An Overview of the Tesseract OCR Engine

Ray Smith
Google Inc.
theraysmith@gmail.com

Abstract

The Tesseract OCR engine, as was the HP Research Prototype in the UNLV Fourth Annual Test of OCR Accuracy[1], is described in a comprehensive overview. Emphasis is placed on aspects that are novel or at least unusual in an OCR engine, including in particular the line finding, features/classification methods, and the adaptive classifier.

1. Introduction – Motivation and History

Tesseract is an open-source OCR engine that was developed at HP between 1984 and 1994. Like a super-nova, it appeared from nowhere for the 1995 UNLV Annual Test of OCR Accuracy [1], shone brightly with its results, and then vanished back under the same cloak of secrecy under which it had been developed. Now for the first time, details of the architecture and algorithms can be revealed.

Tesseract began as a PhD research project [2] in HP Labs, Bristol, and gained momentum as a possible software and/or hardware add-on for HP's line of flatbed scanners. Motivation was provided by the fact that the commercial OCR engines of the day were in their infancy, and failed miserably on anything but the best quality print.

After a joint project between HP Labs Bristol, and HP's scanner division in Colorado, Tesseract had a significant lead in accuracy over the commercial engines, but did not become a product. The next stage of its development was back in HP Labs Bristol as an investigation of OCR for compression. Work concentrated more on improving rejection efficiency than on base-level accuracy. At the end of this project, at the end of 1994, development ceased entirely. The engine was sent to UNLV for the 1995 Annual Test of OCR Accuracy[1], where it proved its worth against the commercial engines of the time. In late 2005, HP released Tesseract for open source. It is now available at http://code.google.com/p/tesseract-ocr.

2. Architecture

Since HP had independently-developed page layout analysis technology that was used in products, (and therefore not released for open-source) Tesseract never needed its own page layout analysis. Tesseract therefore assumes that its input is a binary image with optional polygonal text regions defined.

Processing follows a traditional step-by-step pipeline, but some of the stages were unusual in their day, and possibly remain so even now. The first step is a connected component analysis in which outlines of the components are stored. This was a computationally expensive design decision at the time, but had a significant advantage: by inspection of the nesting of outlines, and the number of child and grandchild outlines, it is simple to detect inverse text and recognize it as easily as black-on-white text. Tesseract was probably the first OCR engine able to handle white-on-black text so trivially. At this stage, outlines are gathered together, purely by nesting, into *Blobs*.

Blobs are organized into text lines, and the lines and regions are analyzed for fixed pitch or proportional text. Text lines are broken into words differently according to the kind of character spacing. Fixed pitch text is chopped immediately by character cells. Proportional text is broken into words using definite spaces and fuzzy spaces.

Recognition then proceeds as a two-pass process. In the first pass, an attempt is made to recognize each word in turn. Each word that is satisfactory is passed to an adaptive classifier as training data. The adaptive classifier then gets a chance to more accurately recognize text lower down the page.

Since the adaptive classifier may have learned something useful too late to make a contribution near the top of the page, a second pass is run over the page, in which words that were not recognized well enough are recognized again.

A final phase resolves fuzzy spaces, and checks alternative hypotheses for the x-height to locate small-cap text.

3. Line and Word Finding

3.1. Line Finding

The line finding algorithm is one of the few parts of Tesseract that has previously been published [3]. The line finding algorithm is designed so that a skewed page can be recognized without having to de-skew, thus saving loss of image quality. The key parts of the process are blob filtering and line construction.

Assuming that page layout analysis has already provided text regions of a roughly uniform text size, a simple percentile height filter removes drop-caps and vertically touching characters. The median height approximates the text size in the region, so it is safe to filter out blobs that are smaller than some fraction of the median height, being most likely punctuation, diacritical marks and noise.

The filtered blobs are more likely to fit a model of non-overlapping, parallel, but sloping lines. Sorting and processing the blobs by x-coordinate makes it possible to assign blobs to a unique text line, while tracking the slope across the page, with greatly reduced danger of assigning to an incorrect text line in the presence of skew. Once the filtered blobs have been assigned to lines, a least median of squares fit [4] is used to estimate the baselines, and the filtered-out blobs are fitted back into the appropriate lines.

The final step of the line creation process merges blobs that overlap by at least half horizontally, putting diacritical marks together with the correct base and correctly associating parts of some broken characters.

3.2. Baseline Fitting

Once the text lines have been found, the baselines are fitted more precisely using a quadratic spline. This was another first for an OCR system, and enabled Tesseract to handle pages with curved baselines [5], which are a common artifact in scanning, and not just at book bindings.

The baselines are fitted by partitioning the blobs into groups with a reasonably continuous displacement for the original straight baseline. A quadratic spline is fitted to the most populous partition, (assumed to be the baseline) by a least squares fit. The quadratic spline has the advantage that this calculation is reasonably stable, but the disadvantage that discontinuities can arise when multiple spline segments are required. A more traditional cubic spline [6] might work better.

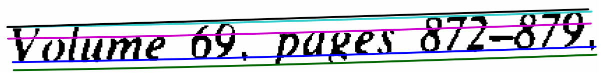


Fig. 1. An example of a curved fitted baseline.

Fig.1 shows an example of a line of text with a fitted baseline, descender line, meanline and ascender line. All these lines are "parallel" (the y separation is a constant over the entire length) and slightly curved. The ascender line is cyan (prints as light gray) and the black line above it is actually straight. Close inspection shows that the cyan/gray line is curved relative to the straight black line above it.

3.3. Fixed Pitch Detection and Chopping

Tesseract tests the text lines to determine whether they are fixed pitch. Where it finds fixed pitch text, Tesseract chops the words into characters using the pitch, and disables the chopper and associator on these words for the word recognition step. Fig. 2 shows a typical example of a fixed-pitch word.

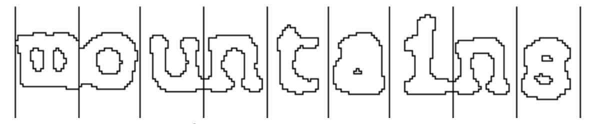


Fig. 2. A fixed-pitch chopped word.

3.4. Proportional Word Finding

Non-fixed-pitch or proportional text spacing is a highly non-trivial task. Fig. 3 illustrates some typical problems. The gap between the tens and units of '11.9%' is a similar size to the general space, and is certainly larger than the kerned space between 'erated' and 'junk'. There is no horizontal gap at all between the bounding boxes of 'of' and 'financial'. Tesseract solves most of these problems by measuring gaps in a limited vertical range between the baseline and mean line. Spaces that are close to the threshold at this stage are made fuzzy, so that a final decision can be made after word recognition.

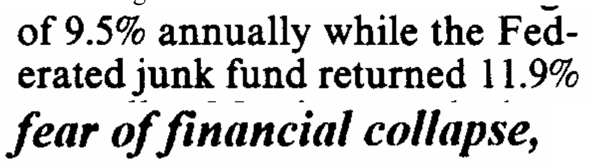

of 9.5% annually while the Fed-
erated junk fund returned 11.9%
fear of financial collapse,

Fig. 3. Some difficult word spacing.

4. Word Recognition

Part of the recognition process for any character recognition engine is to identify how a word should be segmented into characters. The initial segmentation output from line finding is classified first. The rest of the word recognition step applies only to non-fixed-pitch text.

4.1 Chopping Joined Characters

While the result from a word (see section 6) is unsatisfactory, Tesseract attempts to improve the result by chopping the blob with worst confidence from the character classifier. Candidate chop points are found from concave vertices of a polygonal approximation [2] of the outline, and may have either another concave vertex opposite, or a line segment. It may take up to 3 pairs of chop points to successfully separate joined characters from the ASCII set.

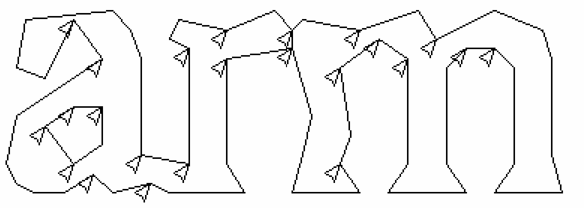


Fig. 4. Candidate chop points and chop.

Fig. 4 shows a set of candidate chop points with arrows and the selected chop as a line across the outline where the 'r' touches the 'm'.

Chops are executed in priority order. Any chop that fails to improve the confidence of the result is undone, but not completely discarded so that the chop can be re-used later by the associator if needed.

4.2. Associating Broken Characters

When the potential chops have been exhausted, if the word is still not good enough, it is given to the *associator*. The associator makes an A* (best first) search of the segmentation graph of possible combinations of the maximally chopped blobs into candidate characters. It does this without actually building the segmentation graph, but instead maintains a hash table of visited states. The A* search proceeds by pulling candidate new states from a priority queue and evaluating them by classifying unclassified combinations of fragments.

It may be argued that this fully-chop-then-associate approach is at best inefficient, at worst liable to miss important chops, and that may well be the case. The advantage is that the chop-then-associate scheme simplifies the data structures that would be required to maintain the full segmentation graph.

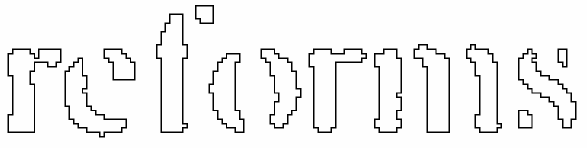


Fig. 5. An easily recognized word.

When the A* segmentation search was first implemented in about 1989, Tesseract's accuracy on broken characters was well ahead of the commercial engines of the day. Fig. 5 is a typical example. An essential part of that success was the character classifier that could easily recognize broken characters.

5. Static Character Classifier

5.1. Features

An early version of Tesseract used topological features developed from the work of Shillman et. al. [7-8] Though nicely independent of font and size, these features are not robust to the problems found in real-life images, as Bokser [9] describes. An intermediate idea involved the use of segments of the polygonal approximation as features, but this approach is also not robust to damaged characters. For example, in Fig. 6(a), the right side of the shaft is in two main pieces, but in Fig. 6(b) there is just a single piece.

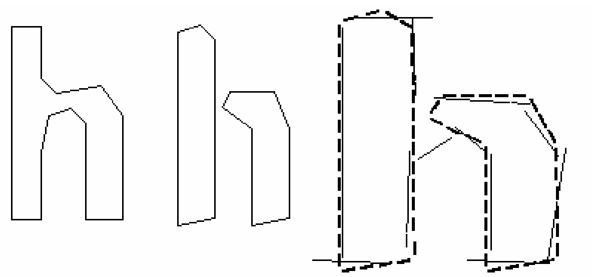


Fig. 6. (a) Pristine 'h, (b) broken 'h', (c) features matched to prototypes.

The breakthrough solution is the idea that the features in the unknown need not be the same as the features in the training data. During training, the segments of a polygonal approximation [2] are used for features, but in recognition, features of a small, fixed length (in normalized units) are extracted from the outline and matched many-to-one against the clustered prototype features of the training data. In Fig. 6(c), the short, thick lines are the features extracted from the unknown, and the thin, longer lines are the clustered segments of the polygonal approximation that are used as prototypes. One prototype bridging the two pieces is completely unmatched. Three features on one side and two on the other are unmatched, but, apart from those, every prototype and every feature is well matched. This example shows that this process of small features matching large prototypes is easily able to cope with recognition of damaged images. Its main problem is that the computational cost of computing the distance between an unknown and a prototype is very high.

The features extracted from the unknown are thus 3-dimensional, (x, y position, angle), with typically 50-100 features in a character, and the prototype features are 4-dimensional (x, y, position, angle, length), with typically 10-20 features in a prototype configuration.

5.2. Classification

Classification proceeds as a two-step process. In the first step, a *class pruner* creates a shortlist of character classes that the unknown might match. Each feature fetches, from a coarsely quantized 3-dimensional look-up table, a bit-vector of classes that it might match, and the bit-vectors are summed over all the features. The classes with the highest counts (after correcting for expected number of features) become the short-list for the next step.

Each feature of the unknown looks up a bit vector of prototypes of the given class that it might match, and then the actual similarity between them is computed. Each prototype character class is represented by a logical sum-of-product expression with each term called a *configuration*, so the distance calculation process keeps a record of the total similarity evidence of each feature in each configuration, as well as of each prototype. The best combined distance, which is calculated from the summed feature and prototype evidences, is the best over all the stored configurations of the class.

5.3. Training Data

Since the classifier is able to recognize damaged characters easily, the classifier was not trained on damaged characters. In fact, the classifier was trained on a mere 20 samples of 94 characters from 8 fonts in a single size, but with 4 attributes (normal, bold, italic, bold italic), making a total of 60160 training samples. This is a significant contrast to other published classifiers, such as the Calera classifier with more than a million samples [9], and Baird's 100-font classifier [10] with 1175000 training samples.

6. Linguistic Analysis

Tesseract contains relatively little linguistic analysis. Whenever the word recognition module is considering a new segmentation, the linguistic module (mis-named the permuter) chooses the best available word string in each of the following categories: Top frequent word, Top dictionary word, Top numeric word, Top UPPER case word, Top lower case word (with optional initial upper), Top classifier choice word. The final decision for a given segmentation is simply the word with the lowest total distance rating, where each of the above categories is multiplied by a different constant.

Words from different segmentations may have different numbers of characters in them. It is hard to compare these words directly, even where a classifier claims to be producing probabilities, which Tesseract does not. This problem is solved in Tesseract by generating two numbers for each character classification. The first, called the confidence, is minus the normalized distance from the prototype. This enables it to be a "confidence" in the sense that greater numbers are better, but still a distance, as, the farther from zero, the greater the distance. The second output, called the rating, multiplies the normalized distance from the prototype by the total outline length in the unknown character. Ratings for characters within a word can be summed meaningfully, since the total outline length for all characters within a word is always the same.

7. Adaptive Classifier

It has been suggested [11] and demonstrated [12] that OCR engines can benefit from the use of an adaptive classifier. Since the static classifier has to be good at generalizing to any kind of font, its ability to discriminate between different characters or between characters and non-characters is weakened. A more font-sensitive adaptive classifier that is trained by the output of the static classifier is therefore commonly [13] used to obtain greater discrimination within each document, where the number of fonts is limited.

Tesseract does not employ a template classifier, but uses the same features and classifier as the static classifier. The only significant difference between the static classifier and the adaptive classifier, apart from the training data, is that the adaptive classifier uses isotropic baseline/x-height normalization, whereas the static classifier normalizes characters by the centroid (first moments) for position and second moments for anisotropic size normalization.

The baseline/x-height normalization makes it easier to distinguish upper and lower case characters as well as improving immunity to noise specks. The main benefit of character moment normalization is removal of font aspect ratio and some degree of font stroke width. It also makes recognition of sub and superscripts simpler, but requires an additional classifier feature to distinguish some upper and lower case characters. Fig. 7 shows an example of 3 letters in baseline/x-height normalized form and moment normalized form.

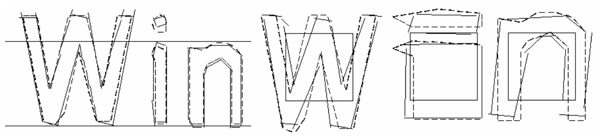


Fig. 7. Baseline and moment normalized letters.

8. Results

Tesseract was included in the 4th UNLV annual test [1] of OCR accuracy, as "HP Labs OCR," but the code has changed a lot since then, including conversion to Unicode and retraining. Table 1 compares results from a recent version of Tesseract (shown as 2.0) with the original 1995 results (shown as HP). All four 300 DPI binary test sets that were used in the 1995 test are shown, along with the number of errors (Errs), the percent error rate (%Err) and the percent change relative to the 1995 results (%Chg) for both character errors and non-stopword errors. [1] More up-to-date results are at http://code.google.com/p/tesseract-ocr.

Table 1. Results of Current and old Tesseract.

		Character			Word		
Ver	Set	Errs	%Err	%Chg	Errs	%Err	%Chg
HP	bus	5959	1.86		1293	4.27	
2.0	bus	6449	2.02	8.22	1295	4.28	0.15
HP	doe	36349	2.48		7042	5.13	
2.0	doe	29921	2.04	-17.68	6791	4.95	-3.56
HP	mag	15043	2.26		3379	5.01	
2.0	mag	14814	2.22	-1.52	3133	4.64	-7.28
HP	news	6432	1.31		1502	3.06	
2.0	news	7935	1.61	23.36	1284	2.62	-14.51
2.0	total	59119		-7.31	12503		-5.39

9. Conclusion and Further Work

After lying dormant for more than 10 years, Tesseract is now behind the leading commercial engines in terms of its accuracy. Its key strength is probably its unusual choice of features. Its key weakness is probably its use of a polygonal approximation as input to the classifier instead of the raw outlines.

With internationalization done, accuracy could probably be improved significantly with the judicious addition of a Hidden-Markov-Model-based character n-gram model, and possibly an improved chopper.

10. Acknowledgements

The author would like to thank John Burns and Tom Nartker for their efforts in making Tesseract open source, the ISRI group at UNLV for sharing their tools and data, as well as Luc Vincent, Igor Krivokon, Dar-Shyang Lee, and Thomas Kielbus for their comments on the content of this paper.

11. References

[1] S.V. Rice, F.R. Jenkins, T.A. Nartker, *The Fourth Annual Test of OCR Accuracy, Technical Report 95-03*, Information Science Research Institute, University of Nevada, Las Vegas, July 1995.

[2] R.W. Smith, *The Extraction and Recognition of Text from Multimedia Document Images,* PhD Thesis, University of Bristol, November 1987.

[3] R. Smith, "A Simple and Efficient Skew Detection Algorithm via Text Row Accumulation", *Proc. of the 3rd Int. Conf. on Document Analysis and Recognition* (Vol. 2), IEEE 1995, pp. 1145-1148.

[4] P.J. Rousseeuw, A.M. Leroy, *Robust Regression and Outlier Detection*, Wiley-IEEE, 2003.

[5] S.V. Rice, G. Nagy, T.A. Nartker, *Optical Character Recognition: An Illustrated Guide to the Frontier*, Kluwer Academic Publishers, USA 1999, pp. 57-60.

[6] P.J. Schneider, "An Algorithm for Automatically Fitting Digitized Curves", in A.S. Glassner, *Graphics Gems I*, Morgan Kaufmann, 1990, pp. 612-626.

[7] R.J. Shillman, *Character Recognition Based on Phenomenological Attributes: Theory and Methods,* PhD. Thesis*,* Massachusetts Institute of Technology. 1974.

[8] B.A. Blesser, T.T. Kuklinski, R.J. Shillman, "Empirical Tests for Feature Selection Based on a Pscychological Theory of Character Recognition", *Pattern Recognition* **8**(2), Elsevier, New York, 1976.

[9] M. Bokser, "Omnidocument Technologies", *Proc. IEEE* **80**(7), IEEE, USA, Jul 1992, pp. 1066-1078.

[10] H.S. Baird, R. Fossey, "A 100-Font Classifier", *Proc. of the 1st Int. Conf. on Document Analysis and Recognition*, IEEE, 1991, pp 332-340.

[11] G. Nagy, "At the frontiers of OCR", *Proc. IEEE* **80(7)**, IEEE, USA, Jul 1992, pp 1093-1100.

[12] G. Nagy, Y. Xu, "Automatic Prototype Extraction for Adaptive OCR", *Proc. of the 4th Int. Conf. on Document Analysis and Recognition,* IEEE, Aug 1997, pp 278-282.

[13] I. Marosi, "Industrial OCR approaches: architecture, algorithms and adaptation techniques", *Document Recognition and Retrieval XIV,* SPIE Jan 2007, 6500-01.

2009 10th International Conference on Document Analysis and Recognition

Hybrid Page Layout Analysis via Tab-Stop Detection

Ray Smith
Google Inc. 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA.
theraysmith@gmail.com

Abstract

A new hybrid page layout analysis algorithm is proposed, which uses bottom-up methods to form an initial data-type hypothesis and locate the tab-stops that were used when the page was formatted. The detected tab-stops, are used to deduce the column layout of the page. The column layout is then applied in a top-down manner to impose structure and reading-order on the detected regions.

The complete C++ source code implementation is available as part of the Tesseract open source OCR engine at http://code.google.com/p/tesseract-ocr.

1. Introduction

Physical Page layout analysis, one of the first steps of OCR, divides an image into areas of text and non-text, as well as splitting multi-column text into columns. This paper does not address *logical* layout analysis, which detects headers, footers, body text, numbered lists, and segmentation into articles.

Physical Layout Analysis is essential to enable an OCR engine to process images of arbitrary pages, such as from books, magazines, journals, newspapers, letters, and reports. Methods for physical layout analysis fall roughly into two categories:

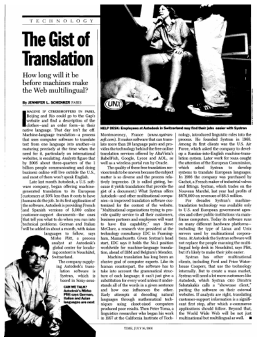
Bottom-up methods are both the oldest [1] and more recently published [2,3] methods. They classify small parts of the image (pixels, groups of pixels, or connected components), and gather together like types to form regions. The key advantage of bottom-up methods is that they can handle arbitrarily shaped regions with ease. The key disadvantage is that they struggle to take into account higher-level structures in the image, such as columns. This often leads to over-fragmented regions.

Top-down methods [4] cut the image recursively in vertical and horizontal directions along whitespaces that are expected to be column boundaries or paragraph boundaries. Although top-down methods have the advantage that they start by looking at the largest structures on the page, they are unable to handle the variety of formats that occur in many magazine pages, such as non-rectangular regions and cross-column headings that blend seamlessly into the columns below.

A third type of method [5-7] is based on analysis of the whitespace in an image. This solves some of the flaws in the recursive top-down methods, by finding gaps between columns by a bottom-up analysis of the gaps, looking explicitly for white rectangles. These algorithms mostly still suffer from the problem of being unable to handle non-rectangular regions.

2. Page layout via tab-stop detection

When a page is laid out, either by a professional publishing system, or by a common word processor, the regions of a page are bounded by *tab-stops.* The margins, column edges, indentation, and columns of a table are all placed at fixed x-positions at which edges or centers of text lines are aligned vertically. Tab-stops distinguish tables from body text, and they also bound rectangular non-column elements, such as inset images and pull-out quotes.


TECHNOLOGY

The Gist of Translation

How long will it be before machines make the Web multilingual?

By JENNIFER L. SCHENKER PARIS

IMAGINE IF CYBERSHOPPERS IN PARIS, Beijing and Rio could go to the Gap's website and find a description of the clothes—and an order form—in their native language. That day isn't far off. Machine-language translation—a process that uses computer software to translate text from one language into another—is maturing precisely at the time when the need for it, particularly on international websites, is escalating. Analysts figure that by 2005 about three-quarters of the 1 billion people communicating and doing business online will live outside the U.S., and most of them won't speak English.

Late last month Autodesk, a U.S. software company, began offering machine-generated translation to its European customers at 50% less than it costs to have humans do the job. In its first application of the software, Autodesk is providing French and Spanish versions of 5,000 online customer-support documents—the ones that tell you what to do when you run into technical problems. German and Italian will be added in about a month, with Asian languages to follow, says Mirko Plitt, a process analyst at Autodesk's global center for localizing content in Neuchâtel, Switzerland.

The company supplying Autodesk's translation software is Systran, which is based in Soisy-sous-Montmorency, France (www.systransoft.com). It makes software that can translate more than 20 language pairs and provides the technology behind the free online translation services offered by AltaVista's BabelFish, Google, Lycos and AOL, as well as a wireless portal run by Oracle.

The quality of these free translation services tends to be uneven because the subject matter is so diverse and the process relatively imprecise. (It is called gisting, because it yields translations that provide the gist of a document.) What Systran offers Autodesk—and other multinational companies—is improved translation software customized for the context of the website. "Multinational organizations trying to provide quality service to all their customers, business partners and employees will want to have this capability," says Steve McClure, a research vice president at the technology consultancy IDC in Framingham, Massachusetts. Given Systran's head start, IDC says it holds the No.1 position worldwide for machine-language translation, ahead of IBM and Belgium's Mendez.

Machine translation has long been an elusive goal of computer experts. Like its human counterpart, the software has to take into account the grammatical structure of each language; it can't just give a substitution for every word unless it understands all of the words in a given sentence and how one influences the other. Early attempts at decoding natural languages through mathematical techniques using closet-sized computers produced poor results. Then Peter Toma, a linguistics researcher who began his work in 1957 at the California Institute of Technology, introduced linguistic rules into the process. He founded Systran in 1968. Among its first clients was the U.S. Air Force, which asked the company to develop a Russian-into-English machine-translation system. Later work for NASA caught the attention of the European Commission, which asked Systran to develop systems to translate European languages. In 1986 the company was purchased by Gachot, a French maker of industrial valves and fittings. Systran, which trades on the Nouveau Marché, last year had profits of $876,000 on revenues of $8.5 million.

For decades Systran's machine-translation technology was available only to U.S. and European government agencies and other public institutions via mainframe computers. Today its software runs on many different hardware platforms—including the type of Linux and Unix servers used by multinational corporations. At Autodesk the Systran software will not replace the people manning the multilingual help desk in Neuchâtel, says Plitt, but it's likely to make their jobs easier.

Systran has other multinational clients, including Ford and Price Waterhouse Coopers, that use the technology internally. But to create a mass market, Systran will need a lot more customers like Autodesk, which Systran CEO Dimitris Sabatakakis calls a "showcase client," putting the software on their external websites. If analysts are right, translating customer-support information is a significant first step, after which e-commerce applications should follow. Beyond that, the World Wide Web will be not just multinational but multilingual as well. ■

HELP DESK: Employees at Autodesk in Switzerland may find their jobs easier with Systran

CAN WE TALK? Autodesk's Mirko Plitt says German, Italian and Asian languages are next

TIME, JULY 16, 2001

Fig.1. Input image.

The tab-stops in the example of Fig. 1 are the column boundaries with an additional tab-stop for the paragraph indentation that is not required for finding the page layout. The non-rectangular inset image, typically, strays outside of the column boundaries.

In some sense, white rectangles match tab-stops, but white rectangles may be disrupted by background noise or background images. Also the ends of white rectangles do not match the ends of the region bounded by the tab-stops, because the white rectangles run on into the perpendicular whitespace.

The proposed algorithm is similar to the whitespace rectangle methods in that it uses a bottom-up method to find a top-down structure, but instead of finding the space between columns, it looks for the tab-stops that mark their edges, and, through further combination of bottom-up and top-down methods, copes easily with non-rectangular regions.

There are for main phases: preprocessing, in which bottom-up morphological and connected component analysis form initial hypotheses over the local data types; bottom-up tab-stop detections; finding the columns layout; and finally applying the column layout to create an ordered set of typed regions. These phases will be detailed in sections 3-6.

3. Preprocessing

The aim of the preprocessing step is to identify line separators, image regions, and separate the remaining connected components into likely text components and a smaller number of uncertain type.

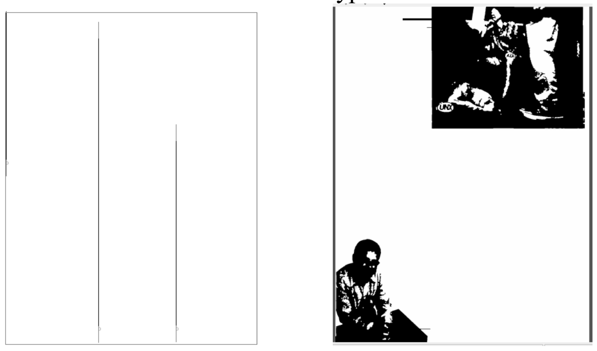


Fig.2. (a) Vertical lines, (b) Image elements.

Starting with the image of Fig. 1, the morphological processing from Leptonica [8] detects the vertical lines shown in Fig. 2(a) and the image mask shown in Fig. 2(b). These detected elements are subtracted from the input image before passing the cleaned image to connected component analysis.

The connected components (CCs) are filtered by width, w, and height, h into small, medium, and large sizes as follows: CCs with $h < 7$ (at 300ppi) are small. The 75th percentile of the heights of the remainder, h_{75}, is computed, and CCs with $h < h_{75}/2$ are small; $h > 2h_{75}$ or $w > 8h_{75}$ are large, and the rest are medium.

This filtration is important, since small CCs (noise or diacriticals) and large non-text CCs, (line drawings, logos, or frames) are likely to confuse the text-line algorithms, but large text headings are important to reading order detection. Large CCs are considered text at this stage if there is a left or right neighbor that has a similar stroke width. On "stressed" fonts, the stroke width is greater on vertical lines than on horizontal lines, so stroke width is calculated separately in both directions. Stroke width is calculated from horizontal and vertical local maxima of the distance function on the binary image of the CC. Fig. 3 shows the CCs are filtered as medium or large text.

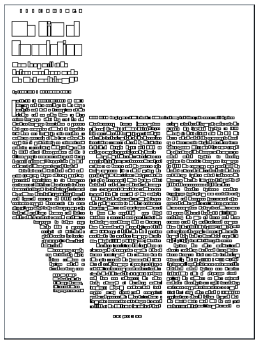


Fig.3. Filtered CCs

4. Finding tab positions as line segments

The process of finding tab-stop line segments has several major sub-steps: candidate tab-stop CCs that look like they may be at the edge of a text region are found and then grouped into tab-stop lines, then connections between tab-stop lines are found, enabling removal of false positives.

4.1. Finding candidate tab-stop components

The initial candidate tab-stop CCs are found by a radial search starting at every filtered CC from preprocessing. Assuming that the CC is at a tab-stop, the search looks for aligned neighbors and neighbors in the gutter where there should be a space. Each CC is processed independently and marked according to whether it is a candidate left tab, right tab or neither. Fig. 4(a) illustrates the candidate tab-stop CCs.

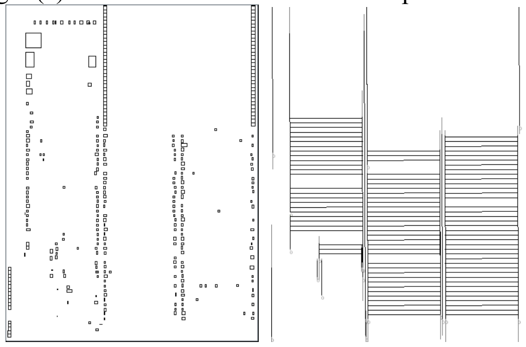


Fig.4. (a) Candidate tab-stop components (b) Fitted tab lines and traces connections.

4.2. Grouping candidate tab components

Candidate tab CCs are grouped into lines, and, where there are sufficiently many CCs in a group, they are kept. A least median of squares algorithm is used to fit a line to the appropriate (left or right) edge of each CC in a group. After finding all tab-stop line segments, all the lines are refitted to the page-mean direction, such that all the member tab CCs fall to one side of the line segment.

4.3. Tracking text lines to connect tab stops

The next step connects tab-stops by tracking text lines from one tab-stop to another. Closely adjacent, vertically overlapping CCs qualify, but large gaps cannot be jumped. Tab-stops that have a text-line connecting them are associated with each other, as being likely opposite sides of a text column. Fig. 4(b) shows the tab-stop lines and connecting text lines. Tab-stop lines that have no connection are discarded.

The most frequently occurring widths of the text lines connecting tab stops are recorded for use in finding the column layout.

4.4. Cleaning up tab stop ends

The final step attempts to make connected tab lines end at the same y coordinate, by allowing the ends to move between the last member CC whose edge was used for the tab line, and the first non-member CC that the line intersects. Fig. 5 shows the final tab line segments.

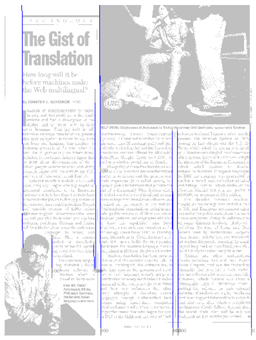


Fig.5. Cleaned tab-stops.

After construction of the tab stops, the CCs are re-classified, as “Text” or “Unknown” using the same text-line tracing algorithm as was used above to find connections between tab stops. If a group of CCs of significant width form a text line, then they are classified as text. Artificial image CCs of about the same size as the body-text CCd are created from the image mask from the morphological preprocessing.

5. Finding the column layout

The next major step is to find the column layout of the page. All the rest of the steps make use of the Column Partition (CP) objects which are created now.

Scanning the CCs from left to right and top to bottom, runs of similarly classified (text, image, or unknown) CCs are gathered into CPs, subject to the constraint that no CP may cross a tab stop line. Fig. 6 shows the result of this process. A collection of CPs from a single horizontal scan are stored in a Column Partition Set (CPset).

Each CPset is potentially a division of the page into columns at that vertical position. Finding the column layout is therefore a process of finding an optimal set of CPsets that best "explains" (see below) all the CPsets on the page, but first some definitions:

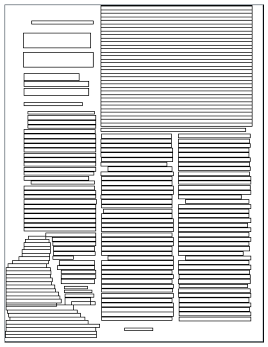


Fig.6. Column Partitions (CPs)

A *good* CP either touches a tab line on both vertical edges of its bounding box, or its width is close to a frequently occurring width. (See 4.3.)

The *coverage* of a CPset is the total width of all the good CPs that it contains.

CPset A *is better than* CPset B if A has greater coverage, or equal coverage, but more good CPs, or equal good CPs, but more total CPs.

CPset A *explains* set B **unless** one or more of the following are true:

1. The edge of one of B's CPs lies outside of all of A's CPs. This is not allowed, as it shows that B has more text than A.
2. The edges of one of B's CPs fall in different CPs of A, and the width of the B CP is a common one. This means that A has split a column of common width.
3. The right edge of one of B's CPs falls in the same A CP as the left edge of the next B CP, and the B CPs are of roughly the same width. It looks like A has a different number of columns to B. The same-width condition allows A to explain B with a pull-out.
4. Both edges of two CPs of B fall in the same CP of A. This means that A has merged two columns of B.

Note that the two edges of one of B's CPs are allowed to fall into two CPs of A, as long as the width is not a common one. This allows headings that merge columns in B to be explained by A.

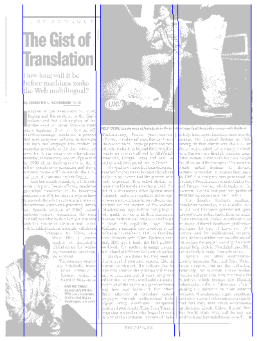


Fig.7. Columns.

A list of column candidates is made from the set of CPsets on the page, ordered best first, and with duplicates eliminated by the A explains B rules above. In this process, all image CPs are ignored.

After the initial candidates are made, they are improved by adding new CPs and widening existing CPs, by using the edge of a CP in a different CPSet while widening doesn't cause overlap of CPs.

An iterative process then labels the longest segment of consecutive (allowing for a very small region of failure) page y-coordinates that is explained by one of the column candidates. Fig. 7 shows the result of this process.

6. Finding the regions

After the columns are found, CPs are given a type according to how many columns they span. CPs within a single column are *flowing*, partitions that touch more than one column, but do not span to the outer edges of either are *pull-out*, and partitions that completely span more than one column are *heading*.

6.1. Create flows of CPs

Each CP chooses its best matching upper and lower partner, being the vertically nearest CP that overlaps horizontally. Since each CP registers itself with its chosen partner, each CP may have zero or more registered upper and lower partners.

The size of the list of registered partners is forced to become zero or one for each of upper and lower, using the following rules in order:
1. Type. If there are multiple types, text can only stay with its own (exact) type, whereas image can stay with any other image type.
2. Transitive partner shortcuts are broken. If A has 2 partners B and C, and also B has C as a partner in the same direction, then delete C as a partner of A, leaving a clean chain A-B-C. Also if A has a partner B, and B has a partner A in the same direction, break the cycle.
3. (Text only) If A still has 2 partners B, C, chase B and C's partners to see which has the longest chain. Delete from A the partner that has the shortest chain, and convert the type of the shortest chain to pull-out.
4. (Image only) Choose the partner CP with the largest horizontal overlap.

All CPs now have 0 or 1 partners. Even so, (re)run rule 1 above. This purifies all chains of text to a single type and splits text chains from image chains. Image chains are purified by setting all CPs in a chain to the most general type in the chain. Fig. 8 shows the final typed CPs, where flowing text is blue, heading text is cyan, heading image is magenta, and pull-out image is orange.

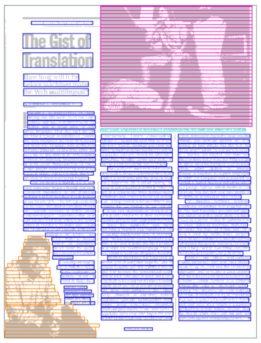


Fig.8. Typed partition chains.

Chains of text CPs are further divided into groups of uniform line-spacing, which make text blocks. Now each chain of CPs represents a candidate region, but the regions have to be ordered.

6.2. Reading order determination

Recall that image and text partitions are typed as one of 3 possibilities: flowing, pull-out, and heading. Also, the page is divided into sections of a consistent column layout. With this information, a reasonable reading order drops out of a few simple rules:
1. Flowing blocks follow by y position within a column.
2. Pull-out blocks follow by y position in an imaginary column between the real columns that they touch.
3. A heading spans multiple columns and follows anything that is above it in the columns spanned, or between them. Anything that lies in the same columns below the heading follows after it.
4. A change in column layout works just like a heading. Anything in any columns that are changed (or between them) goes before anything in the new columns. Unchanged columns are unaffected by a change in column layout.
5. Between headings, the content of columns is ordered from left to right.

6.3. Find the polygon boundary for each region

For simplicity of implementation, the region polygons are isothetic: i.e. edges alternate between being horizontal and parallel to the mean tab line (Approximately vertical.) The polygon edges are chosen to minimize the number of vertices, while satisfying the constraint that all CPs are contained within their region polygon, and no CP from another region intersects. Fig. 9 shows the final blocks created for the input image of Fig. 1.

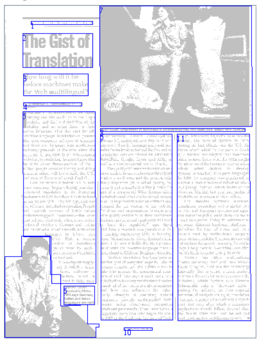


Fig.9. Final blocks.

7. Testing and results

The algorithm described herein is implemented in C++, and the source code is available as part of the Tesseract open source OCR system [9,10]. It runs on a typical 8MPixel image in approximately 1 second on a 3.4 GHz Pentium 4.

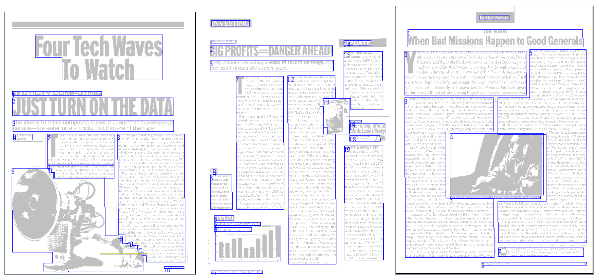


Fig.10. Results on some of the ICDAR2007 set.

Properly testing page layout analysis is a difficult problem [11] with very little publicly available ground-truth for complex magazine pages. The UNLV test set [12], only measures text regions, and counts errors unless figure captions are placed after all the body text.

The ICDAR page layout analysis competitions provide better measurement of overall accuracy, and the results of this algorithm appear in the 2009 competition [13]. Some graphical results are shown in Fig. 10 and numerical comparisons with the entrants in the ICDAR 2007 competition are shown in Table 1. The results in Table 1 are computed on only the 2007 test set, and the author would like to thank Apostolos Antonocopoulos for providing these results. For details on the testing methodology, see references [11] and [13].

Table 1. Results on the ICDAR 2007 set.

Method	Noise	Sep	Text	Image	Overall
		PRImA Metric			
2007-Besus	86.8%	76.9%	37.4%	42.5%	35.9%
2007-TH1	68.0%	79.7%	76.1%	46.2%	67.6%
2007-TH2	67.6%	79.6%	72.9%	48.4%	65.7%
Tesseract	65.6%	74.1%	72.1%	55.3%	68.4%
		F-Measure			
2007-Besus	62.9%	76.2%	95.8%	57.2%	90.2%
2007-TH1	79.2%	80.7%	91.9%	72.1%	88.2%
2007-TH2	79.2%	80.6%	92.3%	72.4%	88.6%
Tesseract	79.2%	70.9%	93.3%	82.0%	91.3%
		Recall			
2007-Besus	65.7%	71.7%	94.9%	67.0%	88.2%
2007-TH1	65.6%	79.5%	96.9%	66.4%	89.8%
2007-TH2	65.6%	79.5%	97.2%	66.9%	90.2%
Tesseract	65.6%	81.4%	97.9%	76.5%	93.8%
		Precision			
2007-Besus	60.4%	81.3%	96.7%	50.0%	92.2%
2007-TH1	100.0%	81.9%	87.4%	79.0%	86.7%
2007-TH2	100.0%	81.7%	87.9%	79.0%	87.0%
Tesseract	100.0%	62.8%	89.0%	88.3%	88.9%

10. Conclusion and further work

Tab-stops make an interesting and useful alternative to white rectangles for finding the column structure of a page. Combining the top-down concept of column structure with bottom-up classification methods enables page layout analysis to easily handle the complex non-rectangular layouts of modern magazine pages without losing sight of the "bigger picture" that often happens when bottom-up methods are used alone.

The algorithm described has no table detection or analysis, but the tab-stops make particularly useful features for both, so table analysis will be added in the future.

11. References

[1] F. Wahl, K. Wong, R. Casey, "Block segmentation and text extraction in mixed text/image documents," *Computer Graphics and Image Processing,* **20**, 1982, pp375-390.

[2] M. Chen, X. Q. Ding, "Unified HMM-based Layout Analysis Framework and Algorithm," SCI CHINA Ser F, **46**(6), Dec. 2003, pp401-408.

[3] S.P. Chowdhury, S. Mandal, A.K. Das, B. Chanda, "Segmentation of Text and Graphics from Document Images," *Proc. of the 9th Int. Conf. on Document Analysis and Recognition,* IEEE, Curitiba, Brazil, Sep 2007, pp619-623.

[4] G. Nagy, S.C. Seth, "Hierarchical Representation of Optically Scanned Documents" *Proc. 7th Int. Conf. on Pattern Recognition,* Montreal, Canada, 1984, pp347-349.

[5] H.S. Baird, S.E. Jones, S.J. Fortune, "Image Segmentation by Shape-directed Covers," *Proc. 10th Int. Conference on Pattern Recognition,* IEEE Atlantic City, NJ, 1990, pp820-825.

[6] T. Pavlidis, J. Zhou, "Page Segmentation and Classification," *CVGIP: Graphical Models and Image Processing,* **54**(6), November 1992, pp484-496.

[7] T.M. Breuel, "Two Geometric Algorithms for Layout Analysis," *Proc. of the 5th Int. Workshop on Document Analysis Systems V,* Springer-Verlag 2002, pp188-199.

[8] Leptonica image processing and analysis library. http://www.leptonica.com.

[9] R. Smith. "An overview of the Tesseract OCR Engine." *Proc 9th Int. Conf. on Document Analysis and Recognition*, IEEE, Curitiba, Brazil, Sep 2007, pp629-633.

[10] The Tesseract open source OCR engine. http://code.google.com/p/tesseract-ocr.

[11] A. Antonacopoulos, B. Gatos, D. Bridson, "ICDAR2007 Page Segmentation Competition," *Proc 9th Int. Conf. on Document Analysis and Recognition*, IEEE, Curitiba, Brazil, Sep 2007, pp1279-1283.

[12] UNLV ISRI OCR testing toolkit and database http://www.isri.unlv.edu/ISRI/OCRtk.

[13] A. Antonacopoulos et. al. "ICDAR2009 Page Segmentation Competition," *Proc 10th Int. Conf. on Document Analysis and Recognition*, IEEE, Barcelona, Spain, Jul 2009.

Adapting the Tesseract Open Source OCR Engine for Multilingual OCR

Ray Smith Daria Antonova Dar-Shyang Lee

Google Inc., 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA.

Abstract
We describe efforts to adapt the Tesseract open source OCR engine for multiple scripts and languages. Effort has been concentrated on enabling generic multi-lingual operation such that negligible customization is required for a new language beyond providing a corpus of text. Although change was required to various modules, including physical layout analysis, and linguistic post-processing, no change was required to the character classifier beyond changing a few limits. The Tesseract classifier has adapted easily to Simplified Chinese. Test results on English, a mixture of European languages, and Russian, taken from a random sample of books, show a reasonably consistent word error rate between 3.72% and 5.78%, and Simplified Chinese has a character error rate of only 3.77%.

Keywords
Tesseract, Multi-Lingual OCR.

1. Introduction
Research interest in Latin-based OCR faded away more than a decade ago, in favor of Chinese, Japanese, and Korean (CJK) [1,2], followed more recently by Arabic [3,4], and then Hindi [5,6]. These languages provide greater challenges specifically to classifiers, and also to the other components of OCR systems. Chinese and Japanese share the Han script, which contains thousands of different character shapes. Korean uses the Hangul script, which has several thousand more of its own, as well as using Han characters. The number of characters is one or two orders of magnitude greater than Latin. Arabic is mostly written with connected characters, and its characters change shape according to the position in a word. Hindi combines a small number of alphabetic letters into thousands of shapes that represent syllables. As the letters combine, they form ligatures whose shape only vaguely resembles the original letters. Hindi then combines the problems of CJK and Arabic, by joining all the symbols in a word with a line called the shiro-reka.

Research approaches have used language-specific work-arounds to avoid the problems in some way, since that is simpler than trying to find a solution that works for all languages. For instance, the large character sets of Han, Hangul, and Hindi are mostly made up of a much smaller number of components, known as radicals in Han, Jamo in Hangul, and letters in Hindi. Since it is much easier to develop a classifier for a small number of classes, one approach has been to recognize the radicals [1, 2, 5] and infer the actual characters from the combination of radicals. This approach is easier for Hangul than for Han or Hindi, since the radicals don't change shape much in Hangul characters, whereas in Han, the radicals often are squashed to fit in the character and mostly touch other radicals. Hindi takes this a step further by changing the shape of the consonants when they form a conjunct consonant ligature. Another example of a more language-specific work-around for Arabic, where it is difficult to determine the character boundaries to segment connected components into characters. A commonly used method is to classify individual vertical pixel strips, each of which is a partial character, and combine the classifications with a Hidden Markov Model that models the character boundaries [3].

Google is committed to making its services available in as many languages as possible [7], so we are also interested in adapting the Tesseract Open Source OCR Engine [8, 9] to many languages. This paper discusses our efforts so far in fully internationalizing Tesseract, and the surprising ease with which some of it has been possible. Our approach is use language generic methods, to minimize the manual effort to cover many languages.

2. Review Of Tesseract For Latin

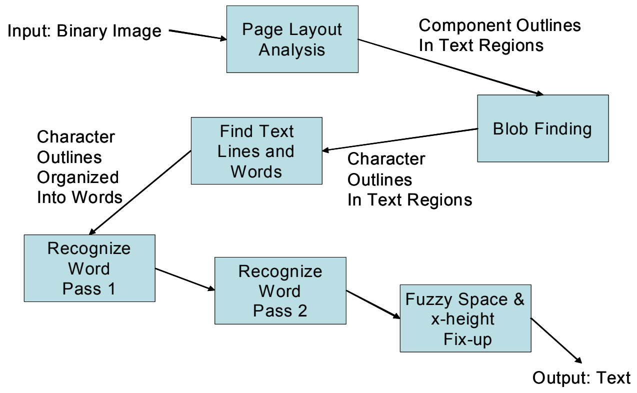

Input: Binary Image
Page Layout Analysis
Component Outlines In Text Regions
Blob Finding
Find Text Lines and Words
Character Outlines In Text Regions
Character Outlines Organized Into Words
Recognize Word Pass 1
Recognize Word Pass 2
Fuzzy Space & x-height Fix-up
Output: Text

Figure 1. Top-level block diagram of Tesseract.

Fig. 1 is a block diagram of the basic components of Tesseract. The new page layout analysis for Tesseract [10] was designed from the beginning to be language-independent, but the rest of the engine was developed for English, without a great deal of thought as to how it might work for other languages. After noting that the commercial engines at the time were strictly for black-on-white text, one of the original design goals of Tesseract was that it should recognize white-on-black (inverse video) text as easily as black-on-white. This led the design (fortuitously as it turned out) in the direction of connected component (CC) analysis and operating on outlines of the components. The first step after CC analysis is to find the *blobs* in a text region. A *blob* is a putative classifiable unit, which may be one or more horizontally overlapping CCs, and their inner nested outlines or *holes.* A problem is detecting inverse text inside a box vs. the holes inside a character. For English, there are very few characters (maybe © and ®) that have more than 2 levels of outline, and it is very rare to have more than 2 holes, so any blob that breaks these rules is "clearly" a box containing inverse characters, or even the inside or outside of a frame around black-on-white characters.

After deciding which outlines make up blobs, the text line finder [11] detects (horizontal only) text lines by virtue of the vertical overlap of adjacent characters on a text line. For English the overlap and baseline are so well behaved that they can be used to detect skew very precisely to a very large angle. After finding the text lines, a fixed-pitch detector checks for fixed pitch character layout, and runs one of two different word segmentation algorithms according to the fixed pitch decision. The bulk of the recognition process operates on each word independently, followed by a final fuzzy-space resolution phase, in which uncertain spaces are decided.

Fig.2 is a block diagram of the word recognizer. In most cases, a blob corresponds to a character, so the word recognizer first classifies each blob, and presents the results to a dictionary search to find a word in the combinations of classifier choices for each blob in the word. If the word result is not good enough, the next step is to chop poorly recognized characters, where this improves the classifier confidence. After the chopping possibilities are exhausted, a best-first search of the resulting segmentation graph puts back together chopped character fragments, or parts of characters that were broken into multiple CCs in the original image. At each step in the best-first search, any new blob combinations are classified, and the classifier results are given to the dictionary again. The output for a word is the character string that had the best overall distance-based rating, after weighting according to whether the word was in a dictionary and/or had a sensible arrangement of punctuation around it. For the English version, most of these punctuation rules were hard-coded.

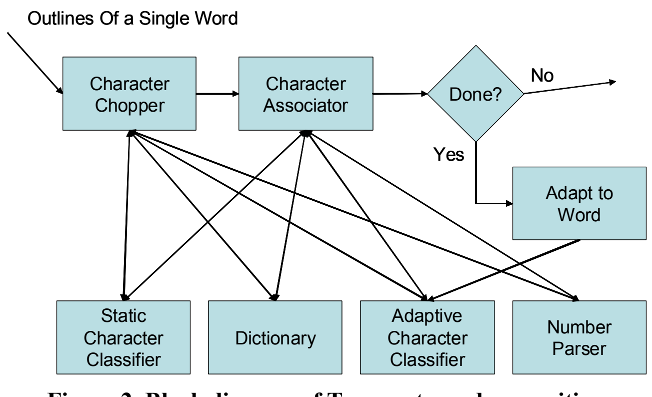


Figure 2. Block diagram of Tesseract word recognition.

The words in an image are processed twice. On the first pass, successful words, being those that are in a dictionary and are not dangerously ambiguous, are passed to an adaptive classifier for training. As soon as the adaptive classifier has sufficient samples, it can provide classification results, even on the first pass. On the second pass, words that were not good enough on pass 1 are processed for a second time, in case the adaptive classifier has gained more information since the first pass over the word.

From the foregoing description, there are clearly problems with this design for non-Latin languages, and some of the more complex issues will be dealt with in sections 3, 4 and 5, but some of the problems were simply complex engineering. For instance, the one byte code for the character class was inadequate, but should it be replaced by a UTF-8 string, or by a wider integer code? At first we adapted Tesseract for the Latin languages, and changed the character code to a UTF-8 string, as that was the most flexible, but that turned out to yield problems with the dictionary representation (see section 5), so we ended up using an index into a table of UTF-8 strings as the internal class code.

3. Layout Preprocessing

Several aspects of the "textord" (text-ordering) module of Tesseract required changes to make it more language-independent. This section discusses these changes.

3.1 Vertical Text Layout

Chinese, Japanese, and Korean, to a varying degree, all read text lines either horizontally or vertically, and often mix directions on a single page. This problem is not unique to CJK, as English language magazine pages often use vertical text at the side of a photograph or article to credit the photographer or author. Vertical text is detected by the page layout analysis. If a majority of the CCs on a tab-stop have both their left side on a left tab and their right side on a right tab, then everything between the tab-stops could be a line of vertical text. To prevent false-positives in tables, a further restriction requires vertical text to have a median vertical gap between CCs to be less than the mean width of the CCs. If the majority of CCs on a page are vertically aligned, the page is rotated by 90 degrees and page layout analysis is run again to reduce the chance of finding false columns in the vertical text. The minority originally horizontal text will then become vertical text in the rotated page, and the body of the text will be horizontal.

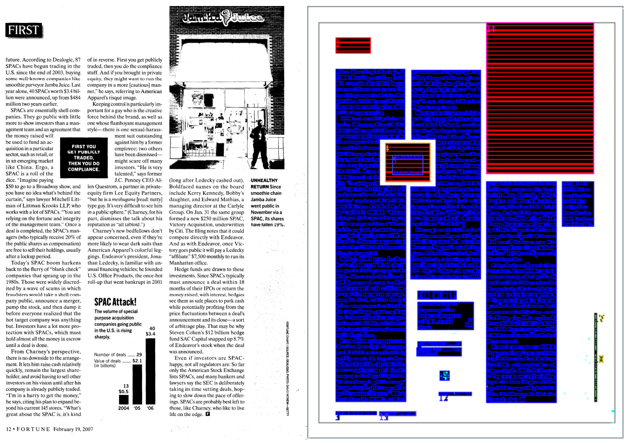


Figure 3. (a) A page containing a verical text region. (b) The detected regions with image in red, horizontal text in blue, and vertical text in yellow.

As originally designed, Tesseract had no capability to handle vertical text, and there are a lot of places in the code where some assumption is made over characters being arranged on a horizontal text line. Fortunately, Tesseract operates on outlines of CCs in a signed integer coordinate space, which makes rotations by multiples of 90 degrees trivial, and it doesn't care whether the coordinates are positive or negative. The solution is therefore simply to differentially rotate the vertical and horizontal text blocks on a page, and rotate the characters as needed for classification. Fig. 3 shows an example of this for English text. The page in Fig. 3(a) contains vertical text at the lower-right, which is detected in Fig. 3(b), along with the rest of the text. In Fig. 4, the vertical text region is rotated 90 degrees clockwise, (centered at the bottom-left of the image), so it appears well below the original image, but in horizontal orientation.

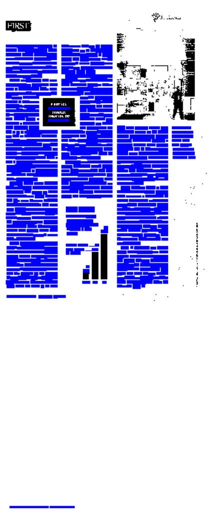


Figure 4. The vertical text is differentially rotated so it is oriented horizontally.

Fig. 5 shows an example for Chinese text. The mainly-vertical body text is rotated out of the image, to make it horizontal, and the header, which was originally horizontal, stays where it started. The vertical and horizontal text blocks are separated in coordinate space, but all Tesseract cares about is that the text lines are horizontal. The data structure for a text block records the rotations that have been performed on a block, so that the inverse rotation can be applied to the characters as they are passed to the classifier, to make them upright. Automatic orientation detection [12] can be used to ensure that the text is upright when passed to the classifier, as vertical text could have characters that are in at least 3 different orientations relative to the reading direction. After Tesseract processes the rotated text blocks, the coordinate space is re-rotated back to the original image orientation so that reported character bounding boxes are still accurate.

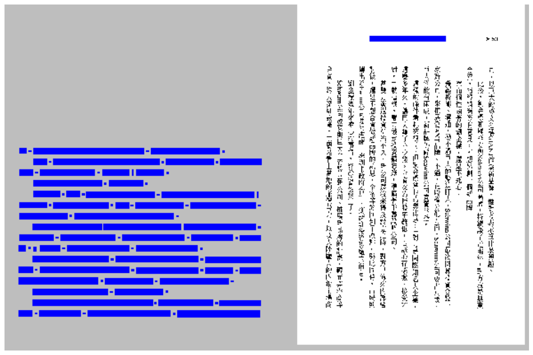


Figure 5. Horizontal text detection for Traditional Chinese. Since the majority of the text is vertical, inside Tesseract it is rotated anticlockwise 90 degrees so it lies outside the image, but the lines are horizontal. The page header, which was already horizontal, remains behind.

3.2 Text-line and Word Finding

The original Tesseract text-line finder [11] assumed that CCs that make up characters mostly vertically overlap the bulk of the text line. The one real exception is i dots. For general languages this is not true, since many languages have diacritics that sit well above and/or below the bulk of the text-line. For Thai for example, the distance from the body of the text line to the diacritics can be quite extreme. The page layout analysis for Tesseract is designed to simplify text-line finding by sub-dividing text regions into blocks of uniform text size and line spacing. This makes it possible to force-fit a line-spacing model, so the text-line finding has been modified to take advantage of this. The page layout analysis also estimates the residual skew of the text regions, which means the text-line finder no longer has to be insensitive to skew.

The modified text-line finding algorithm works independently for each text region from layout analysis, and begins by searching the neighborhood of small CCs (relative to the estimated text size) to find the nearest body-text-sized CC. If there is no nearby body-text-sized CC, then a small CC is regarded as likely noise, and discarded. (An exception has to be made for dotted/dashed leaders, as typically found in a table of contents.) Otherwise, a bounding box that contains both the small CC and its larger neighbor is constructed and used in place of the bounding box of the small CC in the following projection.

A "horizontal" projection profile is constructed, parallel to the estimated skewed horizontal, from the bounding boxes of the CCs using the modified boxes for small CCs. A dynamic programming algorithm then chooses the best set of segmentation points in the projection profile. The cost function is the sum of profile entries at the cut points plus a measure of the variance of the spacing between them. For most text, the sum of profile entries is zero, and the variance helps to choose the most regular line-spacing. For more complex situations, the variance and the modified bounding boxes for small CCs combine to help direct the line cuts to maximize the number of diacriticals that stay with their appropriate body characters.

Once the cut lines have been determined, whole connected components are placed in the text-line that they vertically overlap the most, (still using the modified boxes) except where a component strongly overlaps multiple lines. Such CCs are presumed to be either characters from multiple lines that touch, and so need cutting at the cut line, or drop-caps, in which case they are placed in the top overlapped line. This algorithm works well, even for Arabic.

After text lines are extracted, the blobs on a line are organized into recognition units. For Latin languages, the logical recognition units correspond to space-delimited words, which is naturally suited for a dictionary-based language model. For languages that are not space-delimited, such as Chinese, it is less clear what the corresponding recognition unit should be. One possibility is to treat each Chinese symbol as a recognition unit. However, given that Chinese symbols are composed of multiple glyphs (radicals), it would be difficult to get the correct character segmentation without the help of recognition. Considering the limited amount of information that is available at this early stage of processing, the solution is to break up the blob sequence at punctuations, which can be detected quite reliably based on their size and spacing to the next blob. Although this does not completely resolve the issue of a very long blob sequence, which is a crucial factor in determining the efficiency and quality when searching the segmentation graph, this would at least reduce the lengths of recognition units into more manageable sizes.

As described in Section 2, detection of white-on-black text is based on the nesting complexity of outlines. This same process also rejects non-text, including halftone noise, black regions on the side, or large container boxes as in sidebar or reversed-video region. Part of the filtering is based on a measure of the topological complexity of the blobs, estimated based on the number of interior components, layers of nested holes, perimeter to area ratio, and so on. However, the complexity of Traditional Chinese characters, by any measure, often exceeds that of an English word enclosed in a box. The solution is to apply a different complexity threshold for different languages, and rely on subsequent analysis to recover any incorrectly rejected blobs.

3.3 Estimating x-height in Cyrillic Text

After completing the text line finding step and organizing blocks of blobs into rows, Tesseract estimates x-height for each text line. The x-height estimation algorithm first determines the bounds on the maximum and minimum acceptable x-height based on the initial line size computed for the block. Then, for each line separately, the heights of the bounding boxes of the blobs occurring on the line are quantized and aggregated into a histogram. From this histogram the x-height finding algorithm looks for the two most commonly occurring height modes that are far enough apart to be the potential x-height and ascender height. In order to achieve robustness against the presence of some noise, the algorithm ensures that the height modes picked to be the x-height and ascender height have sufficient number or occurrences relative to the total number of blobs on the line.

This algorithm works quite well for most Latin fonts. However, when applied as-is to Cyrillic, Tesseract fails to find the correct x-height for most of the lines. As a result, on a data set of Russian books the word error-rate of Tesseract turns out to be 97%. The reason for such high error rate is two-fold. First of all the ascender statistics in Cyrillic fonts differ significantly from Latin ones. Simply lowering the threshold for the expected number of ascenders per line is not an effective solution, since it is not infrequent that a line of text would contain one or no ascender letters. The second reason for such poor performance is a high degree of case ambiguity in Cyrillic fonts. For example, out of 33 upper-case modern Russian letters only 6 have a lower-case shape that is significantly different from the upper-case in most fonts. Thus, when working with Cyrillic, Tesseract can be easily misled by the incorrect x-height information and would readily recognize lower-case letters as upper-case.

Our approach to fixing the x-height problem for Cyrillic was to adjust the minimum expected number of ascenders on the line, take into account the descender statistics and use x-height information from the neighboring lines in the same block of text more effectively (a block is a text region identified by the page layout analysis that has a consistent size of text blobs and line-spacing, and therefore is likely to contain letters of the same or similar font sizes).

For a given block of text, the improved x-height finding algorithm first tries to find the x-height of each line individually. Based on the result of this computation each line falls into one of the following four categories: (1) the lines where the x-height and ascender modes were found, (2) where descenders were found, (3) where a common blob height that could be used as an estimate of either cap-height or x-height was found, (4) the lines where none of the above were identified (i.e. most likely lines containing noise with blobs that are too small, too large or just inconsistent in size). If any lines from the first category with reliable x-height and ascender height estimates were found in the block, their height estimates are used for the lines in the second category (lines with descenders present) that have a similar x-height estimate. The same x-height estimate is utilized for those lines in the third category (no ascenders or descenders found), whose most common height is within a small margin of the x-height estimate. If the line-by-line approach does not result in finding any reliable x-height and ascender height modes, the statistics for all the blobs in the text block are aggregated and the same search for x-height and ascender height modes is repeated using this cumulative information.

As the result of the improvements described above the word error

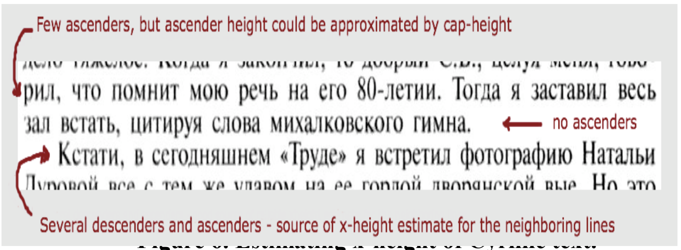

дело тяжелое. Когда я закончил, то добрый С.В., целуя меня, гово-
рил, что помнит мою речь на его 80-летии. Тогда я заставил весь
зал встать, цитируя слова михалковского гимна.
Кстати, в сегодняшнем «Труде» я встретил фотографию Натальи
Дуровой все с тем же удавом на ее гордой дворянской вые. Но это
Few ascenders, but ascender height could be approximated by cap-height
no ascenders
Several descenders and ascenders - source of x-height estimate for the neighboring lines

Figure 3. Estimating x-height of Cyrillic text

rate on a test set of Russian books was reduced to 6%. After the improvements the test set still contained some errors due to the failure to estimate the correct x-height of the text line. However, in many of such cases even a human reader would have to use the information from the neighboring blocks of text or knowledge about the common organization of the books to determine whether the given line is upper- or lower-case.

4. Character / Word Recognition

One of the main challenges to overcome in adapting Tesseract for multilingual OCR is extending what is primarily designed for alphabetical languages to handle ideographical languages like Chinese and Japanese. These languages are characterized by having a large set of symbols and lacking clear word boundaries, which pose serious tests for a search strategy and classification engine designed for well delimited words from small alphabets. We will discuss classification of large set of ideographs in the next section, and describe the modifications required to address the search issue first.

4.1 Segmentation and Search

As mentioned in section 3.2, for non-space delimited languages like Chinese, recognition units that form the equivalence of words in western languages now correspond to punctuation delimited phrases. Two problems need to be considered to deal with these phrases: they involve deeper search than typical words in Latin and they do not correspond to entries in the dictionary. Tesseract uses a best-first-search strategy over the segmentation graph, which grows exponentially with the length of the blob sequence. While this approach worked on shorter Latin words with fewer segmentation points and a termination condition when the result is found in the dictionary, it often exhausts available resources when classifying a Chinese phrase. To resolve this issue, we need to dramatically reduce the number of segmentation points evaluated in the permutation and devise a termination condition that is easier to meet.

In order to reduce the number of segmentation points, we incorporate the constraint of roughly constant character widths in a mono-spaced language like Chinese and Japanese. In these languages, characters mostly have similar aspect ratios, and are either full-pitch or half-pitch in their positioning. Although the normalized width distribution would vary across fonts, and the spacing would shift due to line justification and inclusion of digits or Latin words, which is not uncommon, by and large these constraints provide a strong guideline for whether a particular segmentation point is compatible with another. Therefore, using the deviation from the segmentation model as a cost, we can eliminate a lot of implausible segmentation states and effectively reduce the search space. We also use this estimate to prune the search space based on the best partial solution, making it effectively a beam search. This also provides a termination condition when no further expansion is likely to produce a better solution.

Another powerful constraint is the consistency of character script within a phrase. As we include shape classes from multiple scripts, confusion errors between characters across different scripts become inevitable. Although we can establish the dominant script or language for the page, we must allow for Latin characters as well, since the occurrence of English words inside foreign language books is so common. Under the assumption that characters within a recognition unit would have the same script, we would promote a character interpretation if it improves the overall script consistency of the whole unit. However, blindly promoting script characters based on prior could actually hurt the performance if the word or phrase is truly mixed script. So we apply the constraint only if over half the characters in the top interpretation belong to the same script, and the adjustment is weighted against the shape recognition score, like any other permutation.

4.2 Shape Classification

Classifiers for large numbers of classes are still a research problem; even today, especially when they are required to operate at the speeds needed for OCR [13, 14]. The curse of dimensionality is largely to blame. The Tesseract shape classifier works surprisingly well on 5000 Chinese characters without requiring any major modifications, so it seems to be well suited to large class-size problems. This result deserves some explanation, so in this section we describe the Tesseract shape classifier.

The features are components of a polygonal approximation of the outline of a shape. In training, a 4-dimensional feature vector of (x, y-position, direction, length) is derived from each element of the polygonal approximation, and clustered to form prototypical feature vectors. (Hence the name: Tesseract.) In recognition, the elements of the polygon are broken into shorter pieces of equal length, so that the length dimension is eliminated from the feature vector. Multiple short features are matched against each prototypical feature from training, which makes the classification process more robust against broken characters.

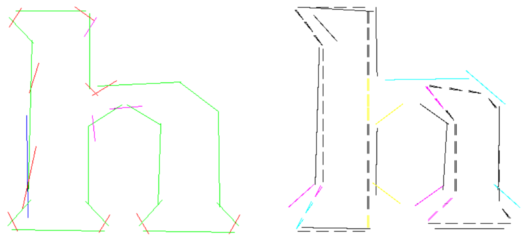


Figure 7. (a) Prototype of h for Times Roman, (b) Match of a broken h against prototytype.

Fig.7(a) shows an example prototype of the letter 'h' for the font Times Roman. The green line-segments represent cluster means of *significant clusters* that contain samples from almost every sample of 'h' in Times Roman. Blue segments are cluster means that were merged with another cluster to form a significant cluster. Magenta segments were not used, as they matched an existing significant cluster. Red segments did not contain enough samples to be significant, and could not be merged with any neighboring cluster to form a significant cluster.

Fig.7(b) shows how the shorter features of the unknown match against the prototype to achieve insensitivity to broken characters. The short, thick lines are the features of the unknown, being a broken 'h' and the longer lines are the prototype features. Colors represent match quality: black -> good, magenta -> reasonable, cyan -> poor, and yellow -> no match. The vertical prototypes are all well matched, despite the fact that the h is broken.

The shape classifier operates in two stages. The first stage, called the *class pruner,* reduces the character set to a short-list of 1-10 characters, using a method closely related to Locality Sensitive Hashing (LSH) [13]. The final stage computes the distance of the unknown from the prototypes of the characters in the short-list.

Originally designed as a simple and vital time-saving optimization, the class pruner partitions the high-dimensional feature space, by considering each 3-D feature individually. In place of the hash table of LSH, there is a simple look-up table, which returns a vector of integers in the range [0, 3], one for each class in the character set, with the value representing the approximate goodness of match of that feature to a prototype of the character class. The vector results are summed across all features of the unknown, and the classes that have a total score within a fraction of the highest are returned as the shortlist to be classified by the second stage. The class pruner is relatively fast, but its time scales linearly with the number of classes and also with the number of features.

The second stage classifier calculates the distance d_f of each feature from its nearest prototype, as the squared Euclidean distance d of the (x,y) feature coordinates from the prototype line in 2-D space, plus a weighted (w) difference of the angle θ from the prototype:

$$d_f = d^2 + w\theta^2$$

This is essentially a generative classifier, in the sense that it calculates the distance from an ideal. The feature distance is converted to feature *evidence* E_f using the following equation:

$$E_f = \frac{1}{1 + kd_f^2}$$

The constant k is used to control the rate at which the evidence decays with distance. As features are matched to prototypes, the feature evidence E_f, is copied to the prototypes E_p. Since the prototypes expect multiple features to be matched to them, and the collection of "best match" is done independently for speed, the sums of feature and prototype evidence can be different. The sums are normalized by the number of features and sum of prototype lengths L_p, and the result is converted back into a distance:

$$d_{final} = 1 - \frac{\sum_f E_f + \sum_p E_p}{N_f + \sum_p L_p}$$

Note that the actual implementation uses fixed-point integer arithmetic and a lot of the scaling constants that would otherwise obscure the calculations are omitted from the equations above.

Part of the strength of the second-stage classifier is in allowing multiple ideals (known as *configs* in Tesseract) within each class label, thus allowing multi-modal distributions that may be caused by arbitrary differences in font or typography. The matching process described above selects the best config when calculating the final distance. In this sense, the classifier is thus effectively a nearest neighbor classifier.

We hypothesize that the class pruner and the secondary classifier work well for large numbers of classes because of their use of voting among multiple "weak classifiers" of small dimension, rather than relying on a single classifier of high dimension. This is the very concept behind boosting [15], except that currently the weal classifiers are not weighted. The dimensions of feature space are quantized to 256 levels, which provide enough precision to store the complex shapes of CJK characters and Indic syllables, and the d_{final} calculation avoids the curse of dimensionality in a similar fashion to the class pruner.

5. Contextual Post-Processing

Tesseract's training process supports partially extending the language model by providing a way to generate dictionaries for new languages from an arbitrary word list. For compactness and fast search, these dictionaries are represented by directed acyclic word graphs (DAWGs). In the original implementation the DAWG data structure was used to sequentially search several dictionaries including the pre-generated system dictionary, the document dictionary (dynamically constructed from the words in the OCRed document) and a user-provided word list.

Originally each edge in the DAWG stored an 8-bit char to represent the letter used for the corresponding transition in the DAWG. This representation, however, was limiting, since manipulating multi-character graphemes and multi-byte Unicode characters in this manner was awkward. The DAWG data structure was modified to store the *unicharset IDs* used by the character classifier instead. This significantly simplified the process of constructing and searching the DAWGs. Another improvement was parallelizing the search over all the DAWGs. To find out whether a given string is a valid dictionary word, the search now starts out with an initial set of "active" DAWGs. As each letter in the word is considered, this set is reduced to only contain those that still "accepted" the partial string. At the end of the process the set of "active" DAWGs consist of only those DAWGs that contain the word. This restructuring allows us to dynamically load an arbitrary number of DAWGs without having to add any custom support for searching each of the newly added DAWGs. It was also one of the necessary modifications to allow Tesseract to support an arbitrary combination of languages - a feature needed for Tesseract to work on multi-language text.

5.1 Constraint Patterns

The punctuation and number state machines in Tesseract were hard-coded and did not generalize beyond the Latin scripts. Even for the Latin scripts, a significant portion of valid punctuation and number patterns were not accepted by the state machines. To help Tesseract handle punctuation and numbers in non-Latin scripts Tesseract's training process was extended with code to collect and encode a set of frequently occurring punctuation and number patterns. The step for collecting these patterns was implemented to be done in parallel with the processing of a large text corpus to construct the dictionaries for a given language. To represent and match the generated patterns, the already existing code for generating and searching word DAWGs was employed. A few modifications to the algorithm that determines whether a given classifier choice is a valid word in the language enabled Tesseract to do a simultaneous search over all the DAWGs containing words, punctuation and number patterns. With this modification it was possible to remove all the language- and script-specific hard-coded rules for numbers and punctuation. The process of generating and searching the punctuation and number patterns was designed to be completely data-driven and so far requires no special casing for any language in particular.

5.2 Resolving Shape Ambiguities

Alongside the pre-trained shape templates, Tesseract's shape classifier includes an adaptive component that learns the patterns of the characters seen in the OCRed document. In order to ensure that the adaptive component is trained on reliable data, the classifier only adapts to the unambiguous dictionary words. The OCRed word is dubbed "unambiguous" if it satisfies two constraints. The first one is that the shape classifier must identify a clear winner among all the alternative choices for the word (i.e. the classifier rating for the top best choice must be significantly higher than the rating of the next best choice). The second constraint is that no dictionary word can exist that is ambiguous in shape to the best choice for the word. This requirement is also important for recognition speed, since (depending on the classifier score) once Tesseract finds such a word choice, it could accept the recognition result and stop further processing of the word.

For Latin scripts Tesseract contained a hand-crafted data file (referred to as "dangerous ambiguities" file) specifying which letter combinations are inherently ambiguous in the majority of the Latin fonts. A scalable solution to enable this functionality for languages using other scripts was to develop an automated way of generating a list of ambiguous n-gram pairs for any given language. A set of n-grams (in this case uni-, bi-grams) whose combined weight accounts for 95% of all n-grams in the language was collected from a large text corpus. The n-grams were rendered with a set of commonly used fonts in a few degradation modes and exposures. Then Tesseract's shape classifier was run on the rendered images to obtain a set of top scoring classifications for each of the n-grams. The resulting classification scores statistics was aggregated for each of the n-grams and the outliers with low classifier scores were discarded (in some fonts and degradation modes the characters were rendered beyond recognizable, and such cases would only pollute the data). Then for each of the incorrectly OCRed n-grams and the corresponding correct n-gram pair an ambiguity score was computed. The ambiguity score was defined as a function of the shape classifier-perceived similarity between the wrong and correct n-grams (aggregated across all fonts and degradation modes) and the frequency of the correct n-gram in the language. In order to achieve the desired balance between Tesseract's speed and accuracy, it was necessary to pick a threshold of the expected number of errors allowed to occur due to the n-gram shape ambiguities (computed from the n-gram frequency and classifier error statistics). To generate the "dangerous ambiguities" file the ambiguous n-gram pairs were sorted in the non-increasing order of their ambiguity scores and the appropriate number of top-scoring ambiguities that ensured the desired expected error rate were included in the file.

With the data files generated by this automated approach it was possible to achieve similar improvements on Latin scripts (EFIGS data set) as compared to using the hand-crafted "dangerous ambiguities" files (although in some languages the results were slightly weaker). Using the automatically generated file on the Russian data set resulted in a 10% reduction in word error rate. Examining the files generated for other languages also showed that the automatically generated files contained a fair number of commonly confused shapes, but further tests on the corresponding data sets will be needed to quantify the improvement.

5.3 Handling Highly Inflected Languages

Tesseract's speed and accuracy are tied to the quality of the dictionary, and it is always a challenge to maximize these, while minimizing the space consumed to store the dictionary. Generating the dictionary from a corpus in a highly inflected language is a particularly difficult task. The frequency of words in highly inflected languages is more evenly distributed, and thus to achieve the same language coverage, a larger dictionary is needed. Moreover, many of the word forms of even the more frequent words might not occur enough times in the training corpus to be included in the dictionary, and thus the dictionary might not generalize well beyond the training corpus. Fig.8 illustrates this problem on a collection of languages by graphing the coverage of the corpus against the number of most frequent words chosen to form the dictionary.

Because of the head and tail compaction of the DAWG data structure, adding an inflected form of a word that already exists in the DAWGs might result in a very small increase in the overall size of the dictionary. This is because the beginning and the ending of the word might already be stored in the DAWG (for example it would be relatively cheap to add the word "talking" to the dictionary if "talk" and "making" have already been inserted).

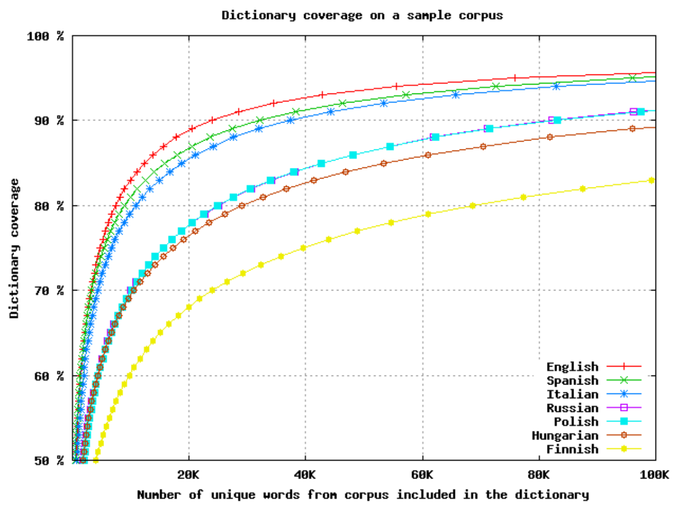


Figure 8. Corpus coverage of varying-size dictionaries in a collection of languages.

To combat the problem of capturing more of the inflected word forms, the dictionary generation process was amended with a step to generate word variants (that were not present in the word list) and add them to the dictionary. First, as previously done, the DAWG is constructed from a word list. Then for each word root in the DAWG a set of suffixes is collected. The sets are clustered using a group-average hierarchical agglomerative clustering algorithm. The suffix sets in the resulting clusters are merged to form expanded suffix sets. Then for each word root and the corresponding suffix set (pre-computed during the first traversal of the DAWG) the closest expanded suffix set is identified. The new words formed by the suffixes from the expanded set are inserted into the DAWG.

6. Status / Experimental Results

Our data set consists of pages from randomly selected books collected by the Google Book Search project. For each language, 100 random books were selected, and 10 pages were randomly selected from each book for manual ground-truthing. Therefore, these pages cover a large variety in every aspect from layout, typeface, image quality, to subject and term usage.

The dataset is then broken into training, validation and test sets, where the training and validation sets are used for learning and benchmarking the algorithms during development, and the test set is reserved for final evaluation during release. Table 1 summarizes the size of the data set and current accuracy for a few languages. For alphabetical languages, we report the error rates at both the character and word level. For Chinese where the meaning of word is ambiguous, we report only the character substitution rate. EFIGSD is a combination of English, French, Italian, German, Spanish and Dutch.

For Simplified Chinese, we noticed there is a large deviation of error rates across different books. The difference can be mainly attributed to variation in fonts and quality. Where the page quality and accuracy are reasonable, the errors are mainly due to confusions between similar or near-identical shape classes. We have plans to increase the capacity of the feature space in the shape matcher, which should help distinguish between similar shapes. On the near-identical cases, the within-class variation across fonts is probably larger than the between-class variation. Fortunately, their usage and priors are so different that they could easily be corrected when we introduce a language model for CJK.

Table 1. Error rates over various languages

Language	No. of chars (millions)	No. of words (millions)	Char error rate (%)	Word error rate (%)
English	39	4	0.5	3.72
EFIGSD	213	26	0.75	5.78
Russian	38	5	1.35	5.48
Simplified Chinese	0.25	NA	3.77	NA
Hindi	1.4	0.33	15.41	69.44

7. Conclusions & Future Work

We have described our experiments with adapting Tesseract to operate on a diverse set of languages, and found that it was surprisingly mostly a matter of engineering. Without any significant changes to the classifier, we were able to obtain good results for a variety of Latin-based languages, Russian, and even Simplified Chinese. The results for Hindi have so far been disappointing, but we have discovered that our test set contains a mix of new and old typography, and a significant proportion of errors are due to the fact that the training set does not contain characters from the old typography. This work does not yet cover languages that are written from right to left, which is mainly another engineering issue, but Arabic has its own set of problems that Tesseract may not be able to address – namely character segmentation. Another language that we have not discussed is Thai, which poses problems of highly ambiguous characters, and like Chinese, does not have spaces between words.

An important future project is to improve the training process to be able to use real data for training instead of just synthetic data with character bounding boxes. This will greatly help accuracy on Hindi. We also need to test Arabic and Thai, where we anticipate more problems. For Chinese, Japanese, and Thai, we need to allow the language model to search the space of arbitrarily concatenated words, since there is no whitespace between the words of these languages. The same capability would also be useful for German, although German compounding has the additional complexity of case changes and inserted letters.

8. References

[1] Nagy, G., “Chinese character recognition: a twenty-five-year perspective” *9th Int. Conf. on Pattern Recognition*, Nov 1988, pp163-167.

[2] Xia, F. “Knowledge-based sub-pattern segmentation: decompositions of Chinese characters” *Image Processing 1994*. Proc. ICIP-94, IEEE Int. Conf. vol.1, 13-16 Nov 1994, pp179-182.

[3] Zhidong Lu, Schwartz, R. Natarajan, P. Bazzi, I. Makhoul, J. “Advances in the BBN BYBLOS OCR system” *Proc. 5th Int. Conf. on Document Analysis and Recognition*, 1999, pp337-340.

[4] Kanungo, T., Marton, G.A., Bulbul, O., “Omnipage vs. Sakhr: paired Model Evaluation of Two Arabic OCR Products” *Proc. SPIE* **3651,** 7 Jan 1999, pp109-120.

[5] Bansal, V.; Sinha, R.M.K**,** “A complete OCR for printed Hindi text in Devanagari script” *Proc. 6th Int. Conf on Document Analysis and Recognition*, 2001, pp800-804.

[6] Govindaraju, V., et. al. “Tools for enabling digital access to multi-lingual Indic documents” *Proc 1st Int. Workshop on document Image Analysis for Libraries*, 2004, pp122-133.

[7] Official Google Blog: http://googleblog.blogspot.com/2008/07/hitting-40-languages.html.

[8] Smith, R., “An Overview of the Tesseract OCR Engine” *Proc 9th Int. Conf. on Document Analysis and Recognition,* 2007, pp629-633.

[9] Tesseract Open-Source OCR: http://code.google.com/p/tesseract-ocr.

[10] Smith, R “Hybrid Page Layout Analysis via Tab-Stop Detection, Document Analysis and Recognition” *Proc. 10th Int. Conf. on Document Analysis and Recognition,* 2009.

[11] Smith, R., “A simple and efficient skew detection algorithm via text row accumulation” *Proc. 3rd Int. Conf. on Document Analysis and Recognition*, 1995, pp1145-1148.

[12] Unnikrishnan, R., Smith, R., “Combined Script and Page Orientation Estimation using the Tesseract OCR engine” Submitted to International Workshop of Multilingual OCR, 25th July 2009, Barcelona, Spain.

[13] Gionis, A., Indyk, P., Motwani, R., “Similarity Search in High Dimensions via Hashing” *Proc. 25th Int. Conf. on Very Large Data Bases,* 1999, pp518-529.

[14] Baluja, S., Covell, M., “Learning to hash: forgiving hash functions and applications” *Data Mining and Knowledge Discovery* **17**(3), Dec 2008, pp402-430.

[15] Schapire, R.E., “The Strength of Weak Learnability” *Machine Learning,* **5**, 1990, pp 197-227.