Master's Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Text Recognition of Different Scripts in the Wild

Bc. Oskar Hollmann

May 2015

Supervisor: prof. Ing. Jiří Matas Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Oskar Hollmann**

Study programme: Open Informatics
Specialisation: Software Engineering

Title of Diploma Thesis: **Text Recognition of Different Scripts in the Wild**

Guidelines:

1. Explore the state-of-the-art in script classification and text recognition for non-latin scripts in the context of the text-in-the wild problem.
2. Create a dataset of photos with text in different scripts (e.g. Cyrillic, Greek, Hebrew).
3. Analyse the dependence of performance of TextSpotter modules [5] on the script.
4. Address the issues found in the previous stage in order to make TextSpot more script-agnostic.
5. Propose and evaluate a method for classification of the script and for non-latin text recognition.

Bibliography/Sources:

1. Pal, U. and Dash, N.: Language, Script, and Font Recognition, Springer-Verlag, 2014
2. Bar-Yosef, Itay et al.: Binarization, character extraction, and writer identification of historical Hebrew calligraphy documents, Springer-Verlag, 2007
3. Belaid, A. and Razzak, M.: Middle Eastern Character Recognition, Springer-Verlag, 2014
4. Rabaev, Irina et al.: Case Study in Hebrew Character Searching, ICDAR, 2011
5. Neumann, L. and Matas, J.: On Combining Multiple Segmentations in Scene Text Recognition. ICDAR 2013: 523-527

Diploma Thesis Supervisor: prof.Ing. Jiří Matas, Ph.D.

Valid until the end of the summer semester of academic year 2015/2016

prof. Ing. Jiří Žára, CSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, March 25, 2015

Acknowledgement / Declaration

I would like to express my sincere thanks to prof. Jiří Matas for his valuable comments and inspiration, Michal Bušta for his patience and many insights into the internals of TextSpotter, and Petr Olšák for this excellent TEX template.

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague on May 11, 2015

....

Abstrakt / Abstract

Rozpoznávání textu v reálných scénách se s vysokou dostupností fotoaparátů a chytrých telefonů stalo zajímavou oblastí výzkumu. Na rozdíl od strojového čtení textu ve skenovaných dokumentech je zatím považováno za nevyřešený problém.

V této práci se soustředíme na problémy, které vznikají, když scéna obsahuje text v různých abecedách. Proto nejprve provádíme analýzu výkonnosti existujícího systému na doposud neznámé abecedě. Systém sice používá co nejobecnější metody, doposud byl ale testován jen na latince a okrajově na azbuce. Identifikujeme, které moduly systému jsou citlivé na výměnu abecedy, a hledáme řešení pro některé nalezené problémy. Poté navrhujeme dvě rozdílné metody na rozpoznávání abecedy, kterou je napsán text ve scéně. Obě metody dosahují výkonu srovnatelného s dostupnou literaturou. Nakonec popisujeme syntézu obou přístupů.

Pro účely vyhodnocování byla vytvořena původní datová sada s nápisy převážně v hebrejštině a latince.

Klíčová slova: detekce textu; rozpoznávání textu; text v reálných scénách; rozpoznávání abecedy; OCR; TextSpotter.

Text recognition in natural images became an interesting research area with the advent of affordable cameras and smartphones. Unlike the traditional character recognition in scanned documents, it is still considered an unsolved problem.

We focus on the problems that arise when different alphabets are present in the scene. Therefore, we first analyse the impact of recognizing a previously unknown script on an existing system. Although developed with generality in mind, it has been tested only on the Latin and Cyrillic scripts so far. We identify which modules are sensitive to changing the script and propose solution for some of the found problems. Second, we propose and implement two different methods for script recognition. Both of the methods show state-of-the-art performance. In the end, we provide a synthesis of these approaches.

An original dataset with mainly Hebrew and Latin inscriptions was collected and annotated for the evaluation purposes.

Keywords: text detection; text recognition; text-in-the-wild; text in natural images; script recognition; OCR; TextSpotter.

Contents

1 Introduction 1
1.1 Architecture of TextSpotter 2
2 State-of-the-Art 4
2.1 Text Recognition of Non-Latin Scripts 4
2.1.1 Middle-Eastern Text Recognition 4
2.1.2 CJK Text Recognition 6
2.2 Script Recognition 8
2.2.1 Recognition of Pre-defined Set of Scripts 9
2.2.2 Script-independent Methods for Script Recognition 10
3 Hebrew Dataset 12
3.1 Character of Images and Inscriptions 12
3.2 Annotation Procedure 14
4 TextSpotter Performance Analysis 16
4.1 Preparatory Tasks 16
4.1.1 Training of the Hebrew Character Classifier 16
4.1.2 Optimizations Prior to the Analysis 17
4.2 Analysis on Hebrew Dataset 18
4.2.1 Evaluation Tools 18
4.2.2 Problems in Hebrew Text Recognition 19
4.2.3 Comparison with the Synthetic Dataset 20
4.3 Analysis on Char74k Dataset 22
4.3.1 Comparison of TextSpotter Pipelines 24
5 Improvements for Hebrew Script 27
5.1 Two-component Letters 27
5.1.1 Removing the Smaller Component 27
5.1.2 Merging the Components 28
5.1.3 Comparison of Methods for Two-component Letters 29
5.2 Issues with the Letter Yod 31
5.2.1 Evaluation of the Yod Optimization 32
6 Script Recognition 35
6.1 Nearest Neighbour Script Recognition 35
6.1.1 Evaluation of FLANN-based Classifier 37
6.2 Bigram Script Recognition 41
6.2.1 Implementation Details of Bigram Classifier 42
6.2.2 Evaluation of Bigram Classifier 43
6.3 Combining Proposed Script Classifiers 44
7 Conclusion 46
7.1 Further Improvements 47
References 48
A Contents of the DVD 51
B Hebrew Dataset Thumbnails 52
C Hebrew Dataset Details 56

Tables / Figures

Tables

3.1. Word and character counts in dataset ... 12
4.1. Fonts for classifier training ... 17
4.2. Comparison with ICDAR dataset ... 18
4.3. FN rates before ... 19
4.4. FP rates before ... 20
4.5. Lost characters before ... 20
4.6. Comparison of synthetic and Wikimedia dataset ... 21
4.7. FP rates for synthetic ... 22
4.8. Fonts for Kannada training ... 23
4.9. Performance on Kannada script ... 24
5.1. He confusion tables ... 29
5.2. Qof confusion tables ... 29
5.3. Improvement in perfect words ... 29
5.4. FN rates after ... 30
5.5. FP rates after ... 30
5.6. Perfect words before and after the Yod optimization ... 33
5.7. FP rates before the Yod optimization ... 33
5.8. FP rates after the Yod optimization ... 34
6.2. Latin and Hebrew Success Rates ... 37
6.1. Fonts for Latin classifier training ... 38
6.3. Success rates of Ablavsky and Stevens method ... 38
6.4. Details about corpora ... 43
6.5. Latin and Hebrew success rates using bigrams ... 43
6.6. Success rates of combined classifier ... 44

Figures

1.1. TextSpotter Modules ... 2
2.1. Over-segmentation example ... 5
2.2. Word spotting features ... 6
2.3. RANSAC char recognition ... 8
2.4. Multi-script text recognition architecture ... 9
2.5. Projection profile example ... 10
2.6. Water reservoir example ... 10
3.1. Traffic signs in dataset ... 13
3.2. Tourist boards in dataset ... 13
3.3. Short signs in dataset ... 13
3.4. Challenging images in dataset ... 14
3.5. An annotated image ... 15
4.2. Synthetic dataset example ... 20
4.1. Confusion matrix before ... 21
4.3. Consonant pairs in Kannada ... 22
4.4. Kannada vowel as diacritics ... 23
4.5. Char74k dataset example ... 23
4.6. Examples of detected Kannada words ... 24
4.7. Comparison of text detection in both pipelines ... 25
4.8. Graph cut often misclassifies background ... 25
5.1. He convex defect ... 28
5.2. Thresholding effect on Mem ... 30
5.3. Confusion matrix after ... 31
5.4. Yod line ... 32
5.5. Confusion matrix after the Yod optimization ... 33
6.1. Example of a successful script recognition ... 37
6.3. Confusion matrix of Latin and Hebrew for FLANN ... 39
6.4. Latin lines classified as Hebrew ... 39
6.5. Hebrew lines classified as Latin ... 40
6.6. Histogram of text lines ... 40
6.7. Normed histogram of text lines ... 41
6.8. Confusion matrix for bigrams ... 43
6.9. Confusion matrix for combined classifier ... 45

Chapter 1
Introduction

Automatic detection and recognition of *text-in-the-wild*, often called *text in natural images*, is an active research area in the field of computer vision for more than a decade. The phenomenon of affordable digital cameras and smartphones enabled us to conveniently collect images of our surroundings thus creating a possibility for many interesting applications of automatic text recognition.

However, many problems arise when we move from the traditional Optical Character Recognition (OCR) systems that operate on scanned documents to text-in-the-wild. The images may contain text in arbitrary positions, scales or wider set of fonts and its skew and other distortions may be significant. Therefore, different methods must be employed. While OCR is largely considered a solved problem for the Latin alphabet, the problem of text-in-the-wild is still open and receives significant attention.

TextSpotter is one of the state-of-the-art end-to-end systems for text recognition in natural images that is being worked on. It is developed at Centre for Machine Perception at Czech Technical University in Prague. A number of papers have been published about its advances, e.g. [1], [2], or [3].

The aims of this thesis are twofold. First is to evaluate and potentially improve performance of TextSpotter on unfamiliar alphabets (scripts) as only Latin and Cyrillic scripts have been tested so far. One of the goals of TextSpotter is to develop a recognition method independent of script and font of the text. Therefore, such an analysis is important for the future direction of development. The analysis should point out which modules of TextSpotter are sensitive to changing the script.

Second aim is to implement a script recognition method in TextSpotter. At present, TextSpotter is able to recognize only one script set by the user before runtime. This is acceptable for some applications but there are many situations where more than one script appear within the image. For instance, more than one script regularly appear next to each other in the streets of countries such as India, Israel, or Ukraine. A method able to recognize a previously defined set of scripts should be developed and evaluated. The classification should be done on text line or word level to be useful in situations when more scripts appear in one image.

For evaluation of both these goals, a dataset containing inscriptions in an exotic script should be collected and annotated. Signs in another script should be present to evaluate the script recognition method.

The rest of this thesis is organized as follows. Chapter 2 provides a light introduction to the topic and state-of-the-art in two areas—non-Latin character recognition and script recognition.

Information about the collected dataset can be found in Chapter 3 including details about the nature of the images, alphabets of the inscriptions, or the annotation process.

In Chapter 4, the prerequisites to the evaluation stage are described first, then the performance of TextSpotter on the Hebrew dataset is measured and compared to a similar Latin dataset and to a synthetically created ideal Hebrew dataset. Section 4.2.2 offers analysis of the found problems. At the end of the chapter, the two pipelines that

coexist in TextSpotter at the moment are compared on an even more exotic Kannada script in Section 4.3.

We can find solutions to chosen problems connected with the Hebrew script in Chapter 5. Each solution is evaluated and the performance is compared with the best previous result.

Chapter 6 explains two different methods proposed to solve the script recognition problem and evaluates them. We have experimented with a combination of these two approaches and we compare the performance.

Finally, Chapter 7 summarizes the problems we discovered during the analysis and discusses the results we had with different methods of script recognition. Possible direction of future work is offered.

1.1 Architecture of TextSpotter

Every aspect of this thesis—be it performance analysis, functional extensions, or dataset annotation—is conducted with respect to one specific text recognition software and that is TextSpotter. Therefore, we need to discuss its architecture and features first. The purpose of this section is to give a quick overview of TextSpotter modules. For a rigorous definition of the problem and the theoretical foundations, please refer to the original paper by Lukáš Neumann and Jiří Matas [1].

Text detection and recognition in TextSpotter can be divided into several subproblems. Although multiple solutions may have been implemented to solve each subproblem, we shall describe the setup that was used throughout this thesis. Please, examine the schema in Figure 1.1 that illustrates the different stages of TextSpotter.

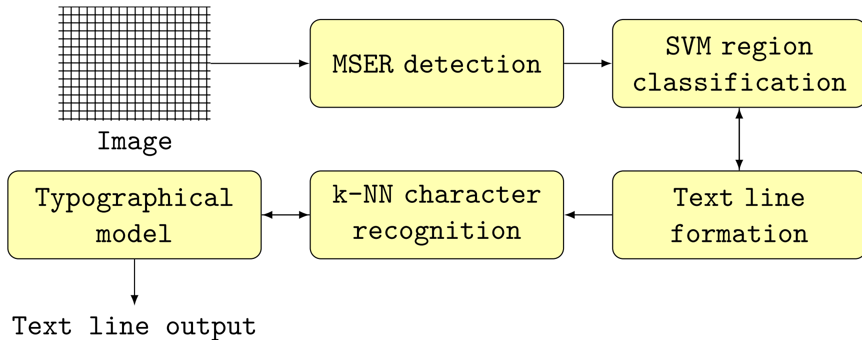


Figure 1.1. Text recognition in TextSpotter has several stages. The schema is based on the original one from [1] but adapted to describe our specific setup. Some aspects that we do not touch in this thesis were left out.

First, Maximally Stable Extrema Regions (MSER) are detected in the image. Ideally, every character would correspond to some extrema region, however, a small amount of characters might be lost even during this first stage.

Since the MSER detection is set to lose as little characters as possible, large amount of extrema regions corresponds to the background. Therefore, a Support Vector Machine (SVM) classifier is used to classify regions as *character regions* or *non-character regions* (background).

The character regions from SVM are then grouped into *text line hypothesis* which filters out large amount of background regions that were incorrectly classified as characters because these background regions appear quite randomly and only rarely form

a text line. Thanks to the feedback loop, character regions that were incorrectly classified as background may be introduced again if they seem to be part of some text line hypothesis. Formation of the line hypothesis is based on the assumption that font, colour, or size of characters rarely change within one line of text.

The next stage is character recognition. An approximate *k-Nearest Neighbour* (k-NN) search (see introduction of Chapter 6.1 for more details) results into several character hypothesis. The k-NN classifier is trained using only synthetic fonts which is one of the biggest advantages of TextSpotter. There is no need to collect a training dataset for the characters. Each vector in the k-NN space corresponds to one image of a character drawn using one of the fonts. It is obvious that the k-NN search can return neighbours corresponding to different letters of the alphabet. Therefore, the classification is often ambiguous.

The typographical model is constructed at this point. The top, middle and bottom line of the text line are extrapolated from the characters (see Figure 5.4 for an illustration) and the characters inconsistent with the model are removed.

We still have several possible classifications for each region on the line. To find the optimal classification for each region, an adjacency graph of the character hypothesis is formed. Every hypothesis has a score comprising of multiple factors (confidence of the classification, deviation from the expected position on the line, and more). Finding the optimal classification for all the characters equals to solving the longest path problem in the adjacency graph by dynamic programming.

At this point, we have extracted and read text lines from the input image. There are other aspects such as segmentation into words, segmentation of connected characters, or employing the language model which we do not cover here.

Chapter 2
State-of-the-Art

The objective of this chapter is to provide an overview of the state-of-the-art in two different areas. First is *text recognition of non-latin scripts* and the second is *script recognition* which is a sub-problem of text recognition and therefore presented as second. In both cases, the overview is done in the context of the *text-in-the-wild* problem.

2.1 Text Recognition of Non-Latin Scripts

Text recognition in the wild has attracted a lot of attention in the past years due to its many applications. Examples of applications are mentioned in [4], including *multimedia retrieval* where reading the text in the images or video can be used to retrieve the relevant multimedia or *automatic translation* of the text in images which has became very interesting with the wide availability of smartphones, lowering language barriers. Just as important are *text-to-speech systems* allowing the visually impaired to navigate in unfamiliar surroundings by being able to read the signs around them. *Industrial automation* is another important application, consider for instance the usefulness of automatic sorting of envelopes based on address in a post office.

The number of interesting applications brings the problem to the attention of researchers in most countries and systems designed specifically to recognize scripts different from the Latin script exist. Before we present the state-of-the-art for various non-latin scripts, let us briefly discuss the challenges of text detection and recognition in natural images.

Compared to the traditional OCR, the scene complexity makes the mere detection of the text non-trivial. The text can be present anywhere in the scene at arbitrary scale or not present at all. When scanning documents, we can guarantee certain resolution of the text and only minimal skew, whereas in natural images we cannot. We do not describe the typical stages of a text-in-the-wild detection and recognition as we have already done so for the specific case of TextSpoter in Section 1.1.

2.1.1 Middle-Eastern Text Recognition

The scripts used in the Middle-East have common origin in the Phoenician alphabet and have a structure different from the Latin script (see [5] for more details about the history and properties of these scripts). Most of the scripts (with the exception of Hebrew) are cursive—meaning the characters of one word are connected. Therefore, segmentation of the words into characters becomes a crucial problem for Arabic and Syriac writing systems. Complex diacritic marks both above and below letters are another common trait. The number of character classes is higher for some of the scripts than in the Latin script due to the letters having different shape depending on their position in the word (start, end, middle and isolated). In the following text, we provide an overview of the methods used to recognize Middle-Eastern scripts with the focus on how they tackle the mentioned characteristics.

An end-to-end system for translation of Arabic text in natural scenes was developed in [6]. The candidates for text regions are extracted at different resolutions of the input image by a set of heuristics to increase speed of the system—these are classified as text and non-text using SVM classifier and *Gradient Boosting Tree* with feature vector consisting of highest values of colour channels (based on the assumption that colour distributions are significantly different in the two classes). A commercial OCR is run on the detected text areas. However, due to the inferior performance of the OCR in the non-optimal conditions of natural images, *a noisy channel model* is applied on the OCR output to achieve higher performance.

Another end-to-end system was proposed in [7] for recognition of Arabic text from TV video sequences for the purpose of extracting metadata. The *Rolling Ball Transform*—a special case of the top-hat morphological transformation—is used to highlight thin and continuous objects in the scene and the structural properties of the Arabic script are used to extract the text. The diacritics is removed and *vertical projection profile* is used for character segmentation since all Arabic letters are connected into words at the baseline (see example of horizontal projection profile in Figure 2.5). The diacritics is then re-introduced and used to correct the mistakes in segmentation—refer to Figure 2.1 for an example. The extracted features include various *projection profiles*, number of *transitions* between black and white in columns and rows and *diacritics marks* which are necessary to discriminate between characters that differ only in the marks. Classification is performed by a k-NN classifier.

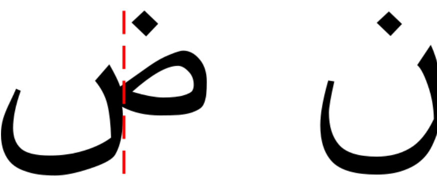


Figure 2.1. Example of an over-segmentation of the letter Dad (left) marked in red dashed line. Although the left part of the segmentation resembles the letter Nun (right), it is missing the diacritical mark, therefore, the segmentation is incorrect and can be removed.

Although *the vertical profile* is used for character segmentation in the previous paper and many others (see [8] for more examples), it suffers from under/over-segmentation problem (an example of over-segmentation is shown in Figure 2.1). So-called *holistic* or *global* approaches attempt to solve the recognition problem for cursive scripts without the need for segmentation. The disadvantage is that such system only recognizes the words from the training set which is not an issue for some applications (e.g. traffic signs).

A recent example of this effort is the method based on *Hidden Markov Models* (HMM) in [9]. The problem of word detection is not covered, the system assumes the words have already been extracted. Each word is divided into $N \times M$ sub-blocks on which *Discrete Cosine Transform* (DCT) is applied and its coefficients with lower frequencies (containing most of the image energy of the sub-block) are used as a feature vector. The feature vector of the word is simply a concatenation of the individual sub-block vectors. The usage of DCT coefficients as features makes the method script-independent and it could be used for other scripts. A HMM classifier was trained using a synthetic dataset of Arabic words in different fonts and it shows promising results on the testing dataset with different fonts and font sizes.

In the area of *text detection*, a novel approach designed with cursive Arabic script in mind has been proposed in [10]. An image operator called *Stroke Width Transform*

was introduced as being able to detect text regardless of scale, skew, colour, font, or language. In natural images, there are many objects with properties similar to text but one feature that sets the text apart is the nearly constant stroke width. First, edges are extracted using the *Canny edge detector*. Second, it is determined which edges belong to text strokes. If a pixel p belongs to the edge of a stroke, its *gradient direction* is perpendicular to the stroke and there is a pixel q on the opposite edge of the stroke with roughly opposite gradient direction. If subsequent pairs of pixels p, q have similar distances, the edges belong to a text stroke. The detected characters (or words in the case of the cursive Arabic script) are grouped together into text lines thus filtering out standalone characters which are likely to be false positives.

In case of the mentioned multimedia retrieval systems, it is not always necessary to recognize every single character but only spot a specific word image. A word-spotting search system proposed in [11] enables to find relevant documents in large volumes of historical Hebrew documents which share many properties with images from natural scenes (such as skew, complicated background or degraded characters). Among the tested features, distance between upper and lower boundary of the letter (see DULP in Figure 2.2) and vertical projection profile proved to be the most effective. Sliding window over text lines is used to extract features of words and Dynamic Time Warping algorithm (DWT) matches it to the feature vector of the query image. Since the recognition is done using simple features, the method is more resistant to degraded characters than traditional OCR.

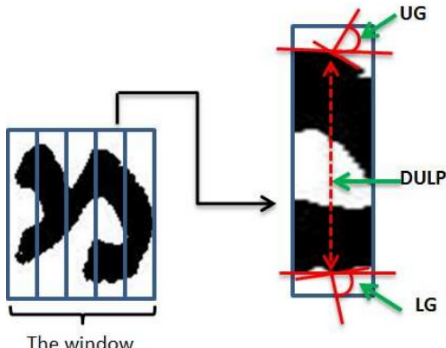


Figure 2.2. Features used for word spotting in [11]. Image taken from [11].

2.1.2 CJK Text Recognition

The scripts developed from the the ancient Chinese Han ideographs are denoted as **CJK**. The abbreviation stands for the most prominent languages using these scripts—Chinese, Japanese, and Korean. Since the scripts are based on ideographs, a common trait is the large number of characters. Although there are tens of thousands of characters, 4000 cover 99% of the characters used in practice. Still, the large number of character classes is the most difficult issue with CJK scripts [12].

Since CJK characters are not connected, segmentation is not a difficult problem compared to the Arabic script and most of the algorithms designed for the Latin script are applicable. However, the recognition phase is fundamentally different from Latin since a CJK character is composed of several non-connected components. As for the classification sub-problem, the same variety of tools is used just like for the Latin script recognition including *Nearest Neighbour*, *Neural Networks*, or *Fisher's Linear Discriminant*. The recent examples of these different approaches follow and the focus is on how the problem of large number of classes is tackled.

An end-to-end system for recognition of Chinese signs in natural images and their translation to English is proposed in [13]. First, *Laplacian of Gaussian* is used to detect edges at different scales. Second, the patches with edges are assigned features (size, intensity, mean, or variance) which are used to filter out unsuitable edges. The edges with similar properties are recursively merged into characters using the fact that all Chinese characters have almost the same aspect ratio and that characters in one context (on one sign) have similar properties. The background is segmented for each character individually using the most promising component of RGB and HSI colour spaces. After affine rectification, the character is divided into 7×7 sub-blocks and *Gabor features* are extracted. *Fisher Linear Discriminant* (FLD) reduces the problem into one dimension and finally NN finds the nearest match in the set of 3755 level 1 Chinese characters. The *vector angle distance* showed superior performance to the Euclidean distance.

A more rigorous approach for detection of Japanese Kanja characters was designed in [14]. The connected components extracted using *Niblack thresholding* are classified as character components or background. First, by an initial classifier with a simple set of rules to reduce number of components and then by a strong classifier constructed using *AdaBoost*. The features of connected components used in the strong classifier include compactness, eccentricity, standard deviation of stroke width, and more. AbaBoost creates different weak classifiers using FLD by decreasing the weight of the already correctly classified samples. The character components are clustered using *Markov random field* into text area candidates (words, lines, or individual characters), which also filters-out some of the remaining noise. The text areas are fed to the OCR module and the output is subjected to post-processing, e.g. areas with many suspicious characters such as "#" or "$" are removed.

A *text detection system* designed and tested on both Chinese and Latin script is proposed in [15]. It focuses on text with arbitrary orientation. The method shows state-of-the-art performance on nearly horizontal text and achieves superior performance in case of arbitrary orientations. SWT (explained in Section 2.1.1) is used to extract connected components. A trained classifier filters out non-character components leaving only character candidates (using contour shape, edge shape, background/foreground ratio, and other features). These are recursively linked into chains based on similar structural properties and another classifier discards chains with low scores (features include candidate count, average probability across candidates, average turning angle from candidate to candidate, and more).

A very different approach from all the methods reviewed so far is [16]. SIFT features and a version of RANSAC generalized to the problem of matching multiple objects is employed. Each character of the particular script has one reference image forming the reference set. During the learning phase, SIFT features are extracted from the reference images and stored in a database. In the classification stage, following operation (see Figure 2.3 for a simplified example) is performed for each $R \in referenceSet$ on the query image Q:

- Take a SIFT feature $f \in Q$ such that also $f \in R$.
- Select $(2 + \epsilon)$ features closest to f.
- Run standard RANSAC on this subset of features.
- If the number of votes is larger than a threshold, project the character R onto Q.

When this procedure is done, there are multiple overlapping (conflicting) characters projected to the image Q. These conflicts are resolved by always removing the character with fewer votes until there are no overlapping characters. The method was tested on

3 widely used Japanese scripts—*Kanji* script derived from Chinese Han characters and modern syllabic alphabets Hiragana and Katakana. The experiments suggest it is suitable for scripts with very complicated characters (Kanji) because more features are extracted from such characters and RANSAC matching then shows better performance. The structurally much simpler Hiragana and Katakana performed worse.

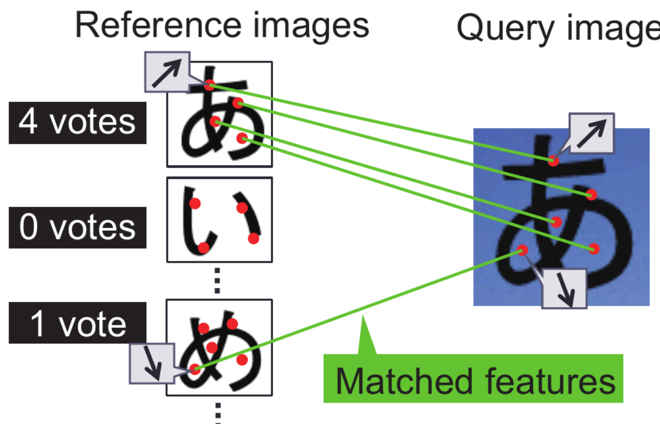


Figure 2.3. A simplified example of the recognition procedure with only one character in the query image. Image taken from [16].

2.2 Script Recognition

Script recognition is a prerequisite for creating an text recognition system that is able to cope with the text in different scripts. As [17] or [18] explain, there are two basic strategies to build such a system:

1. Develop a generalized system that can recognize all the characters in the required scripts.
2. Perform script recognition first and then use a specialized text recognition for the given script (see Figure 2.4).

The first option is considered to be more difficult due to the growing number of character classes. Moreover, the generalized system cannot take advantage of the unique properties of the script which is often used to achieve higher recognition performance. The differences between some scripts are so fundamental that they require different methods for successful recognition. E.g. in Arabic scripts, segmentation of words into letters is necessary prior to the recognition process (e.g. [5] describes many segmentation techniques) whereas in the printed Latin script the segmentation is easy.

Script recognition attracts most interest in countries and regions where multiple scripts commonly occur next to each other such as India (Latin, Devanagari, Bangla, etc.), the Middle East (Hebrew and the variety of different writing styles of the Syriac and Arabic systems) and the Eastern European countries (mainly Latin and Cyrillic scripts). The interest in recognition of the script in the Western countries—where the Latin script is dominant—is somewhat lower although existent [19].

The main issue when composing this review was that the problem of script identification is seldom addressed in the context of text-in-the-wild. Therefore, we had to broaden the scope by reviewing script identification in the context of scanned documents where we focused on *digitalization of historical documents*. This area shares some of its main issues with *text-in-the-wild* such as distortion and low quality of the

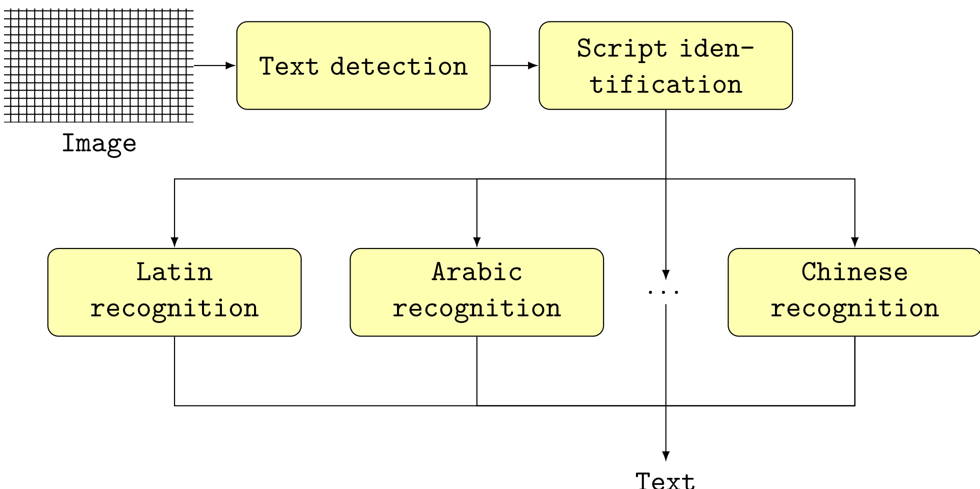


Figure 2.4. Possible architecture of a multi-script text recognition system.

segmented text, complex background, or skew of the text. But it is not burdened by, for example, the difficulty of text localization.

The identification has been done on different levels—page level, paragraph level, line level, and word level [18]. However, for the context of text-in-the-wild problem, only the line and word levels are relevant and we focus on these.

Most of the found publications present methods to recognize a particular set of scripts, e.g. the method proposed in [20] is motivated by the need to distinguish Latin, Cyrillic and Glagolitic scripts. The authors typically examine the scripts and then choose such features that allow them to correctly discriminate the scripts. These schemas show very good performance, usually close to 100%, but have poor generality. Adding another script to the schema requires finding and adding appropriate features (e.g. *horizontal projection profile* in Section 2.2.1 is useful when disciminating Latin and Devanagari but useless for Devanagari and Bengali). This is not only impractical but can lead to problems with dimensionality as features are being added to the set [17].

A review of the interesting methods using a predefined set of features designed to recognize a particular set of scripts with the emphasis on the chosen features is presented in Section 2.2.1. Later, more general approaches whose features are not designed for a specific set of scripts are shown in Section 2.2.2. These methods usually employ some sort of unsupervised learning.

2.2.1 Recognition of Pre-defined Set of Scripts

A simple classifier for English, Tamil and Hindi scripts that operates on the line level was proposed in [21]. The sole feature used to distinguish between the scripts was the *horizontal projection profile* of the line. This feature takes advantages of the fact that the textline has a concept of upper, middle and lower zone and the scripts have different average amount of texture in these these zones. E.g. all characters in Hindi are connected through their top part. This creates one distinct peak in the horizontal histogram in the middle zone whereas an English line typically has two peaks. Refer to Figure 2.5 for an example.

The horizontal profile has been also employed in a more complex scheme in [22] which was successfully used to classify 10 different Indian scripts plus the Latin script. Other interesting features used there include the *water reservoir* analogy. The water could be poured into the characters and some of the characters have such a shape to hold the

Figure 2.5. Example of horizontal projection profiles for Tamil, Hindi and English lines of text. Notice the distinct peak in Hindi text. Image taken from [21].

water (see Figure 2.6). The amount of water that the characters hold from left, right, top and bottom side were used as 4 different features. The usefulness of these features can be demonstrated on the fact that there are 5 characters in the Latin script with the left reservoir (a, s, x, y, z) but in some Indian scripts, most of the characters have left reservoirs.

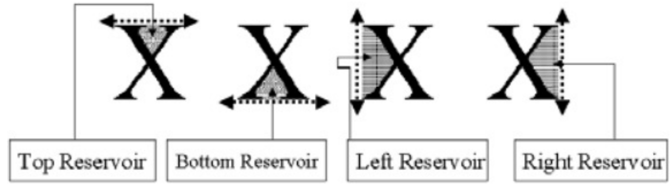


Figure 2.6. Example of top, bottom, left and right reservoirs of the letter X. Image taken from [22].

The method in [20] takes advantage of Latin, Cyrillic and Glagolitic scripts having the concept of lower, middle and upper part of the line. Letters are divided into 4 classes—they can spread only over the middle part (e.g. a, c, e), some also over the lower part (e.g. g, j, p) or the upper part (e.g. A, B, k) or over all the parts (e.g. Q). The text is transformed into coded text by substituting each character by the label of its class. Co-occurence analysis is then done on the coded text to determine the script of the text sample. The paper does not state if the method was tested document-wise or paragraph-wise, but we can assume the performance would be poor on the line level. Nevertheless, we mention this approach due to its unusual procedure.

To identify Latin and Arabic scripts, a different set of features was extracted for each word in [23]. The number of pieces (not characters since the Arabic script is cursive), number of upper and lower diacritical dots, number of loops, number of strokes above and below the baseline, etc. A Multilayer Perceptron is used for classification. The interesting result is that the segmentation of the Arabic words into individual characters is not necessary for successful script identification although being a prerequisite for the character recognition itself.

2.2.2 Script-independent Methods for Script Recognition

The first interesting attempt to solve the problem of script recognition in a general way was found in [19]. Total of 13 scripts including Arabic, Hebrew, Latin, Chinese, or Ethiopic were present. In the learning phase, clusters of similarly looking characters are created. Each character (or word in case of cursive scripts such as Arabic) is rescaled to 30x30 pixels and if there already is a cluster with sufficiently small Hamming distance, the character is added to the cluster. Otherwise, new cluster is created. A template

is made from each cluster by averaging the pixel values of the images in the cluster and thresholding this average image into a binary template. All the templates are then verified to be reliable on the training set. The classification of the script of a new character is simply a matter of finding the best match among the templates using the Hamming distance across the templates in all the scripts.

An improvement in terms of generality was proposed in [24] where they used a large pool of features such as *normalised moments*, *Hu moments*, *compactness*, *number of holes*, *eccentricity* and others to describe the structural features of the characters. Rather than using all the features in a classifier such as AdaBoost, only the features able to discriminate between the required scripts are automatically selected. This is achieved by the **RELIEF-F** algorithm proposed in [25]. It assigns weight to each feature based on its ability to separate one class from all the other classes. The features with low weights are discarted. During classification, k-NN classifier is used to assign a script label to each character. Then a simple voting algoritm is used to determine the script of the text line.

A different approach relying more on the overall appearance of the words or lines is using *Gabor filters*. Gabor filters can be used to approximate the human vision system, capturing some of the important features often used by the human eye [17]. Since the filters can be used to extract structural properties of any script, this method is considered quite general and received a lot of interest. E.g. [26] shows a comparison of Gabor filters and horizontal projection profiles for discriminating the Latin and Tamil scripts on the word level. The performance of Gabor filters was significantly better.

While most of the systems operate on line or word level, the method proposed in [27] operate on the level of individual characters and discriminates between Latin characters, Latin digits, Gurmukhi characters, and Gurmukhi digits. A set of aforementioned Gabor features and another set of gradient features are extracted and used for classification in a SVM classifier. Although the method was tested on scanned documents rather than natural images, we mention this method nonetheless due to the level it operates on and good performance around 98%.

A combination of gradient and texture features was used in [28] to provide a general solution to the problem. Two variants of this approach are discussed in the paper. In the first, Histogram of Gradients (HoG) is computed at the word level to describe the shape and Local Binary Patterns (LBP) are used to capture the texture. The two vectors are concatenated, normalised to compensate for font size changes, and classified using SVM. In the second variant, novel versions of HoG and LBP invariant to text inversion (reading the text upside-down) are proposed. It is argued that these versions are needed because typically a skew correction unit is used before the script recognition and for extreme angles of the text in the input images, it may lead to completely inverting the text. A total of 11 Indian scripts were recognized with the success rate around 97% (depending on the size of the training set).

Chapter 3
Hebrew Dataset

Prior to the evaluation in Chapter 4 and the work on script recognition in Chapter 6, a dataset of an exotic script had to be collected and annotated since we did not find a suitable dataset in our chosen script. Details about the collected dataset are described in the rest of this chapter.

The dataset consist of 100 images—all collected on Wikimedia. The inscriptions are mainly in the Hebrew script, some are in Latin, Arabic, or Cyrillic scripts. For the reasons why we chose to collect a primarily Hebrew dataset, please refer to the introduction to Chapter 4.

The pictures are either in Public Domain or under a permissive licence such as Creative Commons. The nature of the images could be described as photos randomly taken by a tourist in the streets of Israel cities or as images collected from Google Street View in Israel. Details about individual pictures (link, licence, and number of annotated Hebrew words) are available in Appendix C. Each picture is assigned an ID in the table. Appendix B holds thumbnails for all the pictures labeled with the respective ID to allow for easy browsing.

If we were to compare this dataset with some popular public datasets, we should say that it contains much more urban scenes and less images of smaller objects shot indoors than the ICDAR 2013 dataset. On the other hand, the images are comparable to the ones in the Char74k dataset (also used later in this thesis) where signs in urban and nature surroundings dominate.

3.1 Character of Images and Inscriptions

The 100 images contain well over a thousand annotated words. Refer to Table 3.1 for statistics about word and character counts for each annotated script—the occasional Arabic signs are ignored.

	Hebrew	Latin	Cyrillic
words	1141	179	27
characters	6086	1113	205

Table 3.1. Character and word counts of annotations for each script.

The images have been collected with the emphasis on diversity in text density, objects in the scene, background, or lighting conditions. Some of the images contain only one word (e.g. a traffic sign on an empty road), others have up to 200 words (e.g. a board describing a landmark). The annotated inscriptions belong to several prominent categories that shall be described now along with examples.

For instance, common objects in the scene are *traffic signs* in both urban and natural surroundings, their examples can be seen in Figure 3.1.

Figure 3.1. Variety of traffic signs with a diverse background is present.

Figure 3.2. Tourist boards contribute most to the total number of words in the dataset.

Various boards describing landmarks contribute most to the total number of words and correspond with an important application of text recognition—translation of text in images, e.g. for tourists. Examples of such boards are available in Figure 3.2.

The rest of the dataset consists mainly of *street signs* and other *short signs* such as the ones in Figure 3.3. Please refer to the Appendix B for thumbnails of all the pictures. As for the illumination conditions, there are several pictures taken at night but most are in daylight. Since the pictures were collected on Wikimedia, the photos are taken with different cameras with different resolutions ranging from 0.5 MP to 10 MP.

Figure 3.3. The most common objects are street signs and other short signs.

Figure 3.4. Various images are challenging in different aspects such as skew, font, or damaged inscriptions.

A number of *challenging pictures* was selected such as advertising signs in many colours and artistic fonts, banners with severe skew of the inscriptions, or signs with degraded inscriptions. See examples of these challenging images in Figure 3.4.

We believe that the presented variety of images correspond with the tourist scenario and it is safe to say that the dataset suits at least some of the use-cases of TextSpotter, e.g. offer a translation of the text present in the picture to the tourist.

For the purposes of script recognition, we can consider the dataset quite challenging. The text lines are usually short as we can see in the distribution of text line lengths in Figure 6.6. The shorter the text, the more difficult the task of script classification as we learned from the reviewed literature.

3.2 Annotation Procedure

The dataset was annotated using the **pylabelme** annotation tool available at `https://github.com/mpitid/pylabelme` that was adapted at Centre for Machine Perception at CTU for the needs of TextSpotter prior to this thesis.

TextSpotter evaluation tools work best if each annotated rectangle corresponds to one word which means we annotated individual words rather than text lines. This is a bit inconvenient when evaluating the script classifier that works on the level of text

lines. Nevertheless, we did not find this to be an issue since there are no serious reasons not to do the evaluation on the word level. Moreover, annotation on word level is compatible with the rules of the ICDAR competition.

The tool can access TextSpotter interface and create annotation for each MSER detected by TextSpotter and classifies it individually. Such classification does not form the text line and does not take language model into account, therefore, it is often incorrect. Non-character regions are often classified as characters or character regions are assigned an incorrect class. We have corrected all annotation of the MSERs and sorted out the non-character regions. As a result, the dataset can provide ground truth information at the lowest level of individual characters. This was a demanding process as one character in the image can correspond to multiple MSERs. To illustrate the annotation process, we provide a example of an annotation of one selected sign.

Figure 3.5. Screenshot from the annotation tool, both words and characters are annotated in the image. One character is highlighted in **magenta**.

The annotations are organized in the following manner. Each image has several associated text files:

- **imageName.gt** contains bounding box of each word and the word string in plain text as required at the ICDAR competition
- **imageName.lif** contains the same information but in JSON format
- **imageName_regions.json** contains mask images and corresponding annotation of each detected MSER

Moreover, there is a single ground truth file in XML required by TextSpotter containing the bounding boxes and annotation of the words in all images. For convenience, we provide several version of this ground truth file. Each contains words in a different subset of scripts:

- **gt_only_hebrew.xml**
- **gt_hebrew_latin.xml**
- **gt_hebrew_latin_cyrillic.xml**

Should the mask images of all MSERs be required, they can be easily found in a sub-folder with the same name as the corresponding image. A **imageName.txt** file in the same sub-folder contains bounding boxes and annotations for each MSER or question mark in the case of a non-character region. The file ends with a list of MSER IDs forming the words.

Chapter 4
TextSpotter Performance Analysis

One of the goals of TextSpotter is to create a text recognition method that is independent of the used script. Ideally, providing TextSpoter with a diverse set of fonts of a new script would achieve performance comparable to the already tested Latin and Cyrillic scripts. Therefore, one of the objectives of this thesis is to asses the performance of the TextSpotter engine on an unfamiliar script.

Hebrew was chosen as an instance of the unfamiliar script for its fairly similar structural properties to the Latin script (as opposed to e.g. Arabic scripts where different problems arise as explain in Section 2.1.1). Its characters have a clear bounding box and in the printed form of Hebrew do no overlap which makes the basic text recognition method of TextSpotter feasible for Hebrew. Moreover, it has a similar number of character classes as the Latin script (as opposed to the CJK scripts where there may be as many as several thousand classes, see Section 2.1.2).

After familiarising with the interface of TextSpotter and its evaluation tools, a dataset with text in an exotic script had to be created as a prerequisite to the evaluation stage. See Chapter 3 for details about the dataset.

In order to measure performance of TextSpotter when it is used to detect an unfamiliar script, various tools were created besides the standard evaluation framework. The reason behind this decision is that the existing tools do not provide good visualization of the problematic character classes. These tools are described in Section 4.2.1.

The analysis of TextSpotter performance on our Hebrew dataset is presented in Section 4.2.2 along with the list of properties of the Hebrew script that we found problematic. We believe these properties make the recognition of Hebrew a more difficult task than recognition of the Latin script. To validate the problems are universal and not limited to the collected dataset, a comparison with a synthetically created dataset is made to simulate ideal conditions of recognition. Please note that the revision of TextSpotter used during the measurements was **r2462** unless specified otherwise.

When the analysis on the Hebrew dataset was performed, we saw that it would be beneficial to experiment with a script that differs from the Latin script more profoundly. Because an experimental pipeline of TextSpotter was being developed during the work on this thesis, we selected the Char74k public dataset with Kannada inscriptions and compared the performance of both the traditional pipeline and the experimental one. See Section 4.3 for details about the new pipeline and the experiments.

4.1 Preparatory Tasks

Preparatory tasks needed prior to the analysis such as training the Hebrew character classifier or creating the Hebrew dictionary are described in this section.

4.1.1 Training of the Hebrew Character Classifier

In order to classify Hebrew characters, the character classifier must be trained first—using synthetic fonts. The Hebrew alphabet consists of 22 letters and 5 of these letters

have two forms (the special form is used when the character is at the end of the word). In the following text, we treat these forms as different characters which yields 27 classes. The FLANN classifier is used to classify the characters in the conducted experiments. For more details about the FLANN classifier, please refer to Section 6.1.

Total of 19 typefaces were used for training. The typefaces were chosen to represent distinctly different designs of the Hebrew letters. Conservative typefaces such as Times New Roman, Arial, or DejaVu were chosen as well as the more artistic-looking ones such as Stam or Shofar. Monospaced fonts were also present (Miriam Mono, Free Mono). The Table 4.1 shows the font families with the different fonts that were used.

typeface	fonts
Arial	Regular, Italic, Bold, Bold Italic
BABEL Unicode	Regular, Italic
Courier New	Regular, Italic, Bold, Bold Italic
David CLM	Medium, Medium Italic, Bold, Bold Italic
DejaVu Sans	Bold, Bold Oblique
Droid Sans Hebrew	Regular, Bold
Frank Ruehl CLM	Medium, Medium Oblique, Bold, Bold Oblique
FreeMono	Regular, Oblique, Bold, Bold Oblique
FreeSans	Regular, Oblique, Bold, Bold Oblique
FreeSerif	Regular, Italic, Bold, Bold Italic
Hadasim CLM	Regular, Regular Oblique, Bold, Bold Oblique
Keter YG	Medium, Medium Oblique, Bold, Bold Oblique
Miriam CLM	Book, Bold
Miriam Mono CLM	Book, Book Oblique, Bold, Bold Oblique
Shofar	Regular, Regular Oblique, Bold, Bold Oblique
Simple CLM	Medium, Medium Oblique, Bold, Bold Oblique
Stam Ashkenaz CLM	Medium
Stam Sefarad CLM	Medium
Times New Roman	Regular, Italic, Bold, Bold Italic

Table 4.1. Typefaces and their fonts used to train the character classifier.

4.1.2 Optimizations Prior to the Analysis

Two minor optimizations were performed prior to the analysis based on the collected dataset. First, a custom model of the Hebrew alphabet was defined, specifying which characters exceed the baseline (ך, ף, ק, ן, ץ) and which exceed the top line (only ל in Hebrew). This allows TextSpotter to distinguish between several pairs of Hebrew letters which differ only in height. The configuration file describing the model can be found in `testData/config/Hebrew.cfg`.

Second, a dictionary of the most common words in Hebrew was added in order to correct minor errors in classification. A frequency list of words from `https://invokeit.wordpress.com/frequency-word-lists/` was used. The list is constructed from a large sample of Hebrew subtitles available on `http://opensubtitles.org`. It is available under the terms of the Creative Commons – Attribution/Share-Alike 3.0 licence. We used the 150 000 most frequent words as a dictionary for TextSpotter.

The dictionary had very little impact on the performance since only a few words were corrected using it but the custom alphabet model improved the OCR performance by 4%. As performing these tasks is a common routine when adding a new script,

we do not analyse their impact in more detail and we consider them to be more of a preparatory task than optimizations.

4.2 Analysis on Hebrew Dataset

The performance of the entire pipeline was measured using the standard TextSpotter evaluation tools. Recall reached 61% which is comparable to Latin datasets which reach from 60% to 75%. Precision was 38%—much lower than for Latin datasets which reach precision around 80%.

The performance of the OCR module is 68% for the Hebrew dataset which is lower than for the Latin datasets used to test TextSpotter (usually around 90%). From the total of 1141 words, 236 were detected without an error—that means success rate of 21%.

For a more detailed comparison, we provide Table 4.2, where we compare the performance on our Hebrew dataset with the ICDAR 2013 dataset. This dataset contains images of similar nature to the Hebrew dataset although there are much more photos of objects taken from close range and the scenes are often less complex.

	Precision	Recall	OCR Precision	Perfect words ratio
Wikimedia	37.9%	61.1%	67.8%	20.7%
ICDAR 2013	83.1%	66.0%	87.9%	45.7%

Table 4.2. Performance comparison with the ICDAR 2013 dataset. Different key measurements are provided.

We can see the performance is generally lower than for Latin datasets. To achieve a deeper understanding of the problem and to measure progress in the future work, additional tools for analysis were created.

4.2.1 Evaluation Tools

In order to assess which characters are hard to classify, *a confusion matrix* is constructed using the Scikit-learn and Matplotlib libraries in Python (Figure 4.1). The matrix is constructed from the **results.xml** file using a diff tool. The characters for which it is apparent that it has been classified correctly, or for which the diff tool can find a reliable misdetection, are added to the matrix. E.g. if the string “דיל” is detected as “רל”, we cannot be sure if the letter ד was incorrectly detected as ר and the י was lost in the lower stages of the pipeline or if it was the other way around.

To emphasise problematic characters, we provide Tables 4.3 and 4.4 with *false negative* and *false positive rates* for each character ordered by the respective measure. This allows us to quickly glance at the first line to see which characters are often misdetected (FNR) and which characters are often used instead of the correct ones (FPR). The total number of true occurrences is always present so we can filter out the rarely used characters (where the rates do not have significant value).

The last used indicator is the number of cases when each character was missing (not misdetected). In this case, the character was either lost before the OCR stage (the more like scenario) or the OCR engine failed to classify it. Table 4.5 shows the figures for each letter.

4.2.2 Problems in Hebrew Text Recognition

If we examine the FNR and FPR and the confusion matrix, we can clearly see the groups of problematic characters. Very common misdetections (ignoring the characters with only a few occurrences) are the following ones.

- ת is mostly classified as ח or ה.
- ה is often classified as ר
- ו is often classified as ן or י.
- י is often detected instead of a number of characters (ה, ו, ל, ר, . . .)

The letter ת is problematic due to its high similarity with the letters ה and ח. There is no pair of letters in the Latin alphabet so similar to each other, therefore, TextSpotter didn't need such a sensitive OCR so far. The groups of very similar characters has been also stated in [11] as one of the problems in recognition of the Hebrew text.

The letters ה and ק break the assumption of *a single connected component* currently present in TextSpotter (and valid for alphabets such as Latin or Cyrillic). In consequence, these characters are among the top 5 letters with regard to FNR. When the smaller component is eliminated, the letter ה looks like ר and the letter ק does not directly resemble anything, therefore it gets classified as a number of different characters.

The last group of problems is connected to the letter Yod (י). The Latin script does include the concept of the baseline. Some characters are completely above the baseline (e.g. m, n, u) and some do exceed the baseline (e.g. p, q, y). However, the Yod in Hebrew effectively adds an extra line since it covers no more than half of the usual character height. If the shape of the letter were more distinct, this would not pose an issue. But the letter is a simple vertical line and can have drastically different shapes in different fonts. When the shape of a character is not clear, it often starts to resemble Yod (particularly ר or ה).

Furthermore, Yod (י) is the most often lost character in both absolute and relative measures. This is probably due to its small size and indifferent shape which allows font designers a great degree of freedom. To remedy the problems connected with Yod, the position of the character could be taken into account during the classification process.

letter	ת	ה	ק	ף	ך	ג	ד
FNR	100.0%	93.9%	45.5%	40.0%	37.0%	36.2%	29.6%
count	179	147	44	10	27	58	88
letter	ז	ל	ס	ר	כ	ו	ם
FNR	28.1%	26.9%	21.1%	20.4%	18.1%	15.8%	15.8%
count	32	216	76	240	94	392	95
letter	מ	ץ	ט	ן	נ	ב	ש
FNR	14.3%	14.3%	13.7%	12.5%	11.0%	10.9%	10.2%
count	196	7	52	48	163	238	176
letter	צ	פ	י	ח	ע	א	
FNR	9.2%	6.4%	6.1%	5.3%	4.3%	3.9%	
count	65	78	378	113	117	207	

Table 4.3. False negative rates for each Hebrew letter with the number of true occurrences.

letter	ח	ר	י	ו	ן	נ	ה
FPR	4.15%	3.85%	3.29%	2.61%	2.29%	1.33%	1.12%
count	113	240	378	392	48	163	147
letter	ך	ט	ם	מ	ע	ס	ד
FPR	0.71%	0.60%	0.58%	0.42%	0.38%	0.38%	0.35%
count	27	52	95	196	117	76	88
letter	ג	ב	כ	ז	ל	פ	צ
FPR	0.26%	0.24%	0.20%	0.14%	0.09%	0.09%	0.09%
count	58	238	94	32	216	78	65
letter	א	ק	ץ	ף	ת	ש	
FPR	0.06%	0.06%	0.06%	0.03%	0.00%	0.00%	
count	207	44	7	10	179	176	

Table 4.4. False positive rates for each Hebrew letter with the number of true occurrences.

letter	י	ף	ג	ה	ץ	ז	ר
lost	21.9%	21.1%	20.2%	19.1%	15.4%	14.3%	13.7%
count	595	19	84	324	13	49	321
letter	ט	ד	ן	צ	ק	ם	נ
lost	13.5%	13.4%	13.2%	13.1%	13.0%	13.0%	13.0%
count	74	127	68	84	92	131	224
letter	ת	ל	ח	פ	ס	ו	כ
lost	12.5%	12.3%	11.3%	10.7%	10.5%	10.1%	9.7%
count	264	301	151	103	105	515	124

Table 4.5. The percentages of lost characters for each letter and the total number of occurrences for each letter.

4.2.3 Comparison with the Synthetic Dataset

In order to assess the problems in the lower stages of the pipeline (before the OCR module), a synthetic dataset was generated from all the fonts used for training of the character model (see Section 4.1.1) and 25 common Hebrew phrases.

The synthetic dataset does not suffer from the effects present in real-world pictures such as complicated backgroud, skew of the inscriptions, or partial occlusions of the characters. Also, the text covers most of the picture and all the text is in one scale, thus diminishing the problem of text in different scales. These simplifications reduce the task to a simple OCR problem. An example synthetic image can be found in Figure 4.2.

Figure 4.2. Example of an image from the synthetic dataset meaning "Can you speak more slowly?"

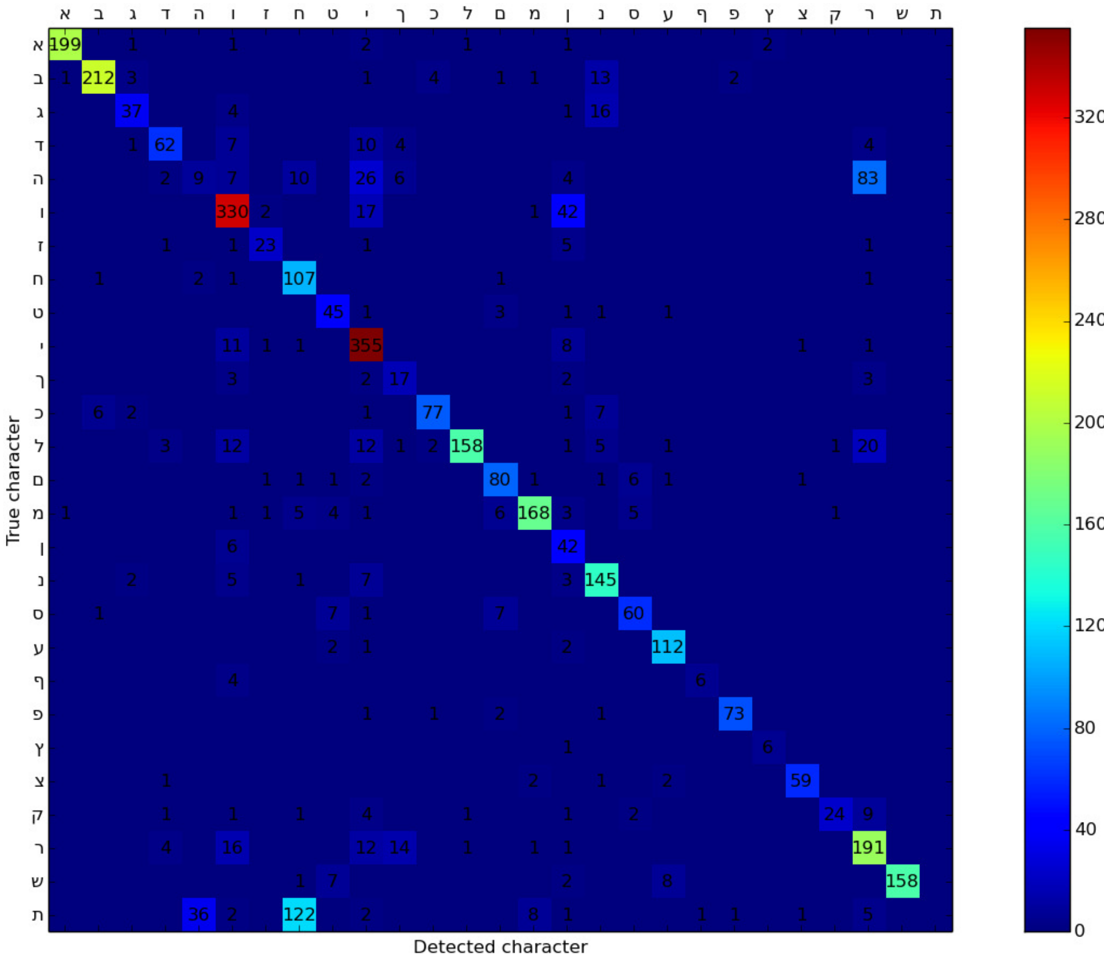


Figure 4.1. Confusion matrix highlighting the problematic classes of letters in the Hebrew script.

	Precision	Recall	OCR Precision	Perfect words ratio
Synthetic	70.1%	58.5%	81.0%	23.9%
Wikimedia	37.9%	61.1%	67.8%	20.7%
ICDAR 2013	83.1%	66.0%	87.9%	45.7%

Table 4.6. TextSpotter performance on the synthetic dataset and the Wikimedia dataset. ICDAR 2013 also included to provide better overview.

The comparison of important metrics for the Wikimedia dataset and the synthetic one is available in Table 4.6. We can see the performance on the synthetic dataset is either comparable or better than on the Wikimedia dataset.

We can come to a conclusion that Hebrew alphabet does not have properties that would had a substantial negative effect on the lower stages of the pipiline (MSER, character candidates detection, and line detection). If this were the case, TextSpotter would perform worse on the Wikimedia dataset. The lower performance can be attributed to the problems in the OCR stage—even on the synthetic dataset, the OCR performance is poorer than on real-world Latin datasets by approximately 10%. Another indicator supporting this theory are the recall rates—recall is lower on the Wikimedia dataset only by 4% compared to the ICDAR 2013 dataset which means a character candidate or a text line have more or less the same chance of being detected no matter the script.

Table 4.7 shows top 7 worst characters regarding the false negative rate. It confirms the most promiment problems of the Hebrew script as the worst 3 letters are exactly the same. In fact, the problems of the first 3 letters stand out even more if we examine the false negative rates. The confusion matrix of the synthetic dataset is left out for the sake of brevity (it shows the same problems as the confusion matrix in Figure 4.1).

letter	ת	ה	ק	ד	ו	ח	ם
FNR	100%	97.5%	56.4%	16.3%	13.7%	9.0%	8.0%
count	554	317	126	178	666	310	224

Table 4.7. Top 7 false negative rates for the characters on the synthetic dataset.

4.3 Analysis on Char74k Dataset

The Hebrew script was a good choice to measure performance of TextSpotter on a script similar to Latin. However, we would like to perform a comparison on a wildly different script as well. The publicly available Char74k dataset gives us such opportunity as it contains inscriptions not only in Latin, but also in Kannada script.

There is another reason for adding evaluation on the Kannada script. While working on this thesis, a completely new pipeline was being developed (refer to [29] for details) and the need to evaluate its performance against a publicly available dataset arose. Moreover, the new pipeline should be more suitable to recognize such a complex script than the traditional pipeline and a comparison of their performance is offered.

Unlike the traditional pipeline in Figure 1.1, the new pipeline classifies found MSERs into 3 classes: *characters*, *multi-characters*—meaning connected characters or whole words—and *background*. The multi-characters are then segmented into individual characters using the *graph cut* method. Therefore, one of the basic assumptions of one character corresponding to one connected component is not needed in this pipeline. As a result, it should be applicable to a wider set of scripts such as Arabic or the mentioned Kannada since it performs the segmentation.

Kannada is one of the official scripts in India with 14 vowels and 36 consonants. There are no capital forms of letters which would mean the number of character classes is similar to the Latin script. However, consecutive consonants form clusters where one consonant is written under the other (see an example in Figure 4.3) or the cluster has a special ligature. Moreover, vowels are often written in the form of diacritics added to the preceding consonant (see Figure 4.4). All the combinations make up several hundred character classes. Unless we want to train the character classifier to recognize hundreds of classes, segmentation is necessary due to the ligatures of consonant clusters and vowels written in the form of diacritics.

Figure 4.3. Two consecutive consonants form a consonant cluster in Kannada.

The Char74k dataset has been created in [30]. The dataset concentrates on character annotations but many words are also annotated (sometimes both). Single character

ಕ + ಆ = ಕಾ

Figure 4.4. A vowel is written in diacritic form if it follows a consonant in Kannada.

annotations are not suitable for evaluation system of TextSpotter, therefore, we created **char74k2icdar.py** script that creates the ground truth file TextSpotter expects from the word annotations only. A total of 249 Kannada words were found in the dataset. An example of an image from the dataset can be examined in Figure 4.5.

Figure 4.5. An example from the Char74k dataset. Notice the multi-script sign and the high similarity between Kannada characters.

The character classifier has been trained in the same manner as in Section 4.1.1 only with different fonts. The lack of wide variety of synthetic Kannada fonts is a serious issue. We were able to collect only 15 different fonts, see Table 4.8. This probably diminishes performance of the classifier but since we use the same one for both pipelines, we are still able to do a comparison.

typeface	fonts
Akshar Unicode	Regular
Arial Unicode MS	Regular
Kedage	Normal, Bold, Italic, Bold Italic
Lohit Kannada	Regular
Malige	Normal, Bold, Italic, Bold Italic
Navilu	Normal
Noto Sans Kannada	Regular, Bold
Sampige	Regular

Table 4.8. Typefaces and their fonts used to train the Kannada character classifier.

We cannot expect high performance since having 2 letters on top of each other in a word is something TextSpotter cannot handle. However, the vowels written as diacritics are effectively a ligature and if the graph cut pipeline were able to segment the diacritics, the recognition of the vowel would be possible since we have trained the diacritics as separate characters.

4.3.1 Comparison of TextSpotter Pipelines

The Char74k dataset was processed by both the traditional pipeline and the experimental graph cut pipeline. The revision **r2616** of TextSpotter was used. We have discovered that Kannada script is very demanding—both pipelines show very poor performance on the Kannada script which is immediately apparent in Table 4.9.

By examining the output, we saw that difficulty of detection of Kannada text is comparable to the detection of Latin text for both pipelines (see Figures 4.7 and 4.8). MSER proved to be a general method able to detect even characters of the Kannada script.

	Precision	Recall	OCR Precision	Perfect words ratio
Traditional	6.0%	15.7%	8.8%	0%
Graph Cut	7.4%	25.6%	2.7%	0%

Table 4.9. Performance comparison of the traditional pipeline and the graph cut pipeline.

The problem is in the OCR stage for both pipelines. Many characters in the Kannada script differ from each other only by a dot in the centre or a different end of the stroke. The consonant clusters are an unfamiliar concept to TextSpotter which is why their lower parts may be mistaken for an independent word which happened e.g. in Figure 4.6. The most notable difference between the pipelines is that graph cut pipeline achieved much higher recall which is most likely due to its ability to detect multi-characters which is advantageous in a script with many ligatures.

Figure 4.6. The unusual properties make text detection of Kannada difficult. Notice the word incorrectly formed from the lower parts of the consonant clusters in **magenta**.

However, our expectations about its ability to perform segmentation on Kannada words to obtain the diacritical form of vowels did not confirm. While useful for the connected characters of the Latin script, the segmentation technique would have to be adapted for the Kannada script.

We have examined the text detected by both pipelines and found that the graph cut pipeline was much more successful in the detection in majority of images. This observation corresponds to the higher recall of the experimental pipeline. To illustrate the difference in performance, we provide several examples of text detection in Figure 4.7.

There are rare cases where the graph cut pipeline does not detect the text but the traditional pipeline does. More serious issue is that the graph cut pipeline quite

Figure 4.7. Text detections by the graph cut pipeline (**green**) are typically more accurate than by the traditional pipeline (yellow).

Figure 4.8. Rarely, the traditional pipeline (yellow) performs better than the graph cut pipeline (**green**). Notice how graph cut often misclassifies background as character or multi-character.

often classifies background as characters or multi-characters. Both these problems are illustrated in Figure 4.8.

To sum up, the new graph cut pipeline showed significantly better performance in text detection but the FLANN-based character classifier used in both pipelines during our measurements is not general enough to handle a complex script such as Kannada.

We believe that even with a larger number of Kannada fonts, the performance would not improve by much. Our results confirm that recognition of Kannada script is a difficult problem which is in accordance with the results published by the authors of the dataset in [30] whose method was not applicable to Kannada as well.

Chapter 5
Improvements for Hebrew Script

In the following chapter, we attemp to extend the existing methods of TextSpotter to be more applicable to the Hebrew script thus improve the performance on the script. In Section 4.2.2, we have discovered inherent structural properties of the script that are not present in the Latin script. Therefore, these were not accounted for in the original design of TextSpotter.

We modify the existing methods rather than develop a specialized character classifier since one of the goals of this thesis is to make the methods used in TextSpotter more general. The modifications are implemented in a manner that does not affect the performance on other scripts.

First problem are the letters comprising of 2 connected components—He (ה) and Qof (ק)—and the solution is offered and evaluated in Section 5.1. Second problem is the high similarity of the letter Yod (י) to a number of other letters in the alphabet due to its indistinct shape. We use the unusual positioning of the letter on the line to solve this issue in Section 5.2.

5.1 Two-component Letters

Two different approaches are presented to solve the problem of the two-component letters. Section 5.1.1 offers a straightforward method of removing the smaller component of the letter while Section 5.1.2 describes a method of merging the components together. The performance of both methods is then compared in Section 5.1.3.

5.1.1 Removing the Smaller Component

The most straightforward solution is removing the smaller component of the letter before training the FLANN matcher. The advantages are its simplicity and not requiring any internal changes to TextSpotter. It is be expected to solve the issue provided the chosen component of the letter is distinct from the other characters. If we examine the letters, the larger part of Qof is quite distinct but the larger part of He is very similar to Resh (ר) and in most of the fonts, it is indeed the same.

The implementation of the solution is as follows. The **ocrTrainGenerator.py** script creates an image for each letter of the script and for each training font. The FLANN matcher is then trained using these images.

As usual, the script creates an image of the letter. Before the image is saved, the following procedure is called to remove all but the largest connected component.

1. binarize the image (pixels with intensity lower than 50 are considered background)
2. label the connected components
3. remove all pixels from the original image except the pixels of the largest component
4. save the image of the letter

Then the FLANN is trained as usual from the set of these images.

5.1.2 Merging the Components

Instead of simply discarding the smaller component of He(ה) and Qof(ק) letters, one might find suitable criteria that would allow to merge the components into one character. While being more complicated than the previous solution, it should solve the issue of He(ה) resembling Resh(ר) after removing the smaller component and also solve the problem in its generality.

If we examine the two letters, we see that they comprise of one curved component (we shall call it *outer*) and one simple stroke (*inner* component). Following criteria for merging the components were derived from the structural properties of the letters.

1. Outer component must have one deep defect in its convex hull.
2. Inner contour cannot have any large defects in its convex hull.
3. The bounding boxes of the components must overlap.
4. The contours of the components cannot overlap.

Let us examine the criteria. Both letters have indeed one large defect in the convex hull of the *outer component*. The defect is marked in Figure 5.1 for the letter He(ה). In some of the more decorative fonts, there are other defects present in the outer part of the letter but these are easily filtered-out using the depth of the defect—the threshold was set empirically but the method is not sensitive to changes in this threshold since the depth difference is quite high.

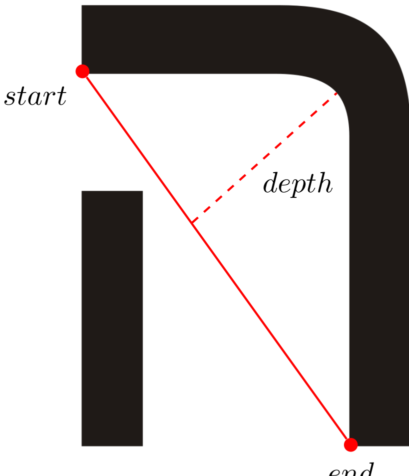


Figure 5.1. The large defect in the convex hull of the curved component is marked in red. Start and end points of the defect are the red dots and the depth of the defect is marked with a dashed line.

The situation is similar for the *inner components*. Although it is a simple stroke, there might be small defects present in the decorative fonts. These are filtered out in the same manner as in the case of the outer components.

The first two criteria filter most of the wrong pairs of components that might be merged but not all, e.g. consider the string “רו”. Therefore, the third criterion is added. For the two letters in question, it is very difficult to find a rotation where the bounding boxes would not overlap which makes this condition suitable yet simple.

The last condition is merely of a technical nature. Since multiple scales of all character candidates are present at this point of the pipeline, there are some bizarre contours whose multiple scales comply with the previous conditions. Such contours are present in the real-world Wikimedia dataset, not the synthetic one.

Unfortunately, characters with more than one component are very rare, therefore, the method cannot generalize to other alphabets. The only other character consisting of two components we were able to find is Jēran in the Runic alphabet. This character breaks the second assumption of our method but Runes are such an edge case that we did not find it necessary to design the method to account for the character. The CJK scripts are completely different, their characters consist of many components where different techniques must be used (refer to Section 2.1.2 for some examples).

5.1.3 Comparison of Methods for Two-component Letters

To assess if the two approaches solve the problem at hand and to compare them, we provide a confusion tables for both characters in Tables 5.1 and 5.2. It was mentioned in the analysis that He and Qof are among the top 5 letters regarding the FNR. The performace improved for both letters when either of the proposed methods was used, especially the true positive rates.

ה – before		ה – removed		ה – merged	
9 TP	38 FP	98 TP	34 FP	81 TP	38 FP
138 FN	3351 TN	73 FN	3327 TN	110 FN	3312 TN

Table 5.1. He (ה) confusion tables showing the performance before and after each method—after removing the smaller component or after merging the components.

ק – before		ק – removed		ק – merged	
24 TP	2 FP	35 TP	14 FP	37 TP	5 FP
20 FN	3490 TN	16 FN	3457 TN	18 FN	3481 TN

Table 5.2. Qof (ק) confusion tables showing the performance before and after each method—after removing the smaller component or after merging the components.

Although Qof is showing an increase in false positive rate, the overall precision of the OCR module was slightly improved by 1% in both cases. The number of perfect words increased slightly when removing the smaller component and significantly when merging the components, see Table 5.3.

before	removed	merged
236	239	277

Table 5.3. Total number of perfect words before and after each method—after removing the smaller component or after merging the components.

The better overall result when using the merging method has two reasons. First is that when the smaller component is eliminated, the TP rate of He(ה) indeed increases but it leads to increasing FN rate in Resh(ר) since the characters are almost indistinguishable (although it still is an improvement). The second reason is that merging of the components helps in other situations as well. E.g. in some fonts, the middle stroke of Shin(ש) is not connected to the rest of the letter and the proposed method correctly merges the two components together.

While experimenting with the first solution in Section 5.1.1, it was discovered that the described preprocessing of the synthetic fonts actually measurably improves the overall performance of the pipeline if done on all the letters of Hebrew alphabet.

The thresholding has the effect of 'sharpening' the letter, possibly removing the parts of the letter that are lost during the recognition process, thus making the learning process more focused. An enlarged example of the preprocessing effect on the letter Mem (מ) can can be found in Figure 5.2, the background is coloured blue to emphasise the difference.

Figure 5.2. The effect of thresholding on the letter Mem (מ). We can see the brightest pixels are removed. The background is coloured blue to emphasise the difference.

Following measurements relate to the case when the preprocessing is done on all letters, not just the aforementioned two, and the merging method is used. In order to provide a thorough comparison to the state before the optimizations, we provide the confusion matrix again in Figure 5.3 and false negative and false positive rates (only for top 7 characters for the sake of brevity) in Tables 5.4 and 5.5. If we compare the tables with the Tables 4.3 and 4.4, we see that the rates have dropped for most of the characters.

Arguably, the most important metric is the total number of correct words. This improved from **277** before the optimisations to **311** after.

In the case of the characters we focused on, the FNR rate for He (ה) has dropped by a third although it is still maintaining the second worst position. Thanks to merging the components, there is no regression in recognition of Resh (ר). In fact, only one Resh was classified as He (ה). The FNR rate for Qof (ק) has dropped from 45.5% to 32.7% which does not seem as much but if we take a look at the confusion matrix in Figure 5.3, we can see the situation has quite improved.

letter	ח	ה	ך	ק	ף	ן	ז
FNR	100.0%	61.4%	34.6%	32.7%	30.0%	28.6%	26.5%
count	166	207	26	49	10	42	34

Table 5.4. False negative rates for the top 7 worst characters after the optimizations.

letter	ר	ח	י	ו	נ	ה	ן
FPR	3.94%	3.05%	2.71%	2.46%	1.32%	1.27%	0.61%
count	261	116	372	405	173	207	42

Table 5.5. False positive rates for the top 7 worst characters after the optimizations.

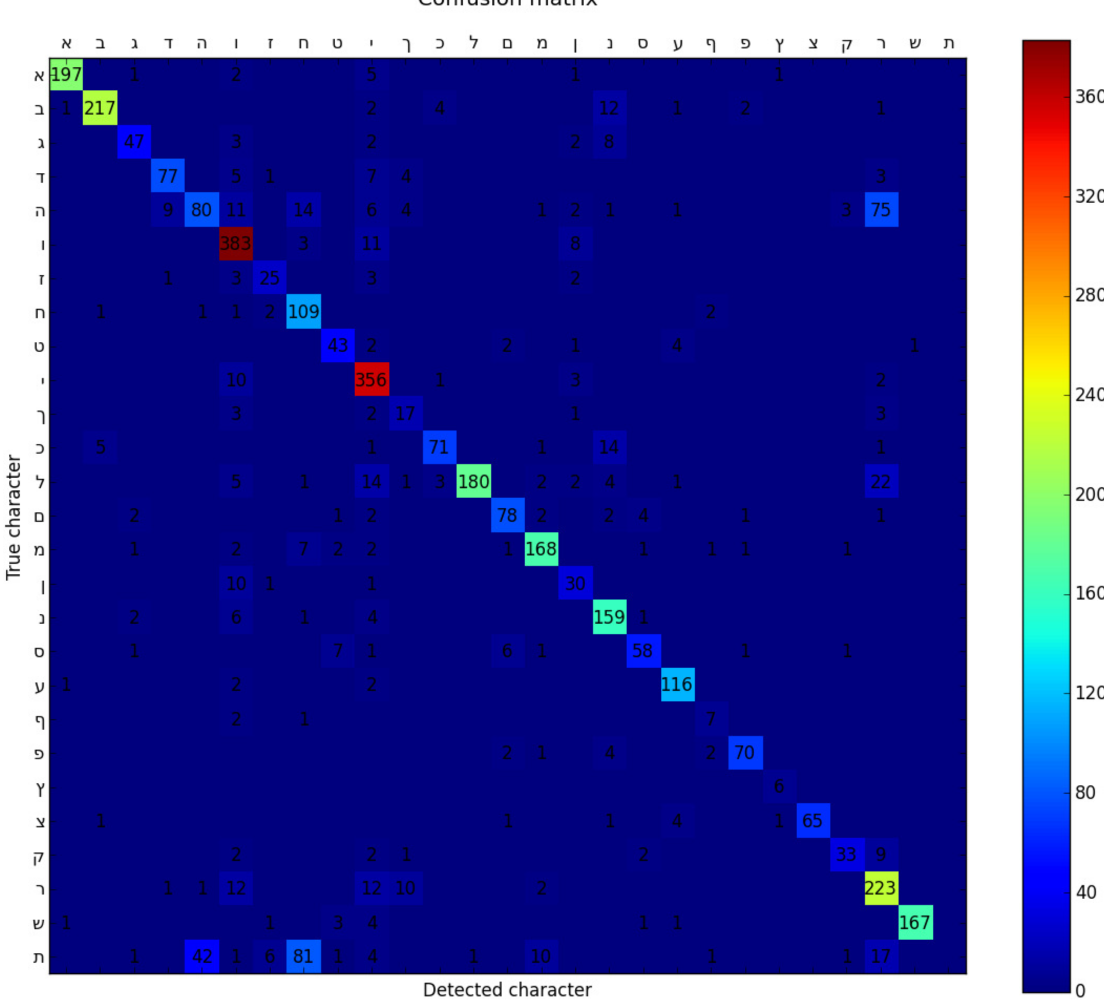


Figure 5.3. Confusion matrix after the all the optimizations—merging of components and the pre-processing.

That being said, we can draw a conclusion that the more complicated method of merging the components together is more suitable to solve the problem of letters with two components. Mainly because of its substantial impact on the total number of perfect words but also because it does not suffer from the regression in Resh (ר) classification.

5.2 Issues with the Letter Yod

We have identified two different problems connected to the letter Yod (י) in Section 4.2.2. First problem is that since it has a very similar shape to the letters Vav (ו) and Nun (ן), the confusion among these character classes is high.

Second problem is that Yod (י) has a very indistinct shape and a number of different letters tend to be classified as Yod (e.g. ה ,ל, or ר), probably when parts of their contours are missing. As a result, FPR of Yod is quite high.

There is only one feature of Yod that can help us distinguish it from other similar characters—its *unique position* on the line. As in the Latin script, the Hebrew line can be divided into 3 regions. Some characters extend under the bottom line (underline letters), some extend above the top line (high letters) and some are between the top and bottom lines (low letters). Yod belongs to the last group of low letters. However,

while it does start at the top line, it does not extend to the bottom line like the other characters. Consequently, we can say it creates a new group of characters that extend from the top line to the Yod line, see Figure 5.4 for illustration of this phenomenon.

Figure 5.4. There is one more region in the Hebrew line of text due to the letter Yod.

The problem of finding the optimal classification of letters in a text line is formulated in TextSpotter as a longest path problem in the adjacency graph of the character hypothesis to find a sequence of hypothesis with maximal score (see [3] for a rigorous definition).

The score assigned to each character hypothesis is a combination of multiple factors. One of these is the character position on the text line. If the character region is classified as the letter "x", its bottom should be close to the bottom line and its top close to the middle line—it is a low letter. The position deviation for a low letter r is defined as

$$\mathrm{dev}(r) = \frac{\mathrm{d}(l_m, r) + \mathrm{d}(l_b, r)}{h},$$

where l_m and l_b refer to the middle line and bottom line. The deviation is normed by the height of the line h.

When Yod was listed among the low letters, it always strongly deviated from the bottom line. To account for this, we introduce a new class of letters containing only Yod. The top of these characters lies on the middle line and the bottom lies on the Yod line (refer to Figure 5.4 again). The position deviation of the Yod character region y is defined as

$$\mathrm{dev}(y) = \frac{\mathrm{d}(l_m, y) + \left|\mathrm{d}(l_b, y) - h/2\right|}{h}.$$

Since we cannot detect the Yod line (there is rarely more than one Yod character within one line), we take advantage of the fact that Yod character covers about half of the text line in most fonts. The distance of the character to the Yod line is then approximated by $\left|\mathrm{d}(l_b, y) - h/2\right|$.

5.2.1 Evaluation of the Yod Optimization

The proposed solution has been evaluated but as it was implemented at a later time, a newer revision **r2616** of TextSpotter was used to conduct the experiments in this section. This revision already includes the improvements from the Section 5.1.

The solution has improved the confusion between Yod and the other classes of similar characters as we can see in the confusion matrix in Figure 5.5 if we compare it to the

before	after
307	321

Table 5.6. Perfect words before and after the Yod optimization.

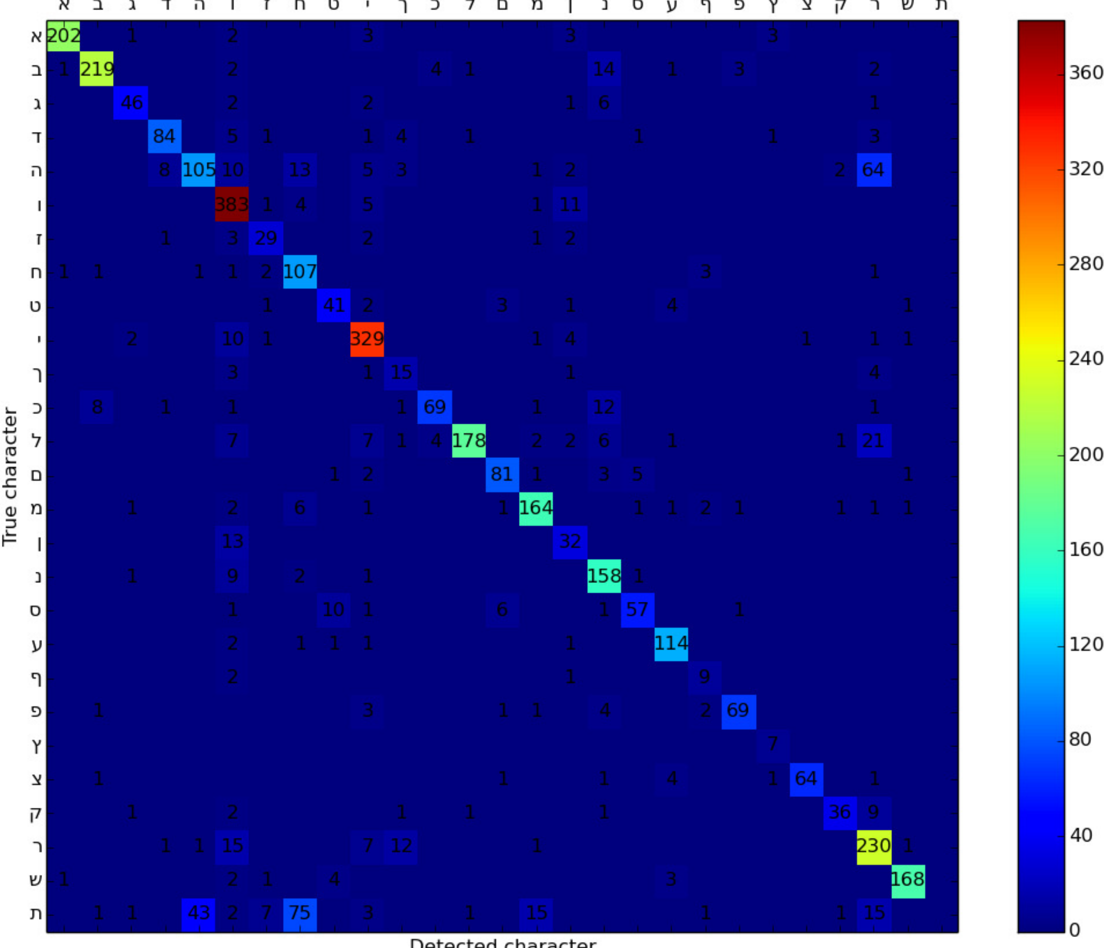


Figure 5.5. Confusion matrix of character classification after the Yod optimization.

previous matrix in Figure 5.3. The most important metric—total number of perfect words—improved as well as we can see in Table 5.6.

The high false positive rate indicated that many characters were misclassified as Yod. Situation improved in this regard as well. We provide Table 5.7 with FPRs before the optimization once more because the used revision of TextSpotter is different and as a result the FPRs slightly differ from the tables in the previous section. Kindly compare the values with the 7 worst FPR after the optimization in Table 5.8. The FP rate of Yod has decreased almost by half while the rest of the FPRs only slightly increased. Some of the Yod characters were lost by re-defining the position deviation but the total performance increased—it is apparent from the increase of total perfect words.

letter	ר	ח	ו	י	ה	נ	ן
FPR	3.59%	2.71%	**2.64%**	2.59%	1.31%	1.23%	0.77%
count	262	115	403	377	210	169	43

Table 5.7. False positive rates for the top 7 worst characters **before** the Yod optimizations.

letter	ר	ו	ח	י	נ	ה	ן
FPR	3.64%	2.94%	2.84%	**1.42%**	1.37%	1.30%	0.80%
count	268	405	117	350	172	213	45

Table 5.8. False positive rates for the top 7 worst characters **after** the Yod optimizations.

Chapter 6
Script Recognition

Script recognition is an important pre-requisite for complex tasks such as recognition of text in multiple scripts in one scene image (explained in Section 2.2) or for identification of the language used in the image. This chapter describes the implementation of script recognition in TextSpotter, which was not present prior to this thesis, and the rational and theory behind the chosen approaches.

First method was proposed and evaluated in Section 6.1. The scripts are discriminated by structural features of individual characters using a combination of several *k-Nearest Neighbour* classifiers—one for each script—and while the method did show promising results, a different approach based on *bigram frequencies* was explored in Section 6.2. Although it showed lower performance than the first classifier, we successfully combined these two approaches to achieve slightly higher performance.

As for the implementation details, **ScriptClassifier** class implements both of the classifiers. If needed, please refer to the enclosed DVD for its source. The model for the script classifier can be trained from the character images and data created by the **ocrScriptTrainGenerator.py** using the **TrainScriptClassifier** method in TextSpotter. The script classifiers were developed and tested using the traditional pipeline of TextSpotter. There is also an experimental graph cut pipeline able to detect multi-characters that we compared to the traditional pipeline in Section 4.3. Plugging the script recognition into this pipeline should not pose an issue.

6.1 Nearest Neighbour Script Recognition

In the original TextSpotter pipeline, character candidates are first detected and assembled into text lines without the need for the knowledge of the script. The proposed method for classification of script is based on following assumptions:

- The text lines and individual character candidates are already detected.
- The script rarely changes within one line of text.
- Different lines of text within an image are likely to be in different scripts.

In order to explain the chosen strategy for script classification, we must briefly discuss the classifier used for *character classification* in the traditional pipeline because we combine them later to create a *script classifier*.

The classification is done via k-Nearest Neighbour search (k-NN) in a high dimensional feature space. Each feature vector f stored during the training corresponds to the feature vector of a letter image i. The image i is created from character $c \in S$ (where S is a script) using one of the synthetic fonts.

To speed-up the computation of k-NN, only an approximate search is performed. *Fast Library for Approximate Nearest Neighbours* (FLANN) is used for this purpose (first proposed in [31]). It is suitable for an approximation of Nearest Neighbour in higher dimensions in sub-linear time. As the authors claim, many approximations of

NN were proposed but the speed often depends on the dataset. FLANN is able to choose the most suitable approximation algorithm for the given dataset.

Since the distances in high dimensional spaces are all huge and differ only slightly from each other, the currently used classification strategy in TextSpotter is *voting* rather than directly assigning the character class c belonging to the nearest vector f. FLANN is used to find the set of K nearest neighbours $F = \{f_1, \ldots, f_K\}$ (specifically $K = 11$) in the feature space. If $\text{label}(c, f)$ is a function returning 1 if feature vector belongs to class c and 0 if not, confidence can be assigned to each character class $c \in S$ quite intuitively as

$$\text{conf}(c, F) = \frac{\sum_{f \in F} \text{label}(c, f)}{K}.$$

In other words, the higher the number of neighbours of the same character class, the more confident we are that this class is the right one.

For the purposes of script classification, a text line L is a sequence of character images. For each character of line L, we obtain its K nearest neighbours using an instance of FLANN trained to recognize the script S. Let us define $L_S = \{F_1, \ldots, F_n\}$ to be the set of outputs. Then the confidence of the whole line when presumed to be in script S is defined as

$$\text{lineConf}(S, L) = \sum_{F \in L_S} \max_{c \in S}(\text{conf}(c, F)),$$

which can be informally formulated as a sum of maximal confidences across all characters of the line. The decision of the classifier is equal to solving following optimization problem

$$S^* = \operatorname*{argmax}_S \Big(\text{lineConf}(S, L)\Big),$$

ergo deciding for the script that provides the highest confidence on the line.

The advantage of this approach is its generality. If the script can be recognized by the traditional pipeline, we can discriminate the script without additional information other than the images of each letter generated using the synthetic fonts. One of the objectives while designing the classifier was to preserve this desirable property of TextSpotter.

Similar approach was proposed in [24] where k-NN search was also employed. However, they used a simpler voting scheme to classify script of the text line. Each character was assigned a script thus voting for the script. The script with the most votes was chosen. We experimented with this approach as well but found its performance lower than using the script with the highest sum of maximal confidences. We believe our approach offers better granularity. Consider for instance recognition of Hebrew and Latin text in an image. The Latin letter “n” is very similar to the Hebrew letter “ח”. In the voting scheme, confidence for one of the scripts would be slightly higher and that script would gain the vote. On the other hand, in our scheme, both characters will contribute to the sum of confidences similarly and other characters with more unique appearance will effectively make the decision.

The performance will most likely depend on the length of the line but this is a common trait of all the methods explored in the literature. The methods showing zero error rates operate on larger portions of the text—paragraphs or whole pages of text.

The proposed script classifier was implemented in TextSpotter and the ability to dynamically switch character classifiers based on the result of the script classification was added. This allows TextSpotter to recognize text in multiple scripts within one

Figure 6.1. Example of a successful script recognition of Hebrew and Latin text lines. The chosen script is in the annotation above the text line.

image which was previously not possible. An example of a successful script recognition may be examined in Figure 6.1.

6.1.1 Evaluation of FLANN-based Classifier

To evaluate the proposed solution, we took advantage of the dataset collected in Chapter 3. There is text in languages other than Hebrew present which was not annotated during the evaluation stage where annotations in only one script were a necessity. The Latin signs were annotated as well at this point and the proposed script classifier was tested on the dataset.

The Hebrew classifier was trained using the same fonts as in Table 4.1. To provide complete information, we list the fonts used for training of the Latin classifier in Table 6.1.

The script classifier was tested on two script classes—Hebrew and Latin—because the number of Cyrillic words is very low and we could not draw any conclusions about the Cyrillic class. The success rates are encouraging, see Table 6.2 for details.

script	Latin	Hebrew
success rate	85.88%	90.71%
word count	177	1077

Table 6.2. Success rates of Hebrew and Latin script classification.

We can argue that the classifier provides state-of-the-art performance when we compare the results to the methods presented in Chapter 2. The classifiers reviewed there were often tested on more favourable datasets of printed documents where the quality of the characters is higher (although the distortion of characters in scanned historical documents is closer the the problem of text-in-the-wild). A viable candidate for comparison is the method for Latin and Cyrillic recognition in scanned documents presented

typeface	fonts
Andale	Mono Regular
Arial	Regular, Black Regular, Italic, Bold, Bold Italic
BABEL Unicode	Regular, Italic
Courier New	Regular, Italic, Bold, Bold Italic
DejaVu Sans	Book, Bold, Oblique, Condensed,
DejaVu Sans Mono	Book, Bold, Oblique, Bold Oblique
DejaVu Serif Mono	Book, Bold, Italic, Bold Italic, Condensed
Droid Sans	Regular, Bold, Mono
Droid Serif	Regular, Bold, Italic, Bold Italic
Inconsolata	Medium
Liberation Mono	Regular, Bold, Italic, Bold Italic
Liberation Sans (Narrow)	Regular, Bold, Italic, Bold Italic
Oxygen	Book, Mono
Times New Roman	Regular, Italic, Bold, Bold Italic
Ubuntu Mono	Regular, Italic, Bold, Bold Italic
Ubuntu	Regular, Light, Medium, Bold, Italic, ...
Ubuntu	..., Medium Italic, Light Italic
Verdana	Regular, Italic, Bold, Bold Italic

Table 6.1. Typefaces and their fonts used to train the Latin FLANN classifier.

by Ablavsky and Stevens in [24]. Their method was tested on two script classes as well and the scanned documents are quite noisy and degraded, see Figure 6.2. The success rates for Latin and Cyrillic scripts measured first on scanned documents and then on more degraded ones can be found in Table 6.3.

Figure 6.2. Examples from the dataset used by Ablavsky and Stevens in [24]. Quite similar to the TextSpotter segmentation output on which the script recognition method operates.

script	Latin	Cyrillic
scanned	86%	99%
scanned copies	51%	99%

Table 6.3. Success rates of a comparable method by Ablavsky and Stevens in [24] on scanned documents and the more degraded scanned copies of documents.

It is difficult to say which case in Table 6.3 is closer to TextSpotter use-case but we can see the success rates on Hebrew and Latin (see Table 6.2) are somewhere in the middle of these two cases. In other works reviewed in Chapter 2, the success rates were typically around 95-99%. We believe it is safe to say the FLANN-based method shows performance comparable to the state-of-art considering the more difficult conditions it operates in.

To be consistent with Chapter 4, we use similar confusion matrices to visualise the problems. The confusion matrix for Hebrew and Latin script classes can be examined in Figure 6.3. We can see the performance is somewhat lower in the case of Latin text.

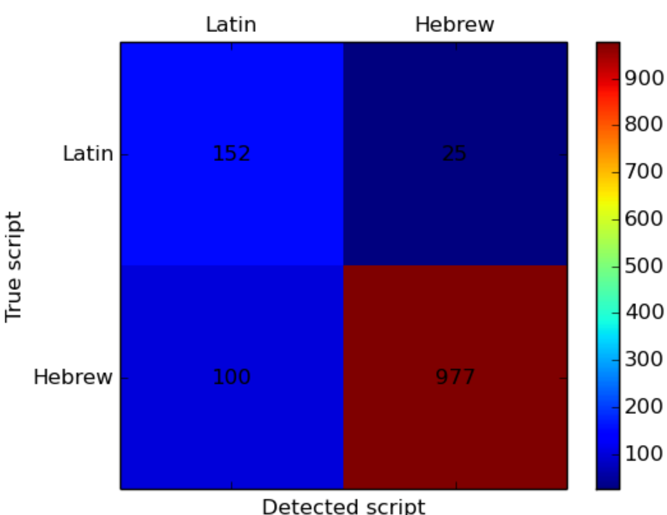


Figure 6.3. Confusion matrix of Latin and Hebrew script classification. The counts refer to the number of words.

To give illustration of incorrect decisions made by the classifier, we provide mask images of the Latin lines classified incorrectly as Hebrew in Figure 6.4 and images of Hebrew lines classified as Latin in Figure 6.5. In both cases, the mis-classified line usually has some challenging aspect such as considerable skew, non-character regions, or severely degraded character contours. Nevertheless, there are lines where the reason of failure is not directly apparent.

Figure 6.4. Mask images of most of the Latin lines classified as Hebrew. The proportions of the images were preserved.

As mentioned before, we expect the classifier to be more successful on longer lines. To validate this assumption, we collected the lengths of lines written in both Hebrew and Latin script for two separate classes—correctly classified and incorrectly classified—and we show histograms of their lengths in Figure 6.6.

We can notice a slight tendency of longer lines being classified correctly but comparison is difficult due to the much larger size of the correctly classified class. Therefore, we provide the same histograms but normalized to form a probability density in Figure 6.7, where the tendency is much more apparent.

Figure 6.5. Mask images of some of the Hebrew lines classified as Latin. The proportions of the images were preserved.

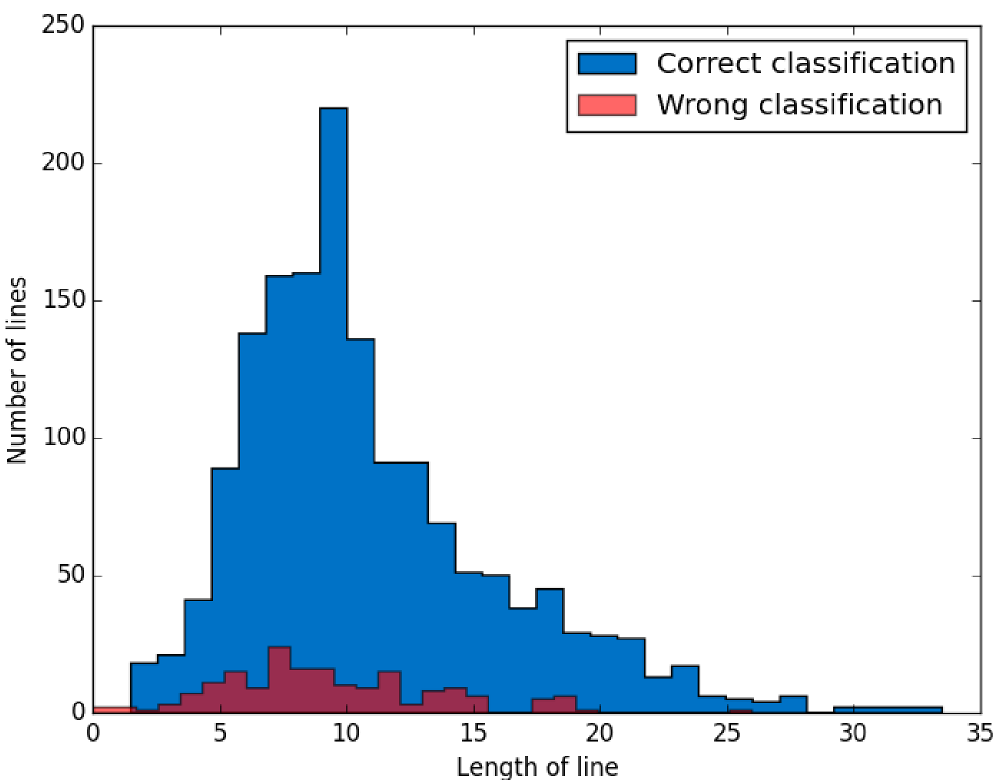


Figure 6.6. Histograms showing the lengths of text lines for the correctly and incorrectly classified lines for both Hebrew and Latin lines.

Another observation that can be made is that text lines comprising of more than 15 characters are quite rare. Although natural for our use-case of text-in-the-wild, it makes our task more difficult. Effectively, the recognition is done somewhere between *line level* and *word level*.

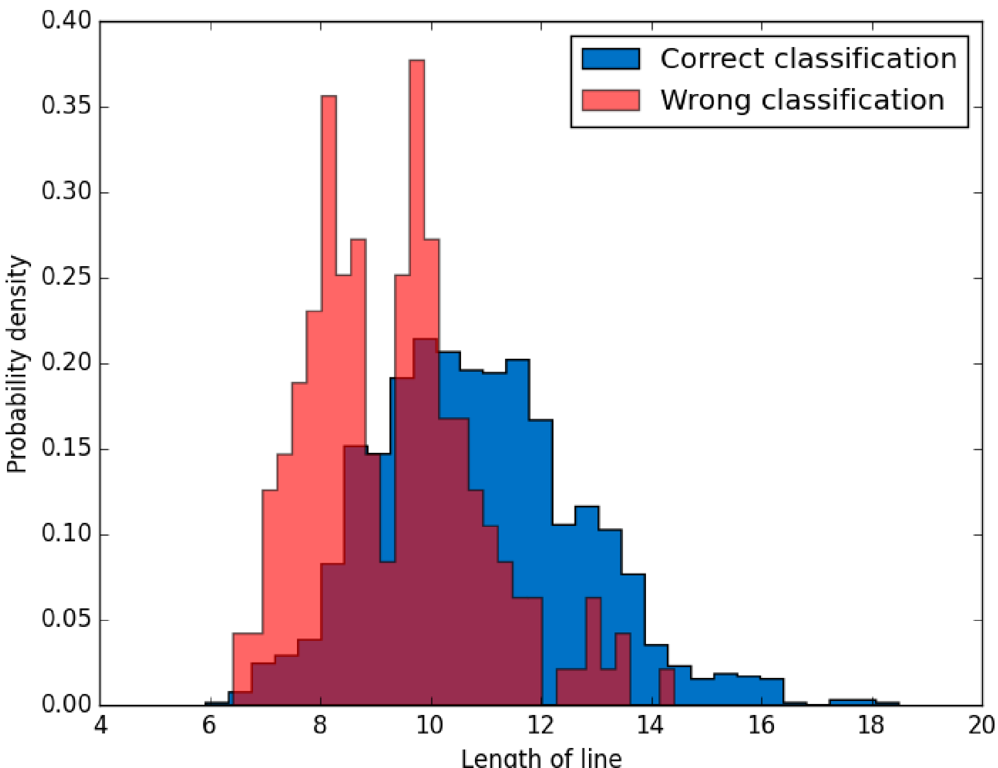


Figure 6.7. The same histograms but normed to form a probability density. The values have been smoothed using moving average to make it more readable. Therefore, short lines up to 5 characters are not visible.

6.2 Bigram Script Recognition

Script recognition methods using structural features such as the one proposed in Section 6.1 might be unsuitable for very similar scripts which share many characters. E.g. there are Cyrillic words that have only one character not present in the Latin script. In this case, the border between *script recognition* and *language recognition* becomes somewhat blurred.

Therefore, we experimented with applying a technique commonly used in language identification in our context of script recognition. A classifier using *n-gram* frequencies is proposed in this section. The usage of n-gram frequencies for language classification is well-established in the field of pattern recognition not only for text processing [32] but also has been generalized for speech recognition [33].

An n-gram is a sequence of consecutive primitives (letters) of length n in a document—be it a paragraph or a text line. E.g. the word "hello" has 4 bigrams: "he", "el", "ll", and "lo". In the case of speech recognition, n-grams are sequences of phonemes (instead of letters) in speech (instead of text). See [32–33] for more details about n-grams.

We can use the FLANN classifiers trained in Section 6.1 to obtain hypothesis of text in each language by choosing the hypothesis with the highest confidence for each character. The text will contain more errors than the final output of TextSpotter because at this stage, we do perform segmentation or construct the adjacency graph of characters (on which dynamic programming is run to obtain the sequence with highest score).

A desirable property of bigrams is their tolerance to noise in the document. It is said in [32] that retrieval systems based on n-grams preserve their performance up to 30% error rate. Due to this, n-grams are a viable option for our task even under the previously explained conditions of noisy text.

The first step in construction of the classifier is to obtain the frequencies of the n-grams for every language we want to recognize. We chose to use bigrams, n-grams of length 2. The benefit of adding also trigrams or four-grams was deemed questionable since we deal with such a noisy text, bigrams are short and therefore less bigrams are affected by one incorrectly recognized character.

The relative frequencies of all possible bigrams in a language are estimated from a large corpus of text. We do realize that we are solving the script recognition problem and that very different languages might be written in one script—this is most pressing for the Latin script, other scripts tend to be specific to only a handful of similar languages. For instance, only Hebrew, Yiddish, Ladino and Judeo-Arabic languages are written in the Hebrew script. However for the testing purposes, we decided to use a corpus in one distinctive language of each script. We believe this is a viable option since some properties are preserved across the languages, e.g. a vowel is more likely to be followed by a consonant than another vowel.

The probability of an occurrence of the bigram xy in a script S is estimated from the corpus as

$$P(xy|S) = \frac{\text{count}(xy)}{n},$$

where n is the total number of all bigrams in the corpus. Analogically, we estimate the probability of the letter x in script S as

$$P(x|S) = \frac{\text{count}(x)}{n},$$

where n is the total number of characters in the corpus. From these relations, we derive the *maximum likelihood* estimation of the conditional probability of the letter y following letter x as

$$P(y|x, S) = \frac{P(xy|S)}{P(x|S)}.$$

Then the likelihood that a text line L is in script S is equal to the product of the conditional probabilities of all letters y given the preceding letter x on the line

$$\mathcal{L}(L|S) = \prod_{xy \in L} P(y|x, S).$$

The classifier chooses the script that maximises the log-likelihood for numerical reasons. In other words

$$S^* = \operatorname*{argmax}_S \mathcal{L}(L|S) = \operatorname*{argmax}_S \sum_{xy \in L} \log(P(y|x, S)).$$

6.2.1 Implementation Details of Bigram Classifier

The details about the corpora of Hebrew and Latin text used for training the probabilities are shown in Table 6.4. The English corpora was collected from news commentaries and was originally used for the purposes of ACL 2013 Workshop on Statistical Machine

script	Latin	Hebrew
source	ACL Workshop	MILA research center
licence	Free	Free for research purposes
URL	http://is.gd/DQsDG4	http://is.gd/gKOosB
# of words	1 510 590	1 851 605

Table 6.4. Details about Hebrew and Latin corpora.

Translation. The Hebrew corpus is a collection of news and articles from the Arutz7 website combined with Spoken Israeli Hebrew corpus, both made available for free for research purposes by the MILA center in [34].

One of the difficulties when using bigrams is that their probability distribution is often very sparse. There are bigrams not present even in large corpora but that can appear in the evaluated text—we would estimate the probability of such bigram to be 0 which is obviously not the case. This can lead to problems during evaluation, as explained in [35]. The most straightforward solution is to add 1 to the occurrences of all bigrams. Although simple, we found this solution to enhance the performance.

While conducting experiments, we found that using only the probability $P(xy|S)$ actually shows better performance than using $P(y|x, S)$, therefore we take advantage of this finding.

6.2.2 Evaluation of Bigram Classifier

Our standard tools were used to measure the performance. The success rates for Hebrew and Latin can be found in Table 6.5 and the confusion matrix can be examined in Figure 6.8. We can see from the success rates that the bigram classifier perfoms better on Latin text lines then the Hebrew textlines.

script	Latin	Hebrew
success rate	90.34%	84.30%
count	176	1083

Table 6.5. Success rates of Hebrew and Latin script classification using bigrams.

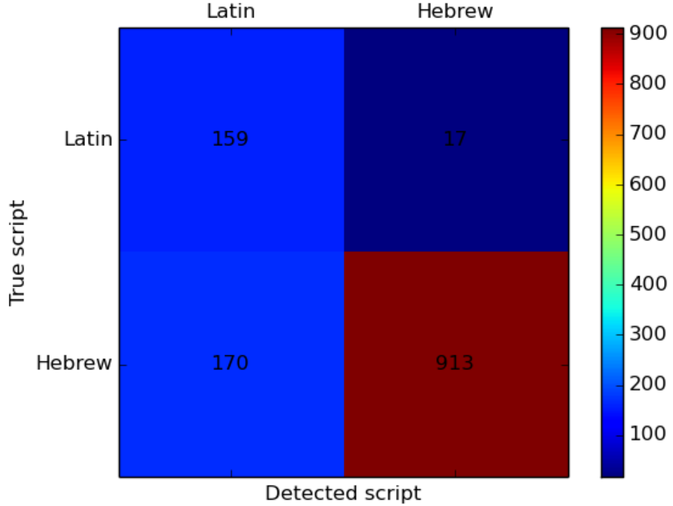


Figure 6.8. Confusion matrix of Latin and Hebrew script classification using bigrams. The counts refer to the number of words.

We showed that bigram-based classification can be used in the context of script recognition even if we use a corpus from one particular language and the signs are in a variety of languages. The classifier shows poorer performance than the purely FLANN-based one on the same set of scripts. In case of more structurally similar scripts (e.g. Latin and Cyrillic), the situation may be reversed. Nevertheless, we combine the two approaches in the next section to achieve superior performance.

6.3 Combining Proposed Script Classifiers

We have presented 2 different classifiers, the first one in Section 6.1 uses structural features of the individual characters and classifies according to the sum of confidences of each detected character on the line. The second one in Section 6.2 takes advantage of different probabilities of two letters appearing consecutively in the text. In this section, we experimented with a combination of these approaches.

The FLANN-based classifier assigns a confidence for each script to the text line. We take advantage of this. The confidences can be compared and used as a measure of the classification ambiguity. If the classification is too ambiguous, we switch from the FLANN-based classifier to the bigram classifier.

The ambiguity of the classification is defined as the difference between two highest line confidences accross the supported scripts. In case of only 2 scripts S_1, S_2, a line classification is considered ambiguous if

$$\left|\mathrm{lineConf}(S_1, L) - \mathrm{lineConf}(S_2, L)\right| < t,$$

where t is a threshold. We set the threshold empirically to $t = 0.05$.

This scheme proved to be more effective than the individual classifiers. The success rates for both classes have improved slightly as we can see in Table 6.6 and again in the corresponding confusion matrix in Figure 6.9. However, the gain was not as significant as we had hoped for.

script	Latin	Hebrew
success rate	93.22%	91.57%
count	177	1079

Table 6.6. Success rates of Hebrew and Latin script classification using the combination of the FLANN-based classifier and the bigram classifier.

This scheme would be more useful in situations in which we knew the domain of the text beforehand. Then the corpus used to train the bigram classifier could be more specific and provide superior performance. For instance, we could create a corpus mainly from city names and other common words occurring often in these circumstances if we knew the system was used to to recognize only traffic signs in cities. However, we do not presume to have such prior knowledge about the text in the scene at this stage. Therefore, we had not explored this possibility.

The advantage of the bigram classifier is that it can be easily used to recognize also the language of the text. A different synthesis could be achieved by using the FLANN-based classifier to classify the script and then use the bigram classifier trained with different corpora for each language of the script to classify the language.

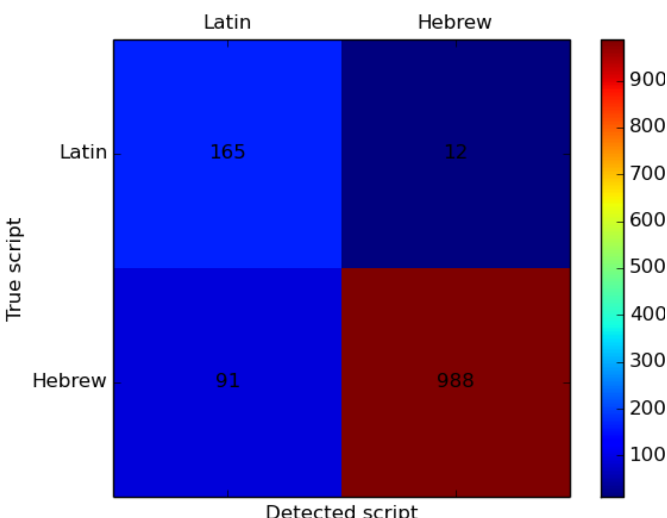


Figure 6.9. Confusion matrix of Latin and Hebrew script classification using the combined of the FLANN-based classifier and the bigram classifier. The counts refer to the number of words.

Chapter 7
Conclusion

In this thesis, we have followed the given guidelines to explore and improve usability of TextSpotter in scenes with text in multiple scripts. In order to do so, state-of-the-art has been examined in Chapter 2.

To learn about the possible challenges and also the specifics of different scripts, we have reviewed the literature about recognition of non-Latin scripts such as the Middle Eastern and CJK scripts in Section 2.1.

Script recognition was reviewed in Section 2.2 to be able to propose an effective method of script recognition in TextSpotter. We struggled with the lack of literature on the topic in the context of text-in-the-wild and reviewed the methods designed for situations closest to ours (historical and degraded documents).

An original dataset compiled from 100 Wikimedia images and annotated for the purposes of evaluation is described in Chapter 3. The dataset contains mainly Hebrew and Latin inscriptions and the images are of similar nature to tourist photographs or Street View scenes.

This dataset was primarily used to evaluate which modules of TextSpotter are sensitive to changing the script in Chapter 4. We showed from several angles that the problems arise not in the lower stages of extrema regions detection or text line formation (which are general enough for a wide range of scripts) but in the OCR stage and are caused by high similarity of some groups of letters in the script or by other fundamental difference from the Latin script. Either by breaking the assumption of one letter being connected component in Hebrew or by the unusual composition of consonant pairs in the Kannada script.

As we mainly concentrated on the Hebrew script, we addressed the found issues and improved the performance of TextSpotter on Hebrew significantly in Section 5 from 236 perfectly recognized words to 321 out of 1147 present in the dataset.

Finally in Chapter 6, we proposed two different methods for script recognition operating on text line level (and often effectively on word level due to our dataset containing very short lines). We implemented a scheme of one k-NN classifier for each script which is used in the literature but improved on its decision strategy. The first strategy was based on the confidence of the k-NN classifiers and the second strategy on the likelihood of bigram occurrences—a technique often used in document analysis for language classification but not for script recognition as far as we are aware. Both strategies show state-of-the-art performance as argued in Section 6.1.1. A combination of these approaches was tested and achieved slightly better performance.

TextSpotter is now able to recognize text in different scripts within one image and the script classifier is trained only from a set of synthetic fonts in the same way as character classifiers or from the fonts and a corpus of text when using the bigram-based classifier. Consequently, the script recognition method preserves the desirable generality of TextSpotter and does not work only on a predefined set of scripts like most of the classifiers in the literature.

7.1 Further Improvements

We propose following improvements or extensions to the work presented in this thesis that occured to us and were not explored due to the time constraints or being out of the scope of the guidelines.

Generalize the method of merging components from Section 5.1.2 to the much more complex case of CJK scripts. This would be the first step to make TextSpotter usable for a new set of scripts that, at the moment, is too different to be recognized by its methods.

Perform language classification using the bigram script classifier. Before this thesis, TextSpotter had no script classification, therefore, no language classification either. Language detection may be useful especially for the Latin script that is used for many languages, e.g. to automatically extend the character set with language specific characters often present in Slavic languages or Turkish.

A classifier sensitive to finer details in the characters would improve performance for some scripts. Although there are characters similar to each other in the Latin script, the tested Hebrew and Kannada scripts have whole groups of very similar characters which makes the performance of the character classifier poorer.

A more specialized script classifier could be developed if the set of scripts was known beforehand using a set of features suitable to discriminate between the scripts. This approach seems to be dominant in the literature. While it could probably provide better performance, our classifier would still be needed due to its generality.

An application specific corpora for the bigram classifier. If the nature of the text in the scenes is known beforehand, e.g. when recognizing traffic signs, the corpora used to train the bigram classifier could be compiled accordingly. This would most likely improve the performance.

References

[1] L. Neumann, and J. Matas. *A method for text localization and recognition in real-world images.* In: Ron Kimmel, Reinhard Klette, and Akihiro Sugimoto, eds. *ACCV 2010: Proceedings of the 10th Asian Conference on Computer Vision.* Heidelberg, Germany: Springer, 2010. 2067–2078.

[2] L. Neumann, and J. Matas. *Text Localization in Real-World Images Using Efficiently Pruned Exhaustive Search.* In: *Document Analysis and Recognition (ICDAR), 2011 International Conference on.* IEEE Computer Society Offices, 2001 L Street N.W., Suite 700 Washington, DC 20036-4928, United States: IEEE Computer Society Conference Publishing Services, 2011. 687–691. ISBN 1520-5363.

[3] L. Neumann, and J. Matas. *Scene Text Localization and Recognition with Oriented Stroke Detection.* In: *2013 IEEE International Conference on Computer Vision (ICCV 2013).* California, US: IEEE, 2013. 97–104. ISBN 978-1-4799-2839-2.

[4] Q. Ye, and D. Doermann. Text Detection and Recognition in Imagery: A Survey. *Pattern Analysis and Machine Intelligence, IEEE Transactions on.* 2014, PP (99), 1–1. DOI 10.1109/TPAMI.2014.2366765.

[5] A. Belaïd, and M.I. Razzak. *Middle Eastern Character Recognition.* 2014. `http://dx.doi.org/10.1007/978-0-85729-859-1_13.`

[6] Y. Chang, D. Chen, Y. Zhang, and J. Yang. An image-based automatic Arabic translation system. *Pattern Recognition.* 2009, 42 (9), 2127–2134. DOI 10.1016/j.patcog.2008.10.031.

[7] M. Ben Halima, H. Karray, and A.M. Alimi. *AVOCR: Arabic video OCR.* In: *I/V Communications and Mobile Network (ISVC), 2010 5th International Symposium on.* 2010. 1–4.

[8] Y.M. Alginahi. A survey on Arabic character segmentation. *International Journal on Document Analysis and Recognition (IJDAR).* 2013, 16 (2), 105–126. DOI 10.1007/s10032-012-0188-6.

[9] A. Krayem, N. Sherkat, L. Evett, and T. Osman. *Holistic Arabic Whole Word Recognition Using HMM and Block-Based DCT.* In: *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on.* 2013. 1120–1124.

[10] B. Epshtein, E. Ofek, and Y. Wexler. *Detecting text in natural scenes with stroke width transform.* In: *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on.* 2010. 2963–2970.

[11] I. Rabaev, O. Biller, J. El-Sana, K. Kedem, and I. Dinstein. *Case Study in Hebrew Character Searching.* In: *Document Analysis and Recognition (ICDAR), 2011 International Conference on.* 2011. 1080–1084.

[12] S. Setlur, and Z. Shi. Asian Character Recognition. *Handbook of Document Image Processing and Recognition.* 2014, 459–486.

[13] X. Chen, J. Yang, J. Zhang, and A. Waibel. Automatic detection and recognition of signs from natural scenes. *Image Processing, IEEE Transactions on*. 2004, 13 (1), 87–99. DOI 10.1109/TIP.2003.819223.

[14] L. Xu, H. Nagayoshi, and H. Sako. *Kanji Character Detection from Complex Real Scene Images based on Character Properties*. In: *Document Analysis Systems, 2008. DAS '08. The Eighth IAPR International Workshop on*. 2008. 278–285.

[15] C. Yao, X. Bai, W. Liu, Yi Ma, and Z. Tu. *Detecting texts of arbitrary orientations in natural images*. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. 2012. 1083–1090.

[16] M. Iwamura, T. Kobayashi, and K. Kise. *Recognition of Multiple Characters in a Scene Image Using Arrangement of Local Features*. In: *Document Analysis and Recognition (ICDAR), 2011 International Conference on*. 2011. 1409–1413.

[17] D. Ghosh, T. Dube, and A.P. Shivaprasad. Script Recognition #x02014;A Review. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*. 2010, 32 (12), 2142–2161. DOI 10.1109/TPAMI.2010.30.

[18] U. Pal. *Language, Script, and Font Recognition*. 2014.
`http://dx.doi.org/10.1007/978-0-85729-859-1_9`.

[19] J. Hochberg, P. Kelly, T. Thomas, and L. Kerns. Automatic script identification from document images using cluster-based templates. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*. 1997, 19 (2), 176–181. DOI 10.1109/34.574802.

[20] D. Brodić, Z.N. Milivojević, and Č.A. Maluckov. An approach to the script discrimination in the Slavic documents. *Soft Computing*. 2014, 1–11. DOI 10.1007/s00500-014-1435-1.

[21] P.K. Aithal, G. Rajesh, D.U. Acharya, M. Krishnamoorthi, and N.V. Subbareddy. *Script Identification for a Tri-lingual Document*. Communications in Computer and Information Science. 2011.
`http://dx.doi.org/10.1007/978-3-642-19542-6_82`.

[22] U. Pal, S. Sinha, and B.B. Chaudhuri. *Multi-script line identification from Indian documents*. In: *Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on*. 2003. 880–884.

[23] S. Haboubi, S.S. Maddouri, and H. Amiri. *Separation between Arabic and Latin Scripts from Bilingual Text Using Structural Features*. Communications in Computer and Information Science. 2011.
`http://dx.doi.org/10.1007/978-3-642-22247-4_12`.

[24] V. Ablavsky, and M.R. Stevens. *Automatic feature selection with applications to script identification of degraded documents*. In: *Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on*. 2003. 750–754.

[25] I. Kononenko. *Estimating attributes: analysis and extensions of RELIEF*. In: *Machine Learning: ECML-94*. 1994. 171–182.

[26] D. Dhanya, A.G. Ramakrishnan, and P.B. Pati. Script identification in printed bilingual documents. *Sadhana*. 2002, 27 (1), 73–82. DOI 10.1007/BF02703313.

[27] R. Rani, R. Dhir, and G.S. Lehal. *Script Identification of Pre-segmented Multi-font Characters and Digits*. In: *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on*. 2013. 1150–1154.

[28] S. Jetley, K. Mehrotra, A. Vaze, and S. Belhe. *Multi-script Identification from Printed Words*. Lecture Notes in Computer Science. 2014.
http://dx.doi.org/10.1007/978-3-319-11758-4_39.

[29] L. Neumann, and J. Matas. Efficient Scene Text Localization and Recognition with Local Character Refinement. *arXiv preprint arXiv:1504.03522*. 2015,

[30] 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*. 2009.

[31] M. Muja, and D.G. Lowe. *Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration.* In: *International Conference on Computer Vision Theory and Application VISSAPP'09)*. INSTICC Press, 2009. 331–340.

[32] S. Gadri, A. Moussaoui, and L. Belabdelouahab-Fernini. *Language identification: A new fast algorithm to identify the language of a text in a multilingual corpus.* In: *Multimedia Computing and Systems (ICMCS), 2014 International Conference on*. 2014. 321–326.

[33] H. Li, B. Ma, and K.A. Lee. Spoken Language Recognition: From Fundamentals to Practice. *Proceedings of the IEEE*. 2013, 101 (5), 1136–1159. DOI 10.1109/JPROC.2012.2237151.

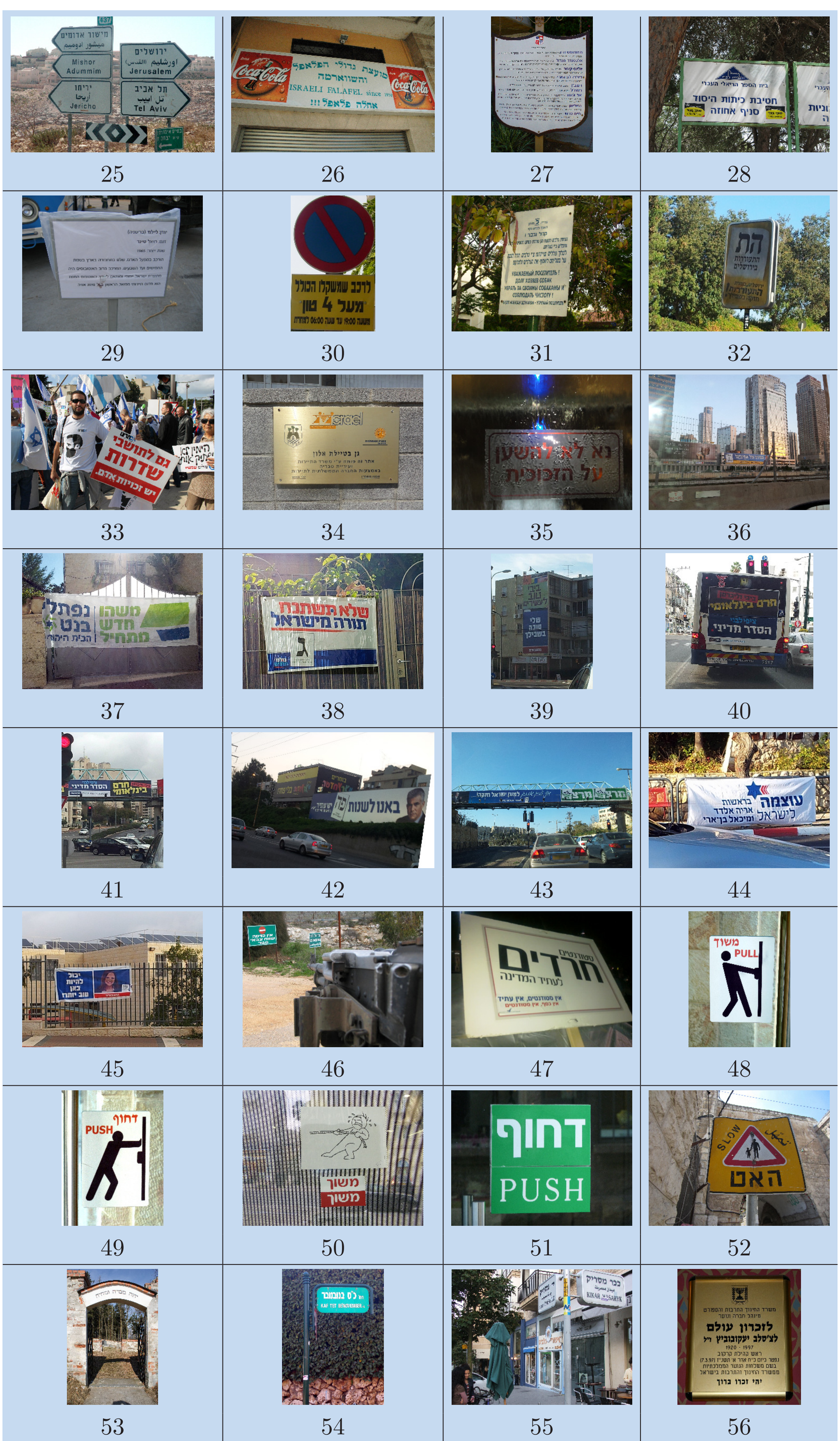
[34] A. Itai, and S. Wintner. Language resources for Hebrew. *Language Resources and Evaluation*. 2008, 42 (1), 75–98.

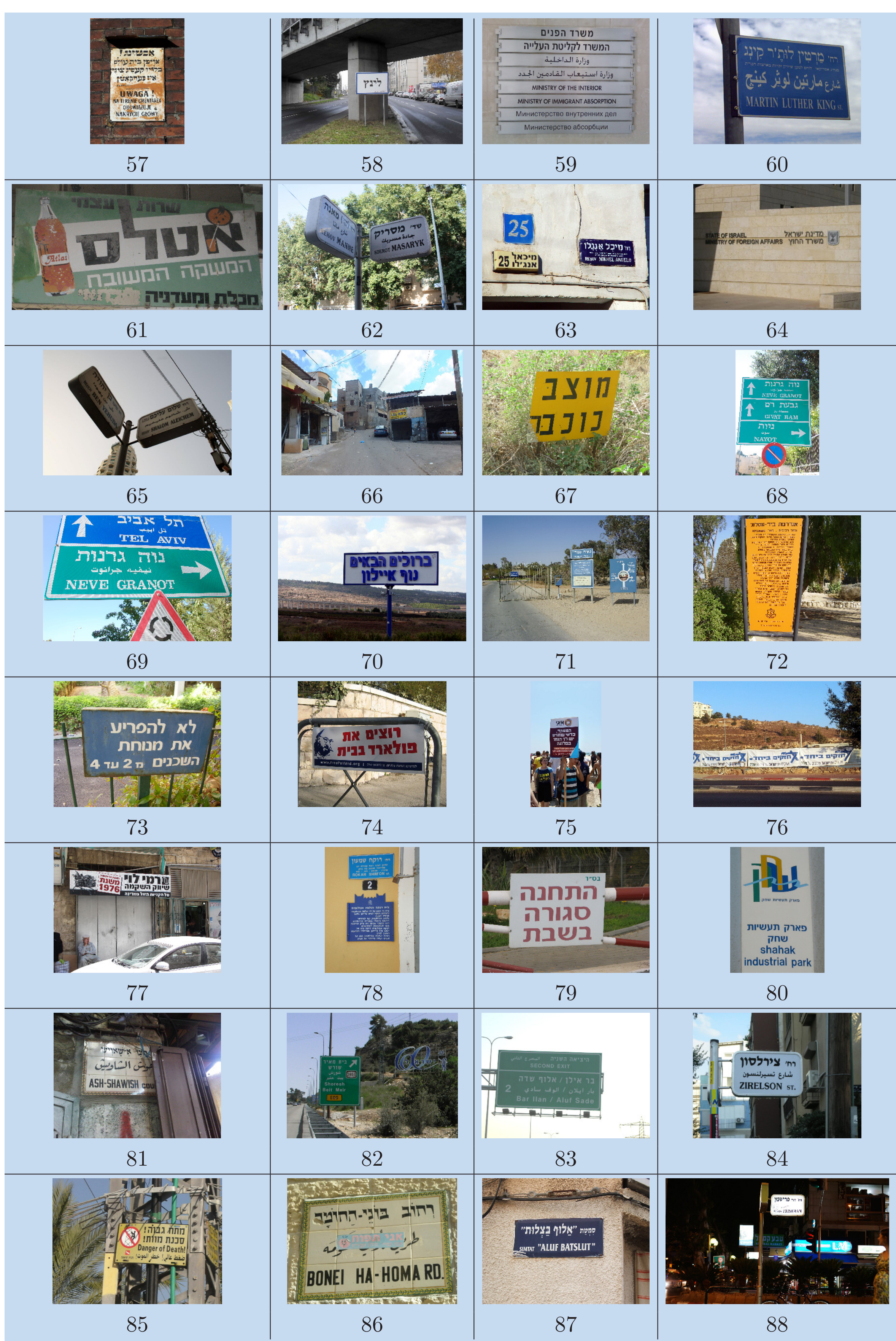
[35] S.F. Chen, and J. Goodman. *An Empirical Study of Smoothing Techniques for Language Modeling.* In: *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*. Stroudsburg, PA, USA: Association for Computational Linguistics, 1996. 310–318.
http://dx.doi.org/10.3115/981863.981904.

Appendix A
Contents of the DVD

```
/
├── corpora/ ............................. corpora used to train bigram classifier
│   ├── Latin.txt ............................................... English corpus
│   ├── Hebrew.txt ............................................... Hebrew corpus
│   └── sources.txt ................................... sources of both corpora
├── hebrew_set/ ............. original Hebrew and Latin dataset with annotations
├── trunk/ ..... complete TextSpotter revision containing all the work in this thesis
│   ├── Sources/ ............................................... actual sources
│   ├── SupportFiles/ ........... Python scripts for generating the training images
│   └── testData/
│       └── config/ ................. configuration files for e.g. Hebrew or Kannada
└── tex/ ............................. TeX source codes for the text of the thesis
    ├── fig/ ........................................... figures used in this text
    └── hollmann_dp.pdf ........... electronic version of the text compiled to PDF
```

Appendix B
Hebrew Dataset Thumbnails


Mishor Adummim
Jerusalem
Jericho
Tel Aviv
ISRAELI FALAFEL
Coca-Cola
משוך
PULL
דחוף
PUSH
משוך
משוך
דחוף
PUSH
SLOW
האט
חרדים
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56


57
58
59
60
MINISTRY OF THE INTERIOR
MINISTRY OF IMMIGRANT ABSORPTION
MARTIN LUTHER KING ST.
61
62
63
64
65
66
67
68
69
70
71
72
TEL AVIV
NEVE GRANOT
73
74
75
76
77
78
79
80
shahak
industrial park
81
82
83
84
ASH-SHAWISH
ZIRELSON ST.
85
86
87
88
Danger of Death!
BONEI HA-HOMA RD.
"ALUF BATSLUT"

WELCOME
89
90
91
92
רח' יבנה
REHOV YAVNE
התנועה
ברוכים הבאים לחיפה
WELCOME TO HAIFA
93
94
95
96
רח' בן סירא
شارع بن سيرا
BEN SIRA ST.
מושב-לצים
MOSHAV LETSIM
97
98
99
100

Appendix C
Hebrew Dataset Details

ID	Image name	Link	Licence	Words
1	amir_peretz_election	http://is.gd/g0T811	CC-BY-SA-3.0	10
2	amshalem_sign_jerusalem	http://is.gd/YqXxPN	CC-BY-SA-3.0	7
3	Anti-Zionism_in_Israel	http://is.gd/GM7ZjC	CC-BY-SA-3.0	6
4	area_a	http://is.gd/i3jrxv	CC-BY-SA-4.0	19
5	armenian_church_in_er2	http://is.gd/qhRMG6	CC-BY-SA-3.0	5
6	assa_kadmoni_str._in_tel_aviv	http://is.gd/SdGG4w	CC-BY-SA-3.0	25
7	Asyrian_convent_road_sign	http://is.gd/RHwB6U	CC-BY-SA-4.0	4
8	beach_of_rlz_10_-_swimming	http://is.gd/Mbu5G9	CC-BY-SA-3.0	6
9	beach_of_rlz_11_-_danger_sign	http://is.gd/4Mt355	CC-BY-SA-3.0	2
10	benamram	http://is.gd/6iu7jH	Attribution	3
11	Benny_Sela	http://is.gd/aC4W5K	CC-BY-SA-3.0	5
12	bethelsign	http://is.gd/CJ6Ra9	CC-BY-SA-3.0	4
13	deleksonol	http://is.gd/MZX7kC	Public Domain	1
14	detail_of_little_windows	http://is.gd/cpLMMj	CC-BY-SA-3.0	3
15	dizengof_center_tel_aviv	http://is.gd/YIsofu	CC-BY-SA-4.0	6
16	eged-says-no-to-passenger	http://is.gd/yztnsB	CC-BY-SA-3.0	21
17	embassy_tel_aviv_6926	http://is.gd/E0FIAp	CC-BY-3.0	4
18	emh_img_6482	http://is.gd/nkFzKs	CC-BY-SA-3.0	20
19	Ezuz	http://is.gd/DCVhCg	CC-BY-SA-3.0	5
20	gllsprng_090	http://is.gd/L4qTW1	CC-BY-SA-3.0	1
21	haifa,_israel_bilingual_sign	http://is.gd/nxZIfI	CC-BY-SA-2.0	6
22	hanukkah_bus	http://is.gd/v2ShDb	CC-BY-SA-3.0	4
23	happy_hour	http://is.gd/TNV7Dd	Public Domain	5
24	hebrew-_do_not_hate_your_brother	http://is.gd/Cy4hyl	CC-BY-SA-3.0	11
25	hebrew_arabic_english_road_signs	http://is.gd/PzcIHQ	CC-BY-2.0	11
26	hebrew_english_grocery_sign	http://is.gd/jWs7Sp	CC-BY-2.0	7
27	hebrew_legend_wall_paintings	http://is.gd/QwIhqm	CC-BY-SA-2.0	207
28	hebrew_reali_school_of_haifa	http://is.gd/jIGp9Q	CC-BY-SA-3.0	19
29	Hebrew_sign_at_auto_display	http://is.gd/JO7SCE	Public Domain	36
30	hebrewnoparkingtrucksign	http://is.gd/un0zk4	Public Domain	5
31	hebrewrussiansignholon	http://is.gd/t2veTj	Public Domain	32
32	hitorerut_party_2013	http://is.gd/3SHr1l	CC-BY-SA-3.0	5
33	HRM_TLV_091211_177	http://is.gd/UorWnF	CC-BY-SA-3.0	8
34	israel_batch_2_(163)	http://is.gd/HOkPTr	Public Domain	18
35	israel_batch_3_(443)	http://is.gd/4vreyH	Public Domain	5
36	israel_elections_2012_amsalem	http://is.gd/EXqt76	CC-BY-SA-3.0	3

Table C.1. Details about dataset images.

ID	Image name	Link	Licence	Words
37	israel_elections_2012_bennet	http://is.gd/3nPT4Q	CC-BY-SA-3.0	7
38	israel_elections_2012_gimmel	http://is.gd/p0PUFv	CC-BY-SA-3.0	4
39	Israel_Elections_2012_Haavoda1	http://is.gd/yBzljw	CC-BY-SA-3.0	7
40	israel_elections_2012_hatnua1	http://is.gd/aVeal0	CC-BY-SA-3.0	15
41	israel_elections_2012_hatnua2	http://is.gd/5t9Sa2	CC-BY-SA-3.0	12
42	israel_elections_2012_lapid	http://is.gd/88cpc1	CC-BY-SA-3.0	10
43	Israel_Elections_2012_Meretz	http://is.gd/Q7N5bx	CC-BY-SA-3.0	11
44	israel_elections_2012_ozma1	http://is.gd/sa2DeW	CC-BY-SA-3.0	8
45	israel_elections_2013_shelli	http://is.gd/RGVaUu	CC-BY-SA-3.0	5
46	israel_lebanon_border	http://is.gd/ZRGsBb	CC-BY-SA-3.0	5
47	Israel_rent_protest_2011	http://is.gd/ssSsUM	CC-BY-SA-2.0	8
48	jerusalem_washington_pull	http://is.gd/IF4tqI	CC-BY-SA-4.0	1
49	jerusalem_washington_push	http://is.gd/opEf45	CC-BY-SA-4.0	1
50	jerusalem_hebrew_pull_sign	http://is.gd/wGu0el	CC-BY-SA-4.0	3
51	Jerusalem_hebrew_Push_sign	http://is.gd/s4XWIW	CC-BY-SA-4.0	1
52	jerusalem_slow!_(6035877073)	http://is.gd/KN9C0h	CC-BY-SA-2.0	1
53	Jewish_cemetery_in_Vodňany	http://is.gd/gUrHQu	Public Domain	3
54	kaf_tet_benovember_st	http://is.gd/gHvFtx	CC-BY-SA-2.5	4
55	Kikar_Masaryk_Tel_Aviv_6929	http://is.gd/cdYfnA	CC-BY-SA-3.0	2
56	krakow_synagoga_tempel	http://is.gd/oGnrkg	CC-BY-SA-4.0	34
57	kraków_1744	http://is.gd/Flq1pF	CC-BY-SA-3.0	8
58	kulturhauptstadt_linz_hebrew	http://is.gd/iQvtpq	CC-BY-3.0	1
59	languages_of_israel	http://is.gd/vHZPJT	CC-BY-SA-3.0	5
60	lutherking_street_jerusalem	http://is.gd/PugDmS	CC-BY-3.0	4
61	Makolet	http://is.gd/dD1Mnt	CC-BY-SA-3.0	4
62	Masaryk_street_Tel_Aviv_1015	http://is.gd/Aq9u6z	CC-BY-SA-3.0	2
63	MichelangeloStreet	http://is.gd/hemLSA	CC-BY-3.0	18
64	Ministry_of_foreign_affairs	http://is.gd/9wXADn	CC-BY-SA-3.0	4
65	multilingualism_in_israel	http://is.gd/yMUsnl	CC-BY-SA-3.0	2
66	nabi_elias	http://is.gd/PqVLpY	CC-BY-SA-3.0	5
67	naharayim_memorial_15	http://is.gd/UhiDfK	Public Domain	3
68	Nayot_1	http://is.gd/qU1d0V	Public Domain	7
69	nayot_3	http://is.gd/ZR9rBM	Public Domain	4
70	nof_ayalon1	http://is.gd/nkUJhZ	CC-BY-SA-3.0	4
71	pikiwiki_israel_18676_entrance	http://is.gd/MvYP4K	CC-BY-2.5	2
72	pikiwiki_israel_5010_bir_asluj	http://is.gd/0gd7XZ	CC-BY-2.5	146
73	pikiwiki_israel_5784_economy	http://is.gd/g55MQR	CC-BY-2.5	7
74	Polard2	http://is.gd/f9o60B	Attribution	7
75	pride_tel_aviv_2014	http://is.gd/cPCtJS	CC-BY-SA-3.0	12
76	Protective_Edge_Patriotic	http://is.gd/Iplu9L	CC-BY-SA-4.0	6
77	rami_levi_original_store	http://is.gd/8zBF2I	CC-BY-SA-3.0	11
78	Rivka_and_Shlomo_Abulafia	http://is.gd/QWXJkm	Public Domain	71
79	ShabatSign	http://is.gd/EZ14pT	Public Domain	4
80	shahak_industrial_park	http://is.gd/BWqiy9	CC-BY-SA-4.0	5
81	shawish_neighbourhood	http://is.gd/9G7nTV	CC-BY-3.0	2

Table C.2. Details about dataset images.

ID	Image name	Link	Licence	Words
82	shelet_60	`http://is.gd/NKL8xF`	CC-BY-SA-3.0	3
83	signsinisrael1	`http://is.gd/leVaBr`	Public Domain	6
84	signsinisrael2	`http://is.gd/leVaBr`	Public Domain	2
85	SignsInIsrael3	`http://is.gd/leVaBr`	Public Domain	4
86	signsinisrael4	`http://is.gd/leVaBr`	Public Domain	3
87	simtat_aluf_batslut	`http://is.gd/nxzVz2`	CC-BY-3.0	3
88	Tel_Aviv_44105	`http://is.gd/DjBjRT`	CC-BY-SA-2.0	5
89	tel_aviv_dog_parka	`http://is.gd/sIWsA5`	CC-BY-3.0	6
90	tel_aviv_yaffo_sign	`http://is.gd/AMnNXz`	CC-BY-3.0	4
91	terra_sancta_sign_(jerusalem)	`http://is.gd/h37cDY`	CC-BY-SA-3.0	33
92	veradim_street	`http://is.gd/8G8p75`	Public Domain	2
93	yad_sarah_signpost	`http://is.gd/GsfQ6o`	CC-BY-SA-3.0	6
94	yavne_street_tel_aviv	`http://is.gd/Ve1wh2`	CC-BY-SA-3.0	2
95	כובע_על_הראש	`http://is.gd/gpNXT9`	CC-BY-2.0	5
96	ברוכים_הבאים_לחיפה	`http://is.gd/G6mW0i`	CC-BY-SA-3.0	3
97	זירת_הפשע	`http://is.gd/b4iHou`	CC-BY-2.0	3
98	מד_גשם_בשמורת_מצוק_העתקים	`http://is.gd/0luRFA`	CC-BY-2.0	17
99	מושב_לצים-שלט_רחוב	`http://is.gd/Ul0WA8`	CC-BY-2.0	2
100	20110223_Israel_0268_Jerusalem	`http://is.gd/JrryPH`	CC-BY-SA-2.0	7

Table C.3. Details about dataset images.