see in Figure 5.20 that the stroke width distributions are different as the rotation angle changes, but the classifier can still correctly classify such samples (even as it was not specifically trained for that).

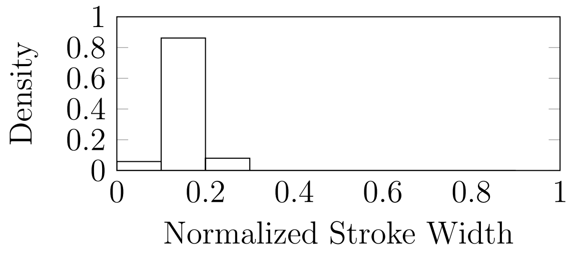
One explanation for the drops at $\theta = 90$ and $\theta = 270$ is that normalization fails for these cases, since at those angles, region width and height are swapped, so the stroke widths are not being normalized by the region width, but by height. This considerably changes the stroke width distribution, causing classification to fail.



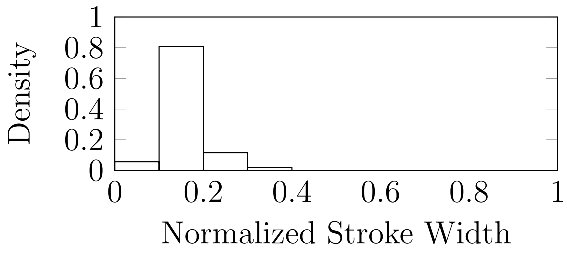
(A) $\theta = 0$



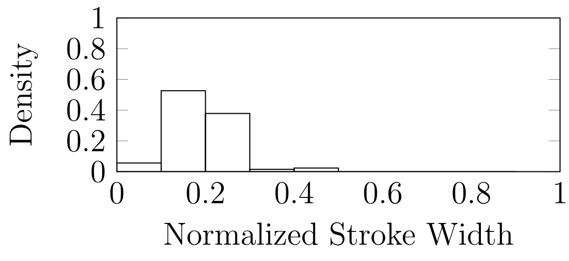
(B) $\theta = 45$



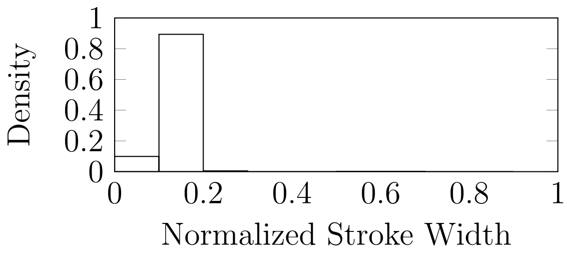
(C) $\theta = 90$



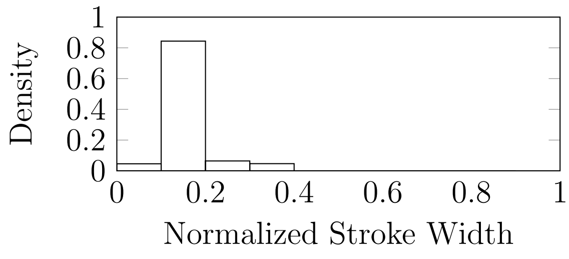
(D) $\theta = 135$



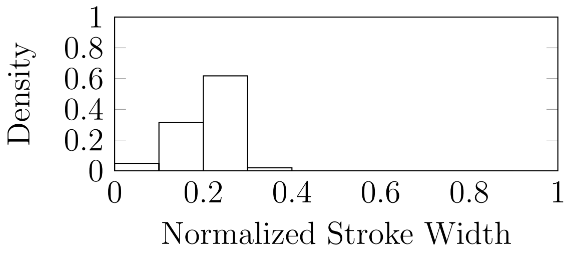
(E) $\theta = 180$



(F) $\theta = 225$

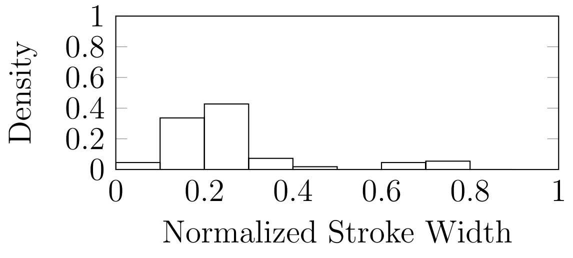


(G) $\theta = 270$

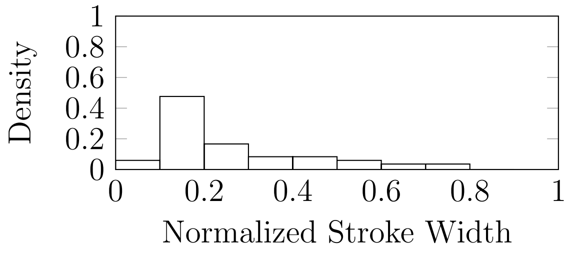


(H) $\theta = 315$

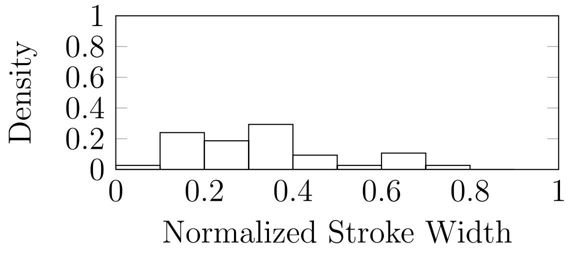
FIGURE 5.19: SWT Histogram of Stroke Widths for rotated versions of the character "A"



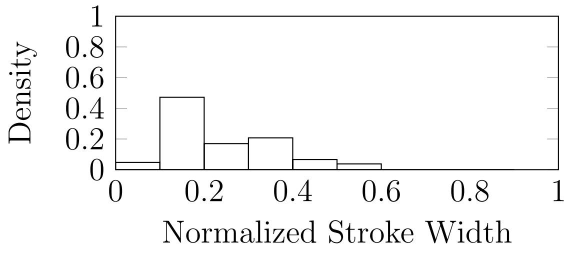
(A) $\theta = 0$



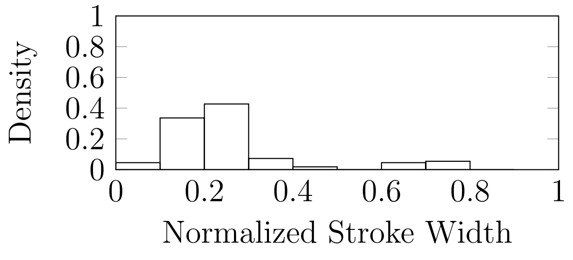
(B) $\theta = 45$



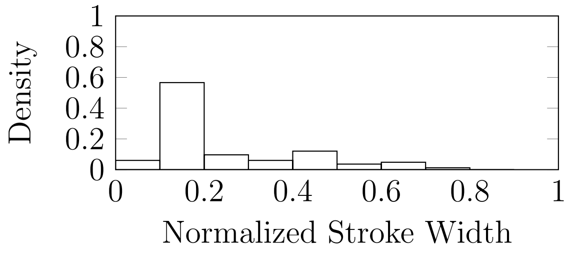
(C) $\theta = 90$



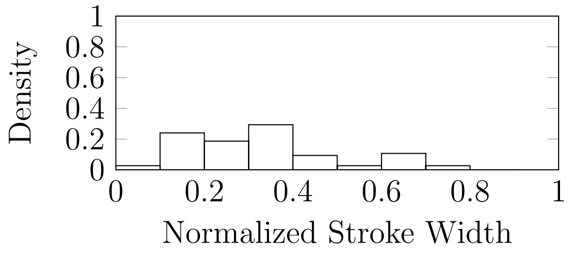
(D) $\theta = 135$



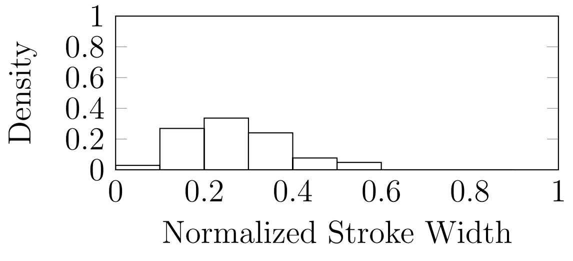
(E) $\theta = 180$



(F) $\theta = 225$



(G) $\theta = 270$



(H) $\theta = 315$

FIGURE 5.20: RL Histogram of Stroke Widths for rotated versions of the character 'A"

5.7 Discussion

In this chapter we have experimentally evaluated our text detection approach. Clearly Run Lengths are the best choice for stroke width extraction for our character classifier and text verifier. Enabling the text verifier has a positive effect on detection performance and on computational performance as well.

Performance under the FTPD is very good, with 72% of the text regions correctly detected, but only 65% of the detections being correct.

By looking at the correct text detections in Figure 5.23 and the incorrect or incomplete text detections (which count as incorrect detections) in Figure 5.22, we can extract the following failure cases:

1. Text detection fails for some cases where one or multiple characters in a word are classified as non-characters. This will produce two text detections in different parts of a word, and they will be counted as incorrect detections.
2. Non-character regions can also be classified as character regions (false positives), and sometimes these regions can also be "attached" to nearby text regions, creating bigger bounding boxes, which will make the detection count as a incorrect one.
3. Very small or noisy regions cannot be detected either as MSER regions or as character regions. This is very noticeable in Figure 5.22d, where "Rommerskirchen" had only three correctly classified or detected characters, and the rest are completely missing. The same applies for "Kreuz" and the number "57" in the same image.

In some cases MSER regions are adjacent or even inside to other regions, and this causes some small issues. This can be seen in Figure 5.21. Note how in that figure, the "n" character has some regions inside it (denoted by a different color), and the characters "r", "t" and "m" exhibit the same behavior.

This is due to noise in the image and could be prevented by merging regions that are adjacent (connected), but doing this would increase the computation time of the

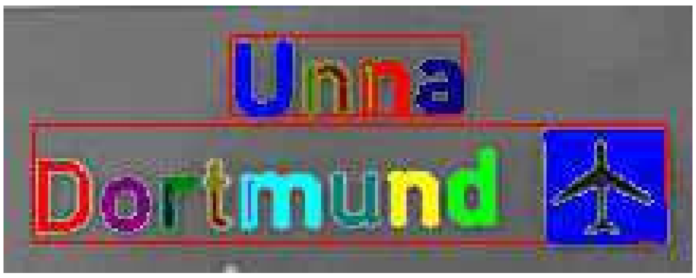


FIGURE 5.21: Regions inside regions produced by the MSER detector

algorithm. Filtering on the input image is also an option.

Our text line grouping algorithm seems to work well, with a very small number of cases where the grouping is incorrect or “oversegments” the image. Such cases would not occur if the character classifier could correctly classify 100% of the character regions and completely remove the non-character regions.

Computational performance of our detector is good, some people [7] would say it is real-time, but more work on a optimized implementation could produce noticeable improvements. To extract MSER regions, we used VLFeat’s MSER implementation [3], but since most MSER implementations are used for tracking, they fit ellipses by means of computation of the moments I_x, I_y and I_{xy} of the region, which can be incrementally computed and only require 3 floating point numbers to store per region in the component tree.

But we had to modify VLFeat’s MSER implementation to produce regions represented as list of points (x, y), which increases computation time and memory requirements, since each region in the component tree contains a list of points. Another approach could produce much better computational performance results.

Finally, the RL-based character classifier has also very good classification performance if the characters are rotated. We believe this happens because the RL-based classifier is much more “tolerant” to different stroke width distributions than the SWT classifier.

[3]http://www.vlfeat.org/overview/mser.html

Initially we thought that the SWT-based classifier would outperform the RL-based classifier. Our experiments show that we were wrong. The reason why the SWT-classifier fails are the same as the reasons for why the SWT itself fails. Gradient information is unreliable because of noise, but also the discretization effects count. Raycasting can fail just because there is no edge in the ray direction, but there would be a edge if the space is continuous instead of discrete.

The number of parameters of our text detector is low, only 4 parameters for the MSER region detector, and only 2 for the text detector itself (number of histogram bins and maximum distance threshold for raycasting). This is low compared with other text detectors, such as Epshtein et al. [8] SWT text detector, with 13 parameters.

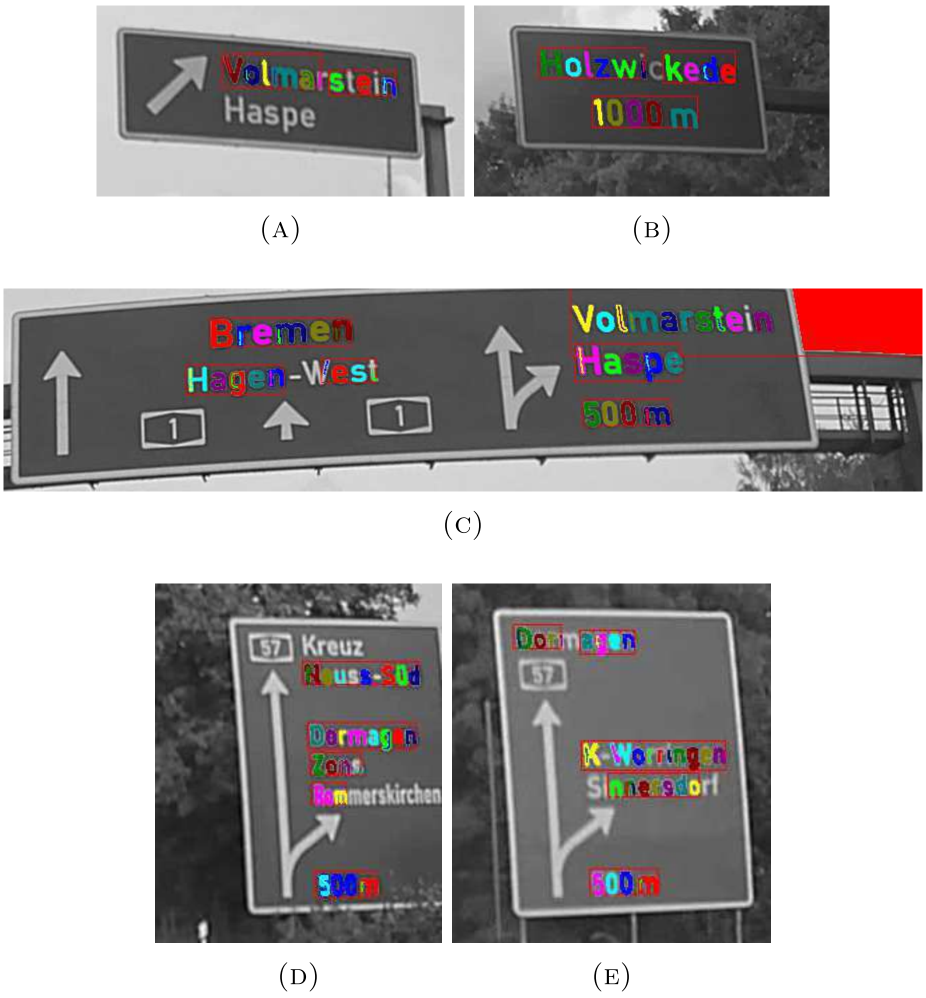
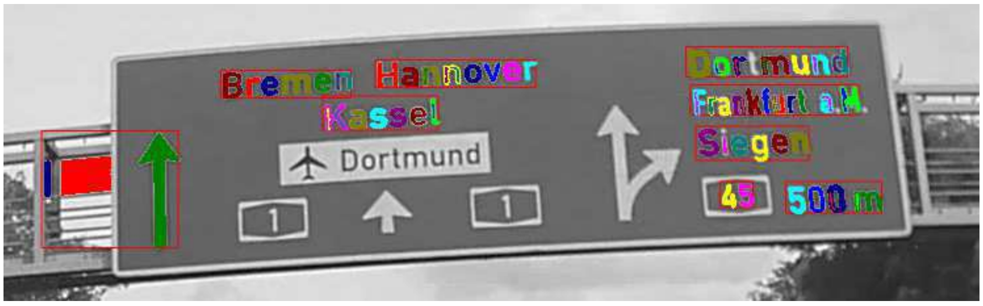
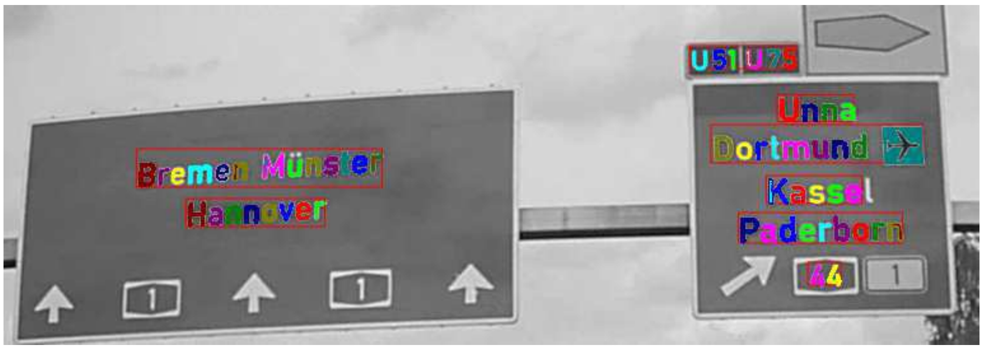


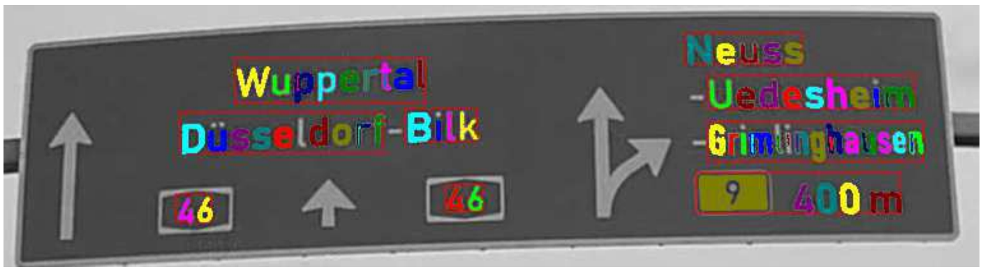
FIGURE 5.22: Examples of incorrect or incomplete text detections



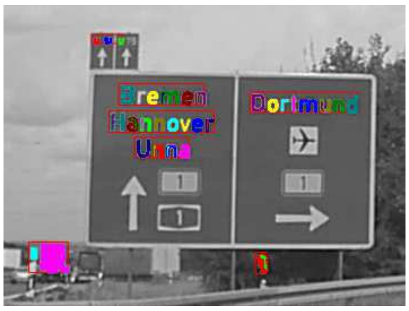
(A)



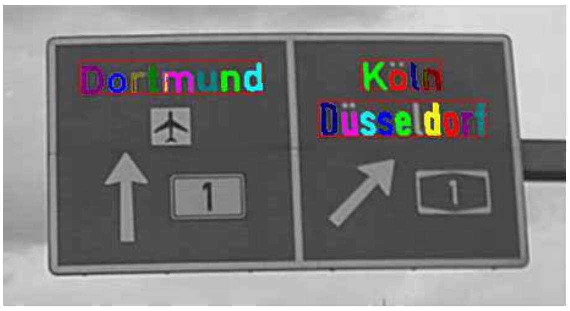
(B)



(C)



(D)



(E)

FIGURE 5.23: Examples of correct text detections

Chapter 6

Conclusions

In this Master thesis we have presented a new feature for character region classification, based on a histogram of the stroke widths. We proposed two ways to extract stroke width information from a region detected by a MSER region detector.

The first way to extract stroke widths is based on the Stroke Width Transform [8], and the second one is based on Run Lengths [28]. Then we compute the histogram of stroke widths, normalized by the width of the region, and by adding two simple and commonly used features (aspect ratio and occupancy ratio), we use a Linear SVM classifier to classify text regions.

This approach can also be used to verify text regions, where the output of a text detector is given to our text verifier, and it will remove text detections that are false positives, improving precision of the detector. Our approach is script-independent and our experiments show that it can generalize very well from training in one script and testing in another different script.

From our approach we built a text detection pipeline with four stages, and proposed a simple way to group character regions into text lines by means of raycasting from one region to the other. We evaluated our character classifier on five different scripts: Latin, Kannada, CJK, Hiragana and Katakana on a dataset of generated character images from computer fonts.

Our character classifier obtained at least 96 % correct classification rates across different scripts, with the lowest classification rate being 89 %. We trained our character classifier with positive examples on one script, and tested with another script, and we obtained very high classification rates in the same range.

To evaluate our text detector on road scenes, we constructed a dataset of images taken from video in the German Autobahn, which contains mostly traffic panels. We denominated this the Fraunhofer Traffic Panel Dataset. The best performing detector configuration is using run lengths for stroke width extraction and enabling the text verifier, which obtains 65 % precision, 72 % recall with an f-score of 69 %.

In comparison, the Stroke Width Transform obtains only 28 % f-score on this dataset, and this is due to the high image noise and very small character sizes. With respect to computational performance, our C++ detector implementation takes 0.9 ± 0.2 seconds to process a frame, where 95 % of the time is spent on MSER region extraction.

Further testing indicated that our SWT and RL character classifiers suffer from some small drawbacks. Some characters such as "E", "F", "N", "W", "X", "Z" and"z" are constantly misclassified as non-characters (with a $30 - 40$ % chance), but in general the character classifier works very well, with > 90 % correct classification rates and low computational complexity.

Our experiments also show that while run lengths stroke width extraction is not rotation invariant, the classifier can still cope and correctly classify rotated characters with similar precision as non-rotated characters.

Finally, in this Master thesis we believe a contribution has been made to the state of the art with a script-independent method for character and non-character region classification. Other contributions are the FTPD dataset, as well as the raycasting method for text line grouping.

6.1 Future Work

Still there is much work to be done in text detection, and in particular for this work as well.

MSER regions were extracted from a grayscale image, which has known problems with image regions that have low contrast or are noisy. This could be improved by using Maximally Stable Color Regions [38].

Our method also takes the assumption that characters will be detected as MSER regions, and this could be violated by many kinds of real-world text. Other authors [40] extract MSER regions in other projections of the image, such as each RGB or HSV color channel, as well as on the gradient of the image. This could improve the detection performance by actually detecting more characters.

Regions were represented as lists of points, and this created space and time complexity problems. We have several ideas about how to deal with this problem, such as extracting only the thresholds and bounding boxes for each region, and then do a postprocessing step that extract the regions using both information. This could reduce the time it takes to extract MSER regions.

Other methods to extract stroke width information should be explored. The histogram of stroke widths does not depend on any particular method of stroke width extraction. The distance transform has also been used to compute stroke widths [39], and could pose as a good option.

Our character classifier was evaluated on generated datasets of non-latin characters, but some datasets also contain real-world images of characters in other scripts, such as Korean and Kannada. We must evaluate our character classifier in such datasets to be able to know the real generalization performance with respect to images in the real-world, which are known to be noisy, blurry, etc.

We also would like to integrate our character classifier into other text detector methods, such as Neumann and Matas [6]. Our character classifier could help improve detection performance for other methods and does not require or rely on our own detector implementation.

One thing that must be done to obtain comparable results is to evaluate our text detector on the ICDAR 2003 dataset. The complexity and size of this dataset makes it a bit difficult to train. Extracting more MSER regions is also needed to obtain good performance, and this could not be done now because of time constraints.

The results about rotation invariance look promising but still more tests should be performed. We normalized by the region width, and other normalization methods should be evaluated, such as the minimum value between width and height of the region. This could avoid edge cases where the classifier fails and could make it truly rotation invariant.

Finally, text detector and character classifier must be evaluated with oriented text. Our text line grouping algorithm is not rotation invariant and should fail with oriented text, but a small modification that includes orientation information for each character could be used to cast a ray in an appropriate direction, which should work for oriented and curved text as well.

Appendix A

Classifier Grid Search Training

N / C	0.01	0.1	1	10	100	1000
10	68.3	83.3	88.3	88.3	91.7	91.7
20	70.0	83.3	86.7	91.7	91.7	91.7
30	70.0	83.3	88.3	91.7	93.3	93.3
40	70.0	81.7	90.0	91.7	91.7	91.7
50	70.0	81.7	86.7	90.0	**95.0**	**95.0**
60	70.0	83.3	93.3	91.7	93.3	**95.0**
70	70.0	81.7	**95.0**	91.7	93.3	91.7
80	70.0	81.7	90.0	91.7	93.3	91.7
90	70.0	83.3	90.0	91.7	90.0	91.7
100	70.0	83.3	91.7	90.0	**95.0**	**95.0**
110	70.0	81.7	93.3	90.0	91.7	90.0
120	70.0	83.3	93.3	91.7	90.0	90.0
130	70.0	83.3	93.3	90.0	93.3	91.7
140	70.0	81.7	93.3	90.0	90.0	90.0
150	70.0	81.7	**95.0**	90.0	91.7	91.7
160	70.0	81.7	93.3	90.0	91.7	90.0
170	70.0	83.3	93.3	88.3	91.7	90.0
180	70.0	83.3	91.7	91.7	91.7	93.3
190	70.0	83.3	91.7	88.3	93.3	**95.0**
200	70.0	83.3	91.7	88.3	91.7	88.3

Table A.1: SWT Text Classifier Grid Search for N and C while trained on the Fraunhofer Dataset

N / C	0.01	0.1	1	10	100	1000
10	71.7	86.7	91.7	**95.0**	**95.0**	**95.0**
20	68.3	83.3	93.3	**95.0**	93.3	93.3
30	70.0	80.0	**95.0**	93.3	91.7	91.7
40	70.0	80.0	**95.0**	93.3	91.7	91.7
50	70.0	80.0	93.3	93.3	91.7	91.7
60	70.0	78.3	91.7	91.7	90.0	**95.0**
70	70.0	78.3	91.7	91.7	91.7	90.0
80	70.0	78.3	91.7	93.3	90.0	90.0
90	70.0	78.3	91.7	90.0	90.0	91.7
100	70.0	78.3	91.7	93.3	90.0	88.3
110	70.0	78.3	91.7	91.7	93.3	91.7
120	70.0	78.3	91.7	91.7	**95.0**	91.7
130	70.0	78.3	91.7	91.7	93.3	93.3
140	70.0	78.3	91.7	91.7	**95.0**	91.7
150	70.0	78.3	91.7	93.3	**95.0**	91.7
160	70.0	78.3	91.7	93.3	90.0	90.0
170	70.0	78.3	90.0	91.7	91.7	93.3
180	70.0	78.3	91.7	91.7	93.3	93.3
190	70.0	78.3	90.0	90.0	93.3	91.7
200	70.0	78.3	91.7	90.0	93.3	91.7

Table A.2: RL Text Classifier Grid Search for N and C while trained on the Fraunhofer Dataset

N / C	0.01	0.1	1	10	100	1000
10	92.1	94.7	97.4	**100.0**	**100.0**	**100.0**
20	92.1	92.1	97.4	**100.0**	**100.0**	**100.0**
30	92.1	89.5	97.4	**100.0**	**100.0**	**100.0**
40	92.1	89.5	97.4	**100.0**	**100.0**	**100.0**
50	92.1	89.5	94.7	**100.0**	**100.0**	**100.0**
60	92.1	89.5	97.4	**100.0**	**100.0**	**100.0**
70	92.1	92.1	97.4	97.4	**100.0**	97.4
80	92.1	92.1	97.4	**100.0**	**100.0**	97.4
90	92.1	92.1	97.4	**100.0**	**100.0**	97.4
100	92.1	92.1	97.4	97.4	**100.0**	94.7
110	92.1	92.1	97.4	**100.0**	**100.0**	97.4
120	92.1	92.1	97.4	97.4	97.4	97.4
130	92.1	92.1	97.4	97.4	97.4	94.7
140	92.1	92.1	97.4	97.4	97.4	97.4
150	92.1	92.1	97.4	97.4	97.4	97.4
160	92.1	92.1	97.4	97.4	97.4	97.4
170	92.1	92.1	97.4	**100.0**	97.4	97.4
180	92.1	92.1	97.4	97.4	97.4	97.4
190	92.1	92.1	97.4	97.4	97.4	94.7
200	92.1	92.1	97.4	97.4	97.4	97.4

Table A.3: SWT Text Verifier Grid Search for N and C while trained on the Fraunhofer Dataset

N / C	0.01	0.1	1	10	100	1000
10	92.1	94.7	97.4	97.4	97.4	97.4
20	89.5	92.1	97.4	97.4	97.4	**100.0**
30	89.5	92.1	97.4	97.4	97.4	97.3
40	89.5	92.1	97.4	97.4	97.4	97.4
50	89.5	92.1	97.4	97.4	97.4	97.4
60	89.5	92.1	97.3	**100.0**	97.4	**100.0**
70	89.5	92.1	97.4	**100.0**	97.4	**100.0**
80	89.5	92.1	97.4	**100.0**	97.4	**100.0**
90	89.5	92.1	97.4	**100.0**	**100.0**	**100.0**
100	89.5	92.1	97.4	**100.0**	**100.0**	**100.0**
110	89.5	92.1	97.4	**100.0**	94.7	97.4
120	89.5	92.1	97.4	**100.0**	**100.0**	97.4
130	89.5	92.1	97.4	**100.0**	**100.0**	**100.0**
140	89.5	92.1	97.4	97.4	97.4	97.4
150	89.5	92.1	97.4	**100.0**	97.4	97.4
160	89.5	92.1	97.4	**100.0**	97.4	97.4
170	89.5	92.1	97.4	97.4	97.4	97.4
180	89.5	92.1	97.4	**100.0**	**100.0**	97.4
190	89.5	92.1	97.4	**100.0**	**100.0**	**100.0**
200	89.5	92.1	97.4	**100.0**	**100.0**	**100.0**

TABLE A.4: RL Text Verifier Grid Search for N and C while trained on the Fraunhofer Dataset

Bibliography

[1] Simon M Lucas, Alex Panaretos, Luis Sosa, Anthony Tang, Shirley Wong, and Robert Young. Icdar 2003 robust reading competitions. In *2013 12th International Conference on Document Analysis and Recognition*, volume 2, pages 682–682. IEEE Computer Society, 2003.

[2] Asif Shahab, Faisal Shafait, and Andreas Dengel. Icdar 2011 robust reading competition challenge 2: Reading text in scene images. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 1491–1496. IEEE, 2011.

[3] Kai Wang, Boris Babenko, and Serge Belongie. End-to-end scene text recognition. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1457–1464. IEEE, 2011.

[4] T. E. de Campos, B. R. Babu, and M. Varma. Character recognition in natural images. In *Proceedings of the International Conference on Computer Vision Theory and Applications, Lisbon, Portugal*, February 2009.

[5] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In *International Joint Conference on Neural Networks*, number 1288, 2013.

[6] Lukas Neumann and Jiri Matas. A method for text localization and recognition in real-world images. In *Computer Vision–ACCV 2010*, pages 770–783. Springer Berlin Heidelberg, 2011.

[7] Lukas Neumann and Jiri Matas. Real-time scene text localization and recognition. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3538–3545. IEEE, 2012.

[8] Boris Epshtein, Eyal Ofek, and Yonatan Wexler. Detecting text in natural scenes with stroke width transform. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2963–2970. IEEE, 2010.

[9] Thotreingam Kasar and Angarai G Ramakrishnan. Multi-script and multi-oriented text localization from scene images. In *Camera-Based Document Analysis and Recognition*, pages 1–14. Springer, 2012.

[10] World Standards. The world's scripts and alphabets. URL http://www.worldstandards.eu/other/alphabets/.

[11] David Nistér and Henrik Stewénius. Linear time maximally stable extremal regions. In *Computer Vision–ECCV 2008*, pages 183–196. Springer, 2008.

[12] Honggang Zhang, Kaili Zhao, Yi-Zhe Song, and Jun Guo. Text extraction from natural scene image: A survey. *Neurocomputing*, 122:310–323, 2013.

[13] Keechul Jung, Kwang In Kim, and Anil K Jain. Text information extraction in images and video: a survey. *Pattern recognition*, 37(5):977–997, 2004.

[14] Lukas Neumann and Jiri Matas. Text localization in real-world images using efficiently pruned exhaustive search. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 687–691. IEEE, 2011.

[15] Seiichi Uchida. Text localization and recognition in images and video. *Handbook of Document Image Processing and Recognition*, pages 843–883, 2014.

[16] Meng Lu, Kees Wevers, and Rob Van Der Heijden. Technical feasibility of advanced driver assistance systems (adas) for road traffic safety. *Transportation Planning and Technology*, 28(3):167–187, 2005.

[17] Cong Yao, Xin Zhang, Xiang Bai, Wenyu Liu, Yi Ma, and Zhuowen Tu. Rotation-invariant features for multi-oriented text detection in natural images. *PloS one*, 8(8):e70173, 2013.

[18] Michael A Smith and Takeo Kanade. *Video skimming for quick browsing based on audio and image characterization*. Citeseer, 1995.

[19] Toshio Sato, Takeo Kanade, Ellen K Hughes, and Michael A Smith. Video ocr for digital news archive. In *Content-Based Access of Image and Video Database, 1998. Proceedings., 1998 IEEE International Workshop on*, pages 52–60. IEEE, 1998.

[20] Datong Chen, Kim Shearer, and Hervé Bourlard. Text enhancement with asymmetric filter for video ocr. In *Image Analysis and Processing, 2001. Proceedings. 11th International Conference on*, pages 192–197. IEEE, 2001.

[21] Xiaoqing Liu and Jagath Samarabandu. Multiscale edge-based text extraction from complex images. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 1721–1724. IEEE, 2006.

[22] Luka Neumann and Jiri Matas. Scene text localization and recognition with oriented stroke detection. In *Computer Vision (ICCV), 2013 IEEE International Conference on*, pages 97–104. IEEE, 2013.

[23] Wenge Mao, Fu-lai Chung, Kenneth KM Lam, and Wan-chi Sun. Hybrid chinese/english text detection in images and video frames. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 3, pages 1015–1018. IEEE, 2002.

[24] Yann Rodriguez. *Face detection and verification using local binary patterns*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2006.

[25] Marios Anthimopoulos, Basilis Gatos, and Ioannis Pratikakis. A two-stage scheme for text detection in video images. *Image and Vision Computing*, 28 (9):1413–1426, 2010.

[26] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[27] Jing Zhang and Rangachar Kasturi. Text detection using edge gradient and graph spectrum. In *Pattern Recognition (ICPR), 2010 20th International Conference on*, pages 3979–3982. IEEE, 2010.

[28] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Pearson/Prentice Hall, 2008. ISBN 9780131687288. URL http://books.google.de/books?id=8uGOnjRGEzoC.

[29] Michael Jones and Paul Viola. Fast multi-view face detection. *Mitsubishi Electric Research Lab TR-20003-96*, 3:14, 2003.

[30] Simon JD Prince. *Computer vision: models, learning, and inference*. Cambridge University Press, 2012.

[31] Rodrigo Minetto, Nicolas Thome, Matthieu Cord, Neucimar J Leite, and Jorge Stolfi. T-hog: An effective gradient-based descriptor for single line text regions. *Pattern Recognition*, 46(3):1078–1090, 2013.

[32] Hilton Bristow and Simon Lucey. Why do linear svms trained on hog features perform so well? *arXiv preprint arXiv:1406.2419*, 2014.

[33] Victor Prisacariu and Ian Reid. fasthog-a real-time gpu implementation of hog. *Department of Engineering Science*, (2310/9), 2009.

[34] Shehzad Muhammad Hanif, Lionel Prevost, and Pablo Augusto Negri. A cascade detector for text detection in natural scene images. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.

[35] Rodrigo Minetto, Nicolas Thome, Matthieu Cord, Neucimar J Leite, and Jorge Stolfi. Snoopertext: A text detection system for automatic indexing of urban scenes. *Computer Vision and Image Understanding*, 122:92–104, 2014.

[36] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[37] Jiri Matas, Ondrej Chum, Martin Urban, and Tomás Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing*, 22(10):761–767, 2004.

[38] P-E Forssén. Maximally stable colour regions for recognition and matching. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.

[39] Huizhong Chen, Sam S Tsai, Georg Schroth, David M Chen, Radek Grzeszczuk, and Bernd Girod. Robust text detection in natural images with edge-enhanced maximally stable extremal regions. In *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pages 2609–2612. IEEE, 2011.

[40] Luka Neumann and Jiri Matas. On combining multiple segmentations in scene text recognition. In *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on*, pages 523–527. IEEE, 2013.

[41] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[42] Cong Yao, Xiang Bai, Wenyu Liu, Yi Ma, and Zhuowen Tu. Detecting texts of arbitrary orientations in natural images. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1083–1090. IEEE, 2012.

[43] Gary R Bradski. Real time face and object tracking as a component of a perceptual user interface. In *Applications of Computer Vision, 1998. WACV'98. Proceedings., Fourth IEEE Workshop on*, pages 214–219. IEEE, 1998.

[44] T Kasar and AG Ramakrishnan. Cococlust: Contour-based color clustering for robust binarization of colored text. *Proc. The Third CBDAR*, pages 11–17, 2009.

[45] Lluis Gomez and Dimosthenis Karatzas. Multi-script text extraction from natural scenes. In *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on*, pages 467–471. IEEE, 2013.

[46] SeongHun Lee, Min Su Cho, Kyomin Jung, and Jin Hyung Kim. Scene text extraction with edge constraint and text collinearity. In *Pattern Recognition (ICPR), 2010 20th International Conference on*, pages 3983–3986. IEEE, 2010.

[47] Tao Wang, David J Wu, Adam Coates, and Andrew Y Ng. End-to-end text recognition with convolutional neural networks. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 3304–3308. IEEE, 2012.

[48] Adam Coates, Blake Carpenter, Carl Case, Sanjeev Satheesh, Bipin Suresh, Tao Wang, David J Wu, and Andrew Y Ng. Text detection and character recognition in scene images with unsupervised feature learning. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 440–445. IEEE, 2011.

[49] Geoffrey Sampson. *Writing systems: A linguistic introduction.* Stanford University Press, 1985.

[50] F. Coulmas. *The Blackwell Encyclopedia of Writing Systems.* Ill., graph. Darst. Wiley, 1999. ISBN 9780631214816. URL `http://books.google.de/books?id=y3KdxBqjg5cC`.

[51] The Unicode Consortium. *The Unicode Standard, Version 7.0.0.* Mountain View, CA, 2014. ISBN ISBN 978-1-936213-09-2. URL `http://www.unicode.org/versions/Unicode7.0.0/`.

[52] Adnan Shaout, Dominic Colella, and S Awad. Advanced driver assistance systems-past, present and future. In *Computer Engineering Conference (ICENCO), 2011 Seventh International*, pages 72–82. IEEE, 2011.

[53] Michael Donoser and Horst Bischof. Efficient maximally stable extremal region (mser) tracking. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 553–560. IEEE, 2006.

[54] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. ISSN 0885-6125. doi: 10.1007/BF00994018. URL http://dx.doi.org/10.1007/BF00994018.

[55] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[56] Ron Kimmel, Cuiping Zhang, Alexander M Bronstein, and Michael M Bronstein. Are mser features really interesting? *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(11):2316–2320, 2011.

[57] Jiri Matas and Karel Zimmermann. Unconstrained licence plate and text localization and recognition. In *Intelligent Transportation Systems, 2005. Proceedings. 2005 IEEE*, pages 225–230. IEEE, 2005.

[58] Jakub Tomasz Lidke. Hierarchical font recognition. Diploma thesis, Phillips University of Marburg, 2010.

[59] Marc Petter, Victor Fragoso, Matthew Turk, and Charles Baur. Automatic text detection for mobile augmented reality translation. In *computer vision workshops (iccv workshops), 2011 ieee international conference on*, pages 48–55. IEEE, 2011.

[60] Nick Barnes. Improved signal to noise ratio and computational speed for gradient-based detection algorithms. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 4661–4666. IEEE, 2005.

[61] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27 (8):861–874, 2006.