User Manual

Introduction

Our User Manual is designed to instruct the reader how we created a framework based on work done by the Early Modern OCR Project (EMOP). Most of the software used is freely available. Aletheia alone requires payment.[[1]](#footnote-1) We recommend you read the guide found [here](http://emop.tamu.edu/outcomes/Franken-Plus) for using everything and [here](http://emop.tamu.edu/Installing-FrankenPlus) for installing everything required.

The software used for training in Part 1 requires a Windows machine. Therefore, we constructed our framework assuming a user would use a Windows computer. In Part 2, however, only a few batch files restrict a Unix computer from following the steps. If the user can obtain a traineddata file without Part 1, and replace the batch files, then he or she can make full use of Part 2.

Part 1: Training

1. Here we describe the steps to create training for a typeface. There are several programs required for the process to work. We followed the instructions at the [EMOP website](http://emop.tamu.edu/outcomes/Franken-Plus) on the Franken+ page. [Here are two videos](http://emop.tamu.edu/tess-training-demo-vids) created by the EMOP team to demonstrate what we will discuss here.

1a. Aletheia

1. Load a .tif file into Aletheia. Note that Aletheia does not appear to read .tiff files so you may need to change the extension. I did this with windows command prompt.
2. Clean the image as much as possible by removing noise that would interfere with training. Aletheia allows the user to do so under the Image tab with the Select Components tool.
3. In the Regions tab, select the Analyse Page button. Choose “Glyphs, words, lines and regions with text” and the language you are OCRing. Click Run.
4. Aletheia does an initial OCR of the page but often errs. Go through the pages and change and fix these mistakes manually. Under the Glyphs tab, skim the letters and make any necessary changes by pressing the Text Content button.
   1. For ligatures, we used the [MUFI website](http://www.abdn.ac.uk/skaldic/db.php?if=mufi&table=mufi_char)
   2. Put the Codepoint value for the desired ligature into the Hex box in the Text Content window. For example, for the ﬅ ligature, type FB05 and click the character that appears in the adjacent box to insert it into the window.
   3. We recommend altering the shapes of the boxes as well so that it only contains the letter or ligature. (Like the letter ‘f’ overhanging whatever follows).
5. When you are satisfied with the quality of the page, save your progress as an xml file with ctrl+s, and then save the tif file by closing the window (not Aletheia, the window that the software opens for tif image) and pressing Yes when the appropriate window pops up. Be sure to put the xml and tif file generated from this in the same folder.
6. Repeat these steps for each page you want to use for training. We used about 12 pages for each font we trained and each page had 100+ words. This allowed us to acquire several examples of most characters.

1b. Franken+

1. Create a language for your typeface.
2. Create a name for your font.
3. In the Aletheia TIF/XML area, navigate to the folder where your tif/xml files from part 1a are saved and click Ingest Glyphs.
4. Franken+ now has lists of all the different characters found in the ingested pages. Press Edit in the Font area to see the lists in a new window. A single list contains each instance of a given character.
5. Below the Font Name, check the box(es) that best describe your font.
6. It is recommended to change the line height to a large number. We used the maximum value, 100.
7. Search the glyphs in the dropdown menu and select the best examples for Tesseract training. Click a letter to highlight it in red and therefore remove it from training. If there are very many examples of a character, then it may be easier to press Remove All and choose the which glyphs you want to keep. We recommend about ten examples per character although quite often we had only a few instances of a rare character (like a Q). For letters with descenders, like ‘p’ or ‘j’, it is necessary to adjust the y-offset. Otherwise, the character will sit on top of the line and tesseract will have a hard time distinguishing lower and upper case ‘p’. The second video on [this page](http://emop.tamu.edu/tess-training-demo-vids) demonstrates this step very well.
8. Save the font
9. You now have a font
10. Repeat steps 2-9 to prepare as many fonts as you wish to train Tesseract
11. Finally, in the synthesize TIF/Box Pair area, add a transcription text and hit Create TIF/Box Pairs. The transcription text is just a plain text file with examples of many characters and ligatures. Here is the [transcription text](http://emop.tamu.edu/sites/all/themes/bluemasters/files/SAA-Workshop/F+TraininigText.txt) the EMOP team used. We modified the EMOP transcription file to include ligatures and accented characters found in our 17th century French texts.

1c. Tesseract Training

1. In Franken+, choose the language that you created in part 1b and click Train Tesseract.
2. Select the fonts you wish to use to train Tesseract created in part 1b.
3. Add a unicharambigs, a frequent word dictionary, and another word list to training.
4. The Training Steps area show what tesseract commands will be performed during training.
5. When you are ready, hit Make Library and wait. This process could take a long time!
6. When it finishes, find the traineddata file in Documents/Franken+/<your language> . Franken+ will open this folder for you. Copy the traineddata file to the tessdata folder in C:\Program Files (x86)\Tesseract-OCR\tessdata . It may be useful to create a shortcut on your desktop.
7. Now you are ready to OCR with Tesseract.

Part 2: OCR

1. Place the tiff images that you wish to be cleaned in a folder in “1\_Input\_Tiff\_Images”
2. Run the cleanup.bat file to run Image Magick on the files and enter a name for the folder of tiff images. This cleans the images and places them in “2\_Output\_Image\_Magick.”
3. Have Tesseract OCR the images. There are two ways to do this in our framework: Franken+ or run a batch file
   1. To OCR pages with Franken+, choose the appropriate language that you used to train Tesseract and click Train Tesseract. At the bottom of the window, there is a Test Library area. Navigate to your folder of tiffs and press Go. Once this is complete, run the move-it.bat file to newly created text files to “3\_Output\_OCR.”
   2. To OCR pages with a single batch file, run ocrTess.bat. This will automatically OCR the pages and put the text files in the proper output folder. As of release, this batch file has some hard-coded components so you will need to make necessary changes to the path and language used.
4. Run the post-processing module main.py. Decode errors obstructed us from running the python script from command prompt so we used PyCharm Community Edition 2016.1.3. Our post-processing module attempts to fix mistakes made during OCR. The new text files are placed in a folder within “4\_Output\_Final\_Text.” Errors that cannot be resolved are written to the text file with error tags around it. For example, <err>word</err>. The post-processing module also writes a log when it is finished and records how many words are fixed at each step.
5. Optionally, you can also run errGenerator.py which may make manual corrections easier. This script searches the post-processed text files for words surrounded by error tags and tries to find possible corrections. It records the incorrect word and its three closest words from the dictionary into a new text file in the Errors folder. We calculate the three closest words using two metrics, Levenshtein distance and python’s built-in sequencematcher.ratio() function.

Documentation

We record here details about how we fine-tuned our framework. Often we experimented with different parameters to yield the best results or we encountered a problem and devised a new method to overcome it. Thus, the framework as we present it may seem illogical to fresh eyes but we undoubtedly spent much time gathering evidence to justify our choices.

Anomalies in Volumes

We analyzed 5 books each split into two volumes printed with a printing press in the late 17th century. The printing press is notoriously prone to errors: too much ink may cause the image of a letter to bleed through a page or make large blotchy characters, too little will cause the letters to appear thin and faded. The scanning process could also be at fault for the quality of the pdf images. These techniques caused some issues that we observed across 10 volumes:

1. Bent text. Some pages were evidently not flush against the scanner and thus appears to bend at the edges. Often a line appears at the edge where the text bends towards the spine and occasionally the text joins this line to become illegible
2. Slanted pages. These pages tend to be straight but oblique. This is also an issue caused during scanning
3. Tiny text. On rare occasions, a small note is made in a smaller font. During our pre-processing procedure, ImageMagick tends to damage the small letters too much and they cannot be OCRed very well.
4. Marginalia. Similar to #3 above but we distinguish this because of its placement along the side of the text. These interfere with our post-processing module and are illegibly OCRed
5. Ink Bleed-through. Typically, this appears as noise interspersed among the text that can be removed by ImageMagick. When the ink is more severe, the shapes of the letters are visible through the page and tend to obscure the text. In this case, it is impossible to for our training to OCR accurately.

Since the marginalia occur on such a small number of pages, we removed it from the page with Photoshop before OCR so that it can be manually added later. Yet, we can find no feasible fix for the other issues due to their high frequency and unpredictability. Hence, we leave these pages for others to manually correct them as it would be excessively difficult to modify our program to deal with these issues.

Dictionaries

We began using a modern French dictionary found on the tesseract GitHub page for both Tesseract training and for our post-processing module. We noticed, however, that many French words were spelled differently in the 17th century. For example, several words that end in an ‘i’ in modern French instead have a ‘y’. Therefore, we borrowed two contemporary dictionaries with permission from the [ARTFL project](https://artfl-project.uchicago.edu/).

Initially, we tried using only the ancient dictionaries in our post-processing script, but we found that the smaller ancient dictionaries lacked too many words and therefore made more mistakes than the single modern dictionary. As a result, we chose to include each of them.

In 2016, we combined each of these dictionaries into a single text file and removed duplicate entries. In doing so, we reduced total amount of words by about 30% from 610,912 to 428,564.

errGenerator.py and three closest words

In our last step of our post processing script, we attempted to guess which word would be the correct replacement. We calculated the closest word using the Levenshtein distance and python’s built in sequencematcher.ratio() function, where the former yields the number of changes to turn word1 into word2, and the latter calculates a ratio of similarity.[[2]](#footnote-2) Thus, the unrecognized word would be compared with those in the dictionaries to obtain the best match. These metrics, however, are not perfect and could easily introduce more error. Therefore, we instead outputted the top three matches into a text file to be resolved manually. This text file can then offer suggestions and may help someone fix remaining mistakes.

Multiprocessing

One of the ways we wished to improve the speed of our post-processing module included parallel processing. We first began looking at multithreading with Python but came across several shortcomings that led us away. Multithreading in Python is subject to something called the Global Interpreter Lock[[3]](#footnote-3) (GIL), which prevents a process from executing more than one thread of python at a time and thus lacks true parallelism. Therefore, while multithreading may be useful for I/O bound programs, it is not suitable for CPU bound programs like our post-processing module. Therefore, we implement multiprocessing instead.

Regular and Italic folders

During the second summer of the project, we proceeded through our framework for a second time and created training for italic letters. We carefully made a list of which pages had any italic font characters on them and which did not. We wondered whether combining italic with regular font training would impair the results. We OCRed a sample of both italic and regular with both training sets.

Unsurprisingly, the italic font performed much better on italic pages. On regular font pages, both training sets had comparable results, and sometimes one erred while the other was correct. Yet, overall the regular font training appeared more accurate. Therefore, we chose to separate the files in order to yield the best results.

Unicharambigs file[[4]](#footnote-4)

After realizing that the EMOP team’s unicharambigs file only contained mandatory changes while ours contained a mix of mandatory and optional changes. We created 4 training sets to compare the differences: 2 regular and 2 italic sets. The results are recorded in project meeting 2. Between the italic font with only mandatory changes and the same font with optional changes, the former consistently made less errors. Thus we chose to use the italic font with mandatory changes for pages with italic letters.

We repeated the experiment with all 4 training sets on regular-font pages. The results here yielded inconclusive results. A pie chart comparing the four training sets revealed that each made approximately the same amount of errors across the pages we used. Yet, the regular font training made slightly fewer errors than italic training and the regular font with the mandatory unicharambigs file did better than the one without it. For these reasons, we decided to use the regular font with only mandatory changes in its unicharambigs file for regular font pages. We would have liked to do more tests to determine whether there was a significant difference but since time was running short we had to move on.

Software Versions and Specifications

Windows 10

[PyCharm Community Edition 2016.1.3](https://www.jetbrains.com/pycharm/)

[Python 3.4.4](https://www.python.org/downloads/)

[Aletheia 2.1](http://www.primaresearch.org/tools/Aletheia) (no longer available but Aletheia Pro should work)

[Tesseract 3.02.02](https://github.com/tesseract-ocr/tesseract)

[ImageMagick-7.0.1-6-Q16-x64-dll](http://www.imagemagick.org/script/binary-releases.php)

[.NET 4.5](https://www.microsoft.com/en-us/download/details.aspx?id=30653)

[My SQL Community 5.7.12.0](http://dev.mysql.com/downloads/installer/)

[Franken+](file:///C:\Users\William\Dropbox\Summer_2016_Mine\Latest%20Franken+%20Release)

Photoshop CS6 (optional)

1. When this project started, Aletheia was free but required users to register. We have not tested the latest version but at a glance it appears that the Pro version is necessary to achieve the same results. [↑](#footnote-ref-1)
2. See RialVisa\_2015 paper [↑](#footnote-ref-2)
3. See the [docs](https://docs.python.org/3.4/c-api/init.html) for more info [↑](#footnote-ref-3)
4. This is used for Tesseract training. [Here](http://emop.tamu.edu/node/48) is a page from the EMOP website that explains how this file is used [↑](#footnote-ref-4)