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

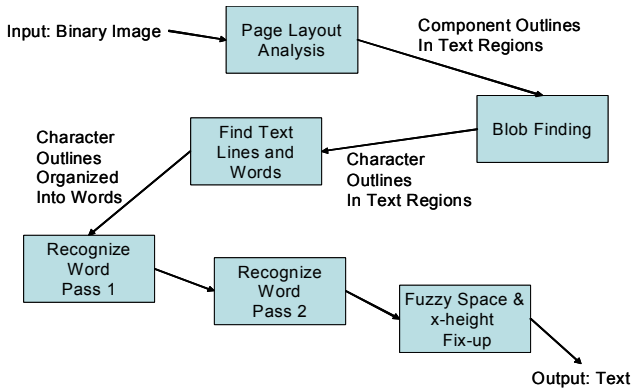


Figure 1. Top-level block diagram of Tesseract.

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.

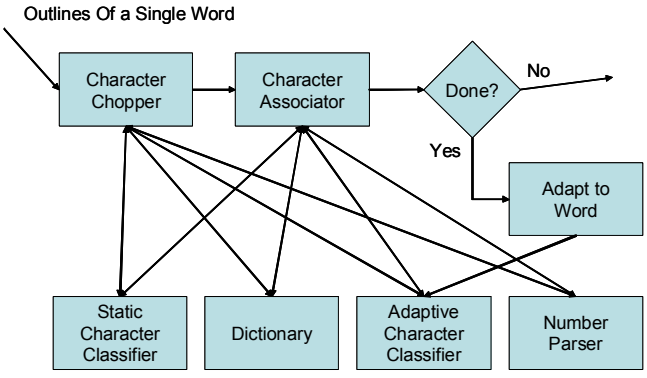


Figure 2. Block diagram of Tesseract word recognition.

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.

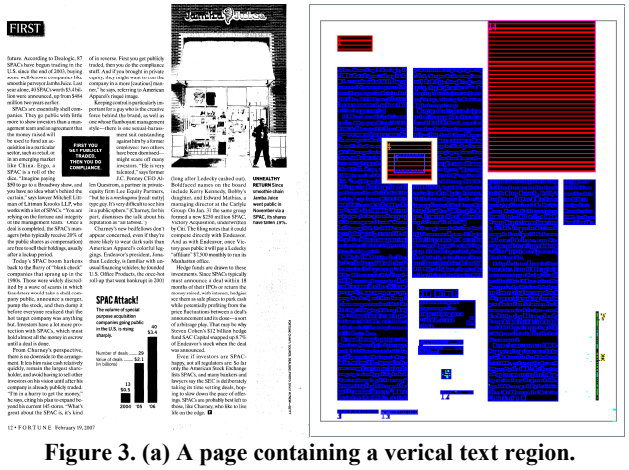


Figure 3. (a) A page containing a verical text region. (b) The detected regions with image in red, horizontal text in blue, and vertical text in yellow.

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.

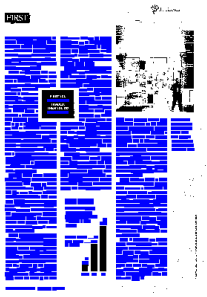


Figure 4. The vertical text is differentially rotated so it is oriented horizontally.

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.

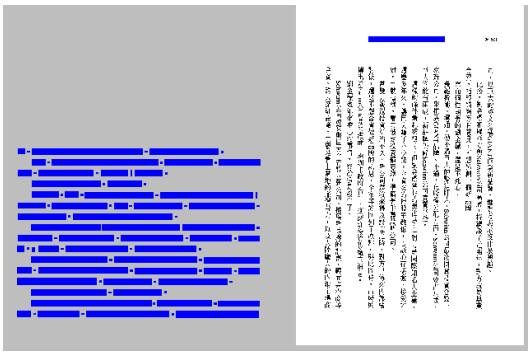


Figure 5. Horizontal text detection for Traditional Chinese. Since the majority of the text is vertical, inside Tesseract it is rotated anticlockwise 90 degrees so it lies outside the image, but the lines are horizontal. The page header, which was already horizontal, remains behind.

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

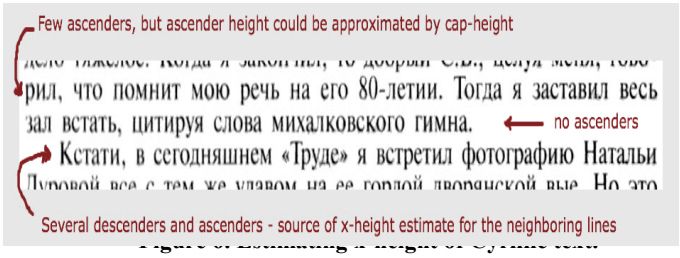
After completing the text line finding step and organizing blocks of blobs into rows, Tesseract estimates x-height for each text line. The x-height estimation algorithm first determines the bounds on the maximum and minimum acceptable x-height based on the initial line size computed for the block. Then, 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 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.

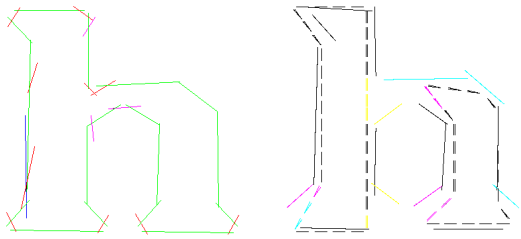


Figure 7. (a) Prototype of h for Times Roman, (b) Match of a broken h against prototytype.

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).

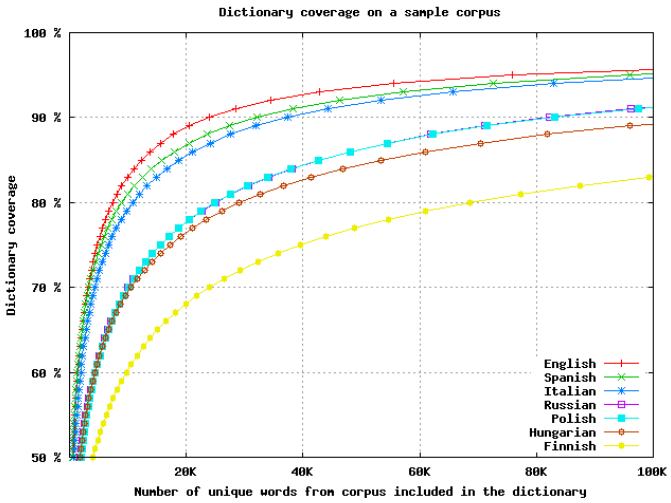


Figure 8. Corpus coverage of varying-size dictionaries in a collection of languages.

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.