My favorites ▼ | Sign in

tesseract-ocr

An OCR Engine that was developed at HP Labs between 1985 and 1995... and now at Google.

Search projects

Project Home Downloads **Wiki** Issues Source Export to GitHub

READ-ONLY: This project has been archived. For more information see this post.

Search Current pages for Search

TrainingTesseract3

How to use the tools provided to train Tesseract3 for a new language.

Featured

Updated Oct 7, 2014 by theraysm...@gmail.com

Introduction

Tesseract 3.0x is fully trainable. This page describes the training process, provides some guidelines on applicability to various languages, and what to expect from the results.

For training Tesseract 2.0x see TrainingTesseract.

Background and Limitations

Tesseract was originally designed to recognize English text only. Efforts have been made to modify the engine and its training system to make them able to deal with other languages and UTF-8 characters. Tesseract 3.0 can handle any Unicode characters (coded with UTF-8), but there are limits as to the range of languages that it will be successful with, so please take this section into account before building up your hopes that it will work well on your particular language!

Tesseract 3.01 added top-to-bottom languages, and Tesseract 3.02 added Hebrew (right-to-left). Tesseract currently handles scripts like Arabic with an auxiliary engine called cube (included in Tesseract 3.0+)

Tesseract is slower with large character set languages (like Chinese), but it seems to work OK.

Tesseract needs to know about different shapes of the same character by having different fonts separated explicitly. This used to be limited to 32 fonts, but the limit has been raised to 64. It is set by the constant MAX_NUM_CONFIGS defined in intproto.h. Note that runtime is heavily dependent on the number of fonts provided, and training more than 32 will result in a significant slow-down.

Any language that has different punctuation and numbers is going to be disadvantaged by some of the hard-coded algorithms that assume ASCII punctuation and digits. To be fixed in 3.0x for x>=2.

You need to run all commands in the same folder where are located your input files.

Additional Libraries required

Beginning with 3.03, additional libraries are required to build the training tools.

```
sudo apt-get install libicu-dev
sudo apt-get install libpango1.0-dev
sudo apt-get install libcairo2-dev
```

Building the training tools

Beginning with 3.03, if you're compiling Tesseract from source you need to make and install the training tools with separate make commands. Once the above additional libraries have been installed, run the following from the tesseract source directory:

```
make training
sudo make training-install
```

Data files required

To train for another language, you have to create some data files in the `tessdata` subdirectory, and then crunch these together into a single file, using combine_tessdata. The naming convention is `languagecode.file_name` Language codes for released files follow the ISO 639-3 standard, but any string can be used. The files used for English (3.00) are:

- `tessdata/eng.config`
- `tessdata/eng.unicharset`
- `tessdata/eng.unicharambigs`
- `tessdata/eng.inttemp`
- `tessdata/eng.pffmtable`
- `tessdata/eng.normproto`
- `tessdata/eng.punc-dawg`
- `tessdata/eng.word-dawg`
- `tessdata/eng.number-dawg`
- `tessdata/eng.freq-dawg`

... and the final crunched file is:

- `tessdata/eng.traineddata`

and

- `tessdata/eng.user-words`

may still be provided separately.

The traineddata file is simply a concatenation of the input files, with a table of contents that contains the offsets of the known file types. See ccutil/tessdatamanager.h in the source code for a list of the currently accepted filenames. **NOTE** the files in the traineddata file are different from the list used prior to 3.00, and will most likely change, possibly dramatically in future revisions.

Requirements for text input files

Text input files (lang.config, lang.unicharambigs, font_properties, box files, wordlists for dictionaries...) need to meet these criteria:

- ASCII or UTF-8 encoding without BOM
- Unix end-of-line marker ('\n')
- The last character must be an end of line marker ('\n'). Some text editors will show this as an empty line at the end of file. If you omit this you will get an error message containing "last_char == '\n':Error:Assert failed..."

How little can you get away with?

You **must** create unicharset, inttemp, normproto, pfftable using the procedure described below. If you are only trying to recognize a limited range of fonts (like a single font for instance), then a single training page might be enough. The other files no longer need to be provided, but will most likely improve accuracy, depending on your application. The old DangAmbigs has been replaced by unicharambigs.

Training Procedure

Some of the procedure is inevitably manual. As much automated help as possible is provided. More automated tools may appear in the future, but will require a complex install/build process. The tools referenced below are all built in the training subdirectory.

Generate Training Images

The first step is to determine the full character set to be used, and prepare a text or word processor file containing a set of examples. The most important points to bear in mind when creating a training file are:

- Make sure there are a minimum number of samples of each character. 10 is good, but 5 is OK for rare characters.
- There should be more samples of the more frequent characters - at least 20.
- Don't make the mistake of grouping all the non-letters together. Make the text more realistic. For example, **The quick brown fox jumps over the lazy dog. 0123456789 !@#$%^&(),.{}<>/?** is terrible. Much better is **The (quick) brown {fox} jumps! over the $3,456.78 <`lazy`> #90 dog & duck/goose, as 12.5% of E-mail from aspammer@website.com is spam?** This gives the textline finding code a much better chance of getting sensible baseline metrics for the special characters.

NEW Automated method

Prepare a utf-8 text file (`training_text.txt`) containing your training text according to the above specification. Obtain truetype/opentype font files for the fonts that you wish to recognize. Run the following command for each font in turn to create a matching tif/box file pair.

```
training/text2image --text=training_text.txt --outputbase=[lang].[fontname].exp0 --font='Font Name' --fonts_dir=/path/to/
```

Note that the argument to --font may contain spaces, and thus must be quoted. Eg:

```
training/text2image --text=training_text.txt --outputbase=eng.TimesNewRomanBold.exp0 --font='Times New Roman Bold' --font
```

There are a lot of other command-line arguments available to text2image. See training/text2image.cpp for more information.

If you can use text2image for your application, great! Now skip to Run Tesseract For Training below.

Old Manual method

- It is sometimes important to space out the text a bit when printing, so up the inter-character and inter-line spacing in your word processor. Not spacing text out sufficiently will cause "FAILURE! box overlaps no blobs or blobs in multiple rows" errors during tr file generation, which leads to FATALITY - 0 labelled samples of "x", which leads to "Error: X classes in inttemp while unicharset contains Y unichars" and you can't use your nice new data files. This situation will improve in the future, as we are working on a solution, but for 3.00 APPLY_BOXES errors remain the most problematic difficulty for people training tesseract.
- The training data should be grouped by font. Ideally, all samples of a single font should go in a single tiff file, but this may be multi-page tiff (if you have libtiff or leptonica installed), so the total training data in a single font may be many pages and many 10s of thousands of characters, allowing training for large-character-set languages.
- There is no need to train with multiple sizes. 10 point will do. (An exception to this is very small text. If you want to recognize text with an x-height smaller than about 15 pixels, you should either train it specifically or scale your images before trying to recognize them.)
- **DO NOT MIX FONTS IN AN IMAGE FILE** (In a single .tr file to be precise.) This will cause features to be dropped at clustering, which leads to recognition errors.
- The example boxtiff files on the downloads page will help if you are not sure how to format your training data.

Next print and scan (or use some electronic rendering method) to create an image of your training page. Up to 64 training files can be used (of multiple pages). It is best to create a mix of fonts and styles (but in separate files), including italic and bold.

NOTE: training from real images is actually quite hard, due to the spacing-out requirements. This will be improved in a future release. For now it is much easier if you can print/scan your own training text.

You will also need to save your training text as a UTF-8 text file for use in the next step where you have to insert the codes into another file.

Clarification for large amounts of training data The 64 images limit is for the number of **FONTS.** Each font should be put in a single multi-page tiff and the box file can be modified to specify the page number for each character after the coordinates. Thus an arbitrarily large amount of training data may be created for any given font, allowing training for large character-set languages. An alternative to multi-page tiffs is to create many single-page tiffs for a single font, and then you must cat together the tr files for each font into several single-font tr files. In any case, the input tr files to mftraining must each contain a single font.

Make Box Files

For the next step below, Tesseract needs a 'box' file to go with each training image. The box file is a text file that lists the characters in the training image, in order, one per line, with the coordinates of the bounding box around the image. Tesseract 3.0 has a mode in which it will output a text file of the required format, but if the character set is different to its current training, it will naturally have the text incorrect. So the key process here is to manually edit the file to put the correct characters in it.

Run Tesseract on each of your training images using this command line:

```
tesseract [lang].[fontname].exp[num].tif [lang].[fontname].exp[num] batch.nochop makebox
```

e.g.

```
tesseract eng.timesitalic.exp0.tif eng.timesitalic.exp0 batch.nochop makebox
```

Now the hard part. You have to edit the file `[lang].[fontname].exp[num].box` and put the UTF-8 codes for each character in the file at the start of each line, in place of the incorrect character put there by Tesseract. Example: The distribution includes an image eurotext.tif. Running the above command produces a text file that includes the following lines (lines 141-154):

```
s 734 494 751 519 0
p 753 486 776 518 0
r 779 494 796 518 0
i 799 494 810 527 0
n 814 494 837 518 0
g 839 485 862 518 0
t 865 492 878 521 0
u 101 453 122 484 0
b 126 453 146 486 0
e 149 452 168 477 0
r 172 453 187 476 0
d 211 451 232 484 0
e 236 451 255 475 0
n 259 452 281 475 0
```

Since Tesseract was run in English mode, it does not correctly recognize the umlaut. This character needs to be corrected using a suitable editor. An editor that understands UTF-8 should be used for this purpose. HTML editors are usually a good choice. (Mozilla on linux allows you to edit utf8 text files directly from the browser. Firefox and IE do not let you do this. MS Word is very good at handling different text encodings, and Notepad++ is another editor that understands UTF-8.) Linux and Windows both have a character map that can be used for copying characters that cannot be typed. In this case the u needs to be changed to ü.

In theory, each line in the box file should represent one of the characters from your training file, but if you have a horizontally broken character, such as the lower double quote „ it will probably have 2 boxes that need to be merged!

Example: lines 116-129:

```
D 101 504 131 535 0
e 135 502 154 528 0
r 158 503 173 526 0
, 197 498 206 510 0
, 206 497 214 509 0
s 220 501 236 526 0
c 239 501 258 525 0
```

```
h 262 502 284 534 0
n 288 501 310 525 0
e 313 500 332 524 0
l 336 501 347 534 0
l 352 500 363 532 0
e 367 499 386 524 0
” 389 520 407 532 0
```

As you can see, the low double quote character has been expressed as two single commas. The bounding boxes must be merged as follows:

- First number (left) take the minimum of the two lines (197)
- Second number (bottom) take the minimum of the two lines (497)
- Third number (right) take the maximum of the two lines (214)
- Fourth number (top) take the maximum of the two lines (510)

This gives:

```
D 101 504 131 535 0
e 135 502 154 528 0
r 158 503 173 526 0
„ 197 497 214 510 0
s 220 501 236 526 0
c 239 501 258 525 0
h 262 502 284 534 0
n 288 501 310 525 0
e 313 500 332 524 0
l 336 501 347 534 0
l 352 500 363 532 0
e 367 499 386 524 0
” 389 520 407 532 0
```

If you didn't successfully space out the characters on the training image, some may have been joined into a single box. In this case, you can either remake the images with better spacing and start again, or if the pair is common, put both characters at the start of the line, leaving the bounding box to represent them both. (As of 3.00, there is a limit of 24 bytes for the description of a "character". This will allow you between 6 and 24 unicodes to describe the character, depending on where your codes sit in the unicode set. If anyone hits this limit, please file an issue describing your situation.)

Note that the coordinate system used in the box file has (0,0) at the **bottom-left.**

The last number on each line is the page number (0-based) of that character in the multi-page tiff file.

There are several visual tools for editing box file - please check AddOns wiki.

Bootstrapping a new character set

If you are trying to train a new character set, it is a good idea to put in the effort on a single font to get one good box file, run the rest of the training process, and then use Tesseract in your new language to make the rest of the box files as follows:

```
tesseract [lang].[fontname].exp[num].tif [lang].[fontname].exp[num] -l yournewlanguage batch.nochop makebox
```

This should make the 2nd box file easier to make, as there is a good chance that Tesseract will recognize most of the text correctly. You can always iterate this sequence adding more fonts to he training set (i.e. to the command line of mfTraining and cnTraining below) as you make them, but note that there is no incremental training mode that allows you to add new training data to existing sets. This means that each time you run mfTraining and cnTraining you are making new data files from scratch from the tr files you give on the command line, and these programs cannot take an existing intproto/pffmtable/normproto and add to them directly.

Tif/Box pairs provided!

Some Tif/Box file pairs are on the downloads page. (Note the tiff files are G4 compressed to save space, so you will have to have libtiff or uncompress them first). You could follow the following process to make better training data for your own language or subset of an existing language, or add different characters/shapes to an existing language:

1. Filter the box files, keeping lines for only the characters you want.
2. Run tesseract for training (below).
3. Cat the .tr files from multiple languages for each font to get the character set that you want and add the .tr files from your own fonts or characters.
4. Cat the filtered box files in an identical way to the .tr files for handing off to unicharset_extractor.
5. Run the rest of the training process.

Caution! This is not quite as simple as it sounds! cntraining and mftraining can only take up to 64 .tr files, so you must cat all the files from multiple languages for the same font together to make 64 language-combined, but font-individual files. The characters found in the tr files **must** match the sequence of characters found in the box files when given to unicharset_extractor, so you have to cat the box files together in the **same order** as the tr files. The command lines for cn/mftraining and unicharset_extractor must be given the .tr and .box files (respectively) in the **same order** just in case you have different filtering for the different fonts. There may be a program available to do all this and pick out the characters in the style of character map. This might make the whole thing easier.

Run Tesseract for Training

For each of your training image, boxfile pairs, run Tesseract in training mode:

```
tesseract [lang].[fontname].exp[num].tif [lang].[fontname].exp[num] box.train
```

or

```
tesseract [lang].[fontname].exp[num].tif [lang].[fontname].exp[num] box.train.stderr
```

NOTE that although tesseract requires language data to be present for this step, the language data is not used, so English will do, whatever language you are training.

The first form sends all the errors to tesseract.log (on all platforms) like it did on windows versions 2.03 and below. With box.train.stderr, all errors are sent to stderr, on all platforms, just like it did on non-windows platforms for versions 2.03 and below.

Note that the box filename must match the tif filename, including the path, or Tesseract won't find it. The output of this step is `fontfile.tr` which contains the features of each character of the training page. `[lang].[fontname].exp[num].txt` will also be written with a single newline and no text.

Important Check for errors in the output from apply_box. If there are FATALITIES reported, then there is no point continuing with the training process until you fix the box file. The new box.train.stderr config file makes is easier to choose the location of the output. A FATALITY usually indicates that this step failed to find any training samples of one of the characters listed in your box file. Either the coordinates are wrong, or there is something wrong with the image of the character concerned. If there is no workable sample of a character, it can't be recognized, and the generated inttemp file won't match the unicharset file later and Tesseract will abort.

Another error that can occur **that is also fatal and needs attention** is an error about "Box file format error on line n". If preceded by "Bad utf-8 char..." then the utf-8 codes are incorrect and need to be fixed. The error "utf-8 string too long..." indicates that you have exceeded the 24 byte limit on a character description. If you need a description longer than 24 bytes, please file an issue.

There is no need to edit the content of the `[lang].[fontname].exp[num].tr` file. The font name inside it need not be set. For the curious, here is some information on the format:

```
Every character in the box file has a corresponding set of entries in
the .tr file (in order) like this
UnknownFont <utf8 code(s)> 2
mf <number of features>
x y length dir 0 0
... (there are a set of these determined by <number of features>
above)
cn 1
ypos length x2ndmoment y2ndmoment

The mf features are polygon segments of the outline normalized to the
1st and 2nd moments.
x= x position [-0.5.0.5]
y = y position [-0.25, 0.75]
length is the length of the polygon segment [0,1.0]
dir is the direction of the segment [0,1.0]

The cn feature is to correct for the moment normalization to
distinguish position and size (eg c vs C and , vs ')
```

Compute the Character Set

Tesseract needs to know the set of possible characters it can output. To generate the `unicharset` data file, use the `unicharset_extractor` program on the box files generated above:

```
unicharset_extractor lang.fontname.exp0.box lang.fontname.exp1.box ...
```

Tesseract needs to have access to character properties isalpha, isdigit, isupper, islower, ispunctuation. This data must be encoded in the `unicharset` data file. Each line of this file corresponds to one character. The character in UTF-8 is followed by a hexadecimal number representing a binary mask that encodes the properties. Each bit corresponds to a property. If the bit is set to 1, it means that the property is true. The bit ordering is (from least significant bit to most significant bit): isalpha, islower, isupper, isdigit.

Example:

- ';' is an punctuation character. Its properties are thus represented by the binary number 10000 (10 in hexadecimal).
- 'b' is an alphabetic character and a lower case character. Its properties are thus represented by the binary number 00011 (3 in hexadecimal).
- 'W' is an alphabetic character and an upper case character. Its properties are thus represented by the binary number 00101 (5 in hexadecimal).
- '7' is just a digit. Its properties are thus represented by the binary number 01000 (8 in hexadecimal).
- '=' does is not punctuation not digit or alphabetic character. Its properties are thus represented by the binary number 00000 (0 in hexadecimal).

```
; 10 Common 46
b 3 Latin 59
W 5 Latin 40
7 8 Common 66
= 0 Common 93
```

Japanese or Chinese alphabetic character properties are represented by the binary number 00001 (1 in hexadecimal).

If your system supports the wctype functions, these values will be set automatically by unicharset_extractor **and there is no need to edit the unicharset file.** On some older systems (eg Windows 95), the unicharset file must be edited by hand to add these property description codes.

NOTE The unicharset file must be regenerated whenever inttemp, normproto and pffmtable are generated (i.e. they must **all** be recreated when the box file is changed) as they have to be in sync. This is made easier than in previous versions by running unicharset_extractor before mftraining and cntraining, and giving the unicharset to mftraining.

Last two columns represent type of script (Latin, Common, Greek, Cyrillic, Han, null) and id code of character given language.

set_unicharset_properties (new in 3.03)

A new tool and set of data files in 3.03 allow the addition of extra properties in the unicharset, mostly sizes obtained from fonts.

```
training/set_unicharset_properties -U input_unicharset -O output_unicharset --script_dir=training/langdata
```

font_properties (new in 3.01)

A new requirement for training in 3.01 is a `font_properties` file. The purpose of this file is to provide font style information that will appear in the output when the font is recognized. The `font_properties` file is a text file specified by the `-F filename` option to mftraining.

Each line of the font_properties file is formatted as follows:

```
<fontname> <italic> <bold> <fixed> <serif> <fraktur>
```

where `<fontname>` is a string naming the font (no spaces allowed!), and `<italic>, <bold>, <fixed>, <serif> and <fraktur>` are all simple 0 or 1 flags indicating whether the font has the named property.

When running mftraining, each .tr filename must match an entry in the font_properties file, or mftraining will abort. At some point, possibly before the release of 3.01, this matching requirement is likely to shift to the font name in the .tr file itself. The name of the .tr file may be either fontname.tr or `[lang].[fontname].exp[num].tr`.

Example:
font_properties file:

```
timesitalic 1 0 0 1 0
```

```
shapeclustering -F font_properties -U unicharset eng.timesitalic.exp0.tr
mftraining -F font_properties -U unicharset -O eng.unicharset eng.timesitalic.exp0.tr
```

Note that in 3.03, there is a default font_properties file, that covers 3000 fonts (not necessarily accurately) in `training/langdata/font_properties.`

Clustering

When the character features of all the training pages have been extracted, we need to cluster them to create the prototypes. The character shape features can be clustered using the `shapeclustering` (available from 3.02 version), `mftraining` and `cntraining` programs:

```
shapeclustering -F font_properties -U unicharset lang.fontname.exp0.tr lang.fontname.exp1.tr ...
```

shapeclustering creates a master shape table by shape clustering and writes it to a file - shapetable. shapeclustering currently should not be used except for the Indic languages.

NOTE: mftraining will produce a shapetable if you didn't run shapeclustering. You **must** include this shapetable in your traineddata file, whether or not shapeclustering was used.

```
mftraining -F font_properties -U unicharset -O lang.unicharset lang.fontname.exp0.tr lang.fontname.exp1.tr ...
```

The -U file is the unicharset generated by unicharset_extractor above, and lang.unicharset is the output unicharset that will be given to combine_tessdata. Mftraining will output two other data files: `inttemp` (the shape prototypes) and `pffmtable` (the number of expected features for each character). (A third file called `Microfeat` is also written by this program, but it is not used.)

```
cntraining lang.fontname.exp0.tr lang.fontname.exp1.tr ...
```

This will output the `normproto` data file (the character normalization sensitivity prototypes).

Dictionary Data (Optional)

Tesseract uses up to 8 dictionary files for each language. These are all optional, and help Tesseract to decide the likelihood of different possible character combinations.

Seven of the files are coded as a Directed Acyclic Word Graph (DAWG), and the other is a plain UTF-8 text file:

To make the DAWG dictionary files, you first need a wordlist for your language. You may find an appropriate dictionary file to use as the basis for a wordlist from the spellcheckers (e. g. ispell, aspell or hunspell) - be careful about the license. The wordlist is formatted as a UTF-8 text file with one word per line. Split the wordlist into needed sets e.g.: the frequent words, and the rest of the words, and then use `wordlist2dawg` to make the DAWG files:

Name	**Type**	**Description**
word-dawg	dawg	A dawg made from dictionary words from the language.
freq-dawg	dawg	A dawg made from the most frequent words which would have gone into word-dawg.
punc-dawg	dawg	A dawg made from punctuation patterns found around words. The *"word"* part is replaced by a single space.
number-dawg	dawg	A dawg made from tokens which originally contained digits. Each digit is replaced by a space character.
fixed-length-dawgs	dawg	Several dawgs of different fixed lengths —— useful for languages like Chinese.
bigram-dawg	dawg	A dawg of word bigrams where the words are separated by a space and each digit is replaced by a *?*.
unambig-dawg	dawg	TODO: Describe.
user-words	text	A list of extra words to add to the dictionary. Usually left empty to be added by users if they require it; see tesseract(1).

```
wordlist2dawg frequent_words_list lang.freq-dawg lang.unicharset
wordlist2dawg words_list lang.word-dawg lang.unicharset
```

For right-to-left languages (RTL) use option *"-r 1"*. Other options can be found in wordlist2dawg Manual Page

NOTE: If a dictionary file is included in the combined traineddata, it must contain at least one entry. Dictionary files that would otherwise be empty are not required for the combine_tessdata step. Words with unusual spellings should be added to the dictionary files. Unusual spellings can include mixtures of alphabetical characters with punctuation or numeric characters. (E.g. i18n, l10n, google.com, news.bbc.co.uk, io9.com, utf8, ucs2)

If you need example files for dictionary wordlists, uncombine (with combine_tessdata) existing language data file (e.g. eng.traineddata) and then extract wordlist with dawg2wordlist

The last file (unicharambigs)

The final data file that Tesseract uses is called unicharambigs. It describes possible ambiguities between characters or sets of characters, and is manually generated. To understand the file format, look at the following example:

```
v1
2       ' '     1       "       1
```

```
1       m       2       r n     0
3       i i i   1       m       0
```

The first line is a version identifier. The remaining lines are tab separated fields, in the following format: <number of characters for match source>

<tab>

<characters for match source>

<tab>

<number of characters for match target>

<tab>

<characters for match target>

<tab>

<type indicator>

Type indicator could have following values:

Value	Type	Description
0	A non-mandatory substitution. This informs tesseract to consider the ambiguity as a hint to the segmentation search that it should continue working if replacement of 'source' with 'target' creates a dictionary word from a non-dictionary word. Dictionary words that can be turned to another dictionary word via the ambiguity will not be used to train the adaptive classifier.	
1	A mandatory substitution. This informs tesseract to always replace the matched 'source' with the 'target' strings.	

Example line	Explanation
2 ' ' 1 " 1	A double quote (") should be substituted **whenever** 2 consecutive single quotes (') are seen.
1 m 2 r n 0	The characters 'rn' may sometimes be recognized incorrectly as 'm'.
3 i i i 1 m 0	The character 'm' may sometimes be recognized incorrectly as the sequence 'iii'.

Each separate character must be included in the unicharset. That is, all of the characters used must be part of the language that is being trained.

The rules are not bidirectional, so if you want 'rn' to be considered when 'm' is detected and vise versa you need a rule for each.

Version 3.03 and on supports a new, simpler format for the unicharambigs file:

```
v2
```

```
'' " 1
m rn 0
iii m 0
```

In this format, the "error" and "correction" are simple utf-8 strings separated by a space, and, after another space, the same type specifier as v1 (0 for optional and 1 for mandatory substitution). Note the downside of this simpler format is that Tesseract has to encode the utf-8 strings into the components of the unicharset. In complex scripts, this encoding may be ambiguous. In this case, the encoding is chosen such as to use the least utf-8 characters for each component, ie the shortest unicharset components will make up the encoding.

Like most other files used in training, the 'unicharambigs' file must be encoded as UTF8, and must end with a newline character. The unicharambigs format is also described in the unicharambigs(5) man page.

The unicharambigs file may also be non-existent.

Putting it all together

That is all there is to it! All you need to do now is collect together all (shapetable, normproto, inttemp, pffmtable) the files and rename them with a `lang.` prefix, where lang is the 3-letter code for your language taken from http://en.wikipedia.org/wiki/List_of_ISO_639-2_codes and then run combine_tessdata on them as follows:

```
combine_tessdata lang.
```

NOTE: Don't forget dot at the end!

The resulting lang.traineddata goes in your tessdata directory. Tesseract can then recognize text in your language (in theory) with the following:

```
tesseract image.tif output -l lang
```

(Actually, you can use any string you like for the language code, but if you want anybody else to be able to use it easily, ISO 639 is the way to go.)

More options of combine_tessdata can be found on its Manual Page or in comment of its source code.

Comment by shreeshrii, Aug 17, 2014

Look at https://code.google.com/p/tesseract-ocr/source/browse/training/tesstrain.sh?spec=svne249d7bcb2d0ed730dd9fbffe5cd228e18a27f00&r=e249d7bcb2d0ed730dd9fbffe5cd228e18a27f00

for new training procedure.

Comment by gautamr...@strose.edu, Jun 29, 2015 is it possible to list the syntax for converting a pdf file into text file Do i have to convert pdf file into image file first

---

Comment by tfmorris, Jul 24, 2015

This wiki is obsolete. The current link for the information on this page is: https://github.com/tesseract-ocr/tesseract/wiki/TrainingTesseract