Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Computer Vision Laboratory
Prof. Luc Van Gool

May 2008
Master's Thesis

Detecting and Reading Text in Natural Scenes

Author:
Martin Renold

Supervisor:
Till Quack

Abstract

This thesis describes a system for detecting text in natural scenes. A Viola-Jones style classifier cascade is used, and new features specifically aimed at text are evaluated. The detected regions are clustered and cleaned up for optical character recognition.

Additionally it is shown how the same approach can be used to locate two dimensional barcodes (like QR and Datamatrix codes) without relying on their finder pattern.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As digital imaging devices are becoming cheaper and widely available, and as collections of images accessible through the Internet are getting enormous, there is a growing demand to analyze and process this type of data in an efficient way. A typical task is to locate certain objects within the image, for example faces in order to identify persons.

The task addressed in this work is to find and read all the text within the image. This is challenging because the content and quality of typical images are rather unpredictable, in contrast to text from a scanned document.

Applications include searching for text within an image database, extracting Internet URLs and loading them into a mobile web browser, or reading signs to help visually impaired persons.

Text that occurs in natural scenes is made to be read by humans. Sometimes it is distorted from the clean appearance in a playful way towards the border of readability (Figure 1.1). Since the human vision system is tolerant and can use lots of context information, a computer program doing the same has to solve a rather difficult task.

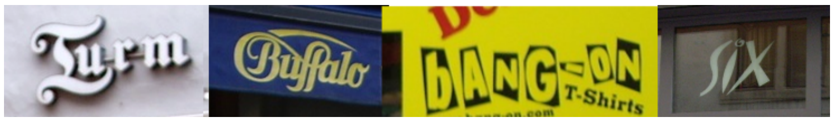


Figure 1.1: Those artistic fonts used for advertising are challenging to read.

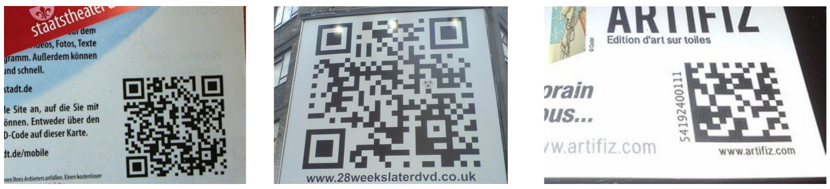


Figure 1.2: Consumer oriented applications of QR (left) and Datamatrix (right) codes.

While already popular for industrial tracking, two dimensional codes, in particular QR codes (Figure 1.2), are becoming increasingly popular as reader software for mobile phones becomes available. Their well specified appearance and the error correction information make them much easier to be read electronically. New applications include advertising, additional product information and business cards. Often the information encoded is just an Internet URL. The task of locating such codes is also addressed in this work using the same system as for locating text.

1.1 Previous Work

In 2001 Viola and Jones[1] have proposed a very successful system for frontal face detection (but not limited to this task) which quickly became popular. Their approach is described in chapter 2 and forms the basis for the text detection framework developed during this thesis.

An overview of early text information extraction systems (until about 2003) can be found in [4]. More recently Chen and Yuille[2] reported success using the Viola Jones approach for detecting text instead of faces. They made the second place in the ICDAR 2005 Text Locating Competition[3] with their system (precision 0.60, recall 0.60) while running over forty times faster than the winning entry (precision 0.62, recall 0.67). A further related work is the master thesis of Ching-Tung Wu[5] where a similar approach was used as part of an email spam classification system. The variant proposed in this thesis is described in detail in chapter 3.

After locating the text, it has to be read. While good programs for optical character recognition (OCR) exist, they are made for scanned-quality documents with a high resolution and constant illumination and are easily confused by non-text objects nearby.

1.2 Organization

The remainder of this report is structured as follows: in chapter 2 the Viola Jones approach for face detection is summarized. In chapter 3 the text detection system is presented in detail. In chapter 4 it is described how the raw detections are prepared for OCR. In chapter 5 the modifications for QR code detection are presented. In chapter 6 the performance of the features and other details of the detection system are discussed.

Chapter 2

The Viola Jones Approach for Object Detection

In their influential paper in 2001 Viola and Jones[1] described an efficient and accurate system for frontal face detection. Due to the extraordinary performance of this approach, many researchers have since proposed modifications applied it to different problems.

Their system uses a classifier to decide whether a square region of the image contains a face or not. This classifier is run at every possible location and scale. The key ideas to make this approach perform well are:

- The features are based on rectangular blocks and can be computed using *integral images* at any scale in constant time.
- A simple threshold is used to turn a feature into a *weak classifier*.
- Adaboost selects a subset of all weak classifiers and combines them into a *strong classifier*.
- A *cascade* of increasingly complex strong classifiers allows to reject easy background early without much computation.

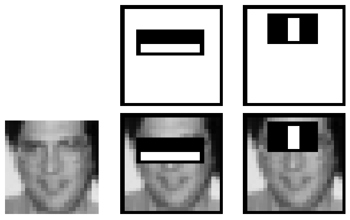


Figure 2.1: The first and second feature selected by Adaboost (from [1] page 11). The features measure the intensity difference between the dark and the bright regions.

Table 2.1: Notation for calculating rectangular blocks.

$I(x,y)$	the greyscale intensity values of the image
R	an arbitrary rectangular region, usually inside the detection window
$\|R\|$	the number of pixels inside R
W	the rectangular region of the detection window
$m(R)$	the mean intensity within R
$s(R)$	the standard deviation of the intensity within R

2.1 The Features

The features are based on the intensity difference between two or more rectangular regions as in Figure 2.1. They are contrast-normalized with respect to the window.

The following properties from Table 2.1 are used:

$$m(R) = \frac{1}{|R|} \sum_{(x,y)\in R} I(x,y) \tag{2.1}$$

$$s(R) = \sqrt{\frac{1}{|R|}\left(\sum_{(x,y)\in R} I(x,y)^2\right) - m(R)^2} \tag{2.2}$$

A contrast-normalized feature f using the two regions R_1 and R_2 is defined as

$$f = \frac{m(R_1) - m(R_2)}{s(W)}. \tag{2.3}$$

Those features are parametrized by their position and size of the rectangles R_1 and R_2 inside the detection window. Viola and Jones have used features with two and three adjacent rectangles to detect edges an lines as those in Figure 2.1, but also a diagonal checkerboard pattern with four rectangles.

2.1.1 Integral Images

Once created, an integral image (defined in Figure 2.2) allows to calculate the sum of a rectangular region very efficiently with only four lookups. The integral image of the image intensities can be used to evaluate the mean in equation 2.1.

The really useful part is that contrast-normalization can also be done with an integral image. The square intensity integral image allows to calculate the sum in equation 2.2 the same way with only four lookups. Since $s(W)$ only depends on the window, it can be evaluated once and used by all features. This makes the whole calculation of a feature f very fast.

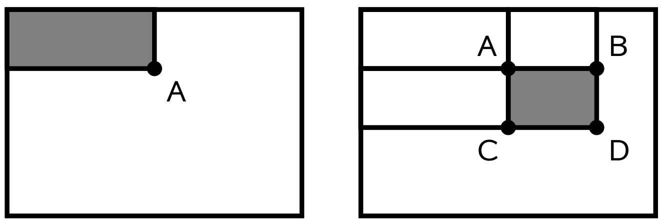


Figure 2.2: At the point A the integral image stores the sum of all pixels to the upper left of A. The sum of pixels in an arbitrary rectangle can be calculated as D - B - C + A.

2.2 Discrete Adaboost

Discrete Adaboost[1] was introduced in 1995 by Freund and Schapire[7]. The idea of boosting is to combine several weak classifiers $h_t(x)$ to a strong classifier $H(x)$. The weak classifiers are only required to be slightly better than chance.

In this setting the input x represents the data in a detection window, and the classifiers have binary $\{-1, +1\}$ output. The Adaboost strong classifier has the form

$$H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x)) \tag{2.4}$$

where T is the number of weak classifiers being used, and α_t are coefficients chosen by Adaboost.

During training Adaboost assigns a weight w_i^t to each training sample i for boosting round t and calls the *weak learning algorithm* to find the best weak classifier under the given weights. Then α_t and the new weights w_i^{t+1} are calculated. This process is repeated until T boosting rounds are done. Since none of the calculations depend on T, any other stopping criterion (like a target error rate) can be used. The complete algorithm (following the wording in [9]) is:

- Given: labeled training samples $(x_1, y_1), \ldots, (x_n, y_n)$ with $y_i \in \{-1, +1\}$
- Initialize weights $w_i^1 = \frac{1}{N}$ for $i = 1 \ldots N$
- For $t = 1 \ldots T$
  1. Use the weak learning algorithm to find the classifier $h_t(x) \in \{-1, +1\}$ that minimizes the error $\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} w_i^t$
  2. Calculate: $\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$
  3. Update weights: $w_i^{t+1} = w_i^t exp(-y_i \alpha_t h_t(x_i))$
  4. Normalize weights such that $\sum_i w_i^{t+1} = 1$
- The strong classifier is $H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$

[1]Sometimes simply called Adaboost. The term *discrete* was introduced later to distinguish it from more recent Adaboost variants that use real-valued output for the weak classifiers.

Advantages of Discrete Adaboost are:

- empirically shown to be somewhat resistant to overfitting [8]
- can still improve after all data is classified correctly
- fast training process
- can be used for feature selection
- easy to implement

Disadvantages are:

- will overfit in the presence of label noise [13]
- greedy learning process can give suboptimal solutions

2.3 Cascade Training

With a cascade structure, the false positives F and the detection rate D of the entire cascade are the products of the false positives f_i and detection rate d_i of the individual stages:

$$F = \prod_{i=1}^{N} f_i \tag{2.5}$$

$$D = \prod_{i=1}^{N} d_i \tag{2.6}$$

To reach some final training goal of the whole cascade (choosen by the user), a training goal for each stage can be calculated, assuming that all stages will have about equal performance. This typically leads to a target detection rate of 0.99 and a false positive rate of 0.30 for each stage, depending on the number of stages.

Adaboost will only try to minimize the missclassification error. However by adding a constant value to the sum in equation 2.4 it is possible to increase the hit rate at the expense of the false positives. Viola and Jones have proposed the following method to train a stage:

1. Let Adaboost choose and add the next weak classifier.

2. Tune the threshold of the current strong classifier[2] such that the desired detection rate is reached on the *validation set*.

3. If the tuned classifier does not reach the target false positive rate on the *validation set*, go back to 1.

In other words, features are added until the training goal is reached. The stage goal is the stopping criterion for Adaboost training.

[2]Since the output of the weak classifiers is binary there is only a discrete number of possible distinctive tunings. In the early stages with few weak classifiers the desired detection rate cannot be reached precisely.

2.4 Adaboost Variants

Since the original Adaboost publication[7], many improved boosting algorithms have been proposed. Most of those works compare to the results reported by Viola and Jones[1] on the face detection problem. Some of the most important variants are summarized below. There are at least ten more variants that are not listed here.

2.4.1 Real Adaboost

Real Adaboost, originally called as *Adaboost with confidence-rated predictions*, was proposed Schapire and Singer [12]. The main difference to Discrete Adaboost is that the weak learner gives a real-valued instead of $\{+1, -1\}$ output.

The task of the weak learner is only to partition data into several bins. The confidence-rated value is calculated from the sample distribution within the bin. Traditionally the weak classifier uses a simple threshold on one feature resulting in two bins, but other variants are possible. For example in [14], 64 bins were used on a single feature, holding the confidence-rated values in a lookup-table.

With this, the learning algorithm of the weak classifier has to be modified to minimize

$$Z_t = \sum_i w_i exp(-y_i h(x_i)). \tag{2.7}$$

The parameters α_t are no longer used; they are folded into the weak classifiers.

2.4.2 Asymmetric Extensions

The original Discrete Adaboost algorithm tries to minimize the number of miss-classifications (this is called the *symmetric* error). However in the cascade structure a false positive (background classified as face) can still be rejected by the later stages, while a false negative (face classified as background) at any stage is a final decision that degrades the overall performance (equation 2.6). In other words the cost of the classification error is not symmetric.

Viola and Jones have initially solved this problem in [1] by simply tuning the final threshold for each stage. However the weak learner is still aiming for a symmetric goal, and the weak classifiers are selected accordingly.

In [10] they proposed a solution to this, based on Real Adaboost. They have modified the weights of the training samples before each boosting round to force more attention to the positive samples. This did both simplify their classifier and made it more accurate.

A more recent approach to asymmetric boosting, based on the statistical interpretation of boosting, was proposed Shirazi and Vasconcelos [11]. The authors derived a modified boosting algorithm that is equivalent to Adaboost in the symmetric case.

2.4.3 Others

Also worth mentioning is Gentle Adaboost[15] which was shown to be effective on the face detection problem in [16]. It seems to be more tolerant to noisy data, in particular if there are wrong labels in the training set.

Another possible modification (not strictly to Adaboost) is exchanging the weak classifier. Traditionally a single threshold on a feature is used (boosting stumps) but boosting decision trees can also be effective. Other weak classifiers include Linear Discriminant Analysis (LDA) or lookup-tables as weak classifiers.

Chapter 3

Detecting Text

This chapter describes the combination of methods which lead to the best text detection results in our experiments. How much the individual choices contribute to the performance is discussed in chapter 6.

3.1 The Dataset

We collected a text dataset consisting of pictures taken from street signs and advertisments in the region of Zürich. There are also some book pages, newspapers and a few urls and numbers displayed on an LCD screen. Additionally about a quarter of the text comes from signs downloaded from Flickr[1]. Only text with roman letters was considered.

Readable text lines were labeled with rectangles, including some space above and below the letters. As in [2] those labels were split into detection windows with width-to-height ratio 2:1 and then used as positive samples for Adaboost. In total there are 599 rectangles labeled in 209 images split into 3423 detection windows.

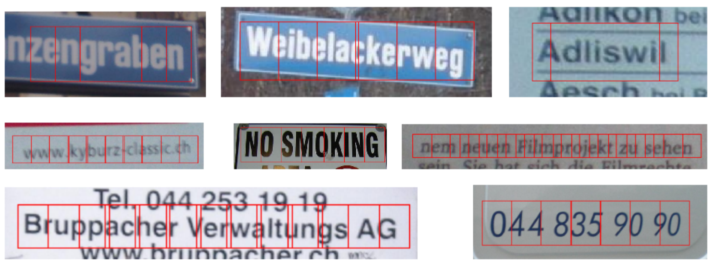


Figure 3.1: Labeled text split into overlapping 2:1 rectangles.

The background regions were also labeled manually. This allowed to leave some text unlabeled, in particular small, unreadable, rotated and heavily distorted text, as well as artistic fonts and graffiti. The training process usually stops because it cannot find enough false positives to train on. Thus additional

[1] www.flickr.com

city scene images were downloaded from Flickr, prefering those with many false positives detected. In total background regions from 632 images were used.

The images were split randomly into three sets of equal size for training, validation and testing.

3.2 Features

Text detection differs from the face detection problem solved in [1] in several aspects. First, the features should be invariant to color inversion, because both white and black text should be detected equally well. Second, while frontal face detection can use well-located features of the face (like the eye region) the position and shape of letters are not fixed. Thus statistical properties (or the texture) are more important.

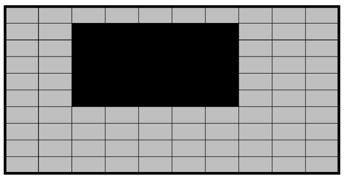


Figure 3.2: Block based features are parameterized by their location and size. All possible rectangles within this 10x10 raster are considered during training.

Most features calculate some property within a subrectangle of the window. The geometry of this block is parameterized as in Figure 3.2. This is the main source of feature diversity.

3.2.1 Intensity based Features

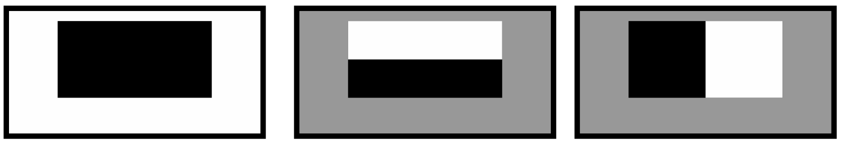


Figure 3.3: The intensity based features used. The absolute intensity difference between the black and the white region is calculated. Left: comparing block to window intensity; right: haar-like edge features

The horizontal and vertical edge features are similar to those used by Viola and Jones in [1]. In contrast to [1] the absolute values were used in order to detect black and white text equally well. The feature using the intensity of the whole window is similar to the three-rectangle feature used in [1]. Its main use (appart from adding diversity) is to check whether the regions above and below the text line have a different color from the text.

Both features are contrast normalized with respect to the whole window, as described in section 2.1.1. They are inherently scale invariant (except for the quantization effects that are shared by all features and described in section 6.5).

3.2.2 Variance based Features

A simple feature is the standard deviation (or the variance) of the gray level values of pixels inside a region. It can be calculated from the square integral image. Regions with very low intensity variance are quite common (sky, uniform surfaces) and rarely contain text. Two variance based features were used.

The first feature is simply the variance inside the whole window. Conveniently this value needs to be calculated anyway to contrast-normalize the intensity based features.

The second feature calculates the ratio between the variance inside a subrectangle and the variance of the whole window (continuing with the notation from Table 2.1):

$$f = \frac{s(R)}{s(W)}. \tag{3.1}$$

This has the effect of contrast normalization. The most obvious use of this feature is to check if a subregion contains only bright or only dark pixels where there should be text.

Note however that the variance does not increase with the number of edges; in a binary image the variance is maximized when 50% of the pixels are black. And the variance can change heavily when the image gets slightly blurred.

3.2.3 Edge based Features

On the original grayscale image $I(x,y)$ edge detection is performed using the Sobel operator, resulting in the gradient images $G_x(x,y)$ and $G_y(x,y)$:

$$G_x(x,y) = I(x,y) * S_x(x,y) \tag{3.2}$$
$$G_y(x,y) = I(x,y) * S_y(x,y) \tag{3.3}$$

with

$$S_x = \frac{1}{8}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \text{ and } S_y = \frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \tag{3.4}$$

Only the absolute values $|G_x(x,y)|$ and $|G_y(x,y)|$ are considered. The idea is to use the property that text has a certain minimum and maximum number of edges. Those statistics are different for horizontal and vertical edges, and also different depending on the position within the detection window (there is usually a blank stripe above and below the text). The challenge to somehow "count" the number of edges with just a few lookups in an integral image.

Let's assume that the original image is a clean black-and-white binary image (it may be slightly blurred). Consider only the vertical edges inside some block R (Figure 3.4). Then the sum along a single horizontal pixel row

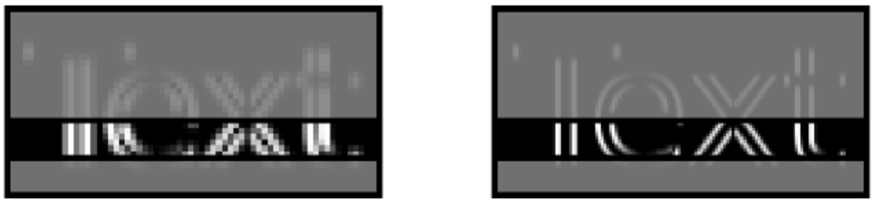


Figure 3.4: The values of $|G_x(x,y)|$ of the same text at different resolutions. We are interested in counting the number of vertical edges inside the horizontal stripe region R, independent of the resolution.

$$g(R,y) = \sum_{x \in R_x} |G_x(x,y)| \tag{3.5}$$

is exactly the number of vertical edges crossing this row, no matter how many pixels there are, no matter whether the image is slightly blurred or not, and no matter whether the edges are perfectly vertical or not. When combining all h pixel rows inside the $w \times h$ rectangle R to get a more robust feature, the values of $g(y)$ have to be averaged to still get an equivalent to the number of edges:

$$e_x(R) = \frac{1}{h} \sum_{(x,y) \in R} |G_x(x,y)| \tag{3.6}$$

and equivalently for horizontal edges

$$e_y(R) = \frac{1}{w} \sum_{(x,y) \in R} |G_y(x,y)|. \tag{3.7}$$

Because the text will rarely be clean black on white, e_x and e_y depend on the contrast of the text and the amount of noise in the window. Three normalization methods were put into the feature pool for Adaboost to choose. The first is the usual window contrast normalization (not solving the noise problem)

$$f_1 = \frac{f_x(R)}{s(W)} \tag{3.8}$$

and, since contrast normalization within the block makes sense too, also

$$f_2 = \frac{f_x(R)}{s(R)}. \tag{3.9}$$

To solve the noise problem a third normalization comparing to the amount of edges within the whole detection window is used:

$$f_3 = \frac{f_x(R)}{f_x(W)}. \tag{3.10}$$

Combining the three normalization methods f_1, f_2 and f_3 with e_x, e_y and $e_x + e_y$ gives nine different feature types. Additionally the ratio of the horizontal edges was used:

$$f_r = \frac{f_x(R)}{f_x(R) + f_y(R)}. \tag{3.11}$$

This makes a total of 10 edge based features.

3.2.4 Scanline based Features

The scanline based features were initially targeted at QR codes, but turned out to help for text detection, too. Starting from a border pixel of the detection window, all pixels along a given scanline are walked through at the full image resolution (Figure 3.5). The idea is to find the minimum or maximum segment length in the binarized image. Binarization is done with a hysteresis to reduce noise effects near the transitions.

The high and low threshold values are centered around the window mean intensity, with a distance choosen by adaboost (between 0.5 and and 2 times $s(W)$, see Table 2.1). Surprisingly Adaboost used this feature differently than expected, choosing a high hysteresis of $1.7s(W)$ and voting against text if the maximum horizontal segment length inside the text area was below 70%.

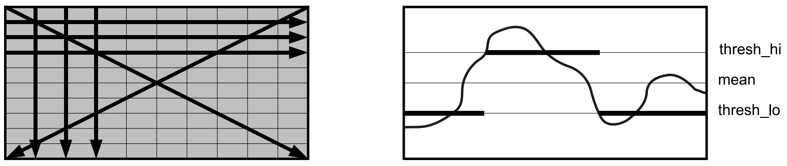


Figure 3.5: 10 horizontal, 10 vertical and 2 diagonal scanlines were in the feature pool. The intensity value is tracked and binarized (with hysteresis) along the selected scanline. The result is the minimum or maximum distance between two transitions.

3.3 Adaboost Training

Discrete Adaboost was used because of its simplicity, but with a modification for asymmetric learning. Apart from that, the approach taken by Viola and Jones (chapter 2) was closely followed.

Background windows were sampled randomly until the amount matched the number of the foreground windows. It was was allowed (for the last stage) to train with at least half as much background. The training stopped when there was not enough background left in the training set.

3.3.1 Asymmetric Boosting

The modification to Discrete Adaboost is to update the weights for each boosting round in an asymmetric way, as described in section 2.4.2. The implementation is based on *Asymmetric Adaboost* by Viola and Jones[10], but applied to Discrete

Adaboost instead of Real Adaboost. While Real Adaboost might have worked better, the results in section 6.3 show that Discrete Adaboost also improves with this modification.

Before each boosting round the weights of the positive samples are multiplied by a factor C while the weights of the negative samples are divided by the same factor. After this the weights have to be normalized. C is defined by

$$C = exp(\frac{k}{T}) \tag{3.12}$$

where T is the total number of boosting rounds and k a constant choosen by the user. The special case $k = 0$ stands for the symmetric case. Good choices for k seem to be $1 < k < 4$. Note that the definition of k is slightly different from the one in [10].

One problem that arises here is that the number of boosting rounds T has to be known before the boosting starts. Because the performance on the validation set is used as a stopping criterion, T depends on how well the selected features work. But this depends again on the choice of C. To resolve this, each stage is trained twice: once with T set to the number of boosting rounds of the previous stage, and once again with T set to the result of the first training. This could be repeated several times until T converges, but one iteration seems to be enough for practical purposes.

3.4 Detector Setup

A minimum text window size of 40x20 pixels was used, with a scale factor of 1.1. The maximum size of the scanning window is limited only by the image dimensions. When scanning at a given scale, a window step of 0.4 (horizontal) and 0.2 (vertical) of the current window size in the respective directioin is used.

Those parameters were also used when scanning for false positives during training.

3.5 Evaluation Method

The performance measure of the ICDAR text locating competition[3] was not used because the problem of word segmentation was not solved during this thesis (this was delegated to the OCR application). Instead the 2:1 detection windows were directly evaluated (before clustering them). The OCR quality is considered separately in section 4.4.

3.5.1 False Positive Rate

The *false positive rate* is simply measured by running the detector on the labeled background regions in the testing set. The positive responses are counted and divided by the total number of classified windows. This way, detections that overlap with a text region are never counted as false positives.

3.5.2 Strict Hit Rate

The hit rate is more difficult to define. Two different evaluation methods were used.

Table 3.1: Per-stage results of the trained classifier (testing set).

Stage	false pos	strict hit rate	permissive hit rate	new features
1	0.147	0.9982	1.0	1
2	0.025	0.9877	1.0	1
3	0.0085	0.9754	0.9965	2
4	0.0026	0.9684	0.9947	7
5	808e-6	0.9544	0.9947	6
6	312e-6	0.9430	0.9921	8
7	115e-6	0.9281	0.9886	18
8	45e-6	0.9140	0.9886	26
9	20e-6	0.9035	0.9877	300

Remember that the labeled rectangles were split into overlapping 2:1 windows. The *strict hit rate* is calculated by running the classifier exactly centered on each labeled window A. This is used during the training to tune the Adaboost threshold and to check if the stage goal has been reached.

3.5.3 Permissive Hit Rate

While the strict hit rate can be evaluated quickly, it does not measure the detection performance in a realistic setup. The first reason is that the final classifier will be evaluated at a limited scanning resolution, and thus miss the target slightly, depending on the scanning resolution. The second reason is that it is not neccessary to get a positive response exactly where each label is, as long as there are enough positive responses in the near neighbourhood.

To exclude those effects when comparing between different detectors, a second measure called the *permissive hit rate* or just *hit rate* is used. With this definition a hit for the ground-truth window A is declared if there is any positive window B detected with an intersection over union value (between the rectangles corresponding to A and B) above a certain threshold:

$$\frac{|A \cap B|}{|A \cup B|} > 0.4 \tag{3.13}$$

The threshold was choosen less strict than the usual value of 0.5 in order to exclude effects caused by missaligned labels. There are some extreme cases in the dataset where the text is in the upper half of the window, and others where it is in the lower half, because the original text was not exactly horizontal but labeled with a single horizontal rectangle.

3.6 Results

The per-stage target for the strict hit rate on the validation set was 0.99 and 0.40 for the false positive rate. Table 3.1 shows the values that were actually reached on the testing set. The first two stages go well beyond their goal using only a single feature. The last stage did not quite reach the target any more (the number of features was limited to 300 per stage).

The theoretical strict hit rate after 9 stages is $0.99^9 = 0.91$. The actual values were 0.96, 0.93 and 0.90 on the training, validation and testing set respectively.

It is worth noting that in the first four stages Adaboost selected only edge based features (section 3.2.3). In the later stages all feature types were selected, with a slight preference for the edge based ones.

The total runtime to produce the raw detection results on a typical 1600x1200 jpeg image is about 0.5 seconds (Pentium 4, 2.40GHz). The actual detection part (after the integral images are calculated) takes 0.180 seconds.

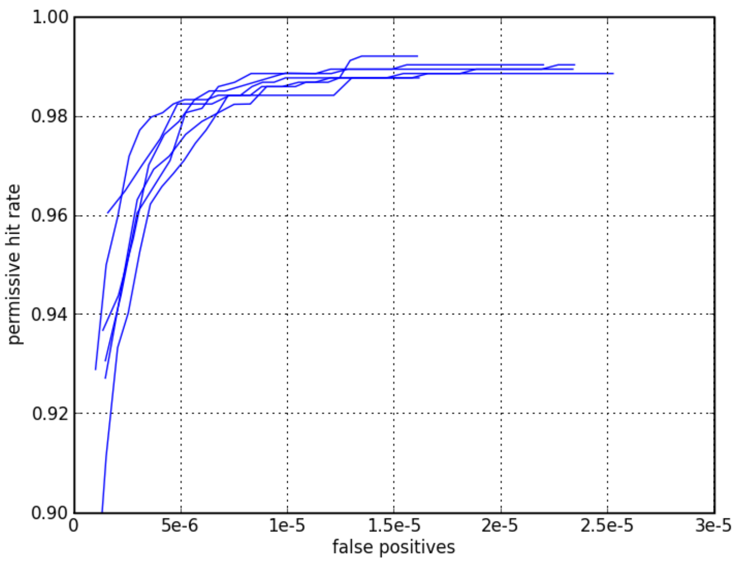


Figure 3.6: ROC curves for six identical trainings. The curves are calculated by adjusting the Adaboost threshold of the last stage. The noise comes from the random sampling of background during the training process.

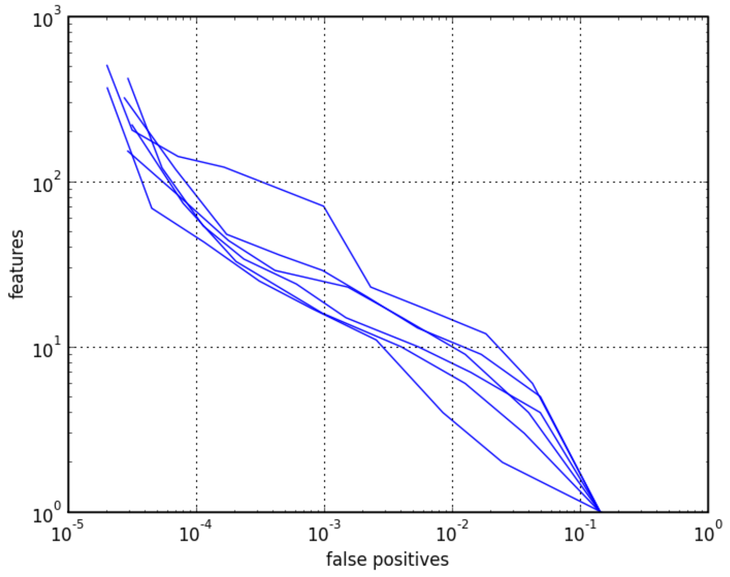


Figure 3.7: Complexity of the cascade for six identical trainings. For each stage there is a point with the final false positive rate and the number of features used (equal to the number of weak classifiers).

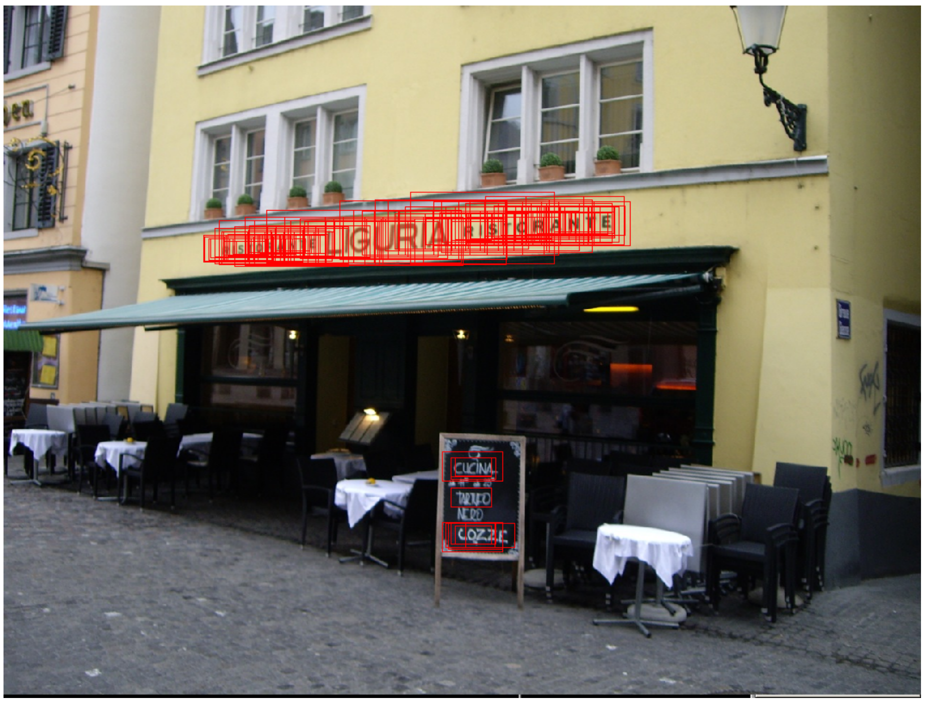


Figure 3.8: A typical raw detection result. Unless otherwise noted all text results are from the best ROC curve in Figure 3.6 at a false positive rate of 5e-6.

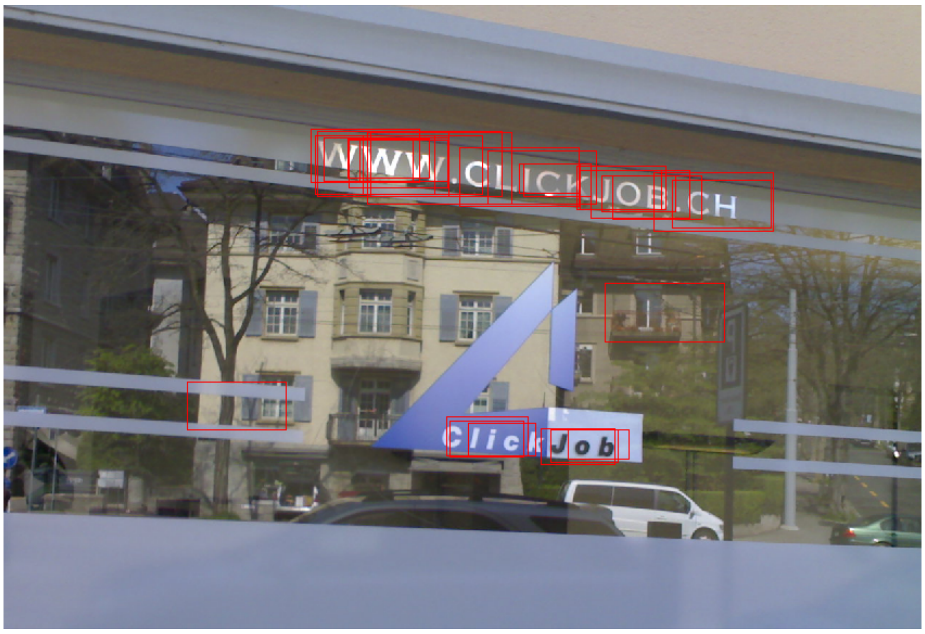


Figure 3.9: This text is rotated by seven degrees but still detected well. The rectangles are clustered less dense since this is slightly harder. Windows on buildings are typical causes for false positives (surprisingly, trees are not).

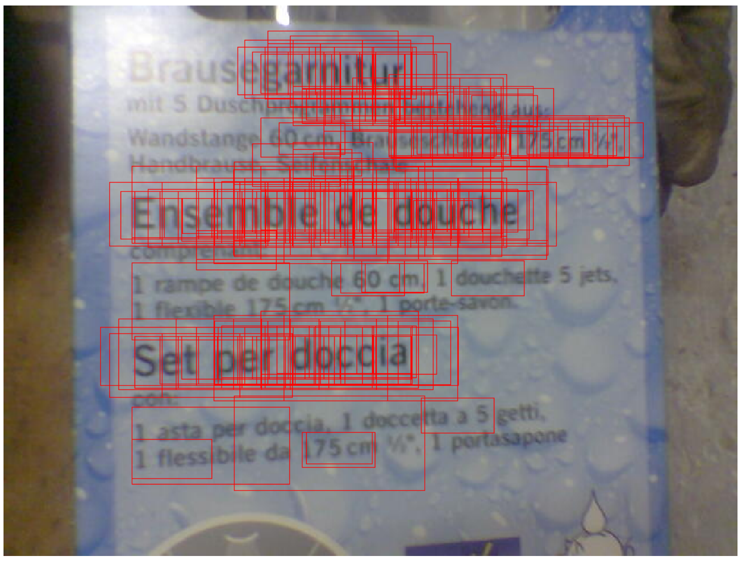


Figure 3.10: The gradient in the upper left of the image, blur and the non-uniform background cause problems.

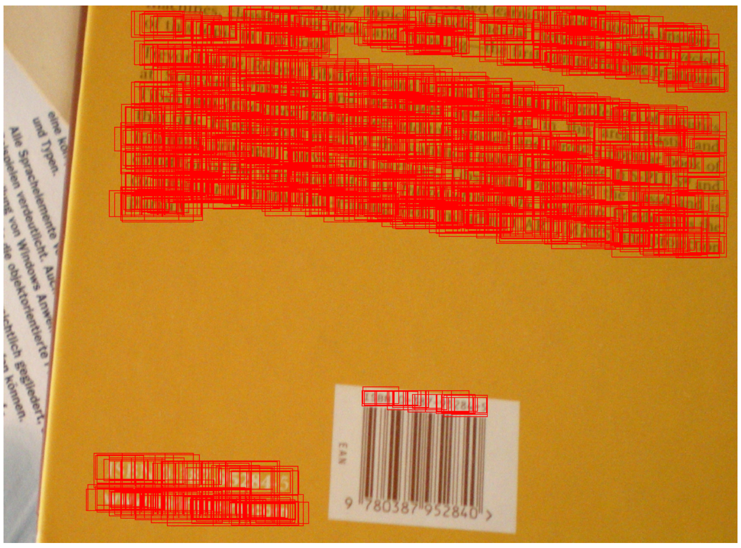


Figure 3.11: Clean text like on this bookcover is located precisely. The number at the bottom is not detected because there is no spacing to the barcode.

Chapter 4

Reading Text

4.1 Combining the Raw Detections

The raw detection windows are clustered, and isolated detections (being usually false positives) are discarded. Two detection rectangles R_1 and R_2 where put into the same cluster if their intersection-over-union measure was above a given threshold:

$$\frac{|R_1 \cap R_2|}{|R_1 \cup R_2|} > 0.4 \tag{4.1}$$

The prior knowledge that text detections are more likely to cluster horizontally than vertically was used by enlarging all raw detection windows horizontally by 1/3 of their original width before clustering. This has the additional advantage of including the first or last letter if they are missing.

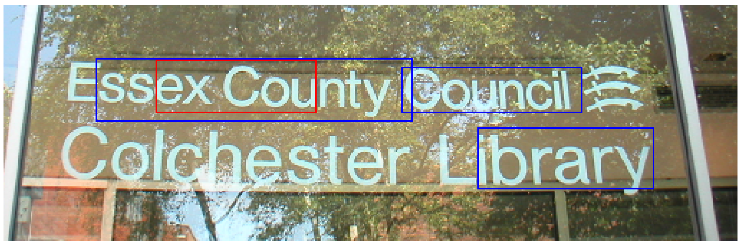
Clusters with less than three rectangles are discarded. For each of the remaining clusters, the text height is estimated by the average logarithmic height within the cluster:

$$\hat{h} = exp(\frac{1}{n}\sum_{i=1}^{n} log(h_i)) \tag{4.2}$$

The final result for each cluster is the union of all rectangles, excluding rectangles that are more than one scale step above the estimated height $\hat{h}$. When all rectangles were included instead, the bounding box of the text was often estimated too large.

4.2 Preprocessing

The detected regions were cut out and prepared for OCR. In order to keep the processing time reasonable, the cropped image is scaled down to a maximum height of 80 pixels. Contrast stretching is done to make the darkest pixel black and the brightest pixel white. This is required for black-on-white detection (described below) and for some of the binarization methods.



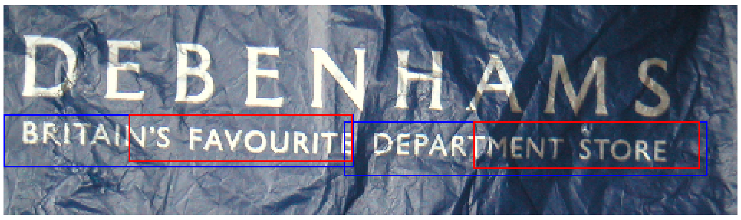


Figure 4.1: Two hard text areas from the ICDAR trial test set. Increasing the number of false positives to 45e-6 allows to find more of this text.

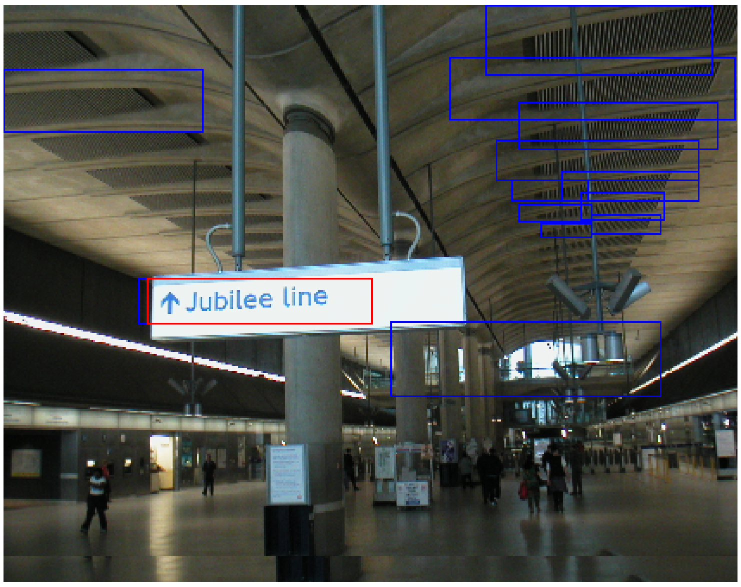


Figure 4.2: Some typical false positives when running at 45e-6 false positives (blue, background). If an image has false positives there are often many of them together, increasing the required OCR processing time a lot. At 5e-6 false positives (red) only the correct text is found.

4.2.1 Binarization

Most OCR programs accept greyscale images as input, however Chen and Yuille[2] reported to get better results when using a slightly modified version of Niblack binarization first. Many methods have been proposed and compared for document binarization, for example in [20].

First note that most binarization methods assume a black on white text and will give very poor results for white on black. Because of this, white on black images must be detected and inverted before binarization. Since there are usually more background pixels than text pixels, this can be done simply by counting the pixels below the mean intensity. The image is inverted if more than 50% of the pixels are below the mean intensity.

Most notable are Otsu's global thresholding, Niblack's adaptive thresholding and Sauvola's algorithm which is a variant of Niblack (details can be found in [20]). Global thresholding might work well on scanned documents, but is of little use when there is a gradient over the image, as in Figure 4.3.

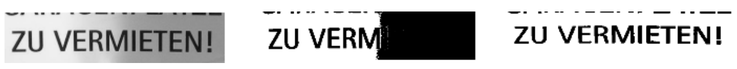


Figure 4.3: Original image, global threshold (Otsu), local threshold (Sauvola).

Local thresholding works by choosing an individual threshold for each pixel, depending on the pixels in the near neighbourhood. Niblack's method calculates the local threshold T from the local mean m and standard deviation s of the pixels within a small rectangle:

$$T = m + k \cdot s \tag{4.3}$$

The parameters are the constant $k = -0.2$ and the size of the local window which is set to $\frac{3}{4}$ of the detection window height. The method is said to be somewhat robust to different window sizes, but the fact that we already have a good estimate of the font size is still an advantage.

The most common problem with this Niblack binarization is that noise is greatly amplified in empty regions of a document, resulting in lots of clutter. Sauvola's method solves this problem with a slightly modified formula:

$$T = m \cdot (1 - k \cdot (1 - \frac{s}{R})) \tag{4.4}$$

with a different parameter $k = 0.5$ and $s = 128$ being the dynamic range of the standard deviation over the whole image.

The local thresholding algorithms can be implemented efficiently using integral images as described in [21], and thus in principle the already calculated integral images could be reused.

4.3 Evaluation

To measure the OCR performance, the ground truth and the OCR output for each image were both treated as a "bag of words". The assumption is that a word has been located correctly if it was read correctly. Three properties were calculated, summarized in Table 4.3. All of them have a maximum value of one.

Table 4.1: Word Evaluation Measures

correct words	ratio of words in the ground truth with an exact match in the OCR output
almost correct words	ratio of words in the ground truth with an almost-exact match in the output (Levenshtein distance smaller than one third of the correct word length)
clutter	ratio of words in the output that do not match any word in the ground truth, according to either of the criteria above

We have annotated the readable text in 88 challenging low-quality images (640x480 with blur and noise) taken with a mobile phone. As an additional dataset the words from the ICDAR train and trial test set were used, ignoring the text location information.

4.4 Results

Three different OCR engines were tested, two of them free software and one commercial: Tesseract[1], GOCR[2] and the ABBYY FineReader Engine[3], called FRE from now. The results can be seen in Figure 4.4. FRE did beat Tesseract in terms of quality, however Tesseract was quite a bit faster.

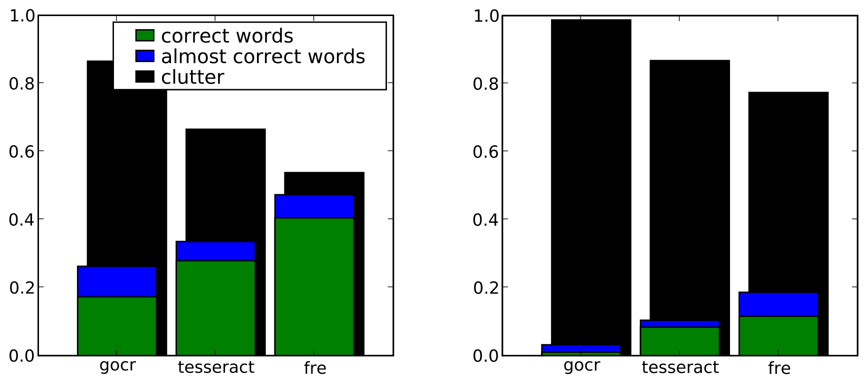


Figure 4.4: Comparing the different OCR engines after Niblack adaptive thresholding, on the ICDAR datasets (left) and on the low-quality dataset (right). This is the result of the complete system, meaning that a missed word can be either unreadable or not located.

The results on the low quality dataset were quite poor. The most common

[1]http://code.google.com/p/tesseract-ocr/
[2]http://jocr.sourceforge.net/
[3]http://www.abbyy.com/

reason for this seems to be the low resolution or the heavy blur of the text in most images. While some text was not found by the detector, much of the cleanly located text could not be read anyway. The result in Figure 4.7 shows that running the detector with a higher hitrate does not help much.

On the ICDAR dataset the text resolution was high in most images. The problems were rather the special fonts, non-uniform backgrounds and single letters or digits. The detector needs a minimum of about three letters, and its training set did not include many special fonts. Training the detector with a more challenging dataset did just make it more complex, without a significant improvement of the precision.

The effect of binarization can be seen best with Tesseract in Figure 4.6. Niblack's method turned out to work best, possibly because the advantage of Sauvola's method would be mainly on empty regions that were rare within the well-located text boxes. Both implementations had two additional hard (non-adaptive) thresholds for very dark and very bright regions, suppressing the most obvious noise.

Attempts to remove some connected components in the binarized image (based on geometry constraints like the aspect ratio) did not work well; this was probably handled better by the OCR engines already.

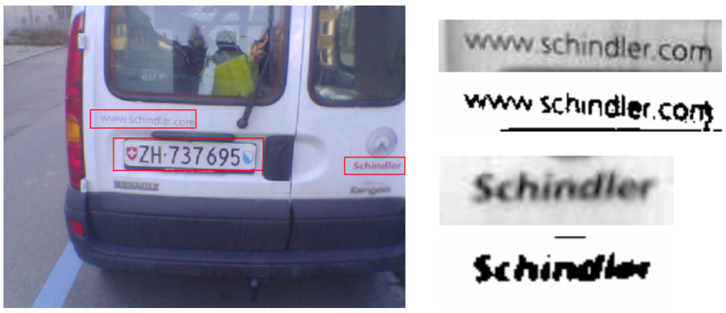


Figure 4.5: A sample image from the low quality dataset. With FRE only the text "QZH-737695" was returned. The text "www schmdJer comm" could also be read after scaling the image up.

Unlike Tesseract, with FRE the unbinarized greyscale images did work almost equally well as with Niblack binarization (see Figure 4.7). It could be that FRE has improved since the results reported in [2]. Scaling the image up did help sometimes, for example in Figure 4.5.

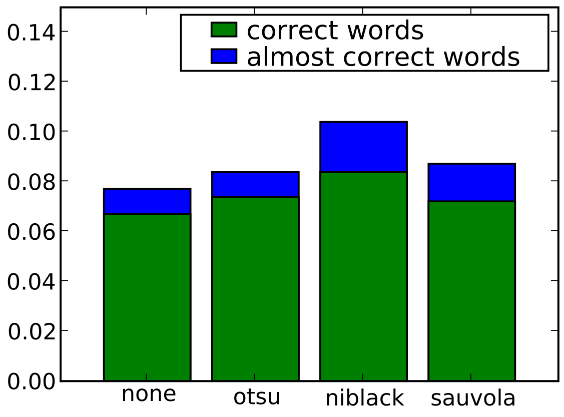


Figure 4.6: Results with Tesseract on the low quality dataset, using the greyscale regions directly, using global thresholding (Otsu) and using an adaptive threshold (Niblack and Sauvola).

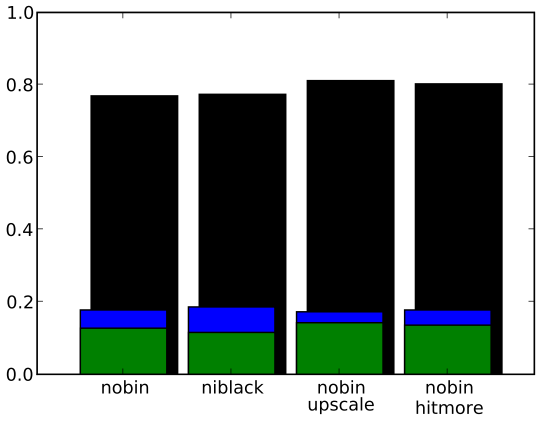


Figure 4.7: Comparing different methods on the low quality dataset with the FineReader Engine: no binarization, Niblack binarization, scaling the images up, and increasing the hitrate of the detector.

Chapter 5

Detecting Codes

QR codes are usually located by their special special finder pattern, for example using the techniques in [22]. This chapter describes how localization can be done using bit-pattern inside the code regions instead. The system is separate from the text detection described in chapter 3, but compatible in the sense that the same integral images are used. Datamatrix codes are also detected since they have a very similar bit pattern. In contrast to the text detector, the QR detector can find codes at any rotation.

Another possibility with the same basic approach would have been to train a detector for the finder pattern or (even better) the entire corner area including a part of the quiet zone. This has three disadvantages: rotation invariance is lost, more training data is required and more windows have to be checked in the detector. (To reliably hit the finder pattern using position-dependent features the detection window would have to hit it with one-bit precision. Many more windows would have to be tested than for only hitting the bit pattern somewhere.)

5.1 The Dataset

QR codes have been labeled with two rectangles each: one around their finder pattern (to have an estimate of the bit size) and one inside the central data region. A fixed number of random squares of two times the marker size were sampled from each central code region and used as positive windows for training Adaboost (see Figure 5.1).

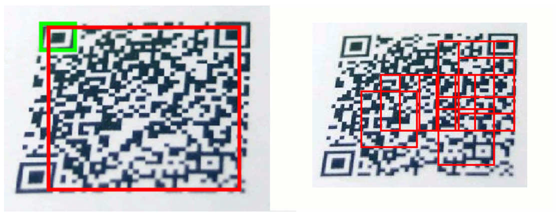


Figure 5.1: QR code labeling (left) and the random square windows used for Adaboost training (right).

The dataset contains mainly testing images of free QR decoding libraries and images with the *qrcode* tag downloaded from flickr[1]. The images from the libraries did contain only the QR code without much background. Blurry, very small and perspectivically distorted codes are included. In total 237 codes were labeled (2280 positive windows), again split into three equally sized sets for training, validation and testing.

The training background was the same as for text detection, with the addition of a few text areas, since text is a typical false positive when searching for codes.

5.2 Boosting Trees

In contrast to text detection, decision trees did improve the performance for code detection slightly. The comparision is in section 6.4.

In the Adaboost literature, decision trees are often choosen as weak classifiers instead of simple feature treshold (called decision stumps). Decision trees are trained on the weighted samples provided by Adaboost, trying to decrease the weighted classification error somehow.

Each node of the tree uses a threshold on a single feature to make a decision. Deep nodes in the tree usually see only a small percentage of the data. The leaves of the tree stand for the final classification result, which is the output of the weak classifier.

The decision trees were grown using the same split criterion as for Classification and Regression Trees (CART) [17, 19]. Starting with an empty tree, the next node to be added is always the one that gives the greatest reduce of the so-called *impurity*.

There are several measures of the impurity of a node. Most common are the missclassification error, the entropy and the Gini diversity index. The Gini diversity index was used. Since it is problematic to find the best feature/threshold combination with a very small fraction of the training data, only splits that take into account at least 5 positive and 5 negative samples were considered. The trees were grown until the maximum size of three nodes was reached. In contrast to CART the trees were not pruned.

5.3 Training and Detector Setup

The same features as for text detection were used. For the intensity based features (section 3.2.1) both absolute and non-absolute intensity differences were considered.

The parameters were set up somewhat different than for text, as can be seen in Table 5.1. Most notably the scale factor was higher and decision trees were used as weak classifiers.

A square scanning window of a 5x5 resolution (for feature selection) was used, with the detection starting at a minimum size of 8x8 pixels. The scale factor was set to 1.25 (in contrast to 1.1 used with text)

[1]http://www.flickr.com/search/?q=qrcode

Table 5.1: Comparing the text and the code detector.

	detecting text	detecting codes
window resolution (features)	20x10	5x5
minimum window size	40x20	8x8
scale step	1.1	1.25
window step dx / dy	0.4 / 0.2	0.35 / 0.35
maximum tree depth	1 (stumps)	3

5.4 Combining and Preprocessing

The raw detection results were clustered using the same overlap criterion as for text (equation 4.1). The detections squares were enlarged to twice their size before the clustering step. This allows them to merge easier and at the same time gives some spacing around the final combined result. This spacing is needed to include all four corners of a code that is rotated by 45 degrees. The final result is the union of the enlarged detections for each cluster (oversized detections are not removed). A minimum of 5 detections is required per cluster.

The resulting regions are scaled up or down such that the code-bits are roughly 6x6 pixels. The code-bit size can be estimated nicely using equation 4.2, since the raw detections all include a more or less constant number of code-bits. No binarization is done, only contrast stretching.

5.5 Results

The ROC and curve of the classifier can be seen in Figure 5.2. Since the last stage has a pretty bad hitrate the second-last stage was used for the detection results below.

The complexity curve is shown in Figure 5.3. It should be noted that an about half as complex classifier can be trained by removing the hard images from the training set.

Those curves look clearly worse than the results for text detection, but it is possible to run the QR detector at a higher false positive rate because many false positives tend to be isolated (see Figure 5.7).

Detecting QR codes by their bit pattern worked surprisingly well for clear codes (eg. Figure 5.4). As long as no blur is involved, the bit pattern is distinctive enough against most backgrounds, including text and vegetation.

The conceptual problem of this approach is that neither the information of the finder pattern nor the quiet zone are used. For low resolution and blurred QR codes those two properties seem to be the main distinctive patterns. The bit pattern is no longer clearly binary and it cannot be distinguished alone (even manually) from many of the background patterns. Often the code can still be read using the redundant error correction information.

The reading performance was evaluated by passing the content of the detection rectangles to the qrcode library[2].

[2]http://qrcode.sourceforge.jp/

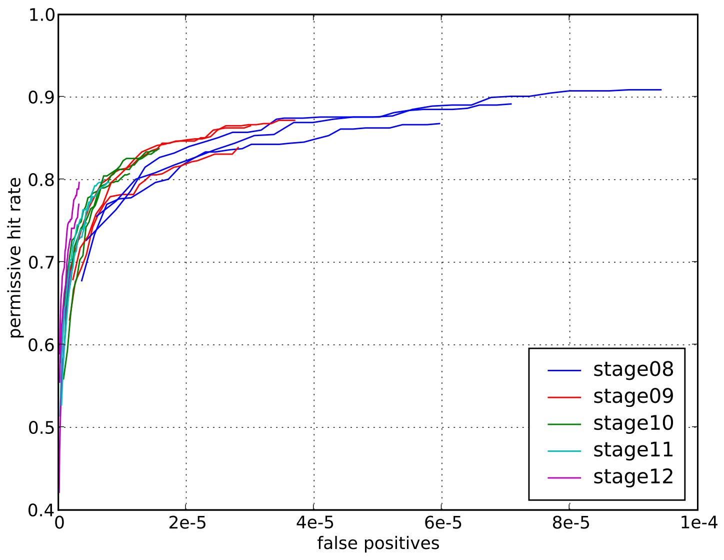


Figure 5.2: ROC curves for three identical trainings, shown for the last few stages. (The permissive hitrate was introduced in section 3.5.3.)

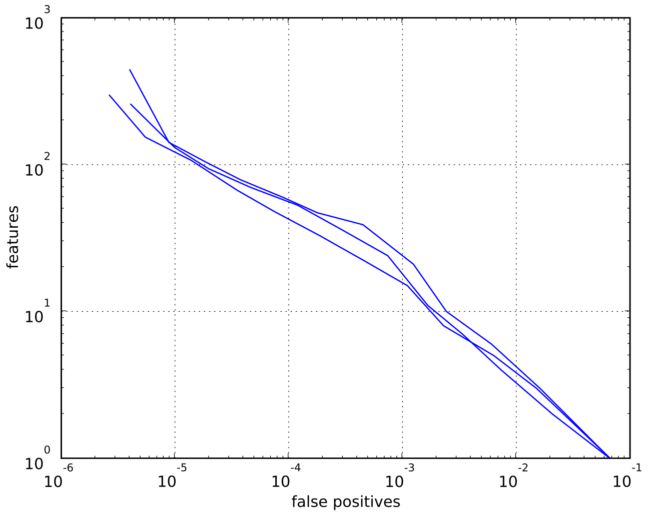


Figure 5.3: Complexity of the cascade for three identical trainings.

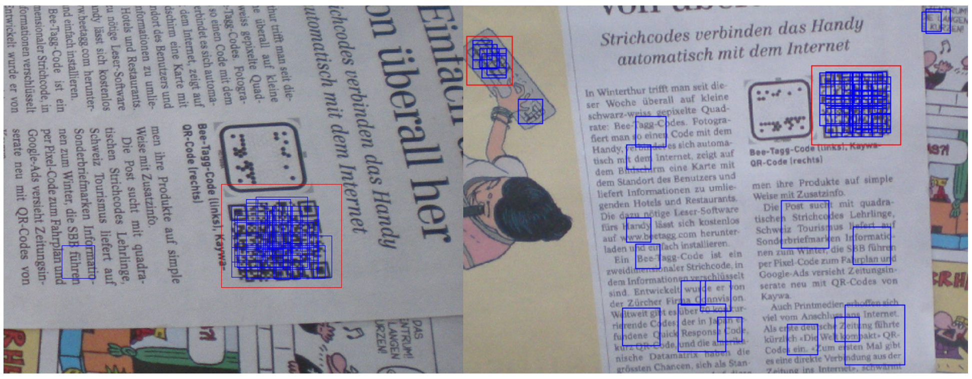


Figure 5.4: A typical detection result at 10e-6 false positives. Blue: raw detection results; red: after combining.

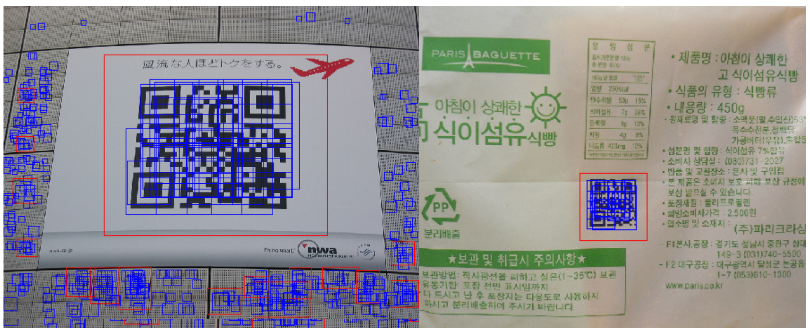


Figure 5.5: The worst case of false positives was the special pattern on this building in the left image. Clear QR codes are detected well and the false positives are not a big problem.

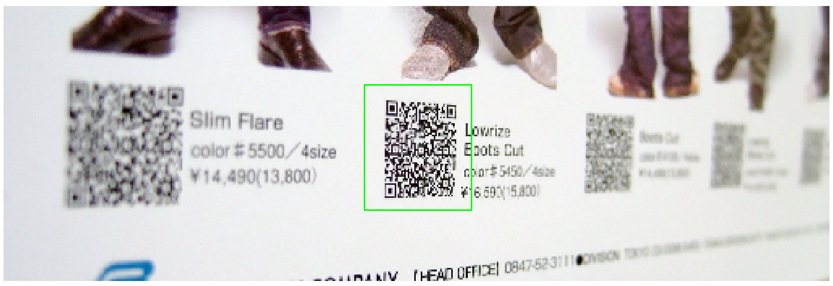


Figure 5.6: Blurred or low-resolution codes are a conceptual problem with this method. The bit pattern is no longer distinctive enough.

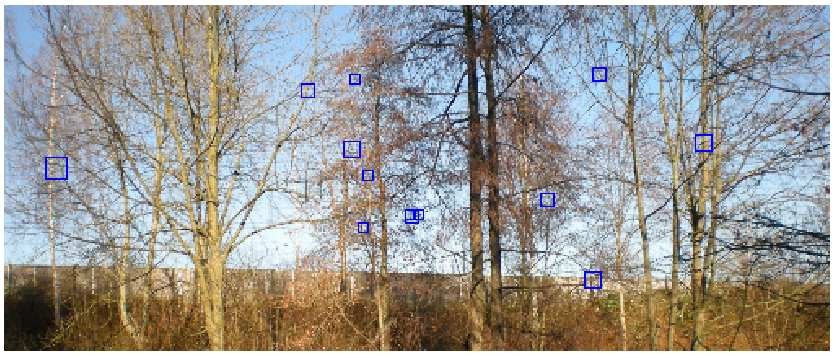


Figure 5.7: In vegetation false positives are quite common, but because they are usually isolated they get filtered out by the minimum cluster size rule.

Table 5.2: Performance of QR code localization.

	detector false pos	codes read	total time
qrcode only (no localization)	(1.0)	178	23min
qrcode after localization	10e-6	136	7min
qrcode after localization	120e-6	151	9min

The number of codes that were readable are listed in Table 5.2. Qrcode alone did read more codes but did so very slowly. There certainly is potential to speed things up further; most time is still spent in qrcode and some time is wasted by encoding the intermediate results as png images. The QR detector itself is equally fast as the text detector.

While the localization did miss some readable codes, there were also cases where the code could only be read after localization. On the other hand sometimes the code was clean and nicely extracted but it could not be read by qrcode any more. So there certainly is room for improvement in the reading step.

Chapter 6

Discussion

6.1 Feature Pool Size

There is a trade-off between the number of features presented to Adaboost and a reasonable training time. The main reason for the high feature count is the variation of position and size of the block-based features.

For text detection initially a window size of 40x20 (44100 sub-rectangles) was used, but the feature count had to be reduced by requiring either full width or full height (Figure 6.1, 419 sub-rectangles). With this restriction a 20x10 window worked just as well in terms of precision. But at this lower size it was feasible to include all rectangles with even x coordinates (Figure 3.2, 3025 sub-rectangles). This final step reduced the false positives by factor two at a hitrate of 94% for text detection.

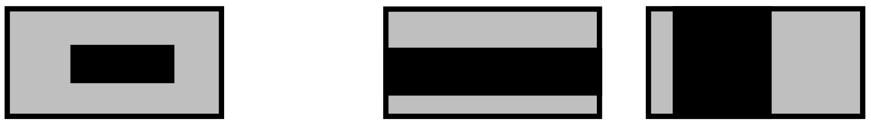


Figure 6.1: Rectangluar regions in the restricted feature pool. Left: arbitrary region (full feature pool); right: geometry restriction to reduce the number of features.

6.2 Performance of the Features

The effect of the feature classes described in section 3.2 for text detection can be seen in Figure 6.2. Each of them did lead to a clear precision improvement together with a reduction of the classifier complexity at the same time.

6.2.1 Features based on Gradient Orientation

We also conducted experiments with features similar to the Sobel edge based features described in section 3.2.3, but based on the gradient magnitude and orientation.

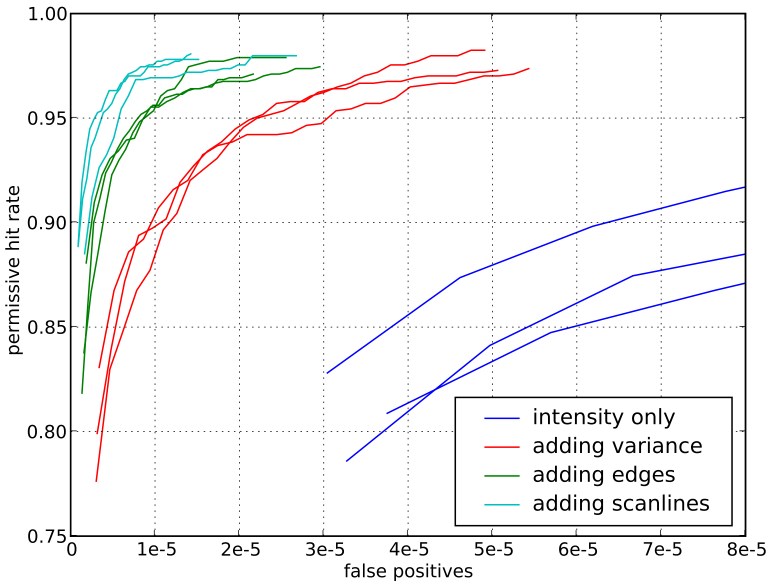


Figure 6.2: ROC curves of the detector when using different feature types. (The geometry restriction in Figure 6.1 was enforced on the feature locations for faster training.)

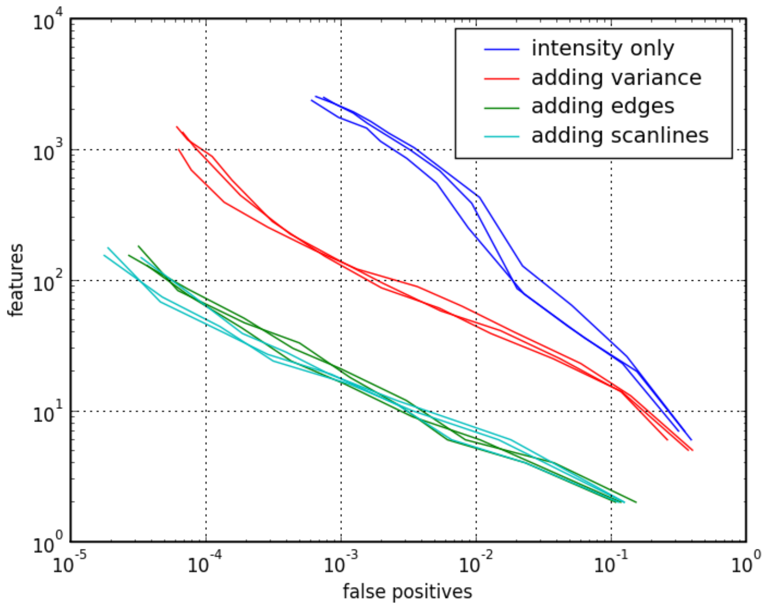


Figure 6.3: Complexity of the classifier when using different feature types.

Six gradient orientation bin images were calculated as proposed by Levi and Weiss[23], each of them containing the gradient magnitude for some gradient orientation range. The simple “Dominant Orientation Features” from [23] were used, and some variants of it tried (eg. a weighted sum of three adjacent orientation bins). Those features are similar to the horizontal to vertical edges ratio in equation 3.11, but extended to six instead of two orientations.

A performance gain over the Sobel edge based features was visible but it was small. The bigger issue was that calculating all the orientation bins and their integral images took lots of processing time and memory in the detector. Because of this the approach was not explored further.

6.3 Assymmetric Boosting

Asymmetric Adaboost as described in section 3.3.1 did give a small but clear precision improvement and a noteable reduction of the classifier complexity. The difference that it made on the final text detection training can be seen in Figure 6.4.

The classifier got simpler especially in the early stages where it is really needed for a fast execution time. The first stage consistently used only one feature instead of two, and in the later stages the number of features was almost a factor two between the best results (note that the complexity plot is logarithmic). It did make even more difference in earlier experiments.

6.4 Boosting Trees

In [16] it was concluded that trees were better suited for the face detection problem. Other sources (eg. [18]) suggest that this depends a lot on the problem and the boosting algorithm used.

The conclusion in this thesis is that boosting trees gave about equal performance as boosting stumps. Trees gave slightly better performance only on the code detection problem, which is shown in Figure 6.5. It is possible that trees help only if the features are not good enough for simple thresholding.

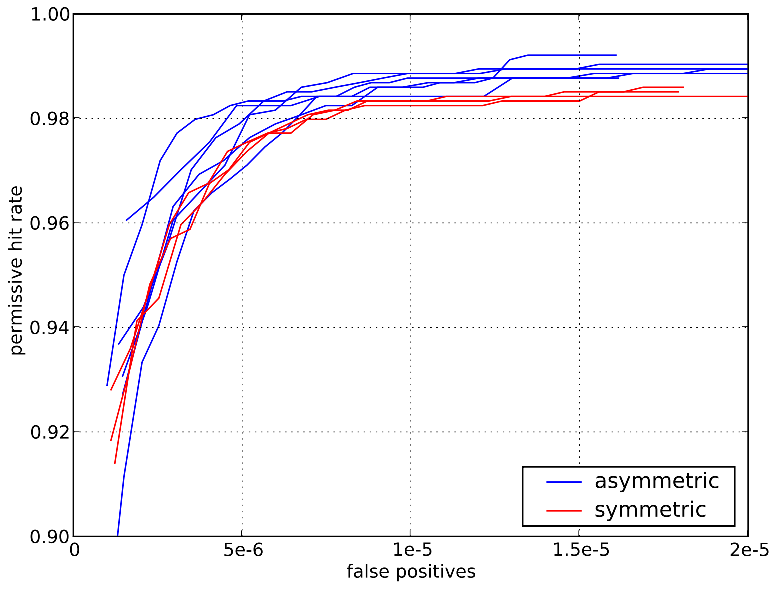
6.5 Quantization effects

When scaling the 20x10 pixels detection window by a factor of 1.1 it is clear that after rounding, some rectangular areas inside the window will grow while others will not. In the report about the OpenCV implementation [16] it was mentioned that there was a big improvement in precision after compensating for the different area ratio caused by rounding errors.

While this is an implementation specific detail, it leaves the question open how bad the remaining quantization effects are. This is important when detecting objects at low resolution.

To find this out, the image in Figure 6.6 was scaled up or down using cubic interpolation and an equally scaled feature was evaluated at fixed locations at the different scales.

Figure 6.7 shows the result for the first intensity-based feature selected by Adaboost for text detection. This feature computes the absolute intensity differ-



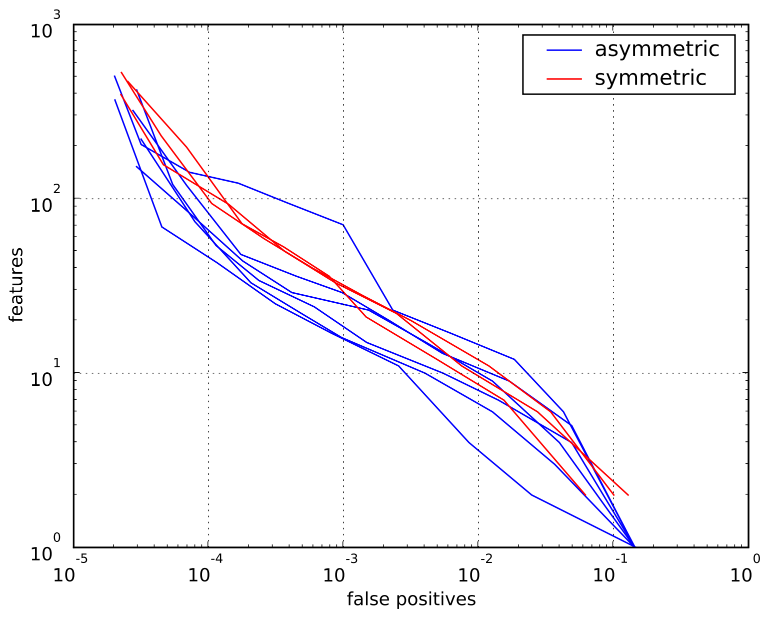


Figure 6.4: ROC and complexity curves of the final text detector with and without asymmetric bossting. The training noise is high but the difference is still clearly visible.

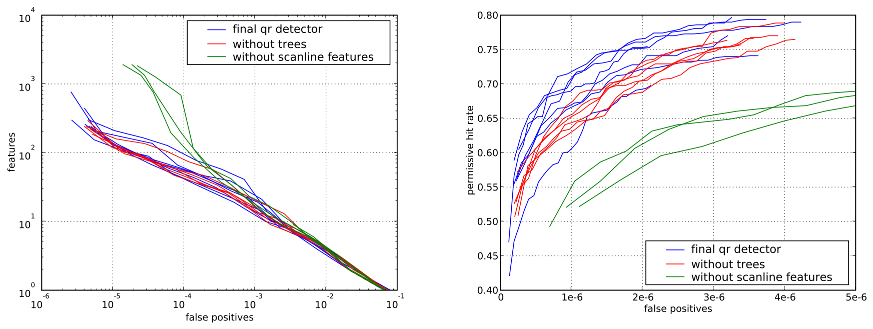


Figure 6.5: ROC and complexity curves of the final QR code detector (last stage only) when boosting trees and when boosting stumps, and also without the scanline based features.

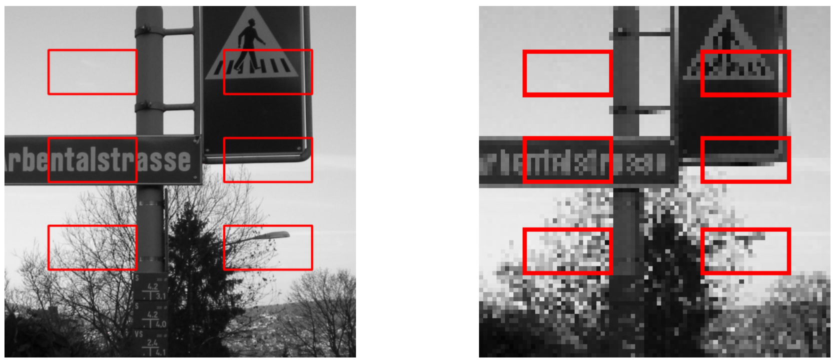


Figure 6.6: The image and feature locations choosen for the scale test. At the smallest resolution, the window has its original size of 20x10 pixels.

ence between a centered 20x6 block and the whole 20x10 window. This feature averages relatively large regions, but Adaboost already has to cope with a lot of quantization noise. Still the value inside the text window stays clearly above the others. Smaller blocks are choosen only at the later stage and can have much more noise.

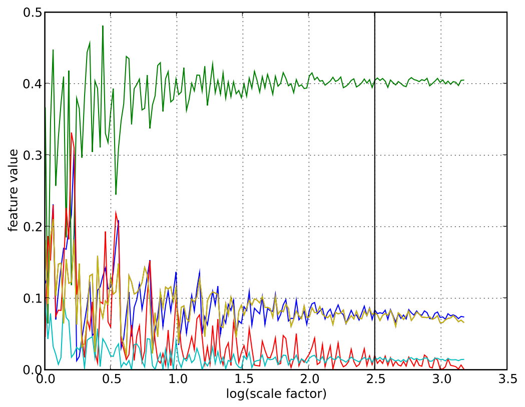


Figure 6.7: The responses of an intensity-based feature at different window scales. Each curve corresponds to a window in Figure 6.6. The image was scaled to match the window size. The leftmost values are at the original window size (scale factor 1.0) and the vertical bar marks the original image resolution.

6.6 Scaling the Edge based Features

Using the same test as in the previous section, it can be judged how scale-invariant the edge-based features really are. As expected the scale invariance is best when only the ratio between the number of edges is compared as in Figure 6.8.

When the number of edges are really "counted", as in the feature shown in Figure 6.9, there is a clear depencency on the scale. This is also expected since at a low scale few details are visible and because the number of visible edges is smaller. Because of noise within the image (and contrast normalization amplifies it) the number of edges increases with the scale even on blank regions of the image.

In fact it may be suspected that this feature type mostly just measures the scale of the detection window. There were few text regions in the training set at the lowest detector scale, and of course the big bulk of the training background windows gets sampled from the smallest scale.

To check this an additional feature that measures nothing but the detection window scale was added into the feature pool. It was never selected by Adaboost. It can be concluded that this feature measures at least some useful property in addition to the scale.

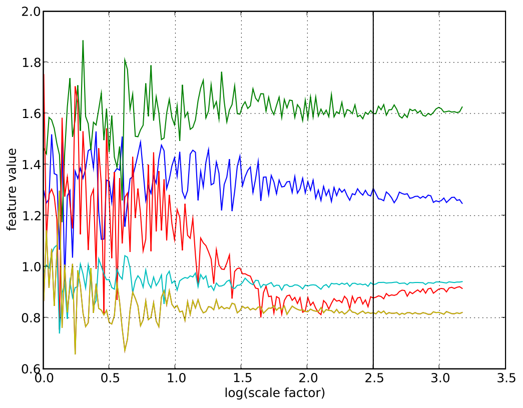


Figure 6.8: The scale responses of the first feature selected by Adaboost for text detection (with threshold 1.04). This feature compares the number of vertical edges inside a horizontal stripe with the same statistic inside the whole window (using equation 3.10).

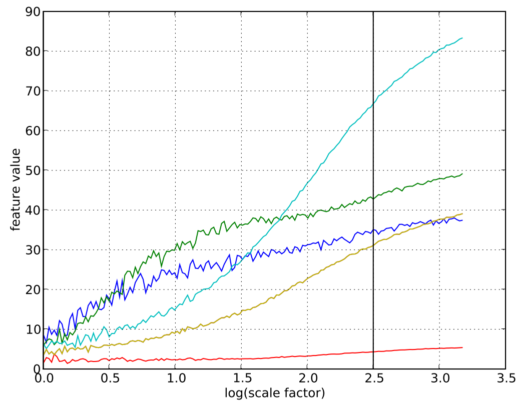


Figure 6.9: The scale responses of a feature "counting" the vertical edges inside a horizontal stripe (using equation 3.9). (The values still grow when the original image is scaled up because the smooting done by the Sobel operator still hides some edges.)

6.7 Different Dataset

In addition to the dataset described in section 3.1 a more challenging superset was tried. This included artistic fonts, some handwritten letters, extreme spacing and three dimensional advertisment signs. Additionally more background was labeled selectively, including only regions with false positives of the final cascade.

As it would be expected, the performance on this dataset is worse because the testing set was more challenging. Much more features were evaluated, also in the early stages. The cascade trained on the simpler dataset seemed to perform about equally well on difficult images.

Chapter 7

Conclusions and Outlook

A system for detecting and reading text in natural scenes has been implemented. Clear horizontal text can be detected and read reliably with very few false positives.

However often the text in natural scenes is not clean in some sense. It may be on uneven background, not evenly illuminated, blurred, perspectivically distorted or written with artistic fonts. Some of this text can still be detected, but most of it is not readable with a modern commercial OCR program. Cleaning up and reading difficult text regions is an active research topic, for example in [6].

The system developed during this thesis can be trained to find other "objects" appart from text. This was demonstrated by locating two dimensional barcodes by their bit pattern. Clear QR and Datamatrix codes (without much blur) can be found reliably and their bit size is estimated at the same time.

7.1 Further Work

The text detection results could be exploited in more sophisticated ways and used as prior knowledge for binarization or OCR. In [24] an expectation maximization algorithm was used for clustering.

Going one step further, the shape of the text line could be estimated and the text isolated from the background using only the detection results. For example there are sometimes bars near the extracted text that are much darker or brighter than both the text and its background, eg. a sign against the bright sky. Those bars are rarely part of a detection window, but if the text is slightly tilted they are included in the result. The extreme intensities confuse both the adaptive binarization and the OCR engine.

More sophisticated binarization methods that can adapt to different background patterns should be tested. Also, color information could be used; while the background pattern can have all possible color, the color of the font is often constant.

Bibliography

[1] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001.

[2] X. Chen and A. L. Yuille Detecting and Reading Text in Natural Scenes. In *CVPR*, 2004.

[3] Simon M. Lucas. ICDAR 2005 text locating competition results. *Proceedings of International Conference on Document Analysis and Recognition (ICDAR)*, 2005.

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

[5] Ching-Tung Wu. Embedded-Text Detection and Its Application to Anti-Spam Filtering. *Master Thesis at the University of California, Santa Barbara*, 2005.

[6] Céline Mancas-Thillou, Bernard Gosselin. Natural Scene Text Understanding. *Vision Systems: Segmentation and Pattern Recognition*, ISBN 978-3-902613-05-9, pp, 307-333, 2007.

[7] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *European Conference on Computational Learning Theory*, pp. 23-37, 1995.

[8] Robert E. Schapire and Yoav Freund and Peter Bartlett and Wee Sun Lee. Boosting the margin: a new explanation for the effectiveness of voting methods. *Proc. 14th International Conference on Machine Learning*, 1997.

[9] Viktor Peters. Effizientes Training ansichtsbasierter Gesichtsdetektoren. *Diplomarbeit im Fach Naturwissenschaftliche Informatik: Technische Fakultät Universität Bielefeld*, 2006.

[10] P. Viola and M. Jones. Fast and Robust Classifcation using Asymmetric AdaBoost and a Detector Cascade. In *Proceedings NIPS01*, 2001.

[11] Hamed Masnadi-Shirazi, Nuno Vasconcelos. Asymmetric boosting. *ICML* 2007, 609-619.

[12] R. E. Schapire and Y. Singer. Improved Boosting Algorithms Using Confidence-rated Predictions. *Machine Learning, 37*, 1999, 297-336.

[13] Yoav Freund. An Adaptive Version of the Boost By Majority Algorithm *Proceedings of the Workshop on Computational Learning Theory*, 1999.

[14] B. Wu, H. Ai, C. Huang, and S. Lao. Fast Rotation Invariant Multi-View Face Detection Based on Real AdaBoost. *Proceedings Sixth International Conference on Automatic Face and Gesture Recognition*, pp. 79-84, 2004.

[15] J. Friedman, T. Hastie, R. Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics 28*, 337–307, 2000.

[16] Rainer Lienhart, Alexander Kuranov, Vadim Pisarevsky. Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection. *Intel Technical Report MRL-TR-July02-01*, 2002.

[17] Chong Yee Seng. Classification and Regression Trees (CART) for Spam Prediction. *Computational Intelligence: Methods and Applications – Assignment 2*, 2006.

[18] S. Charles Brubaker, Jianxin Wu, Jie Sun, Matthew D. Mullin, James M. Rehg. On the Design of Cascades of Boosted Ensembles for Face Detection. *International Journal of Computer Vision*, Special Issue on Learning for Vision, 77(1-3), 2008, pp.65-86.

[19] Trevor Hastie, Robert Tibshirani, Jerome Friedman. The Elements of Statistical Learning. *Springer Series in Statistics*, ISBN 0-387-95284-5, 2001.

[20] Shijian Lu, Chew Lim Tan. Binarization of Badly Illuminated Document Images through Shading Estimation and Compensation. *Document Analysis and Recognition*, ICDAR 2007, pp 312-316.

[21] Faisal Shafaita, Daniel Keysersa, Thomas M. Breuelb. Efficient Implementation of Local Adaptive Thresholding Techniques Using Integral Images. *Document Recognition and Retrieval XV, San Jose.*

[22] Eisaku Ohbuchi, Hiroshi Hanaizumi, Lim Ah Hock. Barcode readers using the camera device in mobile phones *Proceedings of the 2004 International Conference on Cyberworlds*, pp. 260-265.

[23] Kobi Levi and Yair Weiss. Learning Object Detection from a Small Number of Examples: the Importance of Good Features. In *CVPR*, 2004.

[24] Vincent Vanhoucke, S. Burak Gokturk. Reading text in consumer digital photographs. *Document Recognition and Retrieval XIV*, 2007.

Appendix A

Using the Codebase

The classifier is written in C++ and supported by various Python scripts. All the code is in the `classifier` directory. Table A.1 lists the three main scripts. There are a few more specialized scripts. Most of them have a built-in help function that describes their purpose and usage when started without arguments.

A.1 Creating a Dataset

To generate a dataset, background and foreground labels in the `.idl` format have to be given to the `create_dataset.py` script. This file format that was used because it is the output format of the `idledit` tool developed at the institute by Bastian Leibe. Multiple `.idl` files can be combined using `idlmix.py`. The generated dataset directory contains a cropped version of all required images and the new labels (eg. all text detection rectangles have a fixed aspect ratio).

Table A.1: The three central scripts.

Script	Function
`create_dataset.py`	splits the labeled text or code regions into detection windows and randomly distributes the data into training, validation and testing sets; the result is a self-contained *dataset directory*
`traincascade.py`	trains a classifier for the given dataset; for each stage, the detector binary and the evaluation results are saved into the *training directory*
`runcascade.py`	runs the trained classifier on a list of images, invoking external programs for binarization, OCR or QR decoding if requested

A.2 Training the Cascade

The training process will need about 2GB of memory (mainly depending on the number of positive windows). It has to be started within a Subversion sandbox. When running several trainings in parallel each training needs its own sandbox. The Subversion revision and any local modifications will be saved into the training directory.

The configuration file `parameter.py` can be edited to change most settings for the training and the evaluation; the parameters are documented within this file. The training process itself is started with the `traincascade.py` script. It is good practice to append a number to the name of the output directory; ROC curves will be plotted with the same color if only this number is different.

For each stage, a subdirectory is created in the training directory where all stage-specific results are stored, including a compiled `detect` binary as well as all the evaluation results (`evalcascade.py` gets invoked for each stage during the training).

After the training is finished, the ROC and complexity curves can be plotted and compared using either `plotrocs_gnuplot.py` or `plotrocs_pylab.py`.

A.3 Running the Detector

The `runcascade.py` script takes a training directory and a list of images as input. Without parameters, `runcascade.py` will produce a file called `output.idl` with the raw detection results. If OCR is requested the cropped and binarized images are also saved into the `output` directory and the file `output.idl` will contain the combined results instead.

When running the detector a suitable operating point on the ROC curve should be choosen. This is done with the `--thresh=value` option. The value corresponding to each (false positive, hitrate) combination can be found in the last column of the file `traindir/laststage/roc_test.dat`. To get a higher hitrate it is also possible to use an earlier stage of the cascade; just give the stage subdirectory instead of the training directory.

For the final OCR or QR decoding step external programs are called by `detect.py` on the cropped images in the `output` directory. For QR decoding the path to *qrcode*[1] may have to be adapted in the file `decode.py`. For OCR with *gocr* or *tesseract* binarization with the `-b` switch is recommended, which needs the *gamera*[2] framework installed. The results are saved into text files for each image in the `output` directory.

Sometimes it may be neccessary to recompile the `detect` binary after the training, eg. for a different CPU architecture or with code changes that do not require a new training. This can be done with the following command:

```
scons stagedir=traindir/stage09 traindir/stage09/detect
```

substituting the number of the last stage. For predictable results the source code and the relevant settings in `parameters.py` should match those used during training.

[1]http://qrcode.sourceforge.jp/

[2]http://ldp.library.jhu.edu/vhost-base/gamera (svn trunk r1043 was used)

A.4 Datasets

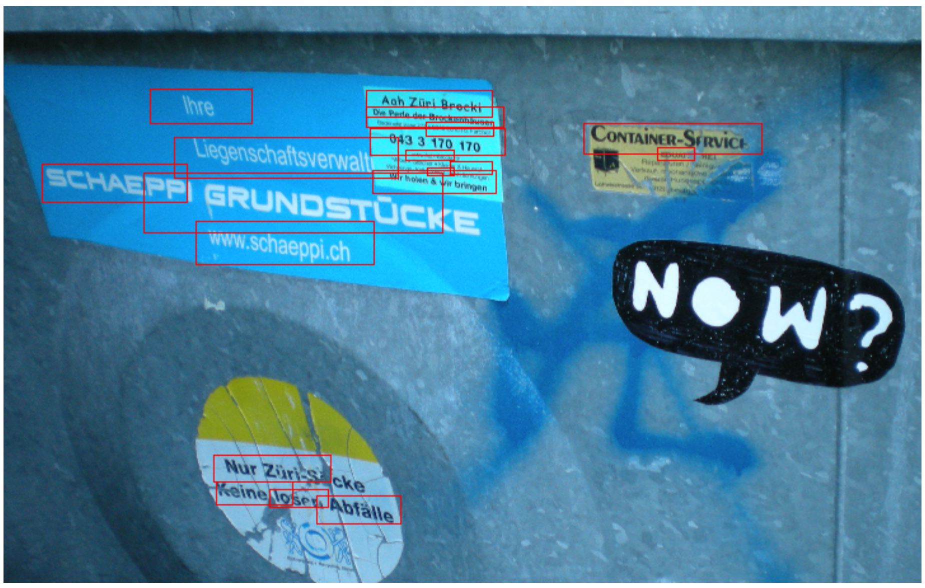
The dataset diretory for the final text training is in `data/datasets/flickr2` and the one for the QR and datamatrix training in `data/datasets/qrdm`. While they are self-contained, the images inside were automatically cropped. The original labels can be found `fg_orig.idl` and `bg_orig.idl` inside each dataset directory. Those files contain references to the original images in `data/images`, `data/flickr`, `data/codes` and `data/codetextbg`.

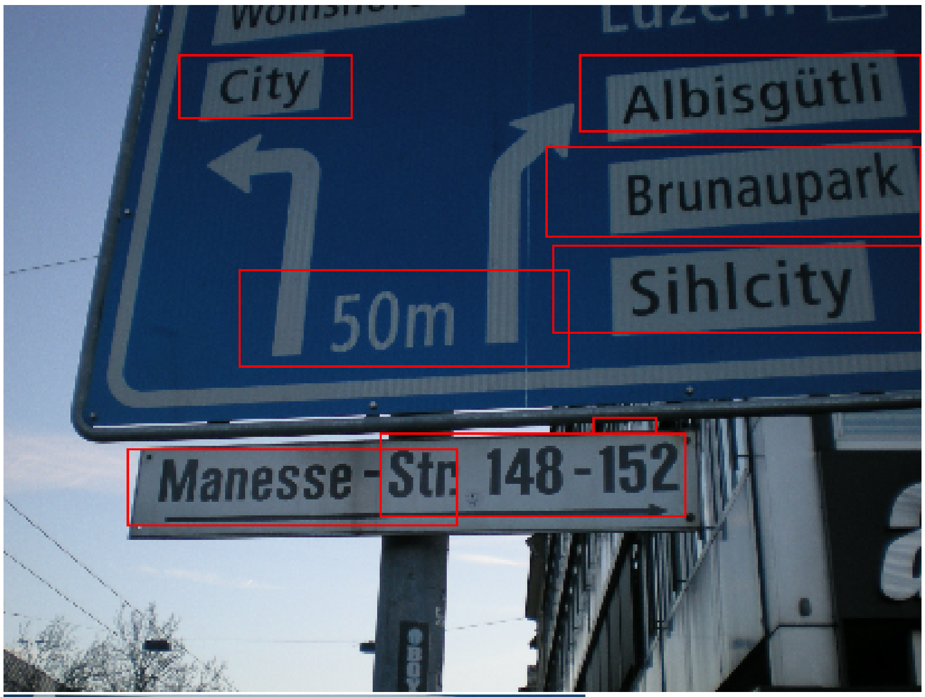
The trained text detector is in `classifier/ffinal1`. To recompile the detector binary in `ffinal1/stage09` the `classifier` directory must be reverted to the svn revision number in the file `classifier/ffinal1/info.txt` (because of a last-minute bugfix in `classifier/feature_sobel.hpp` which turned out to have no influence on performance). The trained QR code detector is in `classifier/qrdm_final9`.

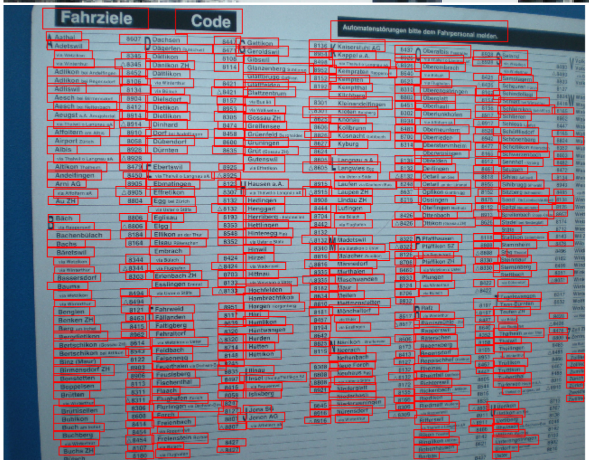
Appendix B

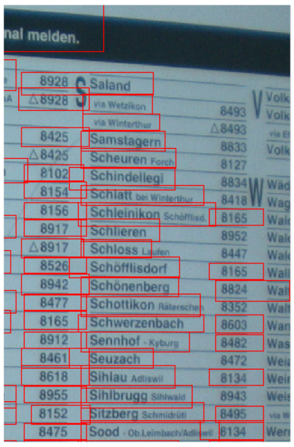
More Text Detection Results

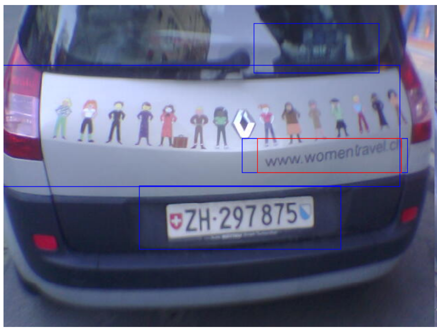
The red labels shown below are all from the same detector as in Section 3.6 running at 5e-6 false positives per window. The blue labels (if given) are drawn behind the red ones and come from the same detector at 45e-6 false positives.

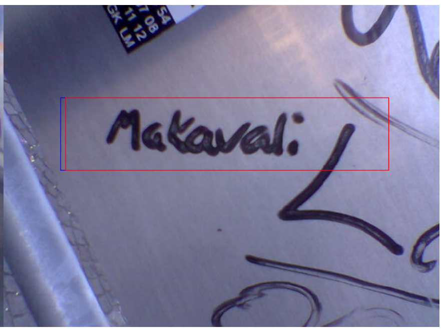


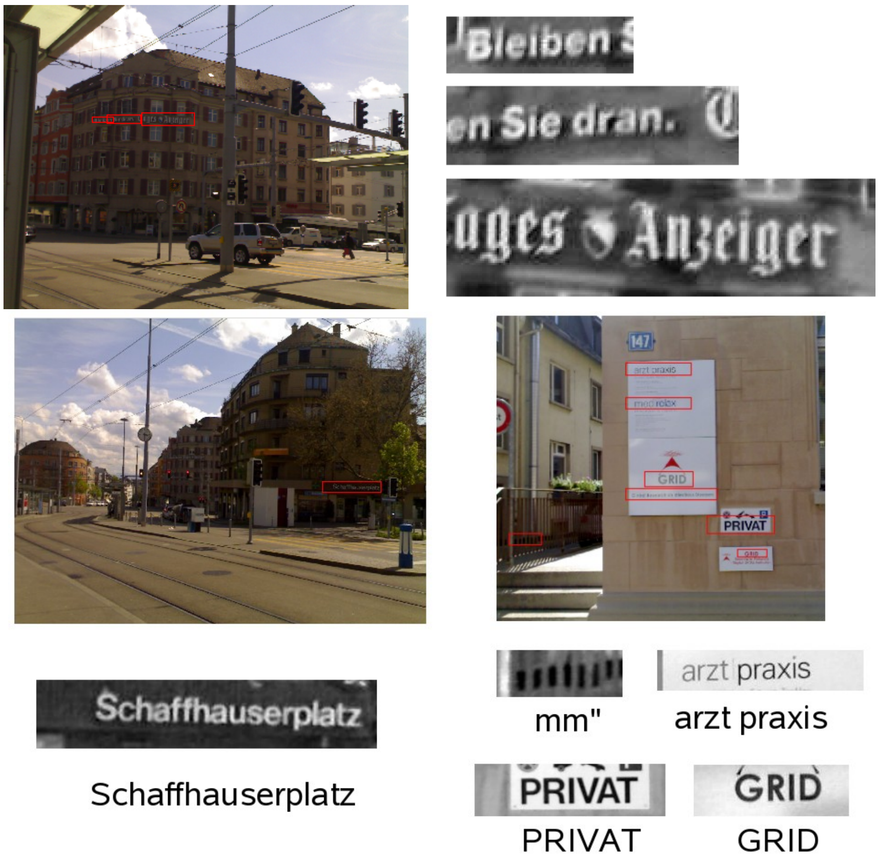

City
Albisgütli
Brunaupark
Sihlcity
50m
Manesse-Str. 148-152


Fahrziele
Code
Automatenstörungen bitte dem Fahrpersonal melden.


nal melden.
8928 Saland
8928 via Wetzikon
via Winterthur
8425 Samstagern
8425 Scheuren Forch
8102 Schindellegi
8154 Schlatt bei Winterthur
8156 Schleinikon Schöfflisd.
8917 Schlieren
8917 Schloss Laufen
8526 Schöfflisdorf
8942 Schönenberg
8477 Schottikon Rätersch.
8165 Schwerzenbach
8912 Sennhof - Kyburg
8461 Seuzach
8618 Sihlau Adliswil
8955 Sihlbrugg Sihlwald
8152 Sitzberg Schmidrüti
8475 Sood - Ob.Leimbach/Adliswil

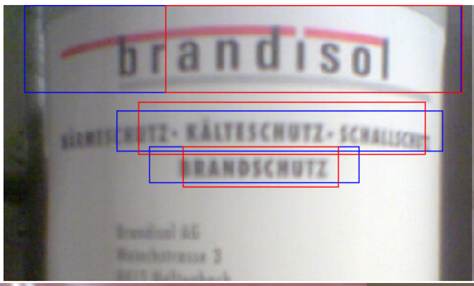

www.womentravel.c
ZH·297875

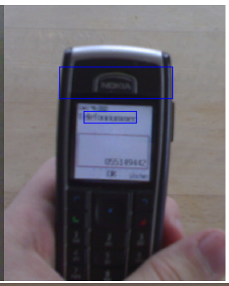

Makavali

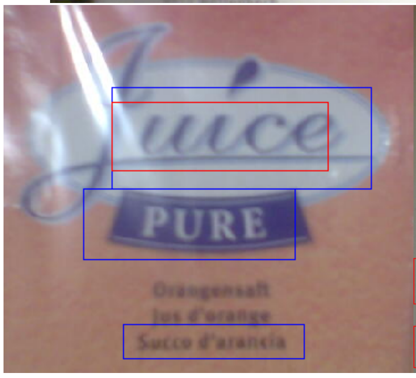

Bleiben
en Sie dran.
uges Anzeiger
Schaffhauserplatz
Schaffhauserplatz
arzt praxis
mm"
arzt praxis
PRIVAT
GRID
PRIVAT
GRID

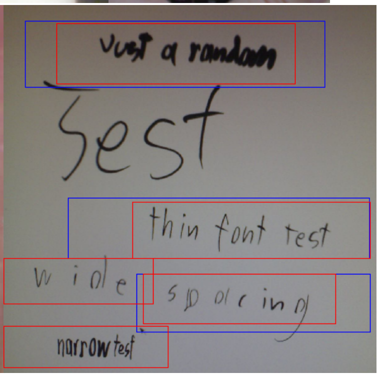
Gebäudereinigungen
...auf Dauer das Sauberste
Baden, Dietikon,
Uster, Zurich
043 322 90 50
RIEN NE VA PLUS
RIEN NE VA PLUS

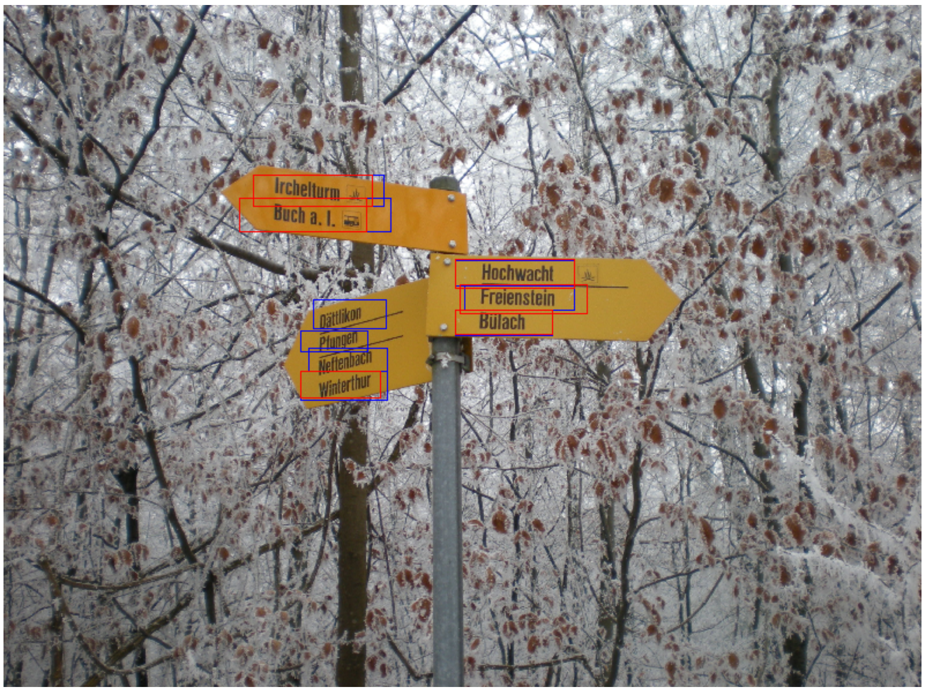
Frauenklinik-Str.
Schmelzberg-Str.
Schützen-Str.


brandisol
WÄRMESCHUTZ · KÄLTESCHUTZ · SCHALLSCHUTZ
BRANDSCHUTZ
Brandisol AG


NOKIA


Juice
PURE
Orangensaft
Jus d'orange
Succo d'arancia


just a random
test
thin font test
w i d e
s p a c i n g
narrow test


Irchelturm
Buch a. I.
Hochwacht
Freienstein
Bülach
Dättlikon
Pfungen
Neftenbach
Winterthur