Ocrocis

A high accuracy OCR method to convert
early printings into digital text

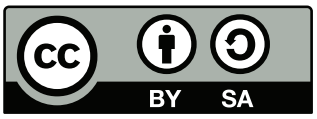
A Tutorial

Uwe Springmann
Center for Information and Language Processing (CIS)
Ludwig-Maximilians-University, Munich

Email: springmann (at) cis.lmu.de

version history:

version 0.95: 05. March 2015
version 0.96: 11. August 2015



Contents

1 Abstract

This tutorial describes the first successful application of OCR to convert scanned images of books over the complete history of modern printing since Gutenberg into highly accurate digital text (above 95% correctly recognized characters for even the earliest books with good scans). This opens up the possibility of transforming our textual culturage heritage by OCR methods into electronic text in a much faster and cheaper way than by manual transcription.

2 Introduction

By a new method of optical character recognition (OCR) based upon recurrent neural networks with long short-term memory (Hochreiter and Schmidhuber, 1997), first introduced into OCR by Breuel et al. (2013), it has become possible to convert not only relatively recent printed material into electronic text, but also very old printings reaching back to the invention of modern printing by Johannes Gensfleisch genannt Gutenberg (ca. 1450). By early printings we here summarily mean all printings from the incunable period (1450-1500) until the 19th century, where conventional OCR machines begin to deliver good results with high character accuracy, defined as the number of correctly recognized characters divided by the number of all characters in a page or document.

As anyone knows, who tried to apply products such as Abbyy Finereader[1] or the open-source engine Tesseract[2] to earlier data (OCR is routinely applied to the scans at `archive.org` and the resulting text is available for download), the results for early printings are rather disappointing: The accuracy of the textual output is badly damaged because of non-standard typography, to which the physical degradation resulting from age and usage add additional barriers. Although Tesseract can be trained to new languages and specific fonts, it seems that the variability arising from early typography and the degradation of the physical images of a typeface, the glyphs, does not lend itself easily to training. Training experiments for both Tesseract and OCRopus (now called Ocropy[3], Breuel (2014)) by converting available text to noisy images using similar-looking computer fonts showed that only moderate success could be reached (see Springmann et al., 2014).

The bad accuracy is understandable, as OCR has been developed in the 20th century with a specific focus on contemporary printings. Any successful application out of its intended domain means that the OCR engine has to be trained on historical fonts with all their peculiarities. The neural-network based Ocropy is able to give high-accuracy results (above 95% accurately recognized characters even for the earliest printings) when trained on real data, i.e. on the scans of printed text lines as opposed to synthetically generated images from existing text and computer fonts.

Unfortunately, this software is neither well-known nor does it come with any kind of documentation apart from an installation Readme file. Furthermore it runs only on Linux systems. It was also not specifically designed to work on early printings and the training on English and Fraktur fonts (see Breuel et al., 2013) relied heavily on synthetic computer generated line images. Still it works very well even on early printings when trained in a proper fashion.

In order to bring these methods to a larger audience, we decided to publish a documentation in tutorial form that would enable any interested individual (or the resident

[1] `www.abbyy.com`, `ocr4linux.com`, `www.frakturschrift.com`
[2] `http://code.google.com/p/tesseract-ocr/`
[3] `http://github.com/tmbdev/ocropy`

information specialist) to train a model tailored to a specific book so that the resulting text from OCR, given a clean original copy with at least 300 dpi grayscale or color scans, will have at least 95% character accuracy. In addition, there is a software part written by David Kaumanns in form of a wrapper script organizing the time consuming process of annotating line images with their electronic representation (“ground truth” in OCR speak) and training a specific model. Because Ocropy has many dependencies on other software and runs natively only under Debian-based Linux systems, we distribute our software in form of a virtual Docker file which already contains all dependencies. We are also working to make it usable on the widely used Mac OS. This tutorial and the software together go under the name of "Ocrocis" (Springmann and Kaumanns, 2015).

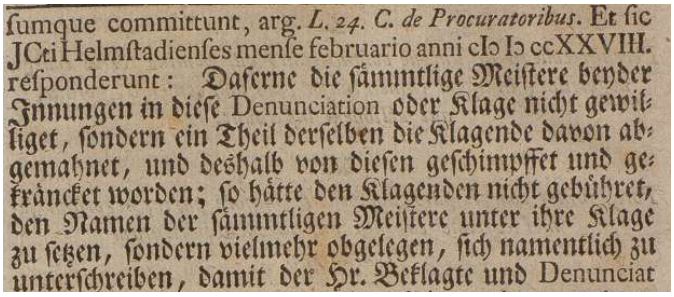
It should be noted that Ocrocis is not a push-button application that delivers excellent results within minutes by just feeding it a pdf file of page images of an old book, nor do the process steps described here lend themselves easily to integration in an automated workflow geared towards mass OCR conversion (that is the subject of ongoing research). Rather, we have the individual researcher ("digital humanist", be it a student, a PhD candidate or an established researcher) in mind who is interested in a specific early work whose electronic text is not available elsewhere and who is willing to invest the amount of manual work necessary to prepare the ground truth needed for the training process. With some hours of diligent effort for preprocessing and annotating of some training pages plus some additional hours of training (this is automatic machine work while the researcher is free to do something else), he or she will be able to extract the text of the complete work plus any additional works with the same typographic characteristics. The resulting text can then be used for searching or as a first draft for further correction with the goal to establish an error-free transcription.

While the method described here is completely general and can be applied to many languages and alphabets (in fact, the results do not use any dictionaries at all, so accuracy may further be improved by methods of automatic and interactice postcorrection), it is especially useful for early printings where there is currently no hope for a decent OCR result from other engines. Applied to recent printings (especially from the 20th century onwards), the possible superior character recognition of Ocropy gets counterbalanced by the much more refined methods of document analysis (differentiating between text and non-text regions on a page), lexical postcorrection and ease of use of market leading products.

To give you an idea of what can be achieved by this method we show you some examples below (see Fig. 1, 2, 3).

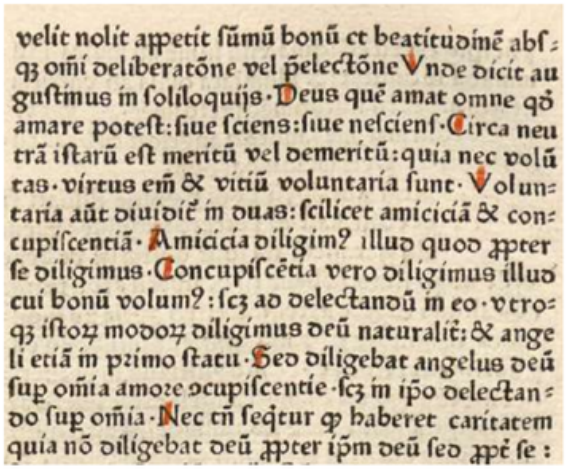
[4] `http://digitalhumanities.org/dhq/vol/3/1/000027/000027.html#p7`

Figure 1: Adam von Bodenstein (1557): Wie sich meniglich … (scanned image from BSB). Right: Uncorrected Ocropy output of a previously unseen page after training on 32,000 randomly selected text lines (images + associated ground truth) from a training set of 34 diplomatically transcribed pages. The OCR result has 8 remaining errors on this page (5 substitutions marked as red characters, 1 deletion marked by blue adjacent characters, 2 insertions marked by a red underscore). The mean accuracy on 5 test pages is 99%. Middle: The output of Abbyy (FineReader Engine SDK 11.1 for Linux) with options `-rl Gothic -rll OldGerman` (Gothic script and old German lexicon) recognizes ſ, but suffers heavily from narrow word spacings, running words together in the output. The narrow space is the result of early printers' attention to right-justification: To get a beautifully justified line, words were both abbreviated and set together very narrowly if needed. Tesseract with language setting `-l deu-frak` achieves 78% on this page. (Springmann 2015, in preparation).


ſumque committunt, arg. *L. 24. C. de Procuratoribus.* Et ſic
JCti Helmſtadienſes menſe februario anni cIↃ IↃ ccXXVIII.
reſponderunt: Daferne die ſämmtlige Meiſtere beyder
Innungen in dieſe Denunciation oder Klage nicht gewil-
liget, ſondern ein Theil derſelben die Klagende davon ab-
gemahnet, und deshalb von dieſen geſchimpffet und ge-
krâncket worden; ſo hätte den Klagenden nicht gebühret,
den Namen der ſämmtligen Meiſtere unter ihre Klage
zu ſetzen, ſondern vielmehr obgelegen, ſich namentlich zu
unterſchreiben, damit der Hr. Beklagte und Denunciat

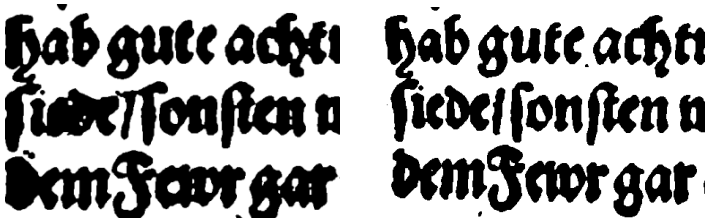
ſumque committunt, arg.L. 24. C. de Procuratcribus.Et ſic JCti Helmſtadienſes menſe februario anni cIↃIↃ ceXXVIII. reſponderunt: Daferne die ſämmtlige Meiſtere beyder Innungen in dieſe Denunciation oder Klage nicht gewilliget, ſondern ein Theil derſelben die Klagende davon abgemahnet, und deſhalb von dieſen geſchimpffet und gekrâncket worden; ſo hätte den Klagenden nicht gebühret, den Namen der ſämmtligen Mettere unter ihre Klage zu ſeten, ſondern vielmehr obgelegen, ſich namentlich zu unterſchreiben, damit der Hr.Beklagte und Denunciat

Figure 2: Augustinus Leyer (1735): In Regis Poloniarvm … Meditationes Ad Pandectas (scanned image from HAB). This is a case of mixed typefaces, Fraktur for German and Antiqua for Latin. Training was done on 40 pages in 47,000 steps (randomly presented text lines). The mean accuracy on 8 test pages is 97%. Abbyy achieves 77%, Tesseract 82%. (Springmann 2015, in preparation).



velit nolit appetit ſũmũ bonũ et beatitudinẽ abſ · q3 oñi deliberatõne vel p̃electõne Vnde dicit au guſtinus in ſoliloquijs · Deus quẽ amat omne qd̃ amare poteſt:ſiue ſciens:ſiue neſcienſ· Circa neu trã iſtarũ eſt meritũ vel demeritũ : quia nec volũ tas · virtus eñ & vitiũ voluntaria ſunt · Vol une taria aũt diuidit̃ in duas: ſcilicet amicicią & con= cupiſcentią · Amicicia diligim⁹ illud quod ꝓpter ſe diligimus · Concupiſcẽtia vero diligimus illud cui bonũ volum⁹ : ſc3 ad delectandũ in eo · vtro= q3 iſtoꝝ modoꝝ diligimus deũ naturaliṫ: & ange li etiã in primo ſtatu · Sed diligebat angelus deũ ſup oñia amore ɔcupiſcentie · ſ3 in ip̃o delectan= do ſup oñia · Nec tñ ſeq̃tur q̃ haberet caritatem quia nõ diligebat deũ ꝓpter ip̃m deũ ſed ꝓp̃t ſe :

Figure 3: Image and OCR result of Vincent of Beauvais' Speculum Naturale (scanned image from BSB), printed by Adolf Rusch in Augsburg before 1476. Ocropus has been trained on 13 pages with accompanying ground truth text, while the resulting character recognition rate was tested on additional 4 pages (average rate: 98%; Springmann 2015, in preparation). The many special glyphs in incunabula signifying abbreviations were the reason for Rydberg-Cox' statement: "Because of the prevalence of these glyphs, incunabula cannot be processed using OCR software. Commercial OCR programs produce almost no recognizable character strings, let alone searchable text. ... Other methods must be explored.[4](Rydberg-Cox, 2009)" For Abbyy and Tesseract, this is till true.



(a) Google (b) BSB

Figure 4: A snippet from "Alchymistische Practic" (Andreas Libavius, Frankfurt 1603). At the the right you see the scan as available from BSB, at the left the same scan with lower resolution as offered by Google. After training a model on each source, the prediction accuracy is 97% for BSB and 94% for Google.

3 The OCR workflow

Before we describe the Ocrocis commands, we give some information on the general OCR workflow with hints to each step.

3.1 Image acquisition

The first step is to locate good scans with high resolution (at least 300dpi), preferably grayscale or color. The advantage of grayscale or color scans is that you have control over the binarization procedure. Primary sources for freely downloadable scans are `www.hathitrust.org`, `www.archive.org`, `www.europeana.eu`, the Bavarian State Library[5] (BSB) as the largest collection of digitized books in Germany and the Herzog-August-Library[6] (HAB) at Wolfenbüttel, one of the finest collections of early modern prints in Europe. There is also Google Books[7], which sometimes offers material also available from other sources such as BSB, but at a reduced resolution and often already binarized in an uncontrollable fashion. This leads to reduced file sizes, but may also negatively impact character definition and legibility and hence a loss in achievable accuracy in the OCR process (see Fig. 4).

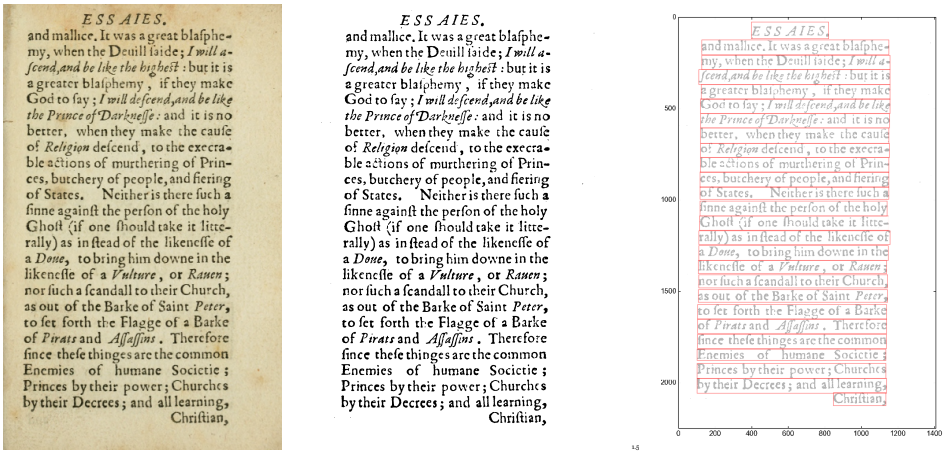
3.2 Preprocessing

Preprocessing of page images consists of several steps such as deskewing, dewarping, despeckling, page segmentation, binarization with the goal to provide the OCR engine

[5] `http://www.digitale-sammlungen.de`, `http://opacplus.bsb-muenchen.de/metaopac/search.do?methodToCall=selectLanguage&Language=en`

[6] `http://www.hab.de`, `http://opac.lbs-braunschweig.gbv.de/DB=2/LNG=EN/`

[7] `http://books.google.com/`



(a) original page image (b) binarized (c) line segmented

Figure 5: Preprocessing done by Ocropy: Deskewing & binarization (b) and text line segmentation (c). Scanned image of Bacon's Essaies (1613) from `archive.org`.

with the best possible input, i.e. a clear separation of glyphs (signal) and background (noise). Although Ocropy (and hence Ocrocis) contains inbuilt functions to do preprocessing, the process is by no means error-free (this is true for all OCR engines). Especially page segmentation can be cumbersome: If the images have a multi-column layout or marginal notes, the recognition will greatly benefit if you either cut out subimages of a page pertaining to different zones such as margins, columns, headings etc. (this process is called "zoning") and subject them to OCR separately, or at least crop the interesting textual material so that no contamination happens from any marginal or other material (like handwritten notes, floral decoration etc.). Footnotes with multilingual alphabets can also be problematic. Ocropy will ultimately resolve a printed page into single text lines, and anything not aligned with these lines is noise that will disturb the training process. The open-source software ScanTailor[8], available both with a GUI and as a command-line utility, has proven to be of good value.

Apart from training, preprocessing is the single most important step for achieving good OCR results, and it even sets the limit for what training can achieve (see Fig. 4 (a) and (b), which both originate from the same scan but treat binarization differently; the same is true for the segmentation in text versus non-text lines).

In order to prepare for training as well as the later recognition step, Ocropy will segment page images into single text lines (see Fig. 5). After binarization of the pages, each page image will get its own directory wherein each file corresponds to a single text line.

[8] `http://scantailor.org/`

3.3 Training

Training is at the heart of the effectiveness of the new OCR method. It works by learning from properly annotated text line images, comparing a horizontal line of pixel inputs with its corresponding "ground truth" correctly transcribed text. Having seen such annotated lines repeatedly and comparing input and desired output, the neural network underlying Ocropy adjusts its internal states in such a way that it is eventually able to predict the correct glyph values when being shown hitherto unseen, new text line images. The adapted internal states of the network are saved as a "model" in a file, so that it can later be loaded into memory, enabling the recognition of new text from page images with the same typographical characteristics as those on which it has been trained (e.g., all the rest of the book from which the training pages have been taken).

Contrary to other OCR engines, however, Ocropy does not use single characters as the smallest recognizable entity. Rather, its "atomic constituents" are vertical slices of pixel values, called "frames", along the text line image. A single text line gets sliced up into as many as 1000 frames, so that each glyph (character or ligature) gets resolved into several slices. By repeatedly comparing the input pixel values for each frame to the desired output (ground truth for a glyph), the neural network adjusts its internal memory states in such a way that it "learns" to produce the correct glyph from all frames corresponding to this glyph, including the white space separating single words.

The first step, therefore, consists in annotating a number of printed text lines with its electronic counterpart.

3.3.1 Annotation

Transcribing printed text lines can conveniently be done in a browser which will display the earlier segmented line images alternated by an empty line into which the user will have to enter the ground truth (Fig. 6).

You have to decide what level of historical detail you want to preserve: For a real diplomatic transcription, you will want to record the long s (ſ) as well as the vowel ligatures Æ,æ and Œ, œ plus all the diacritics that may be present in printing. Otherwise, you just transcribe both s and ſ by s (both glyphs will be recognized and mapped to s, so you avoid the infamous misrecognition of ſ as f present in most OCR results of older books) and resolve the ligatures to ae and oe. On the other hand, you must not map the same (or identically looking) glyph to different characters, as otherwise the system will be confused about the character that is represented by a specific glyph (this means that you have to transcribe printing errors as well!). Fig. 7 gives an example of a glyph with two meanings (m and z) which needs to be annotated by a single character to ensure a stable prediction that does not randomly alternate between two different character annotations. The resolution of this ambiguity must then be dealt with during the post-correction phase.

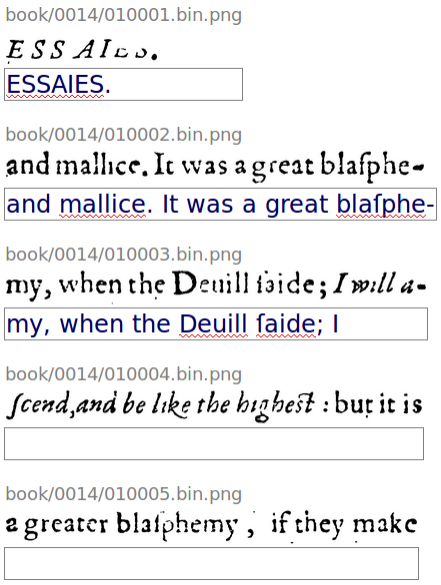
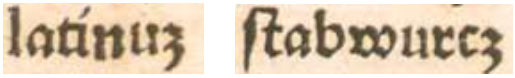


Figure 6: A browser window with line images and additional space into which ground truth is being entered. Spellchecking in the browser is enabled but will highlight historical spellings as well as real transcription errors.



(a) latinum (b) stabwurcz

Figure 7: Johann Wonnecke von Kaub (Johannes de Cuba), Gart der Gesundheit, Ulm 1487. The same glyph is employed for both the occasional designation of m at the end of a word (latinum) and the character z (stabwurcz).

As is obvious from Fig. 3, the (human) recognition and correct transcription of incunabula require palæographical skills. The Medieval Unicode Font Initiative (MUFI[9]) gives a comprehensive catalog of all UTF-8 codes that can be employed for the transcription. You will have to consult your operating system to learn how to input Unicode codes; under Linux/Gnome (including Debian and Ubuntu), it is just a matter of pressing CTRL-SHIFT u followed by the 4-digit hexcode. For letters with diacritics, there often exist different Unicode representations, either as base letter + diacritic (two codepoints, decomposed form) or a single code for the complete symbol (composed form). Use whatever is most convenient for you to input (accents may faster be written by the keyboard's mute keys), as the ground truth will later automatically be normalized to Unicode's "Normal Form Composed" (NFC)[10]. Also, the symbol inventory will be automatically determined from your annotation data and written to the file charset.txt.

In annotating text lines for later training, we need to strive for a fair representation of the kind of glyphs that we are going to see frequently in the book pages. By that we mean that ideally every glyph should be represented with the same frequency. We are saying glyph instead of character because already the form of the representation of a character, be it upright, italic, upper or lower case etc. will make a difference in appearance and therefore need to be learned separately. Individual letter frequencies do vary widely in any real language and we cannot expect to recognize a letter which has never, or only very infrequently, shown up in our training material. The same is true for other symbols such as punctuation or numerals. As we cannot control individual symbol frequencies as long as our text lines consist of real text and not just strings of statistically equiprobable symbols, we can at least make sure we have a good proportion of all those kinds of glyphs in our training material that are going to show up fairly often. Those that are relatively infrequent will be learned less well and have a higher probability of misrecognition than more frequent symbols. If we train on English prose text, the following glyphs can be expected to be underrepresented: capital letters, question marks, italics (if only used for headings), numbers.

The quintessence of this discussion is that we must choose our training pages with some discretion. We cannot just transcribe the first 20 pages of a book, consisting of the title page with unusually large letters, the preface and the table of contents which may be printed in italics, and then hope to get good recognition results on the following 300 pages printed in upright Roman style. Rather, we might take 5 pages of italics to compensate for their later infrequent appearance in headings and citations only, and use another 15 pages of the bulk material of the book. If the scans are not uniform, it is a good idea not to take consecutive pages for training but to sample them over the whole book. With a view on later postprocessing, it is more efficient to get the bulk of a book recognized well and transcribe the title page manually than the other way round.

[9] `http://folk.uib.no/hnooh/mufi/specs/index.html`

[10] Actually, Ocropy will currently normalize to NFKC: normal form compatible composed, lumping e.g. s and ſ to s before training. So you cannot currently get an ſ in recognition output except if you install Ocropy directly and input the used symbols as a separate "codec" in the file chars.py.

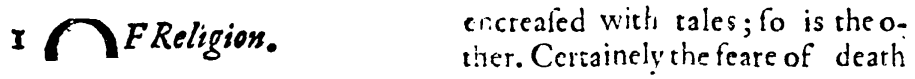


Figure 8: Instances of segmentation errors. Left: An incomplete drop capital has entered the line. Crop it or drop it (leave the annotation blank). Right: Two lines segmented as one. Drop it.

You should also avoid to choose pages with illustrations, marginal notes or decorations. To get pure text lines, we would either have to cut out these items before text segmentation or at least to crop the annotated text lines to reduce noise. At the present stage the goal is not to get a good recognition of the text on a specific page but to get a large number of training lines with the least effort. If one still hits a bad line in the browser (e.g. a line with noise at the end), one could annotate the line and crop the image, or leave its text line blank. In the case of two or more lines segmented as one line, we must leave the text line blank, as it would not yield a meaningful training line. Lines without associated ground truth will be ignored in training.

3.3.2 Training a model

Model training is largely automatic, the only things one has to decide are:

- the number `nlines` of training lines (these have to be annotated as described above)
- the number `ntrain` of training steps
- the number of training steps `savefreq` after which a new model gets written to disk

There are further internal network parameters which are best left at their default values unless you are an expert.

In our experience we got good models with 1,000 ... 5,000 training lines (the higher number for font and typeface mixtures) which required 30,000 to 200,000 training steps. The question is when adding more training data will not noticeably improve the recognition performance. To find that out in an efficient way, one can start with about 300 training lines, train for 100 epochs (an epoch is the number of training steps after which the randomly drawn lines have on average been seen once and therefore equals the number of lines), add more training lines, use the best saved model from the previous step for a recognition of the new lines which then only need to be corrected to generate new ground truth, and so on. Set aside about 10% of annotated lines for testing the model performance. Once the performance shows fluctuating values without any trend, you can stop further training.

Here is our rule of thumbs: `nlines` $= 300$, `ntrain` $= 30,000$, `savefreq` $= 1,000$.

This will ensure 100 epochs of training and 30 saved models per iteration. Training for 30,000 steps will take some hours on a modern PC.

The procedure is then as follows:

1. Identify about 10 good training pages.
2. Annotate at least 300 lines.
3. Set 10% of annotated lines aside for testing.
4. Train on the remaining lines for 30,000 steps.
5. Find the best model by comparing their error rate on the testing lines.
6. Identify additional training pages and segment another 300 lines.
7. OCR these new lines with the best model from the last iteration.
8. Annotate the new lines by correcting the OCR result.
9. Add 10% of the new lines to the testing pool.
10. Add the remaining lines to the training pool.
11. Continue training (start with the last best model) on the training pool for another 30,000 steps.
12. Find the best model from the new iteration.
13. Continue the process until model performance shows fluctuations without trend.
14. Identify the best model and OCR complete book.
15. Save your annotated data and your best model.

If you are impatient, you can annotate more lines per iteration so that you need only one or two iterations until you finish. Then update the training steps and the model saving frequency accordingly.

Testing your models works by using the model to OCR the test line images (predict their characters) and comparing the generated text with the previously annotated ground truth.

You will notice that once the models get really good in reproducing the training data, sometimes the discrepancies between OCR output and ground truth do not arise from faults in the model but from errors in your own annotations! In that case correct your ground truth, otherwise your transcription error rate will set an upper limit for the achievable OCR accuracy.

If you installed Ocropy locally, you can watch your network train by the command

```
ocropus-rtrain -o modelname -d 1 book/*/*.bin.png
```

(-d 1 updates the picture in Fig. 9 with every input line, -d 2 with every second line etc.).

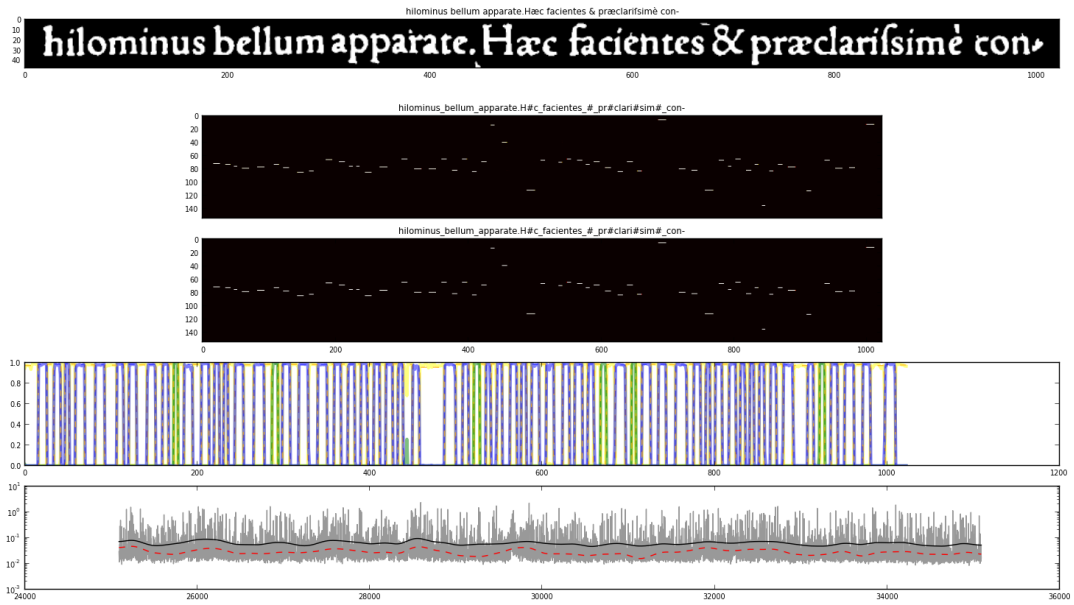


Figure 9: Training in progress. The upmost panel shows a line image with its associated ground truth. The numbers below the line are the frames into which the image gets sliced. The next two black panels give the output of the network: each small light horizontal line corresponds to a number at the left side which enumerates a specific glyph (think of these numbers as a representation of a typographer's type case). The lower numbers (upper part of the panel) correspond to numerals followed by letters and special glyphs in the lower part (the two æ-ligatures are represented by a number larger than 100). As you see, the already well converged model shows mostly letters (middle area). The output of the network closely resembles the annotated ground truth. The next panel shows posterior probabilities for each character. Their rectangular form indicates that the network is fairly sure about which character corresponds to a specific glyph. Inter-word distances are shown in green, characters in blue. The last panel shows the training progress, both per line (the spiky line) and as a moving average of the least squares distance between aligned and unaligned output (the two panels above). The dashed red line is a moving average of the character errors per output line. This panel's vertical number correspond to the learning steps of the network. There has not been much change in the last few thousand steps, so the network has reached a quasi-steady state.

3.4 Character recognition

Once you have a good model, you can run it on a total book directory. The OCR engine will go through the page directories under book/ and generate character string for each line image. In addition to misrecognitions at the character level which got tested with the annotated test lines, now also segmentations faults such as bad text/non-text separation or several lines recognized as one will take their toll. If the scanned material contains a lot of non-text material, a better preprocessing could remedy this error source as discussed above.

The text can be extracted either as pure text (UTF-8) or as hocr with line coordinates.

3.5 Postprocessing

As the OCR result is by no means a perfect representation of the printed text, postprocessing the output will generally be necessary. Depending on your interests the raw OCR output might or might not be what you want. Searching will find many instances but not all; tolerant search will find more instances, but some of them will be false positives. If the discoveries are highlighted in the images one will quickly be able to discard these false positives by just looking at them. The percentage of misses will be lower and what you get might be sufficient (e.g. if you look for some attestations of word or grammar usage, or citations). Tolerant search can also cope with historical spelling patterns by employing correspondence rules between modern and historical spellings.

Another area is the postcorrection of the OCR result, either with or without spelling normalizations. If you need error-free text, a highly accurate OCR result will save you much transcription time when going systematically over each line, comparing it with its printed counterpart and correcting just the errors rather then transcribing the complete line from scratch. Even more efficient is the usage of the tool PoCoTo (for PostCorrectionTool, developed at CIS (Vobl et al., 2014)) for interactive error correction, which presents complete error series based upon a calculated statistical error profile for the document that is to be corrected. After inspecting the error series (e.g., e misrecognized as c), the user approves or changes the correction candidates and can then correct the whole series at the click of a button. The tool is able to distinguish between historical spellings and OCR errors, both of which are at odds with a lexicon of modern word forms. We are working to make the tool and the server calculating the error profiles openly available together with documentation how to install and use it.

4 A practical example

Having learned something about OCR in general and Ocropy specifically, we now take a practical example and run through the above steps, explaining the specific commands

along the way.

Installation of the Ocrocis software is explained in its README[11], for installation of Ocropy see its github repository.[12]. The following assumes that you are working in a project directory, e.g. `OCR/projects/bacon/`. After installation, there is help available for each Ocrocis command, e.g. `ocrocis convert --help`, as well as for the Ocropy commands, if you choose to install Ocropy natively and work with its commands directly. See Sect. 6 for a tabular overview of the input and output associated

We will take the 1603 edition of the Essays of Sir Francis Bacon[13] as an example. Many editions can be found on `www.archive.org` and the electronic text is available in several places on the internet, so we will not uncover something new. It just represents any early printing you might actually be interested in.

4.1 Download the images

The 1603 edition at archive.org[14] is offered in djvu, pdf and jp2 format. The jp2-images have the best resolution, so download the zip archive[15].

Next unpack the zip-file and put all images under a `book/` directory in your project directory.

4.2 Binarize the images

Next issue the command:

```
ocrocis convert
```

which will take the jp2 images, convert them to png, and then do deskewing and binarization. This will take some time, you could use the option `--cpus 4` (4 cpus, if you have them) to accelerate. The convert part is done by ImageMagick's convert, which comes with its own set of options. Other image formats (tif, png, jpg) are understood by Ocropy directly and will be binarized right away. If you start from still other formats, convert them first to png and put the png under the book directory before you issue the `ocrocis convert` command. Make sure you have at least 300 dpi png; sometimes you might need to do a `convert -density 300 <file>.pdf` if you end up with thumbnail-like images.

If you prefer to work with Ocropy commands directly, the equivalent command is:

[11] see `https://code.google.com/p/cistern/wiki/Ocrocis`
[12] `https://github.com/tmbdev/ocropy`
[13] `http://en.wikipedia.org/wiki/Essays_Francis_Bacon`
[14] `https://archive.org/details/essaiesofsrfranc00baco`
[15] `https://ia600400.us.archive.org/22/items/essaiesofsrfranc00baco/essaiesofsrfranc00baco_jp2.zip`

```
ocropus-nlbin <image-dir>/*.png -o book
```

4.3 Segment into lines

Line segmentation is done by the command:

```
ocrocis burst
```

or

```
ocropus-gpageseg book/*.bin.png
```

You now have single page directories under textttbook/ named `0004, 0006, 0007` etc. (the missing numbers correspond to pages where Ocropy could not find any text) with text line files ending in bin.png.

4.4 Choose images to annotate

According to our earlier discussion in Sect. 3.3.1, we are looking for pure text pages with a fair representation of the typefaces employed in the bulk of the book. The text is mostly printed in upright Roman, but italics are also present. We therefore take a few pages of the table of contents with all-italics and frequent numbers, then some pages of regular text. With Ocropy, you would edit an html-file with line images to annotate by, e.g.

```
ocropus-gtedit html book/00[1-2][0-9]/*.bin.png
```

which results in a file correction.html containing the line images of pages 10-29. After you annotated this file, the command

```
ocropus-gtedit extract correction.html
```

will put the annotated gt-lines as text files into the original book image directories (the directory `book/0010/010001.bin.png` will now also contain a file `book/0010/010001.gt.txt` etc., except if you chose not to annotate that specific line). Choose a subset of the annotated pages and copy them to a training directory, the remaining part to a test directory. These are your gold data and no matter what you do later (e.g. if you remove the book directory and start over again), you will not lose your work. You may later add additional annotations to both the train and test directories.

As this process entails a certain amount of bookkeeping, Ocrocis strives to make it easier. Taking the pages 10-29 for your first iteration, you will write:

```
ocrocis next {10..29}
```

A file named Correction.html will be generated under `iterations/01/`. This file needs to be edited in your browser, leaving any badly segmented lines empty. After you have annotated the file, save it to the same name and place.

4.5 Train a model

From the annotation set of page images we will choose a subset for training (the rest will later go into a test directory):

```
ocrocis train --ntrain 30000 --savefreq 1000 {10..24}
```

This will copy the ground truth into `iterations/01/annotation/`, link it to a newly created training directory, determine the set of individual characters (written to `book/charset.txt`) and start model training.

With Ocropus, you have already setup your annotated data in training and test directories. The most basic training command would consist of:

```
ocropus-rtrain -c <train-dir>/*/*.gt.txt <test-dir>/*/*.gt.txt -o <model>
<train-dir>/*/*.bin.png
```

The defaults for training steps are 1,000,000 and for saving frequency 1,000. Watching training proceed in a terminal gives you a rough idea how well the ground truth is being recognized by the network. Every 1,000 steps a model gets saved to disk.

Note that currently Ocropy with the -c option does a NFKC normalization to determine the set of all characters, so an ſ will be lost in the output layer of the network. In order not to lose any annotation labels such as ſ, you will have to NFKC normalize your own ground truth data as well (this can be done by the `uconv` tool; it will convert all ſ into s, among other things). Another option would be to add a string of your additionally used characters to the file `chars.py` (look under `ocropy/lib/python/ocrolib`) as a new codec (e.g., mychars) and add it to the default codec:

```
digits = u"0123456789"
letters = u"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
symbols = ur"""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""
ascii = digits+letters+symbols
xsymbols = u"""€¢£»«›‹÷©®†‡° •∘ ¶§÷¡¿ """
mychars = u"ãẽĩõũỹñm̃áéíóúýàèìòùỳâêîôûŷẹëßſÆæŒœ"
default = ascii+xsymbols+mychars
```

This is just an example, you get the idea. If you did this, the training command would just be

```
ocropus-rtrain -o <modelname> <training-dir>/*/*.bin.png
```

The training progress can be watched on the terminal from which you issued the above commands (another way to watch the training progress in time was shown in Fig. 9). The output looks like this:

```
16011 5.58 (423, 48) train/0010/010009.bin.png
   TRU: u'6 Of Parents and Children.'
   ALN: u'6 Of Parents and Children.'
   OUT: u' dOf Parents and Children.'
16012 9.00 (561, 48) train/0014/010003.bin.png
   TRU: u'my, when the Deuill \u017faide; I will a-'
   ALN: u'my, when the Deuill \u017faide; II will a-'
   OUT: u'my, when the Deuill aide;  will a-'
16013 5.98 (520, 48) train/0014/010015.bin.png
   TRU: u'of Pirats and A\u017f\u017fa\u017f\u017fins. Therefore'
   ALN: u'of Pirats and A\u017f\u017fa\u017f\u017fins. Therefore'
   OUT: u'of Pirats and A\u017f\u017fa\u017f\u017fins. Therefore'
```

The left-justified line gives the training step (here 16011, 16012 and 16013) followed by a measure of uncertainty. It is never zero even for a perfect recognition result as in the third line, because a recognition has a posterior probability between 0 and 100% for each character, and the uncertainty gives the sum of these numbers over the whole line. The next two numbers in parentheses are the pixel width and height of the line image, then the corresponding file name of that line image.

The next three lines give the ground truth (TRU), the network output after it has looked at the data (ALN for aligned) and the real network prediction (OUT). A perfectly trained network would show three lines with identical output (as in the case of the last line). The first line shows that the number 6 has not yet learned (it is confused with the letter d, and the inter-word space has not been correctly recognized). The output is given in ASCII as not all terminals understand UTF-8, therefore UTF-8 sequences are shown for special non-ASCII characters such as `\u017f` for ſ. The second line shows that the network is still uncertain about ſ and I, so these characters are not given. Also, the aligned output shows two II instead of one; both of these errors go away with enough training.

If the network has generally very low uncertainties on its recognition, but sometimes a high value, this more often than not indicates an error not in its recognition but in the ground truth. Correct the ground truth while the training is running. The next time this line comes up for training the error will diminish.

4.6 Test the models

The performance of your model on unseen data is tested by the command

```
ocrocis predict --errors
```

which takes the remaining data not used for training, runs an OCR with either a specified or the last model on it, and compares the output with the annotated ground truth.

If you want to improve the model, you can try do to so with the next iteration of annotations. Follow the steps above for new pages and the annotated data will be added to the training and test data pool.

On the Ocropy side, a model can be tested against unseen test data by first running an OCR with a model:

```
ocropus-rpred -m <modelname> test/*/*.bin.png
```

and then

```
ocropus-errs test/*/*.gt.txt
```

which will give you an output such as:

```
     3     33 test/0045/010009.gt.txt
     5     37 test/0045/01000a.gt.txt
     7     39 test/0045/01000b.gt.txt
     2     34 test/0045/01000e.gt.txt
     1      5 test/0045/01001a.gt.txt
errors         317
missing          5
total         3450
err           9.188 %
errnomiss     9.043 %
```

For each test line, it gives the number of recognition errors, the number of characters in each line and the file name of the ground truth line against which the recognition result is evaluated. The last 5 lines give error statistics: The number of errors summed over all lines, the number of errors in "missing" lines (these lines are too short to give reliable results; Ocropy does not learn so much single characters as characters in the context of a complete line). One of those missing lines is the last one consisting of just 5 characters: This is a case of a so-called catchword which gives the first word or syllable of the next page at the end of the previous page, an age-old typographical habit fallen since out of use. Next comes the total number of characters in the test set and the error rate, both as the fraction of errors to the total number of characters and the smaller rate adjusted for missing lines.

It is also possible to view the confusion matrix (which letters get confused for which others) by

```
ocropus-econf test/*/*.gt.txt
```

which yields:

```
errors          317
missing           5
total          3450
err           9.188 %
errnomiss     9.043 %
13 _ i
8 f f
5 a _
5 s _
5 _ T
4 _ t
4 i _
4 b h
4 u n
4 _ e
0.091884057971
```

This is the same 5 line statistics as above followed by the 10 most frequent confusions, followed again by the error rate. The first column of the confusion matrix gives the frequency a specific confusion happens, the second the recognition result, the third the ground truth. So 13 times an i has not been recognized and left out (deletions are indicated by a _ in the recognition column), 8 times an f has been misrecognized as an ſ (substitution) and 5 times an a has been inserted where it does not belong (insertions, indicated by _ in the ground truth). See the command's help for adding context etc.

If you have a lot of models, the question which one is the best can be answered by running recognition and a subsequent test on each of them. This can easily be effected by a shell script of the form

```
for i in *.pyrnn.gz; do echo "$i" >> modeltest; ocropus-rpred -m "$i"
test/*/*.bin.png; ocropus-errs test/*/*.gt.txt 2>>modeltest; done
```

You will have in the file modeltest the name of each model followed by its 5 line error statistics listed one after the other. Find the model with the least error and keep it; the others might be deleted. For reference, the error levels of our models after every 1,000 training steps both for our training data (344 annotated lines) and our test data (112 lines) are given in Fig. 10. As you see, the error level hits a barrier at about 6% and the network tries several times to reorganize itself, leading to a dramatic increase in its recognition error followed by adaptation again. This situation can usually only be solved by adding more training data. Given our 112 test lines, we should go on and annotated more training lines (1000 at least). But as we already got an accuracy of above 95%, we have proven our assertion and stop here. Had you followed our earlier advice, you would have stopped already at 30,000 steps with the best model at 19,000 steps and an error of 6.2% and added more ground truth which is the sensible thing to do.

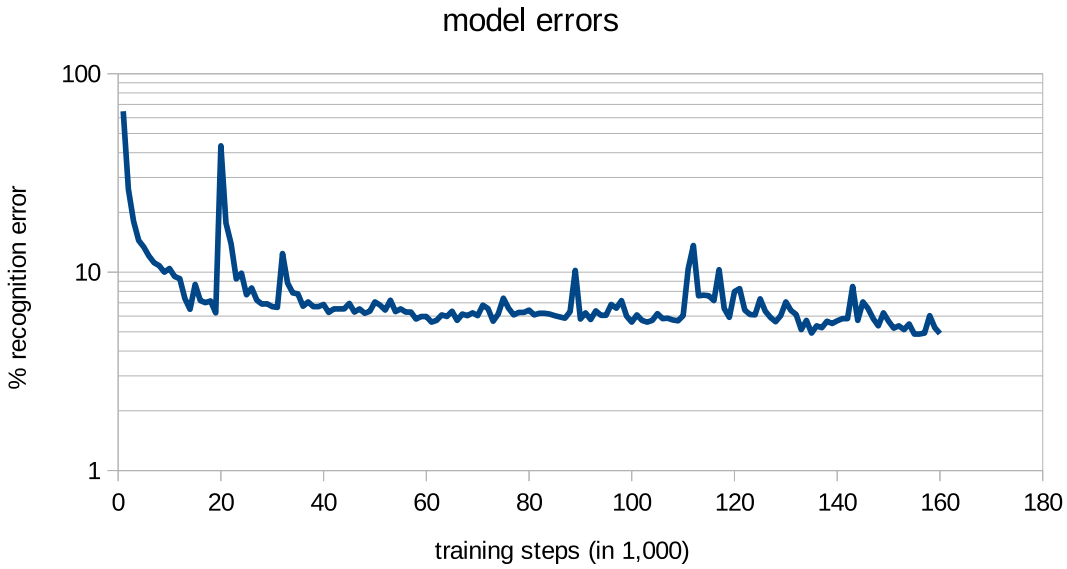


Figure 10: The evolution of recognition errors of different models (at 1000 training steps each) determined from the test data. The best model is the one at 155,000 steps with 4,87% error.

4.7 Recognize the book

With the best model identified, we can now run OCR over the book directory by

```
ocrocis predict --book
```

or

```
ocropus-rpred -m <bestmodel> book/*/*.bin.png
```

4.8 Extract the text

To get the text lines out of the book page subdirectories, we can do several things:

- shell script:
  `for i in book/*/*.gt.txt; do cat "$i" ; echo ""; done > gt.txt`
- `ocropus-gtedit text book/*/*.bin.png`
  (you will have to delete the line identifiers by `cut -b 17- correct.txt > gt.txt`)
- `ocropus-hocr book/*.bin.png`
  This gives you an hocr-file with line coordinates, which can directly be viewed in a browser.

Now you have your OCR text for the whole document and you are ready for postprocessing!

5 Demo data

To enable you to better follow this tutorial and to do your own experiments, we give you the set of our annotated training and testing data together with our best model as a file tutdemo.zip. These data may be downloaded at our Ocrocis repository[16].

6 Overview of the command sequence

Here is an overview of the command sequence from preprocessing page images to getting the OCR'ed text together with input and output of each step for both the Ocropy and Ocrocis case:

Stage	**Ocropy**	**Ocrocis**	**Input**	**Output**
binarization	ocropus-nlbin	ocrocis convert	<page>.png	<page>.bin.png
page segmentation	ocropus-gpageseg	ocrocis burst	<page>.bin.png	<line>.bin.png
annotation	ocropus-gtedit html	ocrocis next	<line>.bin.png	<line>.gt.txt
training	ocropus-rtrain	ocrocis train	<line>.bin.png + <line>.gt.txt	model
test recognition	ocropus-rpred	ocrocis predict	<line>.bin.png + model	<line>.txt
testing	ocropus-errs		<line>.txt + <line>.gt.txt	# errors, accuracy
	ocropus-econf		<line>.txt + <line>.gt.txt	confusion matrix
book recognition	ocropus-rpred	ocrocis predict --book	<line>.bin.png + model	<line>.txt
text extraction	ocropus-hocr		<line>.txt	<doc>.html
	ocropus-gtedit text		<line>.txt	<doc>.txt

Ocropus needs to recognize the test data before it can do testing; Ocrocis does recognition and testing in one step and (later) recognition of the complete book with a slightly different command.

If you install Ocropy natively and Ocrocis with `./install perl-only`, you will have both sets of commands at your disposal.

7 References

Breuel, T. (2014). A Python-based OCR package using recurrent neural networks. `https://github.com/tmbdev/ocropy`.

[16] `https://code.google.com/p/cistern/wiki/Ocrocis`

Breuel, T. M., Ul-Hasan, A., Al-Azawi, M. A., and Shafait, F. (2013). High-performance OCR for printed English and Fraktur using LSTM networks. In *2th International Conference on Document Analysis and Recognition (ICDAR), 2013*, pages 683--687. IEEE.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735--1780.

Rydberg-Cox, J. A. (2009). Digitizing latin incunabula: Challenges, methods, and possibilities. *Digital Humanities Quarterly*, 3(1).

Springmann, U. and Kaumanns, D. (2015). Ocrocis -- a high accuracy OCR method to convert early printings into digital text. `http://cistern.cis.lmu.de/ocrocis/`.

Springmann, U., Najock, D., Morgenroth, H., Schmid, H., Gotscharek, A., and Fink, F. (2014). OCR of historical printings of Latin texts: problems, prospects, progress. In *Proceedings of the First International Conference on Digital Access to Textual Cultural Heritage*, DATeCH '14, pages 57--61, New York, NY, USA. ACM.

Vobl, T., Gotscharek, A., Reffle, U., Ringlstetter, C., and Schulz, K. U. (2014). PoCoTo - an Open Source System for Efficient Interactive Postcorrection of OCRed Historical Texts. In *Proceedings of the First International Conference on Digital Access to Textual Cultural Heritage*, DATeCH '14, pages 57--61, New York, NY, USA. ACM.