Training Tesseract 3.0x for a New Language: A Practical Manual

Isabell Hubert
Department of Linguistics, University of Alberta

April 4, 2016

Abstract

This manual is based on the github page on how to train Tesseract. I have greatly extended it with practical recommendations, added many example calls and detailed explanations, and omitted certain sections that might seem confusing or irrelevant to a beginner Tesseract trainer. Recommendations on how to maximize initial model accuracy while minimizing manual labor are based on the experience gained while training the first OCR models for Northern and Southern Haida (cf. [1]). With this background, the approach laid out here might work well for projects where the print font is unknown, and where the script is Latin-based, but uses an extensive character set. This manual is not meant to be an academic publication, but rather – as the name indicates – a document containing practical tips, encouraging the training of Tesseract for other underresearched languages. As such, it is not geared towards Tesseract experts, but rather towards linguists and other professionals that would like to try training a Tesseract model for the first time. This `pdf` version of the manual was specifically created for submission alongside my first Generals Paper; the best reading experience is provided by reading it on a computer rather than as a print-out, as links are clickable and provide further information. This way, presenting it online is also possible. Alternatively, the LaTeX file it is based on can easily be exported to an `html` file so that the manual can be made available on a server or a website directly.

1 Introduction

This manual will help you train an optical character recognition (OCR) model for the Tesseract OCR engine. Essentially, you will be teaching Tesseract how to recognize text in a language it does not know yet. To achieve this, this manual will take you from a set of `tif` or `png` files with text on them to a Tesseract `.traineddata` file.

A Tesseract `.traineddata` file has all the information that the engine needs to recognize text in this language. Because the file is a one-stop-shop, you can even distribute it to other people should you wish to do so – they can then OCR text in the language you trained the model for without having any extra work.

This manual is based on my work with Tesseract 3.02.02; I will do my best to point out wherever other versions of Tesseract 3.0x differ from 3.02.02, but, as I have not tested any of the other versions, cannot vouch for the correctness of this information. If you are using a version of Tesseract other than 3.02.02, it might hence be advisable to create a "sandbox" directory where you can play around without destroying any important data. This way, you can gauge whether the bash commands I present work as intended for you, or if you might have to make some adjustments for them to work in your version of Tesseract. Generally, I think this is good advice for whenever

you are trying out new things, such as training an OCR model for the first time: Create copies of your data, or even use bogus data in the beginning. Move these data to a separate directory, and try out any and all calls on this copy of your data before you run them on your "real" data. That way, if something goes awry, it'll only have affected your bogus data, or a copy of real data. And even when you decide you can unleash a command on your real data, be sure to always have your data backed up in case something goes wrong.[1]

You will likely be using Tesseract Versions 3.0 or newer if you are planning to train a model rather than just use pre-existing models, as all versions from 3.0 upwards are fully trainable. This feature makes it possible for anyone to train the engine to recognize another language. This manual, based on the github page on how to train Tesseract for a new language, describes the training process, and what to expect from the results. I recommend reading the github page as well as this manual, as the github page is by no means made obsolete by the document you are currently reading. Both documents might contain relevant information for the particular scenario in which you are planning to use Tesseract; the github page in addition contains tips on various contingencies, such as what to do if you come across a certain error message. As my experience stems from training the first OCR models for Northern and Southern Haida (cf. [1]), two highly endangered indigenous languages with rich character sets, I would assume that this manual might be useful in situations where the new language has an extensive character set (with e.g. many diacritics, small capital letters, or superscripts.) While Tesseract can handle any Unicode characters, encoded with UTF-8, there is no guarantee that the OCR process will work flawlessly with your language. However, the steps below should help you achieve at least an adequate recognition accuracy. Also, because this manual is a combination of the github pages and my own experience, it is entirely possible that if you are a seasoned trainer of Tesseract models, you might find that you do things differently. This is ok! There are different ways to go about things. In this manual, I am attempting to provide an easy step-by-step guide to people who have not trained an OCR engine before, but would like to do so.

We will be using a *Source Image* rather than an *Image Generation* approach. What this means will be described in detail below, where the training process is laid out. For now, know that the github training page uses the Image Generation approach – this is one of the key points where this manual will differ from this page. For the Source Image approach, I recommend using just one page of scanned text for model training. If you have the time (and the patience!), it might be recommendable to train two or three different models, all based on a different training page. That way, you are quite likely to arrive at a well-functioning model right at the start.[2] As a final note, I recommend reading the whole manual once in its entirety before getting started on the training process. Many things that might at first seem puzzling will become clearer in a later step, and many calls come with comments that you might miss if you execute every call directly.

1.1 Some Notes on Different Versions of Tesseract 3.0

Be aware that not all versions of Tesseract 3.0, commonly indicated by the term *Tesseract 3.0x*, are created equal: Tesseract 3.01 added support for languages that are written top-to-bottom instead of left-to-right, and Tesseract 3.02 added support for Hebrew, which is written right-to-left. This manual focuses on left-to-right languages, like Haida, so it might not be immediately applicable if you are looking to train an OCR model for a right-to-left or top-to-bottom script. In versions

[1] Also, disclaimer: I cannot be made responsible for any damage to data, person, or property that you incur because you follow along with this manual. I can give you tips and tell you what worked well for me, but as for executing the different steps, you're on your own!

[2] If you are interested in how I came to that conclusion, have a look at [1].

3.00 and 3.01, scripts that did not use the same punctuation characters as English were causing troubles; this has been fixed in version 3.02. Again, this manual is based on work with version 3.02.02, so be sure to check what features are (or are not) supported in the version you are using.

2 Preparation

It is advisable to check the list of languages for which `traineddata` are already available – maybe the language model you need is already available!

I recommend training your model using a Unix-based system (such as a Linux derivation, be it on a workstation or a server, or a Mac), as training Tesseract makes extensive use of the command line. Some familiarity with the Unix command line might be of help, although this manual is specifically not geared towards the Linux expert. It is rather intended for linguists, other professionals, or enthusiastic individuals who find that a language is missing from the Tesseract language data.

I will assume that:

1. you already have Tesseract installed on your machine. If this is not the case, consult the chapter on installation on the Tesseract Wiki for advice on how to get Tesseract to run on your Linux machine, Mac, or even Windows PC. Also note that additional libraries might be required if you use Tesseract 3.03 or newer;
2. you have the source text you would like to OCR available to you in `png` or `tif` format.

3 Image Pre-Processing

In your scan images, it is entirely possible that there are black edges around the sides of the page, or that the lines of the text are not entirely straight. In both of these scenarios, simple image pre-processing might be worthwhile. Tesseract does have an anti-skew mechanism, but I have found that this mechanism works best when you assist it a little. This means essentially that you want to crop away any black edges as they could be interpreted as characters by Tesseract, and that you would want to rotate the image so that the lines are as straight as they can be. In addition to simple skew, book bindings might introduce curvy baselines, which are not easy to get rid of in a simple pre-processing operation. However, I have found that even just eliminating skew can help, so focus on cropping and de-skewing for now, and tackle the de-curving of the lines (by perspective modifications) only if you have lots of time.

Pre-processing is important for both the pages you use for training the models, so that your models will be more accurate, and also for the pages you are planning to OCR with your model later, so that the model has an easier time recognizing text. For this reason, it is best to just go through the whole set and crop and rotate whichever pages might need it. I have used the GIMP for those operations, but your favorite image editor should do the trick. Just make sure you can display grid lines or rulers in this image editor, so you have an easy way of assessing how bad the skew is, and of actually checking whether your rotating operations fixed it. Experiment a little with where you put the center point for the rotation. In this step, you can also get rid of very obvious, large blotches in the images, although this is of less importance.

If your page has exceptionally little text, or exceptionally much, you might want to crop a page, or use two pages for training. I have fared well with using one page of about 1,200 characters for training. If you can, select a page where most (if not all) characters occur more than 50 times each; in the Haida project, statistics revealed that for characters that occurred more than 50 times in training, mean character recognition accuracy turned out to be 97%, whereas characters that occurred less than 50 times in training, mean character recognition accuracy was only 43% [1]. You

probably will not reach 50 occurrences for rare characters, but that's ok. Rare characters mean there won't be as many errors to fix manually anyway!

In the remainder of this manual, we will train one model first, and then – optionally, if you have the time and the energy – one or two more, for which you can use the first one as a basis. This means that, at any given point, we will be working with just one image file, one `.box` file, etc.[3] If you end up using e.g. two pages because each page contains very little text, refer to the github page for how to deal with e.g. multiple `.box` files as input.

Once you have completed image pre-processing, select one (or two, or three, if you decide to train more than one model) page(s) that seem to you to be of good quality, and that have a high number of occurrences for each character (if possible). Choose (a) page(s) that has crisp ink, few blotches, and little to no curvy baselines. Create one new directory for each model you are planning to train, and place one training image each in each directory so that the models are being neatly kept apart.

4 Producing `.box` Files

In this step, the image files we have selected for training will be used as input, and a `.box` file will be produced as output. For training, Tesseract needs one `.box` file each to go with each training image. `.box` files first have to be created, and then "fixed" manually. While the ones that are being created can be quite accurate, there will always be some errors – hence the fixing. `.box` files specify all characters on a page in terms of their location, as delimited by a bounding box and as determined by coordinates, and associate a Unicode character with the shape within this box. In the `.box` file, the characters in the training image are listed in order, one per line. To make our lives easier and to save ourselves some time, we will diverge from the github page instructions here, and we will *not* use English as the default language to create our `.box` files, but specify one particular language. The reasoning is this: English does not have a very big character set. There are no letters with diacritics, no smallcaps letters, and so on. If your language uses diacritics and smallcaps letters, the English model won't be able to recognize any of them. You will then spend a lot of time fixing them by hand. If you choose a language that already has those special characters to generate your `.box` files, you will have to spend less time manually fixing the errors in the `.box` file.

To generate a `.box` file that corresponds to the training image, run this command in the directory where your training image is located:

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

Generally, unless otherwise mentioned, all calls in this manual need to be run in the directory where your image files (and all other files produced by further steps of the process) are located. The parts in rectangular brackets will have to be adapted to your circumstances. A few remarks on this:

[lang] While released `.traineddata` files follow the ISO 639-3 standard with regards to language codes used for naming, [lang] can first and foremost be any string you like. Especially if you are experimenting with different models, it makes sense to differentiate between them. Let's say you are planning to develop an OCR model for the fictional Albertanian language. The official ISO code might be *ale*, so you could use that in your naming: If you are training just one model, `ale` might work well. If you are training more than one model, you could add

[3] This is because the models based on just one page were more accurate than those based on more pages in the Haida project. For details, check out [1].

either incremental numbering to differentiate the models (`ale1`, `ale2`, and `ale3`, for example), or you could add the training page the models are based on (`ale219`, `ale080`, and `ale434`, for example.) Whichever way you choose, make sure your naming makes it easy for you to differentiate between the models you train.

[fontname] If you know what font the text is printed in, insert the font name here. It could be a standard font, like Arial or Times New Roman, or – especially if the text is old – a historic font that is hard to track down. If you cannot find the name of the font, use a unique font name (e.g. the author's name, or the title of the printed source) so that you will remember later where this font name came from.

[num] Insert the page number of the image file here.

[tessLang] This is where we specify the language we use to generate the `.box` file. Ideally, find a language that uses the same (or a very similar) character set as the language you are training an OCR model for. You can choose any language for which a Tesseract model is available. Look up the corresponding code, usually three letters long, on that same page, and put it into your call after `-l`. The original github page does not use this `-l` parameter; if you leave it out, English will be used as the default. Remember: Using the default English is not a good idea if you are trying to OCR a language with an extensive character set.

Note that the name of the image file should follow the naming pattern illustrated in the above and below examples. If your image has a different name when it comes out of the scanning process, make sure to edit the name so it fits the naming pattern of `[lang].[fontname].exp[num].tif` or `[lang].[fontname].exp[num].png`.[4] So if we were to train three OCR models for Albertanian, the first of which is to be based on page 12 of a book by Aldershot McFictitious, written in Albertanian and printed in Arial, our image file should be named like this:

```
ale012.arial.exp012.tif
```

We decide that for example the Swedish language seems to have a similar character set to Albertanian, because we have found that Albertanian also uses the letter å, so we will use this language model to generate the `.box` file. The ISO code listed on the Tesseract languages page is `swe`, so we insert this here. Our call to generate this `.box` file might look like this:

```
tesseract ale012.arial.exp012.tif ale.arial.exp012 -l swe
batch.nochop makebox
```

If we do not know the font, the call might look like this:

```
tesseract ale012.mcfictitious.exp012.tif ale.mcfictitious.exp012
-l swe batch.nochop makebox
```

If you are training more than one model, *do not yet repeat this process for the other image files.* Finish the first model first, which will perform much better on your text than any foreign model. You can then use this first model to generate the `.box` files for the other two pages, the workflow for which is described in Section 13, and you'll save yourself a lot of work![5]

You will notice that we have made use of the scanned images directly to generate `.box` files. This is one of the main points where this manual differs from the github page: The github page uses digitally generated images for training, where we use the source text directly. This approach has worked better in my work with Northern and Southern Haida [1], which is why I am recommending it here.

[4] For the rest of this manual, whenever you see a reference to only `.tif` files, the use of `.png` is also possible, and vice versa; either of the two formats works, so adjust the extension in the function calls to whichever format your image files have.

[5] This part is called Bootstrapping a new character set on the github page.

5 Editing `.box` Files

Now it's time for some manual labor! Yes, I know, I promised in the beginning that we'd be trying to maximize model accuracy while minimizing manual labor. There is, however, always that little bit of manual labor involved, which we'll get to now.

There are two ways you can go about fixing the `.box` file. One is to just open the file in your favorite text editor and fix the file by hand. This is exceptionally onerous, and since what we are looking at are pixel-sized movements indicated by numerical coordinates on a page, this can be very tricky.[6] In our project, I have opted for fixing my `.box` files using jTessBoxEditor. As the name indicates, this is a box editor for Tesseract, and it comes with a graphical user interface. There are other add-ons available that do the same job as jTessBoxEditor, but I have not tested those, and I was very happy with jTess. What you want to do now is start up jTess, switch to "Box Editor" mode up top, and load an image file (yes, an image file; this might seem counterintuitive, but the `.box` file will be opened alongside the image file. This is why giving the image and the `.box` file the same name is really important – otherwise the two cannot be matched!). You should then be presented with a view similar to that in Fig. 1.

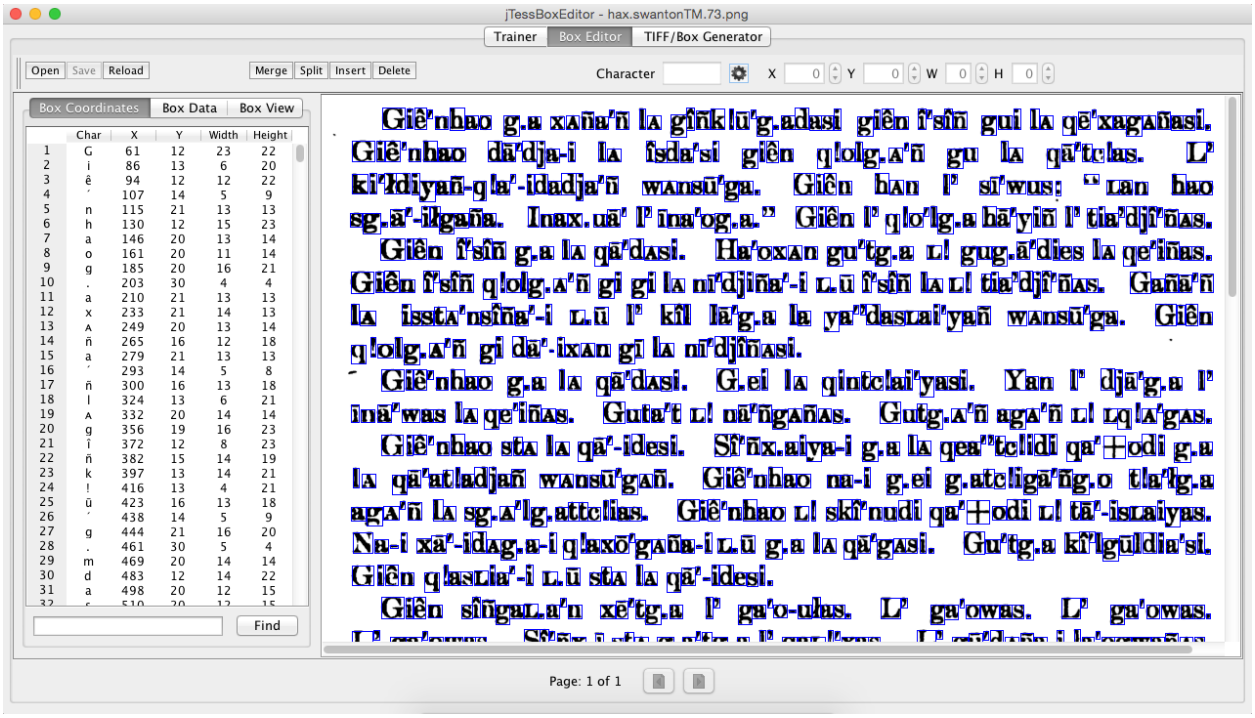

Figure 1: A screenshot of jTessBoxEditor, showing the `.box` file for page 73 in [3].

Now your job is to go through every single box in that `.box` file and make sure it represents the correct character. There are two kinds of problems that can occur: The first kind is that where a box is placed incorrectly. A box could, for example, bound two characters instead of just one; or one character could mistakenly be recognized as two, and thus has two boxes. This can easily be fixed by using the *merge* and *split* operations you can see in the screenshot. Use the *insert* and *delete* buttons to, well, insert or delete a box wherever necessary. Boxes can also be too short, too long, too wide, or too narrow. Just click on a box, and then adjust the values you see above the image to change the box position or size. The great thing is, you'll immediately see the outcome of

[6] If you would like to try manual editing, refer to the github page.

your box-adjusting operations, which you wouldn't get if you used manual editing in a text editor. The second kind of problem regards the actual characters denoted by the boxes. Because you used a "foreign" OCR model to create the `.box` file, Swedish, in our case, there will necessarily be wrong characters in the `.box` file. To change a character to any character that occurs on your keyboard, just click the box, and enter the correct character in the empty field next to "Character." What is vital – and not immediately apparent – is that you must either hit enter, or click the little cogwheel to the right of the character field to save the changes you just made. For any characters that do not naturally occur on your keyboard, which may be a lot if your target language uses an extensive character set, you might have to copy and paste.

I have found the Unicode Character Table to be a prime location to copy and paste characters from. In addition to the full Unicode character table, it has a great search function, and the details page for each character presents you with a wealth of information. Make sure to always use the same Unicode character for the same character on the page; especially various punctuation marks can look alike, bordering the similar, and be completely different Unicode characters. It is very important to be consistent in this step, as otherwise you might have to re-train your model later if you realize you used different Unicode characters for the same character in the image. A shortcut, so that you don't have to look up the same character in the Unicode table over and over, is to copy-paste the characters you need into a separate text document, and then copy them into the `.box` file from there.

For characters that use diacritics, decide whether they should be represented as *decomposed* or *composed* characters. *Decomposed* essentially means that you would enter two characters into the box, the base letter and the diacritic, whereas a *composed* character is one character made up of both the base character and the diacritic. It makes sense to attempt a composed approach for as many characters as possible; for those characters where you cannot find a composed variant in the Unicode Character Table, you can always opt for a decomposed approach. In my experience, hand-correcting a `.box` file of roughly 1,200 characters can take up to an hour; shorter if there are fewer special characters that need copy-pasting. YMMV. Also, and this holds for every file produced in the training process: If any text comes out garbled, with strange symbols or character that you know should not be in there, check your text editor's encoding setting as a first step, and set it to UTF-8 if it isn't yet.

6 Training Tesseract on the `.box` & Image File Pairs

In this step, the image file and the corresponding `.box` file will be used as input, and a `.tr` file (along with an empty text file) will be generated. You will recognize the items in square brackets that we need to adjust (if you can't remember, simply refer back to Section 4):

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

Again, for our first model for the fictitious Albertanian language, the call would look like this:

```
tesseract ale012.arial.exp012.tif ale012.arial.exp012 box.train
```

Unlike for the previous step where we generated the `.box` file, you can safely run this call with the default language parameter. Language data has no influence on the outcome of this call.[7] Make

[7] Apparently you do have to have *some* language data installed in your Tesseract folder, so at least one `traineddata` file has to be present; the language of the `traineddata` file has no influence on the outcome of this call though. Tesseract should come preinstalled with some language models, so only concern yourself with the details of this step if something doesn't work.

sure that the name of the `.box` file matches that of the image file (although, if you've followed along with this manual step by step, this should be the case anyway.) You should now find two new files in your folder:[8] One called `fontfile.tr`, which contains information on the features of each character that Tesseract found on the page, and an (empty) text file with the name of `[lang].[fontname].exp[num].txt`. You do not need to concern yourself with why the text file was created, or why it is empty; you also do not need to edit or concern yourself with the contents of the `fontfile.tr` file.

7 Computing the Character Set

From the `.box` file we created earlier, Tesseract can now compute the full character set that will be used for the new model you are training. As we are using just one `.box` file to train our first model, the call will look like this:

```
unicharset_extractor [lang].[fontname].exp[num].box
```

Or like this, for Albertanian:

```
unicharset_extractor ale012.arial.exp012.box
```

This will generate the `unicharset` file. You again do not need to concern yourself with the contents of this file. In addition to computing the unicharset, you can set additional unicharset properties in Tesseract 3.03.

8 Specifying Font Properties

If your text is printed in a well-known and/or easily identifiable font, you'll have a very easy time with this step. If your text is old, foreign, or otherwise special in terms of typesetting, maybe not so much. In this step, we are specifying the properties of the font that is used in the source text. In Tesseract 3.03, there is a default `font_properties` file that covers 3000 fonts (not necessarily accurately) in the `training/langdata/font_properties` folder; check this file to see if your font is already mentioned in there (and cross-check if its features are correct.) If you do not know the font that is used in your source text, if your font is not listed in this default file, or if you use a Tesseract version older than 3.03, you will have to create the `font_properties` file manually. To do this, create a file names `font_properties` in your favorite text editor. Do not append any extension to this file name, not even `.txt`. Then, we want the contents of the file to specify the following things:

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

We approach this in the way that we again replace the part in square brackets by the name of the font used in the source text. Make sure to use the exact same name you gave the font previously, in Step 4 – make sure that there are no additional spaces and no additional punctuation. The name has to be *exactly* the same. Now we will indicate the properties this font has or doesn't have, using 1 for "feature is present", and 0 for "feature is not present". If you are not familiar with font types, like what "serif" or "fraktur" means, a quick google search can help. Assuming that our Albertanian document uses Arial, the `font_properties` file would look like this:

```
arial 0 0 0 0 0
```

[8] If this call throws errors, consult the github page and/or ask Auntie Google.

Or, if we cannot identify the font that is used in our Albertanian source, if the font is a non-italic, non-bold serif font:

```
mcfictitious 0 0 0 1 0
```

9 Running `mftraining`

In this step, we are combining the `.tr` file with the `unicharset` and the `font_properties` files to generate a number of further files.[9] We will call:

```
mftraining -F font_properties -U unicharset -O [lang].unicharset
[lang].[fontname].exp[num].tr
```

In the Albertanian scenario, assuming Arial as our font:

```
mftraining -F font_properties -U unicharset -O ale012.unicharset
ale012.arial.exp012.tr
```

This step will generate `ale012.unicharset`, which we will need later when we combine all training data; `inttemp`, which specifies shape prototypes; `pffmtable`, the master shape table; and `pffmtable`, which specifies the number of expected features for each character. You do again not have to be concerned with the contents of any of those files.

10 Running `cntraining`

This step uses all `.tr` files as input to generate the `normproto` file, specifying character normalization sensitivity prototypes. In our scenario, training each model from just one page, this means providing just one `.tr` file as input:

```
cntraining [lang].[fontname].exp[num].tr
```

An Albertanian example:

```
cntraining ale012.arial.exp012.tr
```

Again, do not concern yourself with the contents of the `normproto` file.

Optionally, you can manually create a `unicharambigs` file that specifies possible ambiguities between characters. This is not required though, and I have not used it in my project. It might be useful if you find that a letter, or a combination of certain letters, always gets misinterpreted as something else. Find more info on this step here.

11 The Final Steps: Putting it All Together

Almost there! This is the step where we put everything together. This means that all the files that are being required in this last step will have to be given the same language prefix so that `combine_tessdata` knows which files to include. Renaming by hand is possible, but you can also use this quick call as a one-stop shop:

```
for f in inttemp normproto pffmtable shapetable; do mv "\$f"
"[lang].\$f" ; done
```

[9] If you are training a model for an Indic language in Tesseract 3.02, check the shapeclustering section on github; if you are using Tesseract 3.00 or 3.01, check the mftraining section before you execute this step.

This step should have renamed the four files by adding the [lang] prefix to each of them. In our Albertanian scenario:

```
for f in inttemp normproto pffmtable shapetable; do mv "\$f"
"ale012.\$f" ; done
```

The final step is to combine everything:

```
combine_tessdata [lang].
```

Albertanian:

```
combine_tessdata ale012.
```

Note that the period at the end is *not* an error. Do *not* omit it from your call! This should have created the `.traineddata` file, which marks the end of the training process.

To be able to use your new model in Tesseract, copy the `.traineddata` file to your `tessdata` directory. You can do this manually or with the call below. On a Mac, the call would very likely look like this; always make sure to verify the location of your tessdata directory before you run this call though:

```
sudo cp [lang].traineddata /opt/local/share/tessdata
```

Or, in the Albertanian pseudo-project:

```
sudo cp ale012.traineddata /opt/local/share/tessdata
```

12 Running Tesseract to Test your New Model

You can now test whether your model is operating as intended by running the following command in a folder that contains an image you would like to OCR:

```
tesseract image.tif image_OCRed_text -l [lang]
```

Or, assuming you want to OCR e.g. page 13 from the Albertanian source text:

```
tesseract image.exp013.tif image.exp013.ocr -l ale012
```

This call should produce a text file called `image.exp013.ocr.txt` that contains the OCR'ed text from page 13 of your Albertanian source image, and hopefully it'll be quite accurate! If it is not, don't despair: I have made good experiences with passing a different page segmentation mode parameter to the call, like so:

```
tesseract image.tif image_OCRed_text -l psm [psmnum]
```

As an example, we could pass the `psm` parameter `6` to the call in our Albertanian project"

```
tesseract image.exp013.tif image.exp013.ocr -l ale012 psm 6
```

Experiment a little with parameters that match how the text is organized in the scanned image. If the outcome still isn't very good then, check if the image file has any major issues (like black edges, lots of and/or large blotches, or if it is skewed.) Even if you have pre-processed your images prior to starting the training process, you might have missed a feature, or an image. It happens.

Either you are done at this point, and you can run Tesseract on the full set of scanned images, or, if you are planning to train more than one model, you can move on to Section 13 below. Here is an example call for how to run Tesseract on a number of pages, rather than on just one page at a time:

```
for i in *tif; do b=‘basename "\$i" .tif‘; tesseract "\$i"
../specify/output_directory/here/"[lang]_\$b" -l [lang]; done
```

Run this in the folder where your image files are located. This call will prepend the [lang] prefix to the name of the image file, so that you know exactly which output file was created from which image file. Make sure to specify an output folder. In our Albertanian example, if we would like to place the output in a folder that is a sister folder to the one we are working in, the call could look like this:

```
for i in *tif; do b=‘basename "\$i" .tif‘; tesseract "\$i"
../ocr_output/"ale012_\$b" -l ale012; done
```

Of course, you can specify the `psm` parameter here as well. For further options, have a look at the Tesseract command line usage manpage.

If you are interested, you can check some features of your newly minted OCR model by uploading your `.traineddata` file to the Traineddata Inspector. This is completely optional though, and more of a fun way to investigate what you created! If you have set out to create more than one Tesseract model, with the goal of comparing them to each other and then selecting the best one, continue reading Section 13 (also, good for you!). If you are planning on just training one model, you can either skip ahead to Section 14 if you'd like to do some basic quality assessment of your model, or you are done completely![10]

13 Creating Further Models

First of all, you will likely realize that hand-validating the `.box` files required to train further models will take less time than before, the reason being that you can use the one model you already trained to create any further `.box` files. This means that, where we used the Swedish model to create our first `.box` file, you can go ahead and use the model you just created. The images you are planning to use for training should already be pre-processed; if they aren't, just refer back to Section 3. Then, basically what you want to do is repeat Steps 5 to 11 for each model. The bash commands should essentially be the same; just be sure to adapt the page number, and to give the model a different [lang] name. A sample call for the second model we train for Albertanian could look like this (in a scenario where we do not know the font, and where the second model is based on page 434:)

```
tesseract ale434.mcfictitious.exp434.tif ale.mcfictitious.exp434
-l ale012 batch.nochop makebox
```

Run this call in the folder where the image file of page 434 is located (which should be in a separate folder from the first Albertanian model.) Then, just follow along with Steps 5 to 7. In Step 8, you can just copy over the `font_properties` file you created for your first model (unless, of course, the fonts are different, which we will not hope.) Follow along with Steps 9 to 11, and see if your new model works in Step 12. Rinse and repeat if you are planning to train a third model; otherwise, you can do some (optional!) basic quality assurance in Section 14 below, or you are done and ready to enjoy your OCR models!

[10] Note that we are not going to incorporate any dictionary or language data. This is because I have not used any in my project as Haida is a morphologically rich language and word lists are not terribly useful. Refer to the relevant chapter on the github page if you are interested in incorporating dictionary data.

14 Basic Quality Assessments

Most of the time, you'll be able to see right away whether a model performs adequately enough for you to use it for anything (if you trained just one model), or which out of several models performs best (if you trained more than one model.) However, if you'd like to have some hard-and-fast statistics, you can get those too: By using the 2016 version of the ISRI Tools for OCR Accuracy.[11] If you are planning to get some statistical assessment of your model(s), you are going to have to prepare some *ground truth*. This is essentially 100% correct text for the pages that are going to be your testing set. So say you would like to evaluate the performance of a model on three pages of Albertanian text. You would have to produce ground truth for those three pages. The quickest way would be to OCR them with one of the models you created, and then hand-validate the output text file. This is essentially the same process we used in Step 5, but a little less onerous as there are no boxes to adjust; for pages with roughly 1,200 characters, this took about 20 minutes in the beginning, then around 10 for further pages.

Once you have produced your ground truth, you can begin using the ISRI tools. Describing their usage in detail goes beyond the scope of this manual, as it is a training manual, not a quality assessment manual; however, there is a great user guide available [2] that should give you most of the information you need. While it was written for the non-Unicode version of the tools, from the year 1996, the function calls have remained the same. For basic assessment purposes, the `accuracy` and `wordacc` tools should be sufficient. To get a basic understanding of whether the accuracy rates that are being output by those tools are any good, as a proxy for whether your model is any good, you can refer to [1] for some values reported in the literature – or you can google accuracy rates in Tesseract.

References

[1] Hubert, I. (2016). Training & Quality Assessment of an Optical Character Recognition Model for Northern Haida. First Generals Paper, Department of Linguistics, University of Alberta [unpublished].

[2] Rice, S. V. and Nartker, T. A. (2016). The ISRI Analytic Tools for OCR Evaluation - 2016 Port. Ported and extended by Eddie Antonio Santos. https://github.com/eddieantonio/isri-ocr-evaluation-tools, accessed 05 Jan 2016.

[3] Swanton, J. R. (1905). Haida Texts and Myths: Skidegate Dialect. In *Bulletin (Smithsonian Institution. Bureau of American Ethnology)*, volume 29. Government Printing Office, Washington.

[11] There are other tools available as well; this one is my favorite, and it handles Unicode nicely. For other options, see [1].