Book Identification From Bookshelf Images

Pavel Rubinson

***Abstract*—While many websites provide review-based book rankings, brick and mortar stores do not provide easy access to such data. In this project, we develop a system for identifying books from a photo of a bookshelf. This information can then be used to extract any required book-related data from the internet, thus providing the clients of brick-and-mortar bookstores with the same kind of information as is available online. Our system extracts book spines from the photo, segments the text from the background, and groups individual characters into words. An implementation is also provided for training a classifier for character recognition. For segmentation, we propose and implement a simple method suitable for most book spines. For the grouping of characters into words, we develop a new algorithm, suitable for text with a variety of orientations and sizes. Our system achieves successful spine separation in around 94% of cases, successful segmentation of text in 77% of spines, and correct word grouping for 74% of successfully segmented words.**

I. Introduction

Websites such as Amazon.com and Goodreads.com, among many others, provide readers with useful information prior to buying a book, such as user rankings and user reviews. With sufficient number of reviewers, such information allows for smarter purchasing decisions. However, clients of stone-and-mortar bookstores, while having the advantage of physically seeing a book, do not normally have access to such information. In this project we implement a computer vision system for extracting book names from a photo of a bookshelf. Those book names can then be used in internet queries to extract the same kind of information as provided by the above-mentioned websites, in a quick and timely manner.

Figure 1: An example bookshelf photo.

A. Goals

Given a photo of a bookshelf, our initial goal was to correctly identify the book titles of each book. However, due to technical problems with training classifiers for character recognition (this will be discussed in more detail further below), we stop short of actual recognition, and implement all the proceeding processing stages, including the code for training the classifiers.

The main goal is therefore to successfully extract, from each book spine, a binary images of all the individual words on that spine. Given a trained classifier capable of recognizing English characters, this information can then be easily passed to it for actual recognition, with no further processing.

B. Problem Domain

To avoid dealing with several less common and complicated edge cases, several simplifying assumptions were made regarding the input photo. We assume the following:

1) The books are all in English.
2) The books in the photo are all vertically oriented.
3) The text orientation on the books spines is top-to-bottom. This is almost always the case with English books.

C. Overview of Methods

Recognition of book spines presents several challenges. Book spines are often characterized by multiple colors, varying typefaces and font sizes, non-text graphics, more than one text orientations, and varying locations of text. Those properties mainly effect the difficulty of text segmentation and layout analysis. Following is a brief description of out system components and out methodology for dealing with those, and other, challenges.

The process of book recognition is separated into four major parts, listed in their order of application:

1) Book spine separation.
2) Text segmentation (separation of text from background)
3) Layout analysis (grouping of characters into words)
4) Character recognition.

For spine separation, spine boundaries are recognized by applying a Hough transform and looking for horizontal lines. Individual book spines are then cropped out of the original photo. We implement text segmentation by moving a narrow window along the book spine, and applying Otsu's method [10] at each location. Further noise filtering is then applied. For layout analysis we propose and implement a 3-step algorithm that involves creating a path between the connected components, splitting the resulting path into word candidates, then merging some candidates into what would be our the final

(a) (b)

Figure 2: Spine separation stage. (a) Detection of spine borders using Hough transform. (b) Cropped book spines.

words. Finally, for character recognition we provide a working implementation for training an SVM or a K-Nearest Neighbors classifier using the Chars74k [6] dataset, with HOG descriptors [3] as feature vectors. An application for testing the various components of our system is provided with the project, with a user guide at the end of this report.

II. Methods

This section describes in detail each step in our processing pipeline.

A. Spine Separation

Given a photo of a bookshelf with vertically oriented books, our first step is to identify the borders of each individual spine, and separate them in preparation for further processing of each spine. We implement this step by converting the image into grayscale, applying Canny edge detection [1], then applying the Hough line transform on the edge image. The lines extracted from the parameter space are only those with $-20 < \theta < 20$, that is – approximately vertical lines. Nearby lines are merged together to avoid redundant spine edges. The border of each spine is the assumed to be defined by two consecutive lines (with respect to the x axis). Each spine is cropped according to this border, and then wrapped into a rectangular image to compensate for distortions due to perspective. See figure 2.

B. Text Segmentation

Successful text segmentation of book spines required dealing with several challenges. Book spines are often characterized by non-uniform background and non-uniform text properties, such as size and color, as well as non textual graphics. This non-uniformity is apparent when comparing different spines, but is also often present within individual spines: A single spine can have multiple background colors, as well as different text colors and font sizes. See figure 4 for an example of such spines.

Our approach to dealing with text segmentation under those conditions is first to observe that while this non-uniformity is indeed present along the y dimension of each spine (we are assuming that the spine is oriented vertically), the situation is often much simpler when considering only the x dimension for a fixed y. For example: many book spines often consist of two background colors, but they are arranged as one below the other, as opposed to being adjacent along vertical lines. For a fixed $y = y_0$, pixels along this line either take at most 2 values (text and background), or at the very least this line has a uniform "background". This characterizes the vast majority of book spines, though not all of them. This too can be seen in the spines present in figure 4.

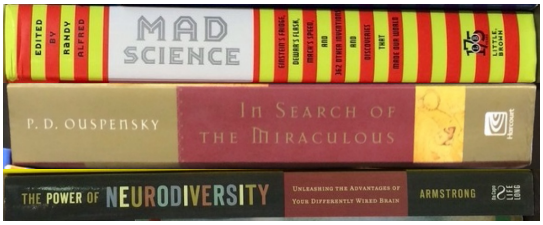
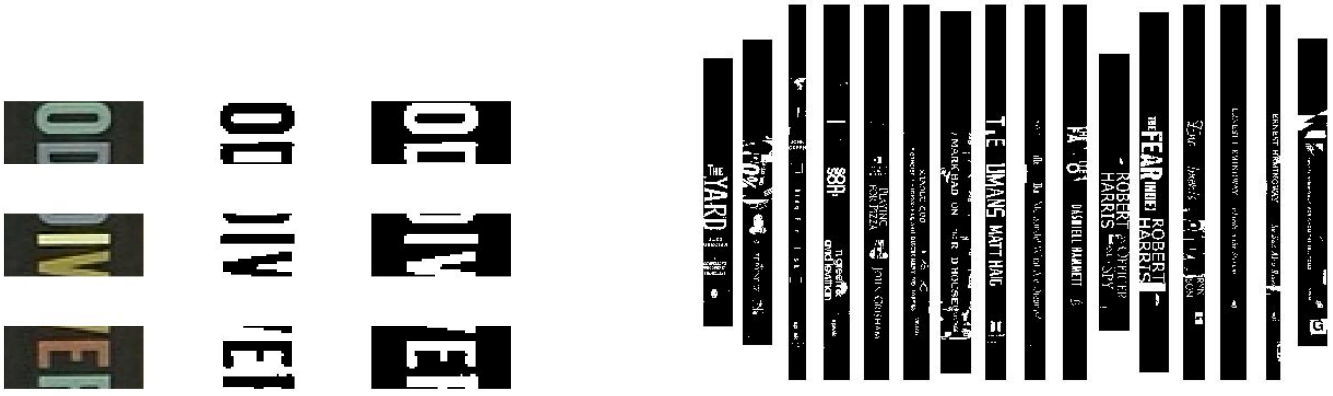


Figure 4: Spines displaying internally varying background and text colors

We therefore propose a simple segmentation method that exploits the above observation: let w be the spine width and h be the spine height. We create a window of size $w \times \frac{h}{n}$ (i.e. window width = w, window height = $\frac{h}{n}$) where n can be any sufficiently high number (we use $n := 80$). We then slide this window along the spine in increments of $\frac{h}{n}$, and for each new location we threshold the window using Otsu's method [10]. Otsu's method calculates a threshold value which is "optimal" in the sense that it minimizes the intra-class variance of pixel values, or equivalently maximized the variance **between** classes. Therefore if each window is characterized by a uniform background which is of sufficiently different value from the value of text pixels, we expect Otsu's method to result in a successful text-background segmentation, as is indeed the case. It should be noted that for windows without text, no segmentation is needed at all, and in fact Otsu's method will only make things worse by forcing a segmentation which doesn't represent neither text nor background. We therefore check, prior to segmentation, the difference between the minimum and maximum pixel values within the window.



(a) (b)

Figure 3: Segmentation results. (a). Results of segmentation for 3 consecutive windows along the spine. Notice that the segmentation classed are flipped (from second to third row) in order to associate the white pixels with text. (b). Segmentation results for an entire bookshelf image.

If the difference is low - we assume the window contains no text, set the window pixels to black, and skip the segmentation altogether.

While the segmentation result for each window is approximately correct in the sense that it successfully separates text and background, we still need make sure that the same single class is reserved for text across the entire spine. i.e., in practical terms, we want all text pixels to be white and all background pixels to be black, across all segmentation windows. This requires checking, for each window that was segmented, which class is text and which is background, and flipping them if necessary. We expect text to be surrounded by background. That is, if both background and text are present – we expect the x coordinate of the leftmost (rightmost) background pixels will be greater (lower) than the x coordinate of the leftmost (rightmost) text pixels. We check this, and flip the classes if necessary. This is surmised by the procedure below. Also see figure 3a for an example of the segmentation for three consecutive windows along a spine.

Algorithm 1 Flip segmentation classes

x_{W_r} = the mean x coordinate of the 5 rightmost pixels from the white class.
x_{W_l} = the mean x coordinate of the 5 leftmost pixels from the white class.
x_{B_r} = the mean x coordinate of the 5 rightmost pixels from the black class.
x_{B_l} = the mean x coordinate of the 5 leftmost pixels from the black class.
$W_d = |x_{W_r} - x_{W_l}|$
$B_d = |x_{W_r} - x_{W_l}|$
if $W_d > B_d$ **then**
    Flip black and white
end if

We now wish to remove white blobs which aren't text. Those can be graphics which are present on the spine, image noise, and background elements which weren't successfully labeled as background. We first find the connected components in the binary image resulting from the segmentation. At this point each connected component is either a character or a wrongly labeled component, which we would want to remove. We remove components with at least one of the following properties:

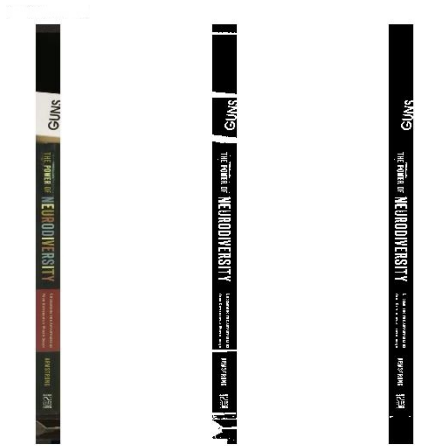
- Height greater than $0.9 \cdot$ spine width
- Width greater than $0.7 \cdot$ spine width and aspect ratio > 5, where we define the aspect ratio as $\max\{\frac{w}{h}, \frac{h}{w}\}$, where w, h are width and height, respectively, of the component's bounding box.
- Fewer pixels than $\left(\frac{\text{spine width}}{30}\right)^2$.
- Aspect ratio between 0.5 and 1.7 and solidity > 0.9, where solidity is defined as the proportion of the pixels in the convex hull that are also in the connected component.

The above geometric constraints were chosen after a lengthy experimentation process. We have encountered some difficulties in achieving a good balance between removing as many noisy connected components as possible on one hand, and making sure valid characters aren't accidentally removed on the other. The result, while manages to remove some noise, fails in several cases, and many book spines still have some amount of non-text CCs remaining at the end of the process. figure 5 shows the results of the segmentation stages on a spine. Also see figure 3b for the results of segmentation on an entire bookshelf image.

We have attempted to further improve our results by using stroke width information as described in [2], using an existing Matlab implementation by MathWorks [8], but it resulted in an increase in the number wrong deletions of valid characters, and we eventually decided to remove it from out implementation.

C. Layout Analysis

The main goal of the layout analysis stage is to group individual CCs (connected components) into words. We do not attempt to further group individual words into lines, etc, as the words by themselves, even out of order, are sufficient



(a) (b) (c)

Figure 5: The segmentation process for a spine. (a) Original spine in RGB. (b) Segmented spine prior to noise removal. (c). Segmented spine after noise removal.

to successfully search for the book in a database. Here again the main challenge is the lack of any uniform text structure in book spines: one cannot infer global letter and word spacing distances, as was proposed, for example, by O'Gorman in 1993 [9] for layout analysis of documents, because no such global value exists – different words have different sizes, different typefaces, different orientations (horizontal and vertical) and also different spacing properties.

Our approach is to consider only properties that can be inferred locally for each CC, or for a small group of CCs. First, starting from an arbitrary symbol, we create a path P consisting of all the individual symbols, by going at each step to the nearest unvisited neighbor (figure 6b). We note that for most occurrences of text, this path has the following property: let $w = a_1 a_2 \dots a_n$ be a word of length n on the spine, where the a_i-s are individual characters, and let a_j be the first character of w to be visited, then either $(a_j, a_{j+1}, \dots, a_n)$ or $(a_1, a_2, \dots, a_j)$ are a subpath of the resulting path. This is simply due to the fact that the nearest character to any character in a word is a character adjacent to it in the same word. Therefore the resulting path will consists mostly of real word segments of length greater than 1. We now wish to separate those segments from each other in order to create word candidates. We do this by splitting the resulting path at the following points, which are suspected as word edges: let (a, b, c) be a subpath of P, we calculate $d_{a,b} = d(a, b)$, $d_{b,c} = d(b, c)$, where $d(a_i, a_j)$ is a the euclidean distance between the two nearest pixels of a_i and a_j. An alternative would be to measure the distance between the centroids of a_i and a_j, however we found that due to differences in character width, this has too high variance when measuring distances between adjacent characters in the same word, and is therefore less suitable for our purposes. We next check if $\frac{d_{a,b}}{d_{b,c}} \geq 1.6$, where the 1.6 value was chosen empirically. If so – we split the path at the edge (a, b). Otherwise we similarly check if $\frac{d_{b,a}}{d_{a,b}} \geq 1.6$, and split the path at (b, c) if so. This splitting process results in a list of word candidates, where each candidate is a subpath of P that cannot be split any further, and is also maximal with respect to this property. Figure 6c shows edges chosen for deletion, and figure 6d show the the word candidates created by the splitting stage.

Since the resulting word candidates are mostly oversegmented actual words, we next wish to merge word candidates that belong to the same word. However now we have additional information to work with: spacing and angle statistics for each candidate. We check if two segments can be merged using the following procedure (an explanation follows):

Algorithm 2 Check if two segments can be merged

1: **procedure** MERGEABLE?(S_1, S_2)
2: $d \leftarrow d(S_1, S_2)$
3: $a \leftarrow angle(S_1, S_2)$
4: $d_1 \leftarrow meanDistance(S_1)$
5: $d_2 \leftarrow meanDistance(S_2)$
6: $s_1 \leftarrow stdDistance(S_1)$
7: $s_2 \leftarrow stdDistance(S_2)$
8: $a_1 \leftarrow meanAngle(S_1)$
9: $a_2 \leftarrow meanAngle(S_2)$
10: $f_d \leftarrow 1.5$
11: $f_a \leftarrow 10$
12: **if** $(d \in [d_1 - s_1 \cdot f_d, d_1 + s_1 \cdot f_d]$ **or** $d \in [d_2 - s_2 \cdot f_d, d_2 + s_2 \cdot f_d])$ **and** $(a \in [a_1 - f_a, a_1 + f_a]$ **or** $a \in [a_2 - f_a, a_2 + f_a])$ **then**
13: **return** True
14: **end if**
15: **return** False
16: **end procedure**

For each segment, we calculate the following values:

- Mean distance between adjacent characters (lines $4-5$).
- Standard deviation of the distances between adjacent characters (lines $6-7$).
- Mean angle between adjacent characters, given in the range of $[0, \frac{\pi}{2}]$ (lines $8-9$).

Additionally, we calculate:

- The distance between the two segments, defined as the distance between the two nearest end-point characters of the segments (line 2).
- The angle between the two segments, defined as the angle between the two nearest end-point characters of the segments (line 3).

We then check for the following two conditions:

1) The distance between the segments is within reasonable limits from the inner distances of at least one of the segments (line 12 – first two conditions). Where "reasonable" is define as within f_d (defined on line 10) standard deviations from the mean distance.
2) The angle between the segments is within reasonable limits from the inner angles of at least one of the segments (line 12 – last two conditions). Where "rea-

sonable" is defined as within distance f_a (defined on line 11) from the mean angle.

Iff both of the conditions are met - we merge the words.

Special care should be taken when considering words of length 2, since the standard deviation in this case is always zero. Therefore in those cases we substitute the standard deviation value by some predefined fraction of the mean. In our implementation this value is set to 0.5.

The actual merging procedure is done recursively: at each iteration we look for two segments that can be merged, merge them, then return the result of the procedure on the updated word list. This implementation has the advantage that new information created by each merge (i.e. distance and angle values for the new merged word) can be used immediately for any following merge. Below is the pseudo code for the algorithm. Figure 6e shows the result following the merging step.

Algorithm 3 Merge into words

```
1:  procedure GRPWRDS(words)
2:      for each pair of candidates S_i, S_j do
3:          if MERGEABLE?(S_i, S_j) then
4:              W ← MERGE(S_i, S_j)
5:              words ← words \ {S_i, S_j} ∪ {W}
6:              return GRPWRDS(words)
7:          end if
8:      end for
9:      return words
10: end procedure
```

D. Character Recognition

Prior to the actual character recognition, we calculate, for each word found in the previous step, its orientation. Under our problem domain assumptions, text appearing on book spines can be in either a vertical or a horizontal orientation. We infer the orientation by calculating the centroids of the first and last characters in each word, then checking if the difference between their x coordinates is greater than the distance between their y coordinates. Assuming the spine is vertical, the former would mean that the word is oriented horizontally, and the latter means it is oriented vertically. Before attempting to recognize each character, we flip it according to its orientation.

Since our system must be able to recognize characters of different typefaces, simple template matching techniques (see [11] for a survey of those methods) aren't likely to work. Our intention was to use a more robust classifier by training a multi-class SVM on Microsoft's Chars74k dataset [6], with HOG descriptors as feature vectors. In practice we were unable to train this classifier on our system, mainly because training the multi-class SVM required training over 1,800 binary classifiers, while in a single night we have managed to train less than 200. As an alternative we have also tried training a K-Nearest Neighbors classifier, however this too failed to work due to running out of memory. Attempting to train the classifiers on university computers didn't work either because the code requires Matlab 2014b or higher. To provide some sort of a working alternative we have included in our application the ability to recognize the extracted words using Matlab's built-in OCR functionality, however unfortunately its performance seems to be very poor even on relatively simple cases, and it rarely manages to recognize anything. Still, since the project includes functioning code for training the above-mentioned classifiers on the Chars74k dataset (based on [7]) , we will briefly describe the components of this system. Additionally, appendix B provides a brief instructions on how to train the classifiers.

1) Chars74k Dataset: The Chars74k dataset [6] was originally used to create a system for character recognition in natural images, using an SVM [5]. The dataset contains training images for 62 classes: letters, capital letters, and digits. Each class has image samples from 256 fonts, with 4 variations for each font (italic, bold and normal). The total number of images is $62,992$, or 1016 per class.

2) HOG Feature Descriptors: HOG stands for Histogram of Oriented Gradients. HOG features for an image are calculated by first dividing the image into a grid of uniformly spaced cells, then – for each cell – calculating a histogram of gradient orientations for pixels within that cell. The descriptor is then the combination of those histograms. HOG descriptors were originally used for the detection of humans in photographs [3], and have since been successfully used for the detection people, vehicles, animals, and additional objects in images [4]. In recent years HOG descriptors were also successfully used for recognition of characters extracted from images or graphics [2] – a task similar to our goal in this project.

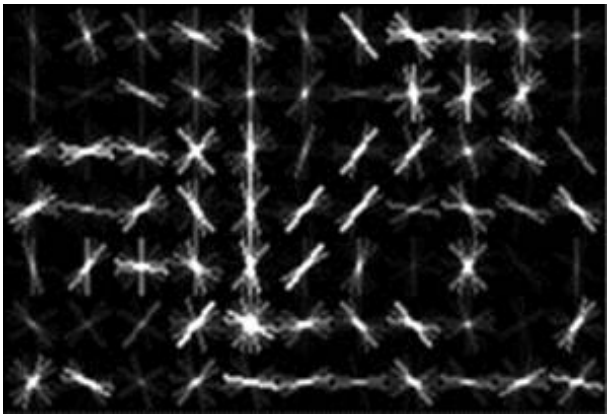
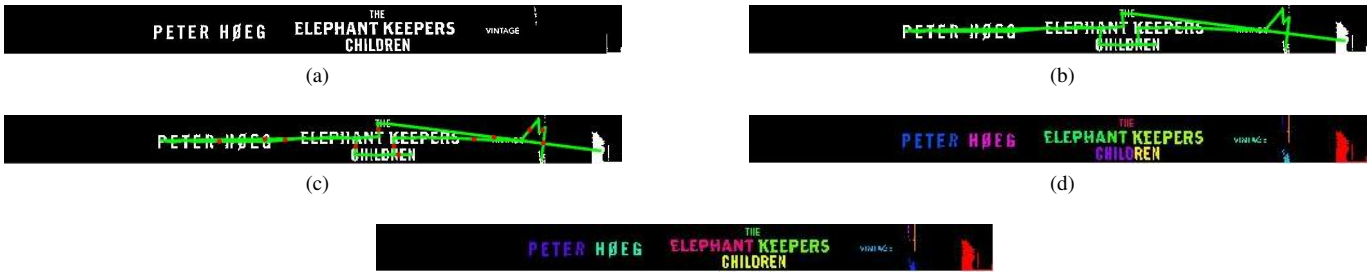


Figure 7: A graphical representation of HOG features for an image. Each block contains a histogram of gradient edges, represented here as edges of varying brightness. The brightness of each orientation signifies its prevalence within that block.

III. Results

We have tested our application on several photos of bookshelves taken in the same bookstore. Because performance of Matlab's OCR functionality is very poor, we do not have a measure the actual text recognition success. We will describe instead the performance of our application in the various processing stages, while listing causes of failure.

While the application succeeds in many cases, no step in the processing chain seems to work perfectly. As all processing



(a) (b) (c) (d) (e)

Figure 6: Layout analyst stages. (a) The spine prior to layout analysis. (b) A traversal of the CCs. (c). Splitting of the path at suspected word edges. Edged to be deleted are marked with red dots. (d) Result of the splitting stage. (e) Result of the merging stage.

steps are dependent upon earlier steps, even minor error in early stages can greatly effect the success of later stages. Following is a description of results for each processing stage.

A. Spine Seperation

Under our problem domain assumptions, spine separation is the least problematic stage of all. Out of 78 books appearing in 4 bookshelf photographs, our application has successfully separated 75 of the books, which amounts to a success ratio of 96%. The main causes of failure seem either misleading book design (i.e. - a straight vertical line along the entire spine), or adjacent books of similar colors.

B. Text Segmentation

Nearly all book spines have a certain amount of non-text blobs left after segmentation and noise removal. This isn't especially troubling for OCR, since most of those can be ignored based on low confidence levels of recognition. However it can negatively affect the layout analysis stage by joining characters to non-character blobs, therefore wrongly skewing the angle and distance statistics for each word candidate. A more serious problem is treating valid text as background. Out of 36 spines in two bookshelf images, around 8 suffer from missing text, with 5 spines showing partial failures, and 3 with almost no text visible at all. This gives a success rate of 77% over a limited sample. This is mostly due to the inherent limitations of our segmentation method: inability to handle background which is non-uniform within small windows along the spine, or text values which are too close to the background values. For books displaying mostly uniform backgrounds, the success rates are significantly higher.

Another problem present is over-segmented or under-segmented characters. Due to the low resolution of the images and small text sizes in some spines, the segmentation stage occasionally results in valid characters being split into two or more non-characters, due to a few (or a single) missing pixels. Similarly, the segmentation stage sometimes results in separate characters being fused together to form a non-existing character. The smaller the text - the higher the probability of this happening.

C. Layout Analysis

We measure the success of layout analysis by considering the ratio of the number of words correctly identified, to the total number of words. In an image consisted of 16 well-segmented books spines, with a total of 71 words, our algorithm has successfully identified 53 words, or 74% of all words. Most identification mistakes result in the grouping several distinct words together. This is often better than mistakes caused by splitting a word into several non-words, as the former is often easier to interpret correctly with the help of a search engine.

Failures are also sometimes caused by word positions and text design which our current system is unable to handle, such as adjacent lines being closer together than adjacent characters in each word.

IV. Conclusions, Limitations, and Future Work.

In this project we have developed a system for recognizing book titles from photos of bookshelves. Our system consists of 4 processing stages: Locating and separating the individual spines, segmenting the text from the background, grouping individual characters into words, and finally recognizing the words. Unfortunately we were unable to test the last stage due to technical difficulties with the training of classifiers, but we provide a working implementation for training and applying the classifiers, with instructions provided in appendix B. The other components of the system are fully implemented and tested in Matlab.

For the segmentation stage we have proposed and implemented a simple method suitable for most book spines, which exploits the uniformity of background along horizontal lines in the spines, and succeeds in correctly segmenting the text in around 77% of the spines, although usually with a certain amounts noise. For the grouping of characters into words, we have developed an algorithm that relies on relationships between each characters and its neighbors, capable of dealing with multiple text orientations, as well as varying sizes, typefaces and spacing, with success rates of over 70%.

Our current system has several limitations, each presents opportunities for future work. Those limitations, possible solutions, and additional improvements, are discussed below.

- **Inability to deal with complicated spine backgrounds**. As mentioned above - our segmentation method relies on book spines having a uniform background along horizontal lines, or within sufficiently small windows. When this isn't the case - our system will usually fail. Dealing with those cases would likely require a different approach to segmentation.
- **Spine angles**. Our system deals only with horizontally oriented spines, and this was an explicit part of our problem domain assumptions. However in practice, books on store bookshelf can appear in almost any orientation, though the horizontal one is the most common. Dealing with this wider array of orientations would require lifting any restrictions on line angles extracted from the Hough transform, and developing a method for correctly extracting book borders from potentially many intersecting lines.
- **Noise removal / filtering of non-text graphics**. Our segmentation stage almost always ends with certain amounts of noise present. As mention earlier - this negatively effects the layout analysis stage. A possible solution would be to apply OCR as part of the filtering process, deleting blobs below a certain recognition confidence level. Additionally, more geometric constrains can be added to our already existing set of constrains. However, we find that discerning the optimal set of such constrains is a very hard task to complete manually, but perhaps this can be done computationally. As mentioned earlier, we have also tried to use stroke-width information [2] for further filtering [8], with poor results, however this might be due to faulty implementation.
- **Oversegmented/undersegmented characters**. This problem can be inherently ambiguous (in the case of conjoined characters we can't say whether a character is an "m" or two "n"s just by looking at the character). An obvious solution would be to use higher resolution images. However this isn't always possible, and some oversegmented/undersegmetned characters can also be the result of the typeface. One possible approach is to to consider all possible interpretations for each word, and check each option against a dictionary of legal words.
- **Improved word detection.** Our current layout analysis algorithm sometimes results in words spanning several lines. This happens when the distance between two lines of text is smaller than the distance between adjacent characters within the same line. At least a certain portion of those cases can be fixed by allowing the splitting of words before or after the merging stage, based on a histogram of angles within a given word candidate.
- **Line detection.** Our current system groups individual characters into words, but does not group individual words into lines or blocks of text. Doing so would aid in the correct recognition of book titles.
- **Training a classifier for character recognition.** This would require either a stronger computer, more training time, using a classifier not trained in Matlab, or some combination of the above.

Finally, we present a Matlab application for testing the system. A brief user guide is provided in an appendix to this report.

REFERENCES

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

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

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

[4] Navteen Dalal and Bill Triggs. Object detection using histograms of oriented gradients. In *European Conference on Computer Vision, Workshop on Pascal VOC 06*, 2006.

[5] Teo de Campos, Bodla Rakesh Babu, and Manik Varma. Character recognition in natural images. 2009.

[6] Microsoft Research India. The chars74k dataset. http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/.

[7] Kenneth Lim. Decaptcha - a captcha breaker using k-nearest neighbor classifiers, support vector machines, and neural networks. https://github.com/kenlimmj/decaptcha.

[8] MathWorks. Automatically detect and recognize text in natural images. http://www.mathworks.com/help/vision/examples/automatically-detect-and-recognize-text-in-natural-images.html.

[9] Lawrence O'Gorman. The document spectrum for page layout analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(11):1162–1173, 1993.

[10] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *Automatica*, 11(285-296):23–27, 1975.

[11] Øivind Due Trier, Anil K Jain, and Torfinn Taxt. Feature extraction methods for character recognition-a survey. *Pattern recognition*, 29(4):641–662, 1996.

APPENDIX A
APPLICATION USER GUIDE

The application allows users to see in action the various parts of our system. It allows to open any bookshelf image, separate the spines, then select a spine for further processing: Text segmentation, word detection, and character recognition using Matlab's built-in OCR functionality. Moreover, the user can choose whether to segment the text with or without noise removal, and also whether to detect words with or without the merging step of our layout analysis algorithm, thus allowing to inspect the precise effects of each step. Following is a brief description of how to use the application.

First, open the app either by running the app.m file from within Matlab, or by running the standalone application (Book_Identifier_icbv151.exe).

When the application is first opened, the only available option is to open a new image. Clicking it will open a load dialogue box allowing to select any bookshelf image from the filesystem.

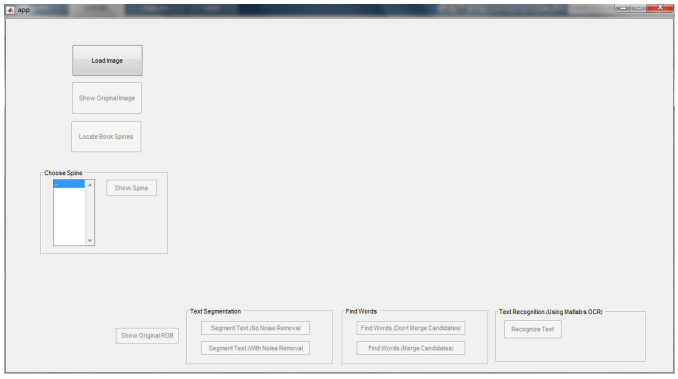


Figure 8: The application screen immediately after opening the app.

The selected image will then be visible on the application screen, along with the option to detect the spines on the image.

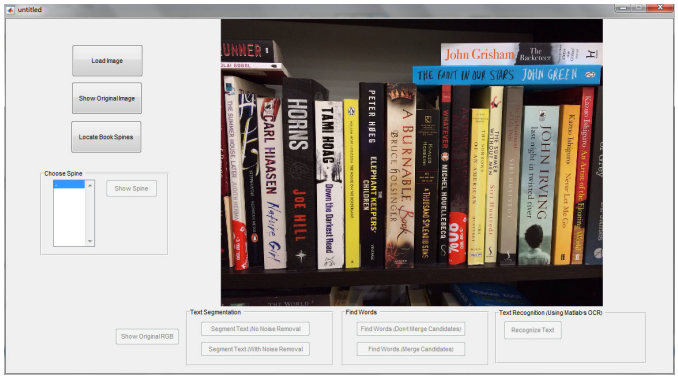


Figure 9: The image is displayed, and the "Locate Book Spines" button is now available.

Clicking the "Locate Book Spines" button will run the spine separation procedure, and display the detected spines, along with a number for each spine. Each spine can be then chose by clicking the corresponding number in the "Choose Spine" listbox, and clicking on "Show Spine".

Figure 10: The detected spines.

Upon clicking on "Show Spine", the main image will be replaced by a horizontal image of the chosen spine. You can next choose to segment the spine, either with or without noise removal.

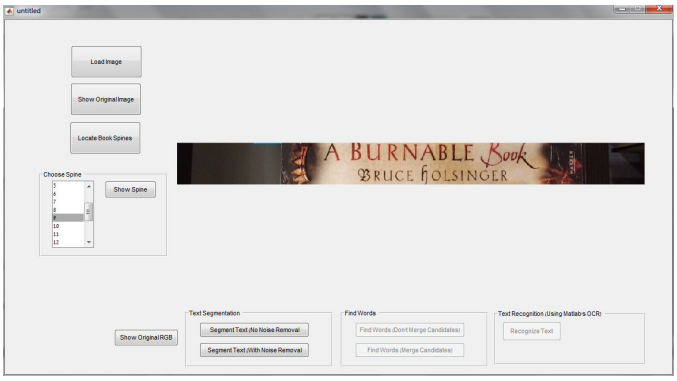


Figure 11: The selected spine.

After segmentation, the segmented spine will be displayed, and the option to detect the words will become available. You can choose to apply the word detection either with or without the merging step of the layout analysis algorithm.

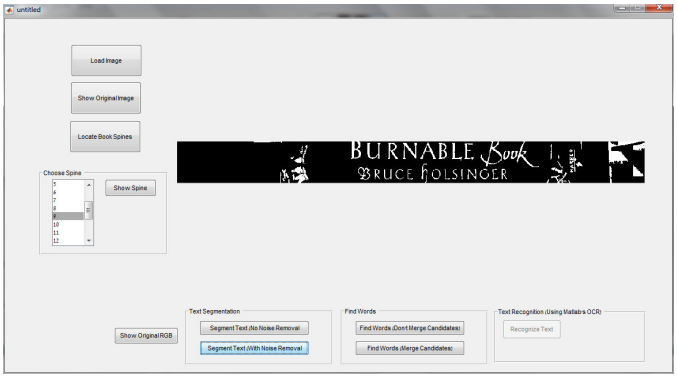


Figure 12: The spine following segmentation.

The detected words will then be displayed, with each word given a different color. You can then choose to attempt and

recognize the text using Matlab's OCR functionality. If you choose to do so, each visible word will be cropped and the resulting image will be passed to Matlab's OCR function (this is transparent to the user). Note that the Matlab's OCR performance is very poor and most of the time you will get inaccurate results.

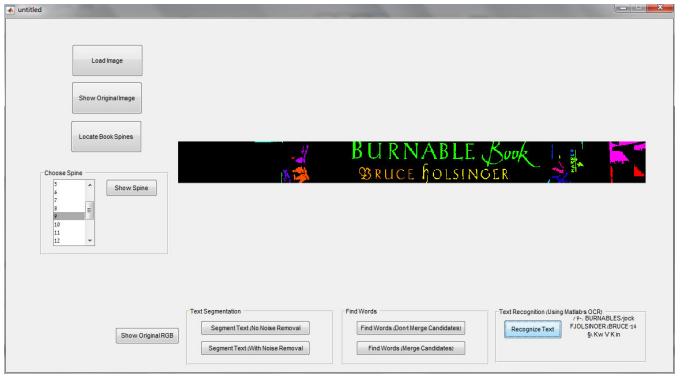


Figure 13: The detected words.

Note that at any step you can choose to go back or reapply any of the previous processing stages, including opening a different image.

Appendix B
Training a Classifier

The code for training the classifiers is based on [7], with minor fixes. To train the character recognition classifier, follow the following steps:

1) Download and extract the EnglishFnt.tgz file from http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/. Open Matlab and add the extracted folder to your Matlab Path.
2) Add the classifier folder (provided with this project) to your Matlab path.
3) In Matlab, run `list_English_Fnt;`
4) Run `fontChars = genFontChars(list);`
5) After completion, you can choose to train an SVM by running `svmRecognizer = genSVMRecognizer(fontChars, list.ALLlabels);`, or a k-Nearest Neighbors classifier by running `knnRecognizer = genKnnRecognizer(fontChars, list.ALLlabels);`
6) Cross your fingers and wait for a very long time. We never got past this point.
7) To classify a word, create a cell array `chars` of the word characters (each character – a binary image), then run `recSvmCaptcha(chars, svmRecognizer)` for the SVM classifier, or `recKnnCaptcha(chars, knnRecognizer)` for the KNN classifier.

Consult [7] for further information and instructions.