Automatic Processing of Document Annotations

Jacob Stevens†, Andrew Gee† and Chris Dance‡

†University of Cambridge
Department of Engineering
Cambridge CB2 1PZ
[94jts|ahg]@eng.cam.ac.uk

‡Xerox Research Centre Europe,
61 Regent Street
Cambridge CB2 1AB
dance@xrce.xerox.com

Abstract

A common authoring technique involves making annotations on a printed draft and then typing the corrections into a computer at a later date. In this paper, we describe a system that goes some way towards automating this process. The author simply passes the annotated documents through a sheet-feed scanner and then brings up the electronic document in a text editor. The system then works out where the annotated words are and allows the author to skip from one annotation to the next at the touch of a key. At the heart of the system lies a procedure for reliably establishing correspondences between printed words and their electronic counterparts, without performing optical character recognition. This procedure might have interesting applications in document database retrieval, since it allows an electronic document to be indexed by a printed version of itself.

1 Introduction and outline

Documents are usually written using an iterative process. A first draft is typed into a word processor and then printed out for editing, either by the author (who is tired of looking at the screen) or by another party. The corrections are then typed into the word processor and the process repeats, until a satisfactory final draft is produced. This extremely common practice is not very efficient, since the corrections are processed twice, once by the editor and then again by the typist (who may or may not be the same person). Furthermore, the process is prone to human error, as annotations are not always spotted when quickly skimming a page. In this paper, we describe a system that goes some way towards automating this process. The idea is that the annotated draft is passed through a sheet-feed scanner and the corrections made *automatically* to the electronic version.

Such a system requires the solution of a number of difficult pattern processing problems:

BMVC 1998 doi:10.5244/C.12.44

1. The handwritten annotations need to be detected on the print-out.
2. Individual printed words must be segmented, and the annotations associated with the appropriate word.
3. Correspondence needs to be established, on a word to word basis, between the hard copy and the electronic document.
4. The semantics of the annotations must be determined, and the necessary changes made to the electronic document.

Item 4 is an example of the generic off-line handwriting recognition problem, which has attracted considerable research interest over the years [2]. We make no attempt to address this significant problem, apart from noting that the task could be simplified, without sacrificing too much functionality, by limiting the annotations to the standard set of proof-reader's symbols: text for insertion would still need to be typed manually.

Item 1 is a typical image segmentation problem, which can be tackled using a variety of approaches depending on the nature of the hard copy and the annotations. In this paper, we assume that the hard copy is printed in black and white and the annotations are made in red. The segmentation problem is then solved using a simple thresholding scheme in RGB space.

The word segmentation stage of item 2 is a component of the standard optical character recognition (OCR) pipeline: it is best tackled using connected component analysis [8, 9]. Annotations can then be matched to words using a simple nearest neighbour scheme, with some minor modifications to ensure that underlines are not misinterpreted as overlines.

Perhaps the most interesting of the problems is item 3. The modern world is full of electronic documents, mostly stored as ASCII text or in proprietary word processor formats, but also sometimes as bit-mapped images in document archives. Item 3 amounts to a content-based retrieval of electronic documents from large databases: the documents are retrieved using printed versions of themselves. This is a generic problem which has potential applications beyond the annotation system which is the subject of this paper.

In our approach to item 3, we assume that the hard copy contains only alphanumeric text, and that the electronic document is stored as pure ASCII text. In a more general setting, different algorithms would need to be developed for the various word processor formats to strip out images and symbols and leave the pure ASCII text. Similar preprocessing operations would need to run over the hard copy to blank out images and other non-textual entities.

Even given this simplification, item 3 is not trivial. The correspondence needs to be invariant to the appearance of the hard copy, which could have been printed using any shape and size of font and any sensible formatting style. One approach might be to subject the print-out to OCR, and then match the OCR output with the ASCII text in the electronic document. While this is a perfectly valid approach, it must be noted OCR is not 100% reliable, so the matching algorithm would need to be of the *approximate* string matching variety [5][1]. Given this inescapable truth, we believe that there is little to be gained from a full OCR approach. Instead, we take the OCR process as far as the word segmentation

[1] Such algorithms have been the subject of much research recently, predominantly for the important task of DNA string sequencing [3].

stage, and then match the printed word lengths (in mm) to the electronic word lengths (in characters). Since the hard copy may be produced using proportional fonts, and there will invariably be segmentation errors, this too takes the form of an approximate string matching problem. The advantages of a word length approach are threefold: first, the algorithm is faster, since the final stage of the OCR process is omitted; second, we need to resolve much less detail in the hard copy, so we can scan a typical document at 100 dpi, instead of the 300 dpi required by OCR packages, or even image the document using an over-the-desk camera [10]; and third, the presence of the annotations disrupts the OCR output far more than the word lengths, so a word length approach is more robust.

Our matching scheme can reliably establish correspondence between a 51,900 word electronic document (in our experiments, the entire libretto of Wagner's *Ring*) and a few paragraphs printed using any normal font in any normal format. Since we make no attempt to interpret the annotations, our implementation is rounded off with an extension of the Emacs text editor which allows the user to effortlessly jump between annotations.

The paper is organised as follows. Section 2 describes how we detect and classify the annotations, and segment individual words from a page of text. The matching algorithm is explained in Section 3, where we also suggest a procedure for correcting any segmentation errors using the matcher's output. The extension of the Emacs text editor is presented in Section 4, along with some illustrative results. Finally, we draw some conclusions and suggest some avenues for future work in Section 5.

2 Detecting annotations and words

Detecting annotations

Under our assumption that the printed document comprises only black and white text with red annotations, detecting the annotations simplifies to a straightforward colour segmentation problem. The documents are scanned at 100 dpi in 24 bit colour, using most of the dynamic range, and then pixels are classified according to the following simple rule:

if (2$\times$ red – green – blue $>$ T) then pixel = annotation
else pixel = background

We have found that the threshold T does not need to be set within narrow tolerances: a value of 45 seems to work for many different types of red pen. Alternatively, T could be determined automatically by evaluating the discriminant at all pixels and locating the centres of the two resulting clusters. Following the colour segmentation, isolated red pixels are removed, and individual annotations are identified by grouping the remaining red pixels into connected components. Pixels are deemed connected if they share a common 10×5 pixel neighbourhood. Thus, multi-word annotations are treated as a single entity.

Classifying annotations

In our current implementation, annotations are classified into one of three types according to the shapes of their bounding boxes. **Lines**, which may be horizontal and vertical, are long and thin. Annotations which are not lines are classified as **blobs**. For each type of annotation, we define a **focus point**, which is where the cursor will move to

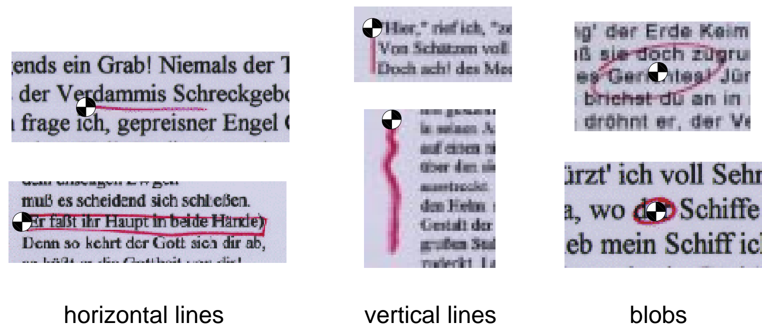


Figure 1: **The three classes of annotation and their focus points.**

when that annotation is selected in the text editor. For horizontal lines, the focus point is centered vertically at the left of the bounding box; for vertical lines, the focus point is centered horizontally at the top of the bounding box; for blobs, the focus point is at the centre of gravity of the bounding box — see Figure 1. The classification is performed by thresholding the bounding box's aspect ratio.

Detecting words

The document image then undergoes a series of operations to prepare it for word segmentation — see Figures 2(a)–(d). It is first de-skewed using a standard algorithm [1, 11], so that the rows of text are horizontal. Next, the annotations are suppressed by replacing the red pixels with white ones. Finally, high image gradients, which correspond to text, are detected by the application of a Sobel operator [6] followed by binary thresholding of its output. This produces a compact image, which is insensitive to illumination and page coloration, and clearly shows the outlines of the words [11].

Individual words are segmented in the binary gradient image using a hierarchical series of connected component analyses. First, the page is segmented into individual columns using a large, vertically elongated connectivity neighbourhood. Then, using a thinner neighbourhood which favours horizontal connections, each column is segmented into individual rows. Finally, using an even thinner neighbourhood, which is elongated in the vertical direction to catch the dots on i's and j's, each row is segmented into individual words. The entire process is illustrated in Figures 2(e)–(g), and the dimensions of the connectivity neighbourhoods are given in Figure 3. We have found that these neighbourhoods work reasonably well for documents scanned at 100 dpi and containing fonts sized between 10 pt and 16 pt, as long as the line spacing is not reduced below the standard single-spaced norm. The entire word detection process can be performed on a standard personal computer in less than one second per A4 page.

The resulting word segmentation suffers from occasional split and merged words, as one would expect given the low scanning resolution and the *ad hoc* setting of connectivity neighbourhoods. The subsequent matching stage needs to be robust to these errors.

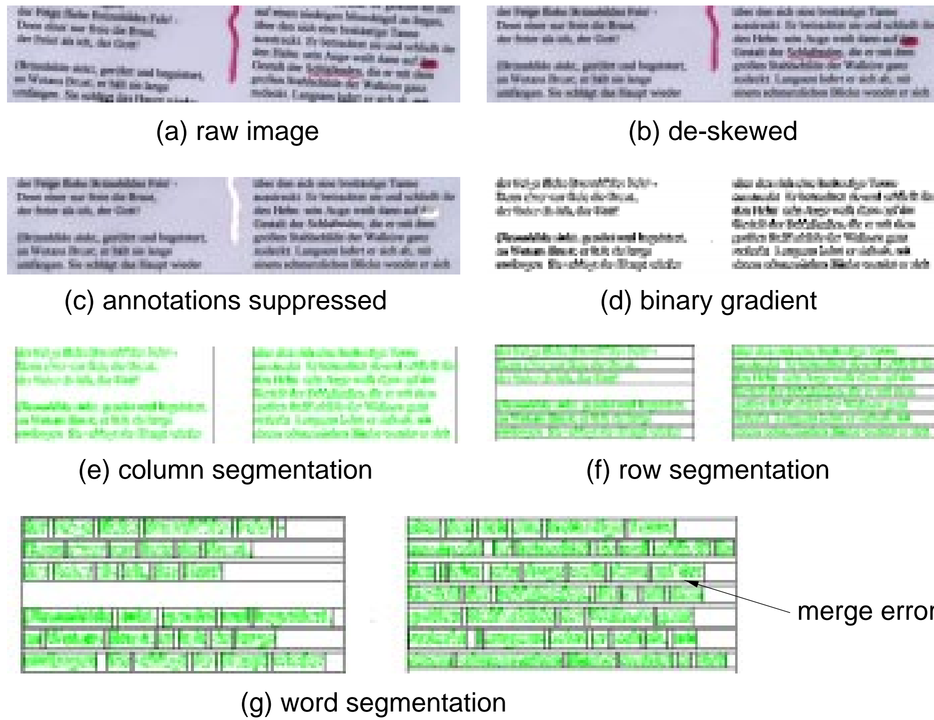


Figure 2: **Detecting words**. In (a)–(d), the raw image is processed to de-skew the rows, suppress the red annotations and enhance the text. In (e)–(g), individual columns, rows and finally words are identified using connected component analysis. The text has been lightened in (e)–(g) in order to highlight the segmentation boundaries.

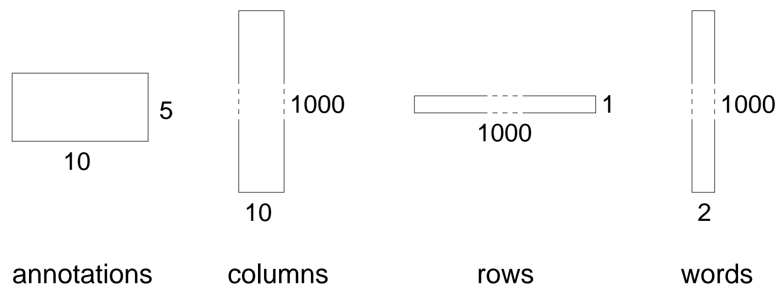


Figure 3: **Connectivity neighbourhoods used in the various connected component analyses**. Dimensions are in pixels. For computational efficiency, the 1000 pixel dimensions can be reduced to, say, 10, and the resulting connected components merged (in a post-processing stage) if they overlap in the '1000' direction.

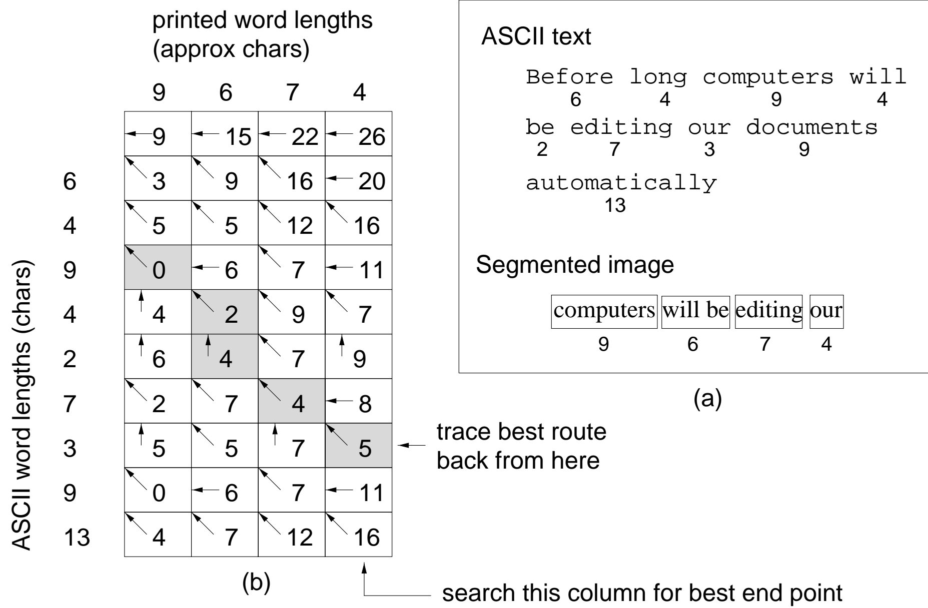


Figure 4: **Approximate string matching by dynamic programming**. The ASCII and printed text are represented by strings of word lengths (a), then matched using an approximate string matching algorithm (b). Note the merge error in the segmentation of the printed text.

3 Establishing correspondences

Establishing correspondences between the printed and electronic words is an approximate string matching problem [5]. The electronic document is represented as a string of word lengths (in characters). Following the hierarchical segmentation, it is possible to obtain a word order for the printed document and represent it too as a string of word lengths (in pixels). A character-to-pixel conversion factor can be estimated by comparing the median word lengths in the two strings: all elements of the printed document string are then scaled by this conversion factor and rounded to the nearest character. We have tried using ratios of adjacent word lengths as a scale invariant length feature, but found this to be inferior to the median method, since the ratios are less robust to split and merge errors.

With perfect segmentation and fixed spaced fonts (eg. Courier), we could then search the electronic document string for a sub-string that exactly matches the printed document string: this is the *exact* string matching problem [4]. With proportional fonts, however, the printed word lengths will not be directly related to the character count, and there will also be split and merge segmentation errors. For these reasons, we need to employ an *approximate* string matching algorithm, which can match multiple words in one string with a single word in the other, and can tolerate small discrepancies between the lengths of matched words.

Our approximate string matching algorithm is built around a simple dynamic pro-

gramming approach [7]. If $W(i,j)$ is the best cost for matching substring $a_i, \ldots, a_j$ of string A to any substring of B, then we exploit the dynamic programming equation:

$$W(1,i) = \min_{j<i}\{W(1,j) + W(j+1,i)\}$$

To find the best match for the entire printed word string, we construct a table, where the rows are indexed by the ASCII words and the columns by the printed words — see Figure 4. The entries in the table indicate how each word is matched. A $\leftarrow$ symbol indicates that the printed word is not matched against any ASCII word: this incurs a penalty equal to the length of the printed word. A $\uparrow$ symbol indicates that the ASCII word is not matched against any printed word: this incurs a penalty equal to the length of the ASCII word. Finally, the $\nwarrow$ symbol indicates that the ASCII and printed words are matched: this incurs a penalty equal to the absolute difference between the two word lengths. The individual matches can be chained together and interpreted as paths through the table. We are looking for the lowest cost path which spans the table from left to right, thus accounting for every word in the printed word string.

The table is constructed row by row from the top left hand corner, keeping track of the *cumulative* penalties incurred up to the current point. Each of the three possible match types is considered at each entry, and the one which results in the lowest cumulative cost is recorded. In the event of a tie, a positive match (type $\nwarrow$) is preferred.

In the first row, only $\leftarrow$ matches are possible. So when we have reached the end of the first row, we see that we can match the entire printed word string against an empty ASCII string with a penalty of 26 characters. We now start to construct the second row. For the first entry, we consider a possible $\nwarrow$ match, which incurs a penalty of 3 characters, and also the $\uparrow$ match, which incurs a penalty of 6 characters plus the 9 above, giving a total cumulative penalty of 15 characters. We choose the lower cumulative penalty, and record the match type and cumulative penalty in the cell.

Since each cell only needs to know about the accumulated penalties above, to the left, and diagonally up and to the left, we can construct the table in raster order in one pass. We then look for the lowest cost match for the entire printed word string: in other words, we locate the smallest entry in the rightmost column. Finally, we backtrack from this point, following the path back to the leftmost column to generate the following string match:

computers	$\leftrightarrow$	computers	editing	$\leftrightarrow$	editing
will	$\leftrightarrow$	will be	our	$\leftrightarrow$	our
be	$\leftrightarrow$	NULL			

The correctly segmented words in the printed document have been perfectly matched with their electronic counterparts. Merge segmentation errors are conveniently flagged by $\uparrow$ matches, whereas split errors show up as $\leftarrow$ matches. We can now go back to the image, and use the known ASCII word lengths to correct all segmentation errors.

There is one slight complication with the correction of merge errors. Suppose the segmentation error had joined the words “be” and “editing”, instead of “will” and “be”. The matcher would come up with the following match:

computers	$\leftrightarrow$	computers	editing	$\leftrightarrow$	be editing
will	$\leftrightarrow$	will	our	$\leftrightarrow$	our
be	$\leftrightarrow$	NULL			

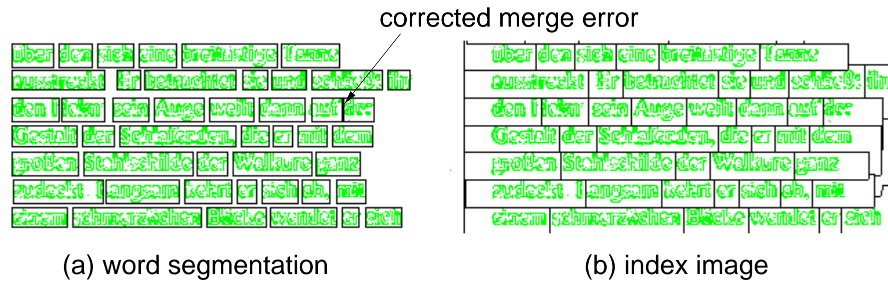


Figure 5: **Corrected word segmentation and the index image**. Compare (a) with Figure 2(g). In the index image (b), the white space between rows is associated with the words *above*. This ensures that underlines are not misinterpreted as overlines.

This has exactly the same path through the match table as before. In other words, the $\nwarrow\nwarrow\uparrow\nwarrow\nwarrow$ match string does not in itself tell us where the merge error is. The $\uparrow$ indicates that *either* the current printed word *or* the one on the right contains a merge error. The ambiguity can be resolved by looking at the $\nwarrow$ matches on either side of the $\uparrow$ match, and noting which has the higher cost. Figure 5(a) shows a portion of Figure 2(g) after automatic correction of segmentation errors: notice how the merge error has been fixed.

At this stage, with an accurately segmented page of text, we match the annotations to words. This is done by first constructing an **index image**, which associates every pixel in the image with a particular word. The index image is constructed using a modified nearest neighbour scheme which ensures that underlines are not misinterpreted as overlines — see Figure 5(b). Each annotation is then matched to a particular word by using its focus point to access the index image.

4 The Emacs interface

We have embedded the entire system into an extension of the standard Emacs text editor — see Figure 6. The ASCII text file is loaded into the editor and a pull-down menu is used to run the annotation system on any specified image file. After matching, the user can tab backwards and forwards through the matched annotations. The system keeps track of the annotated words as the editing session progresses.

Some typical results are shown in Figure 7. In each case, the printed extract was successfully matched against the entire 51,900 word libretto of Wagner's *Ring*. Underneath each extract, Figure 7 shows the words selected by the tab key in the Emacs text editor. In all cases, these correspond to the annotated words. A variety of fonts and type sizes are represented.

5 Conclusions and further work

We have presented a novel system that allows authors to effortlessly and reliably locate incidences of annotated text in electronic documents. At the heart of the system lies a

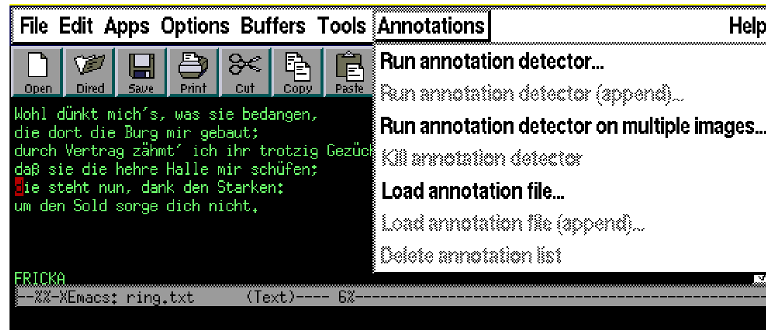


Figure 6: **The Emacs text editor extension.**

technique for establishing correspondences between printed words and their electronic counterparts, without resorting to full OCR. Indeed, the system works well at much lower scanning resolutions than are used for OCR.

While the annotation application is interesting, some users might still prefer the standard search facilities offered by their word processors. Nevertheless, we believe that the matching technique has wider, significant applications. In particular, it provides a content-based retrieval mechanism for electronic document databases, whereby an item can be indexed by a printed version of itself. The particular matching algorithm presented here is based around dynamic programming and has good complexity (order mn, where m is the number of words in the electronic document and n is the number of printed words). However, for large electronic databases, which can run into millions of words, this is not good enough. In the future, we plan to investigate hierarchical document representations, coupled with coarse-to-fine matching, which should improve the speed of the matcher.

References

[1] D. S. Bloomberg, G. E. Kopec, and L. Dasari. Measuring document image skew and orientation. In *SPIE Volume 2422, Document Recognition II*, pages 302–315, 1995.

[2] R. M. Bozinovic and S. N. Srihari. Off-line cursive script word recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):68–83, 1989.

[3] W. I. Chang and E. L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12:327–344, 1994.

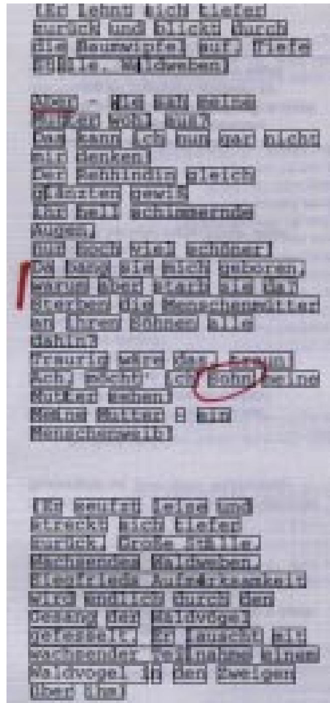
[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, pages 853–885. MIT Press, 1990.

[5] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4:33–72, 1988.

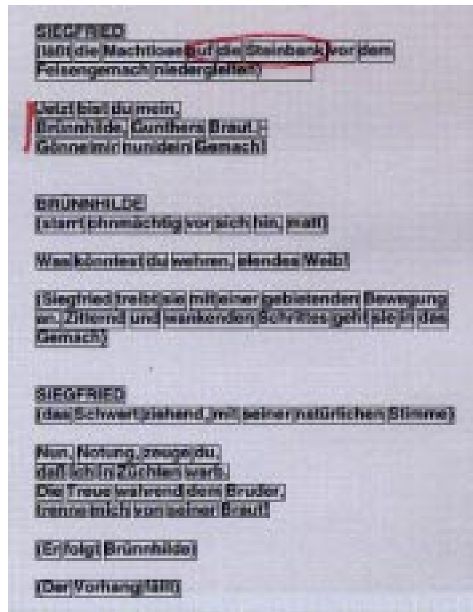
[6] A. K. Jain. *Fundamentals of digital image processing*. Prentice Hall, 1989.

[7] D. Lopresti and A. Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181:159–179, 1997.

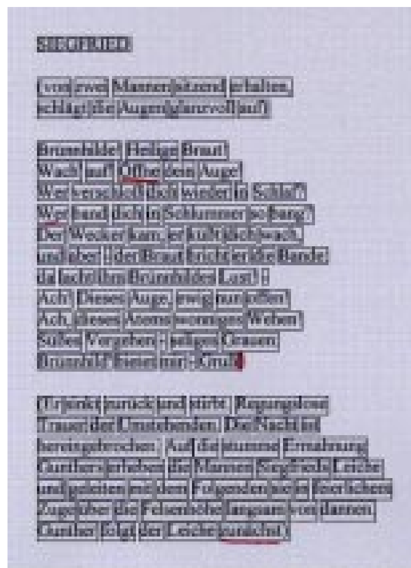
[8] L. O'Gorman and R. Kasturi. *Document Image Analysis*. IEEE Computer Society Press, 1995.



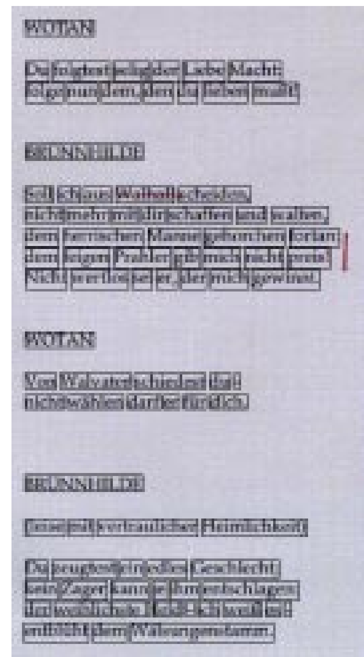
Aber .. Da .. Sohn (12 pt Courier)



auf .. Jetzt (11 pt Helvetica bold)



Offne .. Wer .. Gruβ.. zunächst
(12 pt Times New Roman)



Walhall .. fortan (14 pt Palatino)

Figure 7: **Some typical results.** The figure shows the final word segmentation overlaid on the de-skewed images, along with the ASCII words matched to the annotations.

[9] J. Schürmann, N. Bartneck, T. Bayer, F. Franke, M. Eberhard, and M. Oberländer. Document analysis — from pixels to contents. *Proc. IEEE*, 80(7):1101–1119, 1992.

[10] M. J. Taylor and C. R. Dance. Enhancement of document images from cameras. In *SPIE Document Recognition V*, pages 230–241, 1998.

[11] A. R. Zappalá, A. H. Gee, and M. J. Taylor. Document mosaicing. In *Proceedings of the British Machine Vision Conference*, volume 2, pages 600–609, Colchester, 1997.