OCR Tutorial

Niko Partanen

2020-01-10

Contents

Chapter 1

Prerequisites

This tutorial is about current OCR technologies, and Niko Partanen's own practices when building new OCR models. These are not necessarily *best practices*, but certainly on a road toward that, and all feedback that helps improving the methodology presented here is most welcome.

All examples are taken from the National Library of Finland's Fenno-Ugrica collection.

This workshop assumes that the participants are comfortable with installing software from the source, and are used to working on command line. All examples are supposed to be complete and repeatable, but this is very much ongoing work.

Chapter 2

Introduction

This workshop will be built around the materials presented by Partanen and Rießler (2019). That study contained few experiments with training Ocropy models for various languages for which Unified Northern Alphabet was used. The result was that bootstrapping of a new OCR system is extremely fast when the domain is very narrow, i.e. font is known, and there are enough examples of different characters. The study also showed that the system performed comparably well in monolingual and multilingual conditions, indicating that having an OCR system for this writing system, and not one for each language using it, should be a reasonable goal for further work.

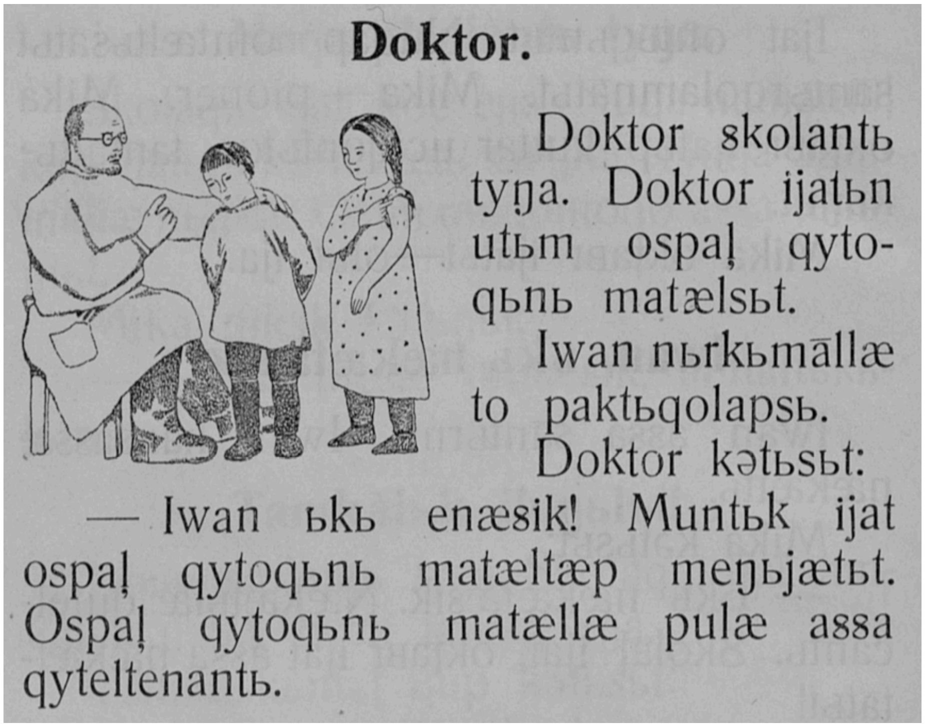
In Chapter 3 I go through the main tools we can currently use in OCR. Handwritten text recognition, HTR, is somewhat beyond the capabilities of this software, and in that domain Transkribus system has a very strong position. Thereby that is discussed separately in section 7.

Since text recognition is closely connected to layout analysis, that is discussed in 4. I have not personally explored much the field of layout detection, although there certainly is a lot to gain in that front. I would even say that most of our current technical problems relate to layout analysis, more than text recognition itself.

In order to start training the models, we need to acquire or create Ground Truth datasets. These are discussed, through various examples, in Chapter 5.

In the Chapter 6 I finally get into actual model training, and in the Chapter 8 I provide examples and code for using the models we trained. As will be shown, proofreading Ground Truth and training the model creates a very fast and rewarding loop, where generating more data that improves the model gets increasingly faster.

Comments, corrections and additions are more than welcome, either by email (niko.partanen@helsinki.fi), or through GitHub Issues in the project repository.



Doktor.

Doktor ꞩkolantь tyŋa. Doktor ijatьn utьm ospaļ qytoqьnь matælsьt.

Iwan nьrkьmāllæ to paktьqolapsь.

Doktor kətьsьt:

— Iwan ьkь enæꞩik! Muntьk ijat ospaļ qytoqьnь matæltæp meŋьjætьt. Ospaļ qytoqьnь matællæ pulæ aꞩꞩa qyteltenantь.

Figure 2.1: Example from a book in Selkup

Chapter 3

Tools

The OCR tools discussed here are:

- Ocropy
- Calamari
- Tesseract

I will not discuss Abbyy FineReader at length, besides saying that even though it fits well to casual use, I think it is not really the best choice for creating scientific datasets that are discussed here.

In the end of the workshop also Transkribus is discussed. It is a very exciting project that has derived impressive results on handwritten text recognition. It is very recommandable to take a look into it.

Chapter 4

Layout analysis

It is important to understand that OCR systems áre primarily about working with the text content itself, traditionally at character level, but at the moment line is the normal minimum unit. The system takes a line and returns the predicted text, but it is an entirely different question how we retrieve these lines.

I would even say that OCR itself is much more a solved problem than layout detection. If we have nice lines getting out from them a relatively high accuracy text is very easy. But with complex documents a lots of work is still left in finding all the text areas, lines within them, and how all those connect together into nice running text.

Of course the argument can also be made that for variety of purposes it is not even crucial to have the lines and sections connect to one another perfectly. This could be the case, for example, in topic detection tasks etc. It is important not to fall into trap where we think that as something doesn't work perfectly we cannot use it.

Indeed, as the OCR model training does not care about anything beyond an individual line, it is not of any importance there how the lines connect to one another and whether the texts are complete.

When we use the OCR model, see section Doing OCR, it is necessary that the lines we get from the line segmentation tool we use are similar to the lines we did the model training with. Thereby it is important to think about the whole pipeline before getting too far.

In my experience the Tesseract's layout analysis tool is very good, and often gives a very sensible result. Also Transkribus has some excellent layout detection capabilities. So running the layout detection in these programs, and extracting the line bounding boxes from the XML returned is a good option. In section

Doing OCR I also have an example of how to run the whole pipeline in Python.

Chapter 5

Ground Truth creation

The main challenge in creating Ground Truth is that we need a comfortable environment for doing the proofreading, with safety that we know the software used will save the edited file back without any structural changes.

Lots of programmers have got the idea to build their own proofreading environment. In practice this is very complicated. Tools that allow editing beyond individual lines usually break something in the XML structure.

In principle proofreading tools / environment can be extremely simple, and this is illustrated by Ocropy in the next section.

5.1 Examples

We have training data in folders `data/batch_1_orig`, `data/batch_2_orig`, `data/batch_3_orig`, `data/batch_4_orig`, `data/batch_5_orig` and `data/batch_6_orig`. Each batch has 2 pages.

We are using Ocropy in this section, so please install Ocropy.

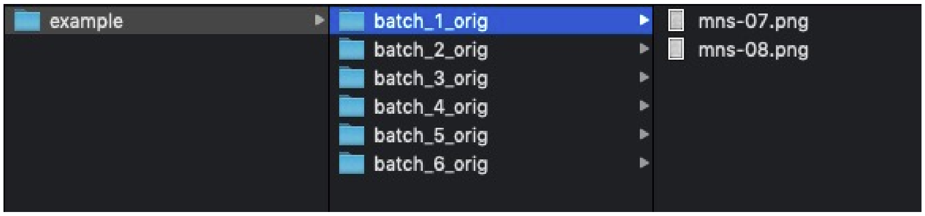
This is the starting position:

```
ocropus-nlbin ./example/batch_1_orig/*.png -o ./example/batch_1
```

This tool can be used to create segmented lines. The system stores somehow information about the line locations, but moving the files around is apparently not a good idea.

```
ocropus-gpageseg ./example/batch_1/*.bin.png
```

Now, let's pretend we are without any OCR system for this script. Then we would need to add start writing from the scratch. This could be started with the following command:

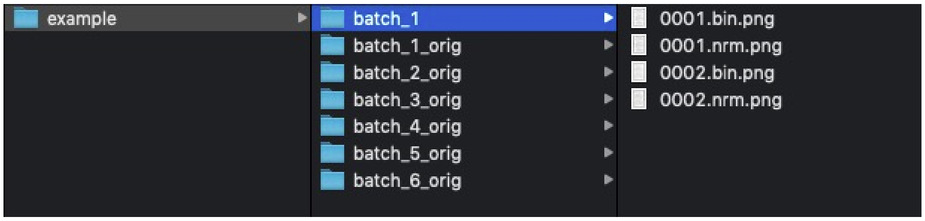


example

batch_1_orig
batch_2_orig
batch_3_orig
batch_4_orig
batch_5_orig
batch_6_orig

mns-07.png
mns-08.png

Figure 5.1: Just scanned images

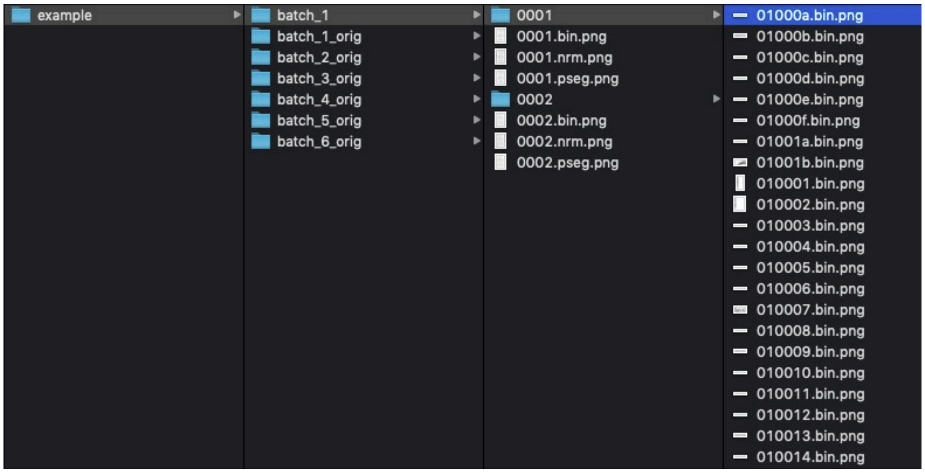


example

batch_1
batch_1_orig
batch_2_orig
batch_3_orig
batch_4_orig
batch_5_orig
batch_6_orig

0001.bin.png
0001.nrm.png
0002.bin.png
0002.nrm.png

Figure 5.2: Binarized pages



example

batch_1
batch_1_orig
batch_2_orig
batch_3_orig
batch_4_orig
batch_5_orig
batch_6_orig

0001
0001.bin.png
0001.nrm.png
0001.pseg.png
0002
0002.bin.png
0002.nrm.png
0002.pseg.png

01000a.bin.png
01000b.bin.png
01000c.bin.png
01000d.bin.png
01000e.bin.png
01000f.bin.png
01001a.bin.png
01001b.bin.png
010001.bin.png
010002.bin.png
010003.bin.png
010004.bin.png
010005.bin.png
010006.bin.png
010007.bin.png
010008.bin.png
010009.bin.png
010010.bin.png
010011.bin.png
010012.bin.png
010013.bin.png
010014.bin.png

Figure 5.3: Segmented lines

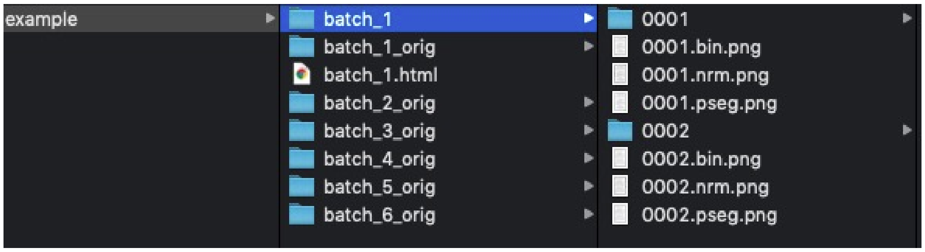


Figure 5.4: HTML file appears

```
ocropus-gtedit html ./example/batch_1/*/*.png -o ./example/batch_1.html
```

This outputs an HTML file:

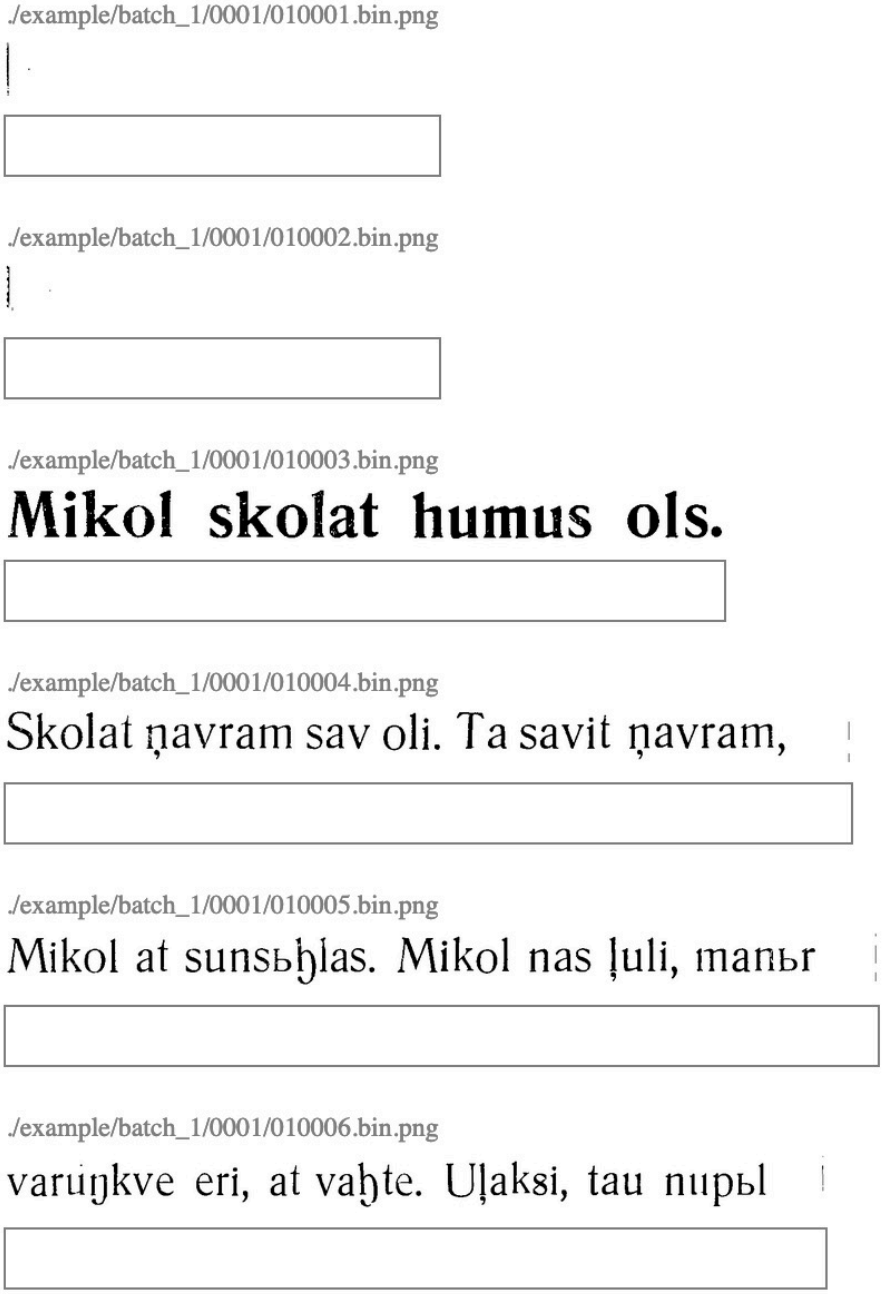
However, as we have a model from an earlier work, let’s use it for now.

```
ocropus-rpred -Q 4 -m ../unified-northern-alphabet-ocr/models/ocropy/mixed_model.pyrnn.gz ./exampl
```

As we already see from output, the result is sensible:

```
INFO:  ./example/batch_1/0001/010003.bin.png:Mikol skolat humus ols.
INFO:   ./example/batch_1/0001/010007.bin.png:lavs:
INFO:  ./example/batch_1/0001/010004.bin.png:Skolat ņavram sav oli. Ta savit ņavram,
INFO:   ./example/batch_1/0001/010008.bin.png:- Ja! t -unten
INFO:  ./example/batch_1/0001/010006.bin.png:varuŋkve eri, at va te. Uļak i, tau nup l
INFO:   ./example/batch_1/0001/010009.bin.png:tuŋkve patev.
INFO:  ./example/batch_1/0001/010005.bin.png:Mikol at suns las. Mikol nas ļuli, man r
INFO:   ./example/batch_1/0001/01000b.bin.png:Sistam  olen.
INFO:   ./example/batch_1/0001/01000a.bin.png:Mikol, hani tah-
INFO:  ./example/batch_1/0001/01000f.bin.png:-  emen luvtuŋkve eri.
INFO:   ./example/batch_1/0001/01000e.bin.png:Hani tan hum lavs:
INFO:  ./example/batch_1/0001/01000c.bin.png: emen skolan joht s. Skolat ņavram t
INFO:  ./example/batch_1/0001/010011.bin.png:hurataves. Puŋkane luvtuŋkve haņ ulaves.
INFO:  ./example/batch_1/0001/01000d.bin.png:sistam  ole t.  emen paŋk ŋ  joht s.
```

These lines are saved with the images.



./example/batch_1/0001/010001.bin.png

./example/batch_1/0001/010002.bin.png

./example/batch_1/0001/010003.bin.png

Mikol skolat humus ols.

./example/batch_1/0001/010004.bin.png

Skolat ņavram sav oli. Ta savit ņavram,

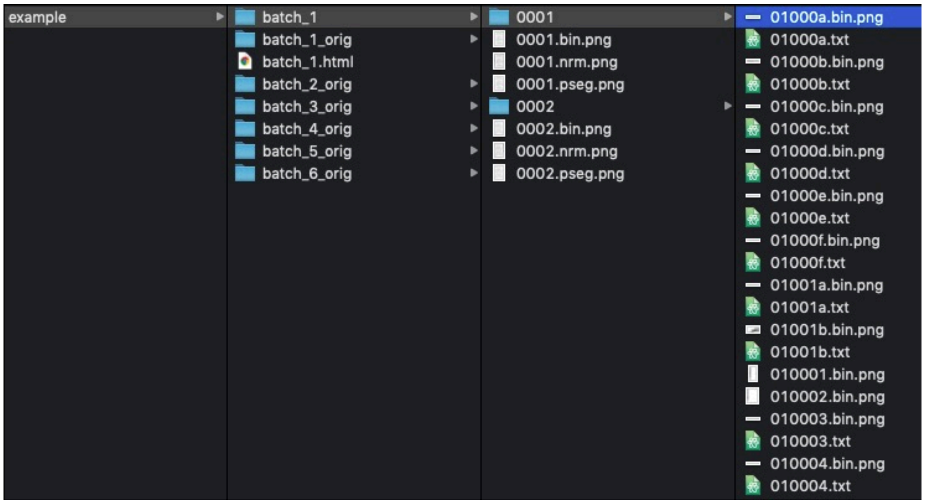
./example/batch_1/0001/010005.bin.png

Mikol at sunsьɧlas. Mikol nas ļuli, manьr

./example/batch_1/0001/010006.bin.png

varuŋkve eri, at vaɧte. Uļaksi, tau nupьl

Figure 5.5: Empty HTML from Ocropy (**has to be edited in Firefox**)



example

batch_1
batch_1_orig
batch_1.html
batch_2_orig
batch_3_orig
batch_4_orig
batch_5_orig
batch_6_orig

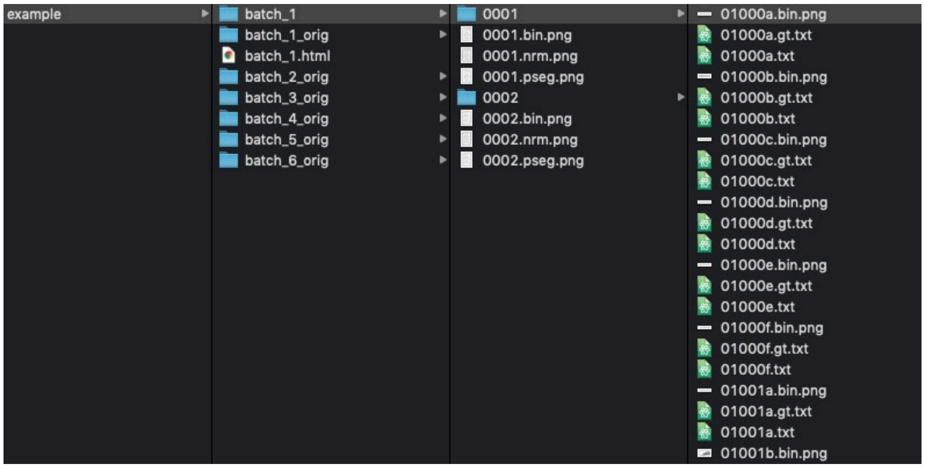
0001
0001.bin.png
0001.nrm.png
0001.pseg.png
0002
0002.bin.png
0002.nrm.png
0002.pseg.png

01000a.bin.png
01000a.txt
01000b.bin.png
01000b.txt
01000c.bin.png
01000c.txt
01000d.bin.png
01000d.txt
01000e.bin.png
01000e.txt
01000f.bin.png
01000f.txt
01001a.bin.png
01001a.txt
01001b.bin.png
01001b.txt
010001.bin.png
010002.bin.png
010003.bin.png
010003.txt
010004.bin.png
010004.txt

If we edit the HTML, and the save the file, the edited lines can be saved. This happens with:

```
ocropus-gtedit extract -O ./example/batch_1.html
```

This saves the edited lines with extension `.gt.txt`.



example

batch_1
batch_1_orig
batch_1.html
batch_2_orig
batch_3_orig
batch_4_orig
batch_5_orig
batch_6_orig

0001
0001.bin.png
0001.nrm.png
0001.pseg.png
0002
0002.bin.png
0002.nrm.png
0002.pseg.png

01000a.bin.png
01000a.gt.txt
01000a.txt
01000b.bin.png
01000b.gt.txt
01000b.txt
01000c.bin.png
01000c.gt.txt
01000c.txt
01000d.bin.png
01000d.gt.txt
01000d.txt
01000e.bin.png
01000e.gt.txt
01000e.txt
01000f.bin.png
01000f.gt.txt
01000f.txt
01001a.bin.png
01001a.gt.txt
01001a.txt
01001b.bin.png

In this point we can do:

```
cat example/batch_1/**/*gt.txt | wc -l
> 48
```

More than enough! Let’s go onward!

5.2 Summary

- `.txt` files are collected to the HTML
- Proofread lines are exported from HTML
- The wanted outcome is pairs of `.bin.png` and `.gt.txt` files
- These can be used when training the models

The idea is that you go now to section @ref(training} about model training, train the first model with what we have, and then the workflow described here is applied to `batch_2`.

Chapter 6

OCR Model training

Once we have a ground truth dataset, we can start the model training. Most of the time this is relatively simple process, and we just run the training command and tell it where the training files are, and how we want to name the model. The system we use takes care of image preprocessing, which will then be applied also when the model is used.

There are few things we usually should keep in mind while training the model:

- Documenting which training files are used
  - Use Git commit hash in model name?
- Checkpoint frequency
  - Too high eats your harddisk space
  - Too low is maybe difficult to monitor
  - As the model accuracy can go up and down pretty wildly, it is important to notice when it is in the period of confusion, and use the model before or after that
- If you have lots of data (10,000–100,000 lines), then let it train for as long as you can
  - Same if you want to release something more publicly
  - **Hot take:** With small amount of data nothing significant happens after first few hours. If you are in Ground Truth creation loop, iterating the new models fast is a good idea.

6.1 Training Calamari

First, install Calamari, something like:

```
pip install calamari_ocr
pip install tensorflow
```

Or:

```
git clone https://github.com/Calamari-OCR/calamari
conda env create -f environment_master_cpu.yml
```

Calamari model can be trained with a following command:

```
calamari-train --files train/*png --output_model_prefix komi-test- --output_dir models/ -
```

This would save the model into path `models/komi-test-000200....` New model would be saved every 200 training steps. The models can be fairly large.

With our demo dataset there is the problem that Ocropy and Calamari prefer bit different filenames, so that Calamari doesn't want `.bin.png` ending. So let's collect the files we have into one folder, that is a nice practice anyway. I often do it with Bash like this:

```
mkdir train

for gt_line in `ls ./example/*/*/*gt.txt`

do

  bin_png=$(echo $gt_line | sed 's/gt.txt/bin.png/g')
  png=$(echo $gt_line | sed 's/gt.txt/png/g')

  cp $gt_line ./train/"${gt_line##*/}"
  cp $bin_png ./train/"${png##*/}"

done
```

So we just find all Ground Truth lines, and rename + copy them into directory `train`. We could, in this point, split it into `train` and `test`, but as the data is less than 50 lines, this is maybe a bit early. It is a good idea to use tools like SciKit Learn's `train_test_split` in Python, but in this point it isn't that complicated what we are doing.

In the case of our demo dataset, the command would be:

```
calamari-train --files train/*png --output_model_prefix una-batch_1- --output_dir models/

> Resolving input files
> Found 26 files in the dataset
> Preloading dataset type DataSetMode.TRAIN with size 26
> Preloading data: 100%|                                            | 26/26 [00:03<00:0
> Computing codec: 100%|                                           | 26/26 [00:00<00:00
> CODEC: ['', ' ', '!', ',', '-', '.', ':', 'A', 'H', 'M', 'O', 'P', 'S', 'T', 'U', 'a', 'e',
```

Then the training starts, and what we get it something like:

```
#00000100: loss=103.64814411 ler=1.00000000 dt=1.01209140s
```

```
  PRED: '*,'
  TRUE: '*Ņavram t tot joŋhes t. Uļak i lavs:,'
Storing checkpoint to '/Users/niko/github/ocr-tutorial/data/models/una-batch_1-00000200.ckpt'
#00000200: loss=88.01333866 ler=1.00000000 dt=1.13691821s
  PRED: '*,'
  TRUE: '*Ņavram t tot joŋhes t. Uļak i lavs:,'
#00000300: loss=84.42394615 ler=1.00000000 dt=12.90764643s
  PRED: '*,'
  TRUE: '*vos ols.,'
Storing checkpoint to '/Users/niko/github/ocr-tutorial/data/models/una-batch_1-00000400.ckpt'
#00000400: loss=66.31870094 ler=0.88571429 dt=2.39815145s
  PRED: '*p l  , l jte .,'
  TRUE: '*tau nup l lavs, a m l jemte n. Ok-,'
#00000500: loss=33.71355089 ler=0.78357143 dt=0.87875572s
  PRED: '*aki kee up l las,'
  TRUE: '*Uļak i  emen nup l lavs:,'
Storing checkpoint to '/Users/niko/github/ocr-tutorial/data/models/una-batch_1-00000600.ckpt'
#00000600: loss=17.30537902 ler=0.68943453 dt=0.98521040s
  PRED: '*use avra t jŋhe t. kol at,'
  TRUE: '*Pusen ņavram t joŋhe t. Mikol at,'
WARNING:tensorflow:Method (on_train_batch_end) is slow compared to the batch update (0.302853). Chec
#00000700: loss=9.67660141 ler=0.59846268 dt=0.94566187s
  PRED: '*Uļak i pioner  oli. au okţarat sart,'
  TRUE: '*Uļak i pioner  oli. Tau okţa rat sart,'
Storing checkpoint to '/Users/niko/github/ocr-tutorial/data/models/una-batch_1-00000800.ckpt'
#00000800: loss=6.00772715 ler=0.52365484 dt=0.86985343s
  PRED: '*p ri i  oli.,'
  TRUE: '*p ri i  oli.,'
#00000900: loss=4.19488548 ler=0.47200692 dt=0.79613184s
  PRED: '*Hohsan ht-lajen, joŋhuŋkve t -,'
  TRUE: '*- Hohsan hot-lajen, joŋhuŋkve t -,'
Storing checkpoint to '/Users/niko/github/ocr-tutorial/data/models/una-batch_1-00001000.ckpt'
#00001000: loss=2.85367827 ler=0.42480623 dt=1.01420514s
  PRED: '*harte t!,'
  TRUE: '*harte t!,'
#00001100: loss=2.35952931 ler=0.39628849 dt=1.01120380s
  PRED: '* opitel.,'
  TRUE: '* opiteln.,'
Storing checkpoint to '/Users/niko/github/ocr-tutorial/data/models/una-batch_1-00001200.ckpt'
#00001200: loss=1.79808081 ler=0.36326445 dt=0.93737005s
  PRED: '*jajen.,'
  TRUE: '*jajen.,'
#00001300: loss=1.41390861 ler=0.33532103 dt=1.03504477s
  PRED: '*Ņavram ten Sano joŋhuŋkve untuves.,'
  TRUE: '*Ņavram ten Sano joŋhuŋkve untuves.,'
Storing checkpoint to '/Users/niko/github/ocr-tutorial/data/models/una-batch_1-00001400.ckpt'
```

```
#00001400: loss=1.39289295 ler=0.31136953 dt=1.04986981s
  PRED: '*Ņavram t Uḷak i huntle t.,'
  TRUE: '*Ņavram t Uḷak i huntle t.,'
#00001500: loss=0.89948882 ler=0.29061156 dt=1.21357172s
  PRED: '*vos ols.,'
  TRUE: '*vos ols.,'
Storing checkpoint to '/Users/niko/github/ocr-tutorial/data/models/una-batch_1-00001600
#00001600: loss=0.80192822 ler=0.27423405 dt=1.04396018s
  PRED: '*Semel part hot-os eln. Hasne rakt,'
  TRUE: '*- Semel part hot-os eln. Hasne rakt,'
#00001700: loss=0.76085947 ler=0.26146398 dt=1.28815659s
  PRED: '*tau nup l lavs,a mm l jemte n. Ok-,'
  TRUE: '*tau nup l lavs, a m l jemte n. Ok-,'
```

Of course the system is only repeating the same small number of lines, so it eventually just learns them.

After having it run 2000 steps we stop, and let's test that model:

```
calamari-predict --checkpoint ./models/una-batch_1-00002000.ckpt.json --files ./mixed/*.
```

Then we test it:

```
calamari-eval --gt ./mixed/*.gt.txt
```

What we get is:

```
Resolving files
Loading GT: 100%|                                              | 800/800 [00:03<00:00, 2
Loading Prediction: 100%|                                          | 800/800 [00:02<00:0
Evaluation: 100%|                                              | 800/800 [00:01<00:00, 5
Evaluation result
=================

Got mean normalized label error rate of 24.43% (5495 errs, 22494 total chars, 5585 sync errs
GT       PRED     COUNT    PERCENT
{ə}      {a}           237      4.24%
{d}      {ol}          195      6.98%
{m}      {n}           153      2.74%
{n}      {}            133      2.38%
{m}      {nn}           98      3.51%
{w}      {v}            90      1.61%
{æ}      {ae}           85      3.04%
{q}      {op}           82      2.94%
{ḷ}      {}             80      1.43%
{c}      {o}            73      1.31%
The remaining but hidden errors make up 69.81%
```

Error rate of 24.43% means that every fourth character needs to be fixed, but that is already much better than what we had in the beginning.

6.2 Training Tesseract

Training Tesseract can be a bit intimidating process. In last years many improvements have been done, and at the moment training is possible on both Linux and Mac. There is, additionally, tessmake project that very conveniently wraps the training process into a Makefile.

```
make training MODEL_NAME=komi-test GROUND_TRUTH_DIR=train/
```

If you want to change the parameters, play around with the Makefile.

This gives a very good Tesseract model if you have enough data. The models, to be foundable for Tesseract, have to be in so called tessdata directory. This can be also specified when using Tesseract by specifying `--tessdata-dir`.

Chapter 7

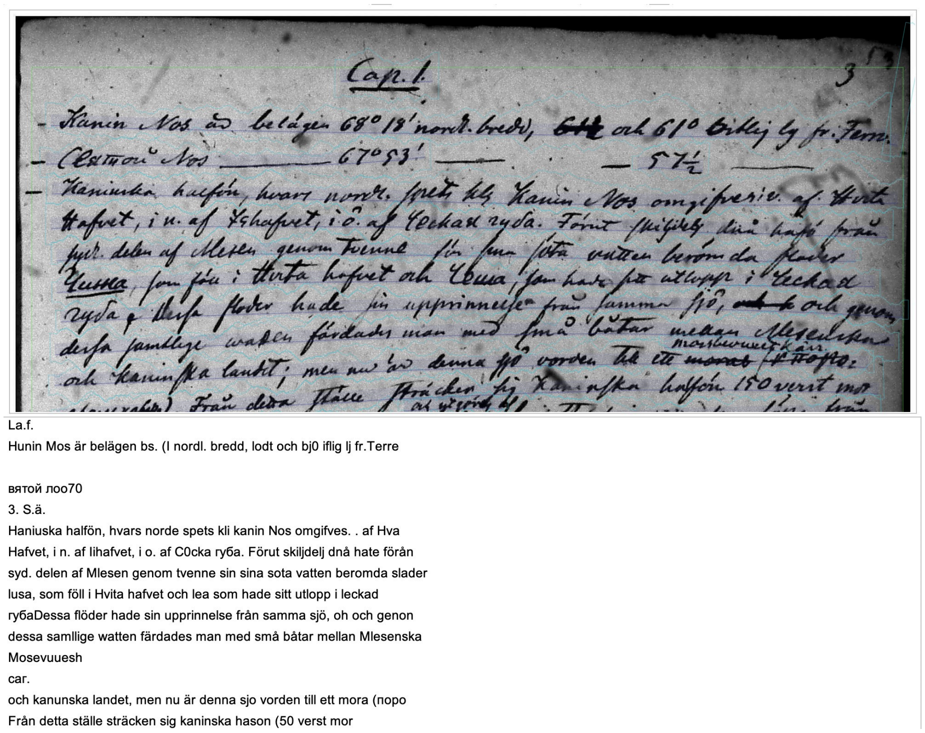
Handwritten text recognition

At the moment Transkribus project offers the best platform for handwritten text recognition, as well as for lots of OCR related tasks. Training also OCR models with the Transkribus system seems to work extremely, even ridiculously, well, so that is certainly worth playing around.

One example of how collections edited in Transkribus can be made available can be seen in this interface for National Archives of Finland's data.

Experiments on Finnish dialectal transcription have also been very promising.

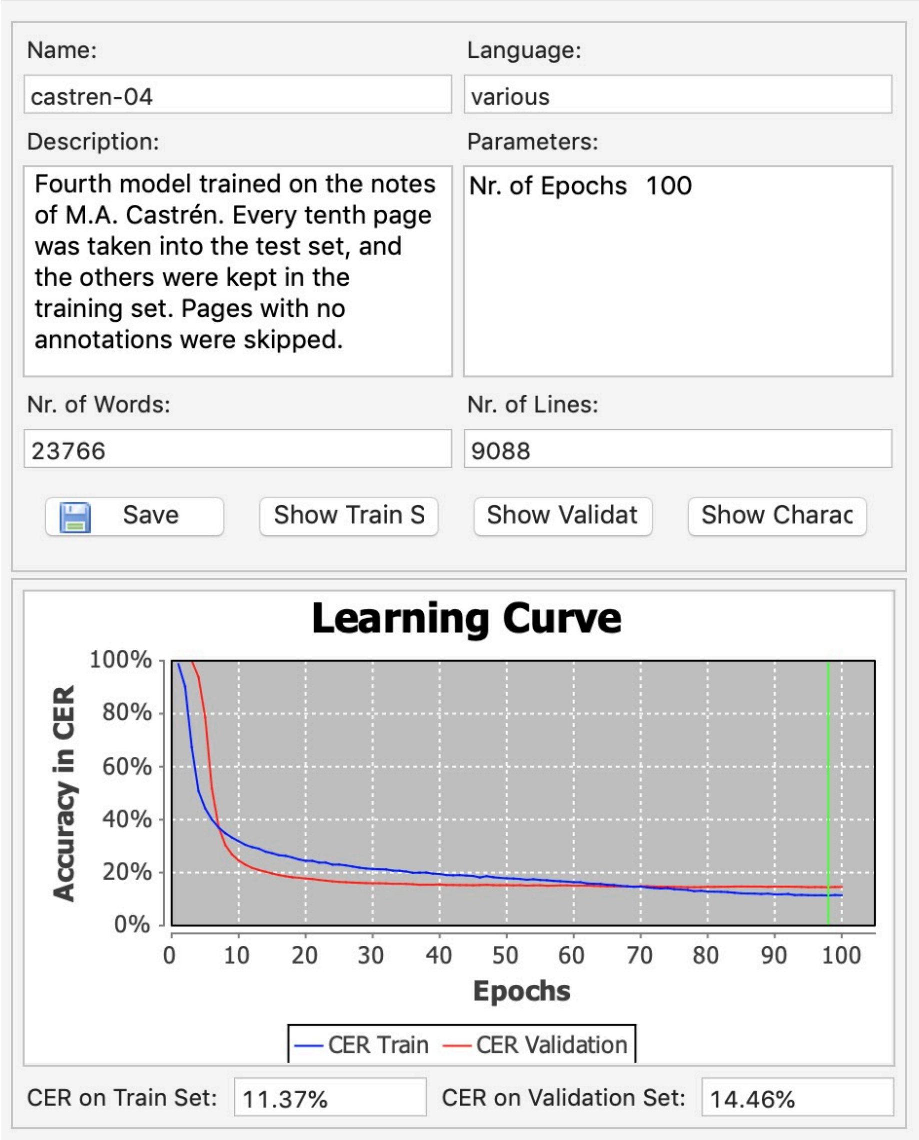
There is also an extremely well done online editing interface.



La.f.
Hunin Mos är belägen bs. (I nordl. bredd, lodt och bj0 iflig lj fr.Terre

вятой лоо70
3. S.ä.
Haniuska halfön, hvars norde spets kli kanin Nos omgifves. . af Hva
Hafvet, i n. af lihafvet, i o. af C0cka губа. Förut skiljdelj dnå hate förån
syd. delen af Mlesen genom tvenne sin sina sota vatten beromda slader
lusa, som föll i Hvita hafvet och lea som hade sitt utlopp i leckad
рыбаDessa flöder hade sin upprinnelse från samma sjö, oh och genon
dessa samllige watten färdades man med små båtar mellan Mlesenska
Mosevuuesh
car.
och kanunska landet, men nu är denna sjo vorden till ett mora (nopo
Från detta ställe sträcken sig kaninska hason (50 verst mor

Figure 7.1: Example from Castrén's Swedish



Name: castren-04

Language: various

Description: Fourth model trained on the notes of M.A. Castrén. Every tenth page was taken into the test set, and the others were kept in the training set. Pages with no annotations were skipped.

Parameters: Nr. of Epochs 100

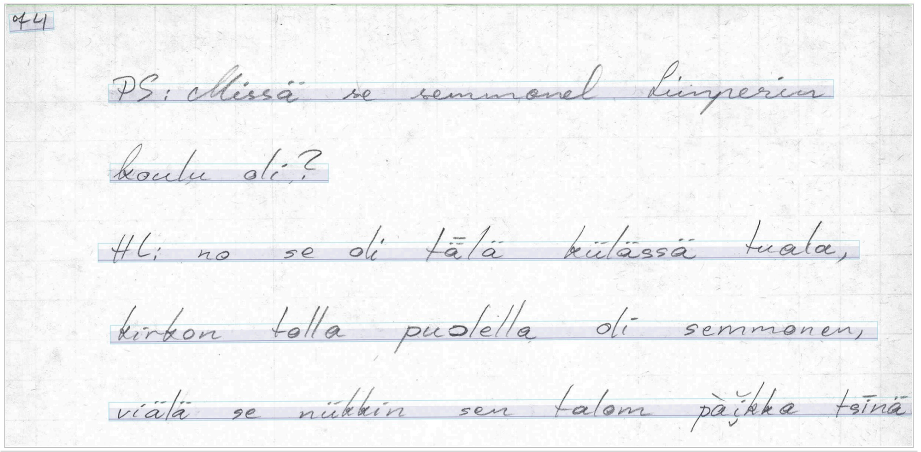
Nr. of Words: 23766

Nr. of Lines: 9088

CER on Train Set: 11.37%

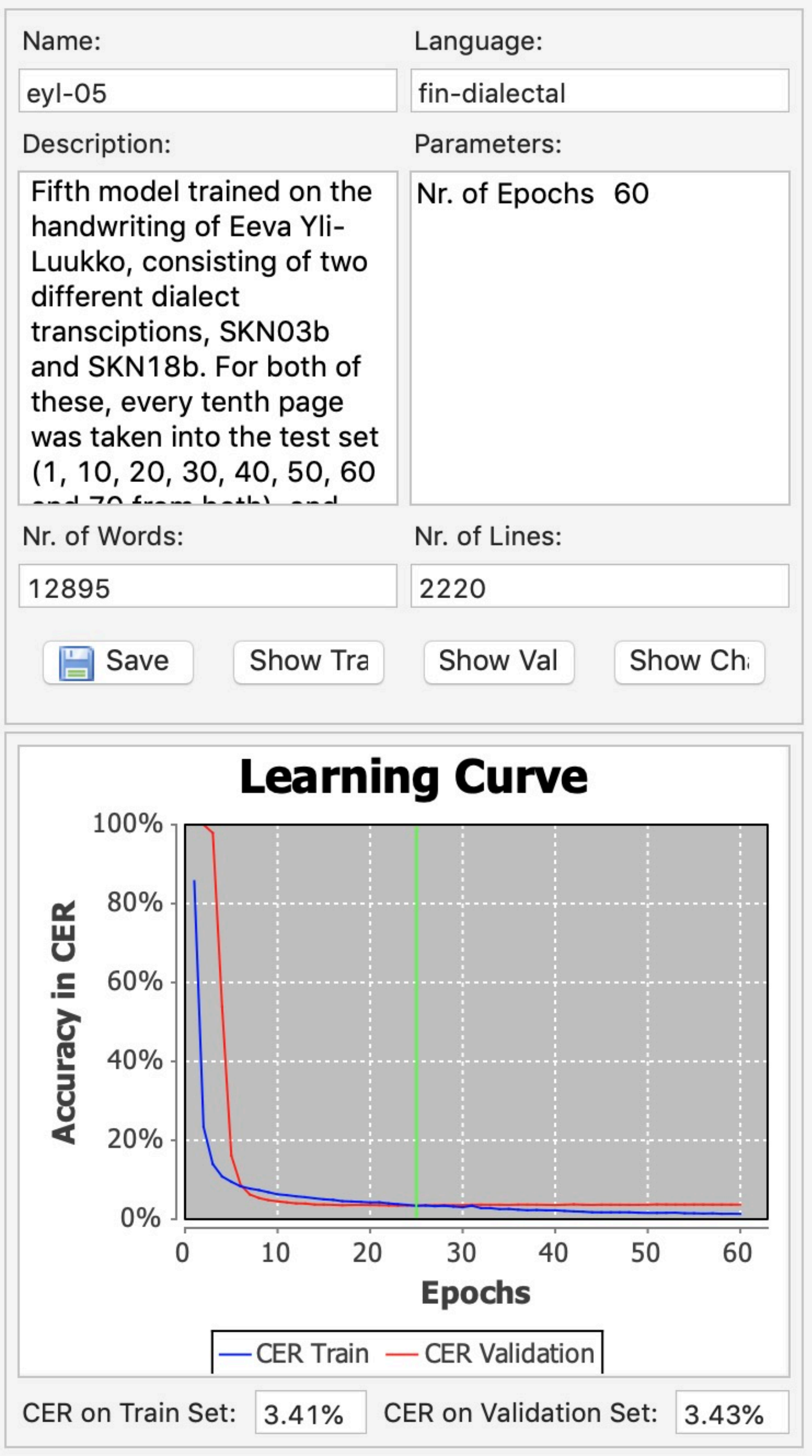
CER on Validation Set: 14.46%

Figure 7.2: Castrén's currently best model



74
PS: Missä se semmonel Limperin
koulu oli?
HL: no se oli tå̄llä külässä tuala,
kirkon tolla puɔlella oli semmonen,
viälä se nükkin sen talom paikka tsīnä
on semmonen, koulumestarik sanottīn
sitä, se oli sunti₍ammena samalla
kirkossa ja se, opetti lapsi₍a siäll‿
oli sitte peŋkit oekēj ja semmone
huɔne, ja, āpiset sitte oli, meitillä,
josta tavvāmāj ja āpiset, -- ne äkko¬
set‿opetettī. nī, se oli limperiŋ

Figure 7.3: Eeva Yli-Luukko's transcription (© Institute for the Languages of Finland



Name: eyl-05

Language: fin-dialectal

Description: Fifth model trained on the handwriting of Eeva Yli-Luukko, consisting of two different dialect transciptions, SKN03b and SKN18b. For both of these, every tenth page was taken into the test set (1, 10, 20, 30, 40, 50, 60 and 70 from both), and

Parameters: Nr. of Epochs 60

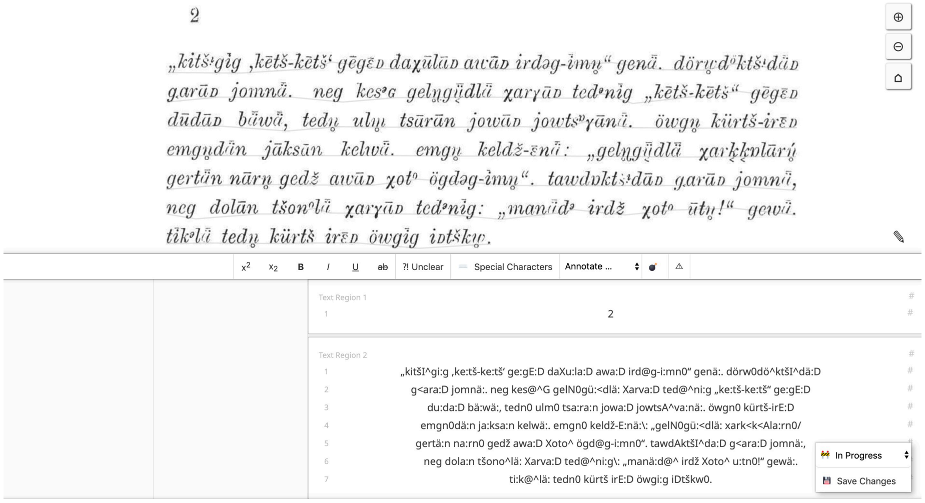
Nr. of Words: 12895

Nr. of Lines: 2220

CER on Train Set: 3.41%

CER on Validation Set: 3.43%

Figure 7.4: Best Finnish dialect transcription model



Text Region 1

2

Text Region 2

„kitšI^gi:g ‚ke:tš-ke:tš‘ ge:gE:D daXu:la:D awa:D ird@g-i:mn0“ genä:. dörw0dö^ktšI^dä:D
g<ara:D jomnä:. neg kes@^G gelN0gü:<dlä: Xarva:D ted@^ni:g „ke:tš-ke:tš“ ge:gE:D
du:da:D bä:wä:, tedn0 ulm0 tsa:ra:n jowa:D jowtsA^va:nä:. öwgn0 kürtš-irE:D
emgn0dä:n ja:ksa:n kelwä:. emgn0 keldž-E:nä:\: „gelN0gü:<dlä: xark<k<Ala:rn0/
gertä:n na:rn0 gedž awa:D Xoto^ ögd@g-i:mn0“. tawdAktšI^da:D g<ara:D jomnä:,
neg dola:n tšono^lä: Xarva:D ted@^ni:g\: „manä:d@^ irdž Xoto^ u:tn0!“ gewä:.
ti:k@^lä: tedn0 kürtš irE:D öwgi:g iDtškw0.

Figure 7.5: Transkribus online interface

Chapter 8

Using OCR models

In this section we go through with practical examples how OCR models can be used.chr

8.1 Using Tesseract

Tesseract can be used as:

```
tesseract image.jpg image -l kpv alto
```

This works when we have language model called `kpv` in so-called Tessdata directory. Directory can also be specified as an argument. To get XML output, one can have `alto` or `hocr` in the end of the command.

R has Tesseract bindings that work well.

Pytesseract is a good option for Python. I had some problems while using different language models on it, but for layout analysis it was really convenient.

8.2 Using Calamari

Calamari can be used from command line, so that it is given the model and location of line images.

```
calamari-predict --checkpoint path_to_model.ckpt --files your_images.*.png
```

It can also be used directly from Python.

```
from calamari_ocr.ocr import Predictor, create_dataset, DataSetType, DataSetMode
import tensorflow as tf
```

```
predictor = Predictor('./models/komi-latin-bin00004500.ckpt')

for prediction in predictor.predict_raw(images = line_list, progress_bar=False):
    print(prediction.sentence)
```

More complete example could look like this:

```
def read_alto(alto_file, version = 2):

    tree = ET.parse(alto_file)
    root = tree.getroot()

   xmlns = {'alto': '{http://www.loc.gov/standards/alto/ns-v' + str(version) + '#}'}

    data = []

   unit = root.find('.//{alto}MeasurementUnit'.format(**xmlns)).text

   max_height = root.find('.//{alto}PrintSpace'.format(**xmlns)).get('HEIGHT')
   max_width = root.find('.//{alto}PrintSpace'.format(**xmlns)).get('WIDTH')

   for block in root.iterfind('.//{alto}TextBlock'.format(**xmlns)):

        block_id = block.get('ID')

     for line in block.iterfind('.//{alto}TextLine'.format(**xmlns)):

            content = {}

            content["block_id"] = block_id
            content["height"] = line.get('HEIGHT')
            content["width"] = line.get('WIDTH')
            content["top"] = line.get('VPOS')
            content["left"] = line.get('HPOS')
            content["unit"] = unit
            content["max_height"] = max_height
            content["max_width"] = max_width

            line_strings = []
        for string in line.findall('./{alto}String'.format(**xmlns)):
                line_strings.append(string.get('CONTENT'))
            content["text"] = ' '.join(line_strings)

            data.append(content)

    return(data)
```

```
def extract_line_array(pil_image, height, width, top, left):

   cropped_example = pil_image.crop((int(left), int(top), int(left) + int(width), int(top) + int(hei

    cropped_example_bw = cropped_example.convert("L")

    image_array = numpy.array(cropped_example_bw)

    image_array_binarized = binarize_array(image_array, 150)

   #image_array_binarized = Binarizer(image_array, threshold = 150, copy=False)

    return(image_array_binarized)

def predict_page(page_image, alto, predictor):

    alto_content = read_alto(alto)

    pil_image = Image.open(page_image)

    line_list = []

    for line in alto_content:
     extracted_line = extract_line_array(pil_image, line['height'], line['width'], line['top'], li
        line_list.append(extracted_line)

    pred_list = []

   for prediction in predictor.predict_raw(images = line_list, progress_bar=False):

        pred_list.append(prediction)

    return(pred_list)
```

Calamari returns very nicely list of confidences for each character. This info can be used, for example, to distinguish areas where the model is most confident, which seems to indicate correct script.

The model used here was quite horribly bad, but still good enough for this.

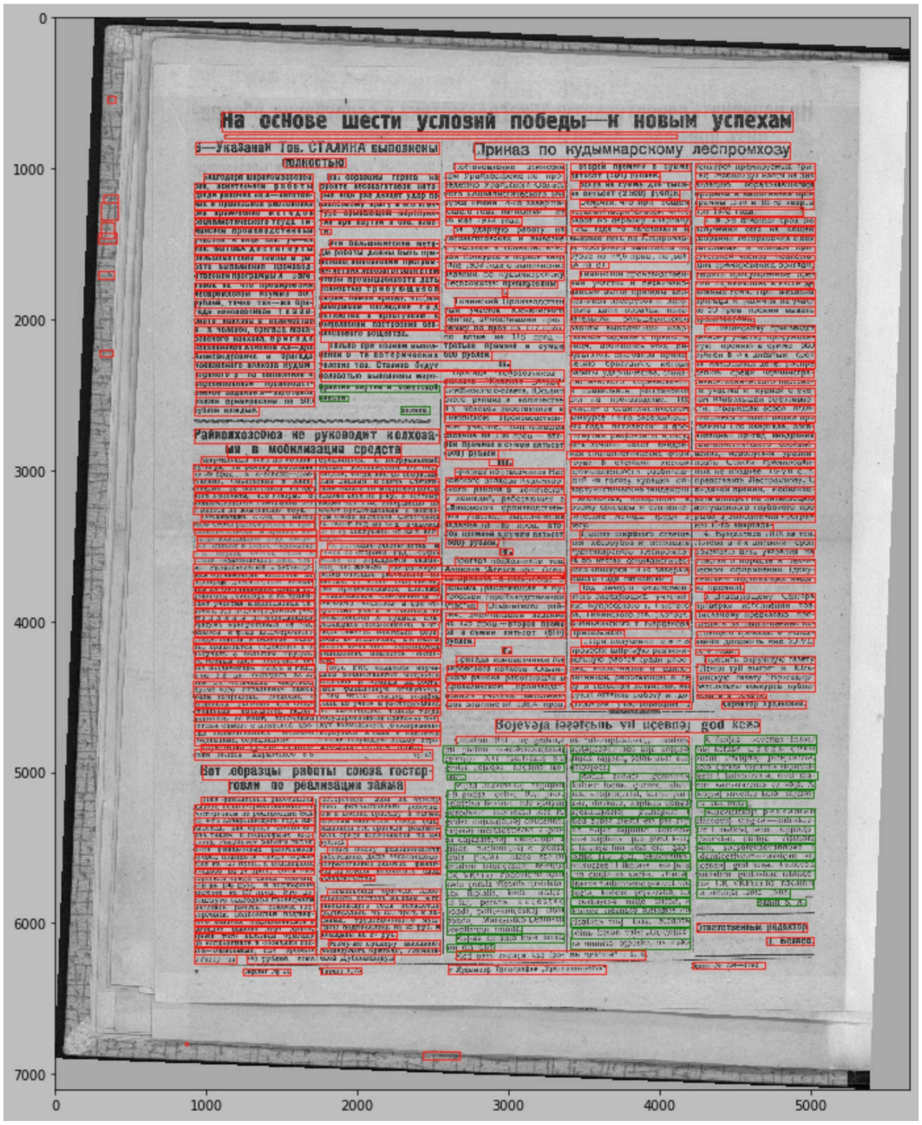

На основе шести условий победы—к новым успехам
Приказ по кудымкарскому леспромхозу
0
1000
2000
3000
4000
5000
6000
7000
0
1000
2000
3000
4000
5000

Figure 8.1: Calamari used in script detection

Bibliography

Partanen, N. and Rießler, M. (2019). An ocr system for the unified northern alphabet. In *Proceedings of the Fifth International Workshop on Computational Linguistics for Uralic Languages*, pages 77–89.