Profiler Manual

Florian Fink
Centrum für Informations- und Sprachverarbeitung (CIS)
Ludwig-Maximilians-Universität München

2015-08-25
last updated: 2016-02-29

Contents

1 Introduction

This manual covers both the language aware OCR document error profiler and the profiler web service. It explains how to obtain and compile all tools necessary to create OCR error profiles for documents in various languages and covers the deployment of the profiler web service.

The profiler is a tool that is also able to calculate historical spelling variants used in (historical) documents and to distinguish them for real OCR errors. It needs to be trained on various external language resources in order to do its work. The profiler web service is a web service that uses the profiler in the background to generate language aware error profiles for different languages. It relies on number of pre-compiled language resources – the so called language backend of the profiler web service.

This manual assumes that you have a running Linux/Unix system and that you are proficient using the command line and build files from source. It also assumes that you use some variation of a POSIX compliant shell. The compilation of the profiler and the deployment of the web service require some additional tools. You need to be able to install those requirements on your system if they are not already installed.

2 The language aware OCR document error profiler

The profiler calculates historical spelling variants and OCR errors of a (historical) document using a modern dictionary. It is therefore able to differentiate between *real* OCR Errors and historical spelling variants of words.

First of all this part of the manual covers the general workings of the profiler. Then the more technical parts – the compilation and installation – of the tool are explained in more detail. Finally the usage of the tool and the generation of specific language aware OCR document error profiles are covered.

2.1 Overview

In OCR'ed text in general and in OCR'ed historical text in particular there appear to be number of words, that do not match a regular entry in a dictionary of the text's language. There are different explanations for these *unexplained* words. On possibility is, that the automatic text recognition engine made a mistake on character level and the word is not a proper dictionary entry; we call these errors OCR errors. Another possibility is – particular in historical texts – that the OCR engine did recognize the word with no errors, but the historical spelling of the word differs from its modern spelling – these cases are called historical spelling variations. It is possible as well that both OCR and historical spelling variations overlap on the same words.

The profiler tries to find good explanations for any of these *unexplained* words in the text, using the profile of the actual document. It therefore tries to map unrecognized words to dictionary entries with a minimum Levenshtein distance. In some circumstances there can be a lot of words with similar Levenshtein distances to a particular word, so the profiler uses additional pattern rules to find more fitting candidates.

These pattern rules manifest themselves both in common OCR error patterns as well as in external historical spelling variations. They describe common character level transformations that can be used

to describe common error patterns in historical OCR'ed documents.

Consider the pattern rule `e -> c`. This rule describes a common OCR error pattern, where the letter `e` in the source text is recognized as a `c` by the engine. For another example consider the rule for a historical spelling pattern `t -> th`. It explains historical (German) word forms like `theil` that should map to the modern form `teil`. So if the profiler finds a lot of unexplained words that could be explained with a `e -> c` OCR Error pattern, this pattern would get a higher priority for calculating correction candidates. On the other hand if the profiler would encounter a word `theil` that he could explain using historical patterns, he would not recognize this word as an OCR error and would assume a historical spelling variation of the word `teil`.

The profiler uses modern dictionaries and external historical pattern rules to calculate *the best explanations* for unexplained words for a given input text. It generates a list of correction suggestions for these unknown words. These correction suggestions can be used among others to postcorrect historical OCR'ed documents. See the PoCoTo manual for more information about the postcorrection of OCR'ed documents.

2.2 Requirements

In order to build and use the profiler you need to install a number of external tools[1] on your machine. Make sure that all of the required tools are installed on your system before you proceed.

- CMake
- Git
- The Gnu C++ Compiler
- Make
- Git
- Xerces XML Parser
- CppUnit
- Java Virtual Machine (JVM)
- ICU development library
- In the case of Xerces XML Parser and CppUnit make sure that you have the development headers of the according libraries installed as well. They are normally called `*-dev` in the different package management systems. Make also sure that you install the C version of the Xerces XML Parser[2].

2.3 Building and installing the profiler

In order to use the profiler, you have to download its sources, compile the binaries and install them manually. Before you proceed make sure that you have installed all required additional tools and libraries.

[1] All required tools are freely available and should be already included in your distribution's package management system

[2] Sometimes called `xerces-c` or something similar.

2.3.1 Downloading the sources

The source code of the profiler is maintained using a git repository. Just use git to clone the source code:

```
$ git clone https://github.com/cisocrgroup/profiler.git
```

Download and unpack the sources.

2.3.2 Building the profiler

In order to compile and install the profiler you need to know the path to your system's java instalation[3]. Simply export this path via the `JAVA_HOME` environment variable into your current environment:

```
$ export JAVA_HOME=/usr/lib/jvm/java-7-openjdk
```

It is not necessary to install the profiler globally. You can install it to some convenient place in your home directory. If you want to install the profiler in your home directory (recommended) instead of a system directory like `/usr/local/bin` you need to set the `PREFIX` environment variable to a installation directory in your home directory. In order for the system to find the installed executables, make sure that your systems `PATH` variable points to this directory:

```
$ export PREFIX=$HOME
$ export PATH=$PATH:$HOME/bin
```

Now you can start to build and install the profiler. From the Profiler top level directory simply call the `make` utility:

```
$ make
$ make install
```

You can also skip the step of exporting the `JAVA_HOME` and `PREFIX` variables and combine it all into one call to `make`:

```
$ make JAVA_HOME=/usr/lib64/jvm/java-7-openjdk
$ make PREFIX=$HOME install
```

Make compiles the sources and installs the binaries into the given location. If make encounters any errors, the build process terminates and nothing will get installed. In this case make sure that all your given paths are valid and that all required tools and libraries are installed on your system and can be found by the compiler linker.

After the installation has finished, test if you can execute the installed tools. Just type:

```
$ profiler --help
$ compileFBDic --help
$ trainFrequencyList --help
```

[3] Normally it can be found in `/usr/lib/jvm/` or `/usr/lib64/jvm` system directories. It is called `java-7-openjdk` or something similar.

If the installation was successful you should see the help text of the matching tools. If you get an error saying something like `profiler: command not found` make sure that your `PATH` variable points to the install directory of the profiler.

```
$ echo $PATH
$ export PATH=$PATH:/path/to/prefix/bin
```

2.4 Generating a language model

In order to avoid a confusion of terminology, we make the following note: The `language aware OCR document error profiler`, or profiler for short, is a program calculating correction candidates for statistical series of suspected OCR errors in historical documents. Its output is a document containing the original OCR tokens together with correction candidates and further information (such as the historical and OCR patterns giving rise to the OCR token, its Levenshtein distance from a correction candidate and its weight) which might be called the `error profile` of the document which has been OCR'ed. The profiler uses *as input* various language dependent files such as a sorted lexicon of wordforms, a file of historical rewrite patterns describing spelling variations and a ground truth text. These files need to be compiled into a special form and represent language resources needed for the profiler. They are called a `language model` in the following and must not be confused with either the language aware error profile (output) or the profiler itself (program).

After you have installed the binaries as described in the previous chapter, you can now build a language model. This profiler can then be used with this language model to get a list of correction candidates for historical spellings in various document written in this language.

The installation process did compile and installed 3 different programs. Make sure that all of them are working before you proceed:

- `profiler` the main profiling tool
- `trainFrequencyList` tool to train the profiler on a historical ground truth file (see next chapter)
- `compileFBDic` tool that compiles the dictionary.

2.4.1 External language resources

In order to build a language modele you need a number of external lexical resources:

- At least one sorted full form dictionary of modern words
- A file of historical pattern rules
- A text with historical spelling variations as ground truth

All these files must be UTF-8 encoded. The profiler does not support any other encoding. If you use files that are not UTF-8 encoded you will get strange errors and segmentation faults while using the tools. You can check the encoding of your files using the `file` utility and convert their encoding using the `iconv` utility.

```
$ file -i myfile
$ iconv -t UTF-8 < myfile > myutf8file
```

Make sure that all your input data and all your external resources (dictionary, patterns, historical ground truth) has the same encoding and that the files are in the same unicode normalization form (either composed or decomposed). You can use the `uconv` tool of the `icu` package to convert between different encodings and / or normalization forms.

```
$ uconv -f utf8 -t utf8 -x [nfc|nfd] -o output.txt input.txt
```

If you decide to use the decomposed normalization form for your input files, consider to increase the maximum allowed Levenshtein distance in the configuration file. The profiler does not distinguish between real and combining characters. For example the profiler would calculate a Levenshtein distance of 2 between `quâm` and `quem` if they were encoded in the decomposed normalization form, whereas in composed form the distance would just be 1.

2.4.1.1 Dictionary

The dictionary is a simple text file that contains each word on a separate line. It cannot contain any duplicate entries and must not be sorted using your environment's default locale. Sort it using the C locale, e.g.:

```
$ LC_ALL=C sort file | LC_ALL=C uniq > outfile
```

If you get an error message saying `your input is not in sorted order` while you are using any of the profiling tools, the dictionary file you use is not properly sorted and you have to use the given command to sort the dictionary correctly.

On the other hand, if you should get an error saying `error: empty key` your dictionary contains at least one empty line. You can remove the empty lines from the dictionary using `sed`:

```
sed -e '/^[[:space:]]*$/d' input.txt > output.txt
```

2.4.1.2 Simple creation of a modern dictionary with hunspell

If you do not have any language resources available, you can use the spelling dictionaries of LibreOffice to create your own for usage with the profiler. You need to install hunspell and the hunspell tools in order to expand ("unmunch") Hunspell's dictionary (.dic) and affix (.aff) files into a list of wordforms needed for the profiler.

Download a *spelling* dictionary of your desired language. The downloaded LibreOffice extension is a zip archive that you must unzip before you proceed. You need two files from the archive, the `.dic` dictionary file and the `.aff` file containing the different affixes of the language. For historical reasons, the files will be most likely encoded in ISO-8859-1. Make sure to convert the `.dic` and `.aff` files to UTF-8 before you proceed:

```
iconv -f ISO_8859-1 -t UTF-8 de_DE.aff > de_DE.utf8.aff
iconv -f ISO_8859-1 -t UTF-8 de_DE.dic > de_DE.utf8.dic
```

Also, change the line `SET ISO-8859-1` in the affix file to `SET UTF-8`.

Now you can use the `unmunch` utility to generate the full forms from the affix and the dictionary files. The `unmunch` command generates composita components for some languages. You need to remove these from your dictionary, since the profiler cannot handle these entries.

```
unmunch ./de_DE.dic ./de_DE.aff | grep -E -v "/|-" > mydictionary.dic
```

The `grep -E -v "/|-"` expression in the above command is used to filter out all composita components from the dictionary file. The resulting file can now be processed in the same way as described above.

Note that in the case of Latin we already provide sorted word lists. The somewhat involved procedure to generate it is described in the README accompanying our lexica.

German and Ancient Greek lexica are also provided.

2.4.1.3 Pattern file

The pattern file contains pattern rules that describe how to transform modern spellings to historical spellings. Each line of the file should contain exactly one pattern rule of the form `mod:hist`. You can use `@` to mark the start of a word and `$` to mark the end of a word if you want to specify pattern rules that will only apply at the beginning or the end of words. Empty lines or lines starting with a `#` are ignored.

If you want to encode the pattern rules you need to describe the variation of the modern word form against the spelling of the respective historical form. Consider for example the German word `Teil` and its historical spelling `Theyl`. You could use the following two pattern rules in your file to describe the variation of the two words:

```
t:th
ei:ey
```

Note that the case of the characters are generally ignored. Use `@` and / or `$` to mark the beginning and end of words, e.g:

```
bar$:lich$
```

Note also that it does not appear to be possible to escape any of those special characters to their literal meaning. If you need to encode pattern rules that contain any of the characters `#`, `$` or `@`, you are out of luck.

2.4.1.4 Historical ground truth

The ground truth file is a simple UTF-8 encoded text file that contains words with historical spellings as described in the pattern rules. This ground truth is used to train the initial pattern weights for the profiler.

The profiler uses these training files just as starting weights for its calculations. It always tries to optimize the pattern weights on the actual input text and uses the initial weights as mere hints for good pattern weights.

The time the training of the initial weights takes depends heavily on the size of the training file. If the file is very large the training can easily take days to finish. Since the training just calculates the initial weights for the patterns, its not a big deal to reduce the size of the file if the training takes too long.

2.4.2 Compilation of a language model

In order to compile a language model, you need at least the three external resource files described in the previous chapter: a lexicon of modern wordforms, a file of historical rewrite patterns and a ground truth text. It is recommended to create each language model in its own subdirectory. It is also possible to use a different layout for your profile. You just have to adjust the paths of the respective files accordingly.

The creation of a language model involves three steps:

- compilation of the dictionary
- creation of the profile configuration (`.ini`) file
- training of the initial pattern weights

The following description assumes that all your external resources are in a common directory. All additional files that are generated by the profiler toolchain will eventually end up in this common directory as well. After you have compiled the language model you can remove both the sorted dictionary and the historical ground truth file in order to save some hard disk space. You must *not* remove the historical pattern file and the compiled dictionary since the profiler needs them in order to generate correction candidates later on.

2.4.2.1 Compilation of the dictionary

You can use the tool `compileFBDic` to compile the dictionary for the profiler. Make sure that your dictionary file is sorted as described in section Dictionary. Simply run `compileFBDic input-file output-file` in the common language directory.

Assuming that you have an unsorted dictionary file `lang.lex`, you must sort the dictionary into the sorted dictionary file `lang.sorted.lex` and then compile the dictionary to the file `lang.fbdic`. You can accomplish that using these two simple steps:

```
$ sort LC_ALL=C lang.lex | uniq > lang.sorted.lex
$ compileFBDic lang.sorted.lex lang.fbdic
```

2.4.2.2 Creation of the `.ini` file

The profiler needs a configuration file that defines the different dictionaries[4], the pattern file and the initial weights that the profiler should use. You can simply generate such a configuration file using the profiler's `--createConfigFile` command line option:

```
profiler --createConfigFile > lang.ini
```

This default configuration file contains a lot of comments on how to configure the profiler. Just open up the configuration file in the editor of your choice and adjust the settings accordingly. The file format is a simple `ini` format. The expression `[block]` start a new namespace / block for the configuration options, the expression `key=value` is used to set configuration parameters.

It is possible to create custom variables that can be referenced in the configuration file:

[4] You can use multiple dictionaries for the profiler; see chapter Using multiple dictionaries for more information on how to define them.

```
BASE_PATH = /absolute/path/to/directory
path = ${:BASE_PATH:}/file.txt
```

You should at least consider to edit the following variables (there are certain entries that are not generated automatically; they are all marked with †. Make sure to add these to the configuration file as well):

- `ocrPatternStartProb` should be set to `1e-5`†.
- `donttouch_hyphenation` should be set to `1`†.
- `donttouch_lineborders` should be set to `0`†.
- `patternFile` should contain the name (path) of the file that contains the patterns.
- `patternWeightsFile` should contain the name (path) of the file where the profiler will store the initial pattern weights after the training on the historical ground truth.
- `corpusLexicon` should contain the name (path) of the file where the profiler will store the lexicon of the ground truth file†.
- `freqListFile` should contain the name of the compiled frequency list that is also used for the initial pattern weights.
- at least one block that contains information about one compiled dictionary. Such a dictionary block must be named with a leading `dict_` or it will not be considered active (even if it is set to active in the configuration block). Each block must contain the following parameters:
  - `dict_type` should be set to `simple`†.
  - `path` should contain the name (path) of the compiled dictionary.
  - `ocrError` should contain a maximal Levenshtein distance used to find similar words in the dictionary. Words in the dictionary that have a larger distance than this value are not considered to be valid correction candidates for any given word.
  - `ocrErrorsOnHypothetic` sets the maximal distance used to find similar token with a variant pattern application. Just use the default value `1`.
  - `histPatterns` sets the the maximal number of patterns that can be applied to one word.
  - `cascadeRank` sets the rank of the dictionary[5] (see Using multiple dictionaries).
  - `priority` sets the priority of the dictionary[6] (see Using multiple dictionaries).
  - `active` should be set to `true` if the dictionary entry should be used, to `false` otherwise†.

For example assume you have one compiled dictionary `lang.fbdic` and the pattern file `lang.pat`. You want to store the training weights and frequency information in the files `weights.txt` and `freqlist.binfrq` in the same dictionary. This is what the important parts of your configuration file should look like:

```
#...
[language_model]
patternFile = "lang.pat"
patternWeightsFile = "weights.txt"
freqListFile = "freqlist.binfrq"
```

[5] Higher rank dictionaries are not even considered if lower rank dictionaries generate valid correction candidates.

[6] The priority orders dictionaries of the same rank and the order of the correction suggestions.

```
corpusLexicon = "corpusLexicon.bin"

#...
[dict_modern]
dict_type = simple
active = true
path = "lang.fbdic"
histPatterns = 2 # default
ocrErrors = 2 # default
ocrErrorsOnHypothetic = 1 # default
priority = 100 # highest priority
cascadeRank = 0 # default
#...
```

2.4.2.3 Training of the initial pattern weights

In the last step you have to train the initial pattern weights for the language model. This step generates the three files that you set in the configuration file using the keys `patternWeightsFile`, `freqListFile` and `corpusLexicon`. These files contain the training weights, the frequencies and the token of the training file. The profiler needs these files to generate correction candidates.

In order to train the profiler with the configuration file `lang.ini` and the ground truth file `lang.gt` execute the following command:

```
$ trainFrequencyList --config lang.ini --textFile lang.gt
```

If you get any errors during this step

- Make sure that all the paths given in the file on the command line are valid.
- Make sure that all the paths given in the configuration file are valid.
- Make sure that you have defined all required configuration parameters as stated in Creation of the `.ini` file

After the command has finished (this can take a while depending on the size of the training file) make sure that the three files mentioned above were actually created by the training.

2.5 Using multiple dictionaries

As mentioned above it is possible to use multiple dictionaries with a language model. For example you could use one general purpose dictionary and additional specialized dictionaries for named entities (persons, locations, organizations) or some other specialized dictionaries that contain foreign technical terms etc.

In order to use multiple dictionaries you have to add appropriate dictionary blocks into the configuration file. Each of these additional dictionaries need to be compiled the same way as explained in chapter Compilation of the dictionary. Make sure that each new dictionary's block name starts with `dict_` – otherwise it will be ignored by the profiler.

Consider this extract of the configuration file taken from a German language model:

```
###################  RANK 0 ##############################################
[dict_modernExact]
dict_type = simple
active = true
path = "${:DICT_PATH}/staticlex_de.frq.fbdic"
histPatterns = 0
ocrErrors = 0
ocrErrorsOnHypothetic = 0
priority = 100
cascadeRank = 0
#...
###################  RANK 3 ##############################################
[dict_modernError]
dict_type = simple
active = true
path = "${:DICT_PATH}/staticlex_de.frq.fbdic"
histPatterns = 0
ocrErrors = 2
ocrErrorsOnHypothetic = 0
priority = 100
cascadeRank = 3
# ...
[dict_latin]
dict_type = simple
active = true
path = "${:DICT_PATH}/LatinForms.fbdic"
histPatterns = 0
ocrErrors = 1
ocrErrorsOnHypothetic = 0
priority = 70
cascadeRank = 3
```

As you can see, this language model uses a lot of different dictionaries. Each new dictionary block has its own block and configuration parameters. Note that all entries must contain the path to the compiled dictionary, the type of the dictionary and are set to active. They are ordered both by their rank and priority.

If you want to use multiple dictionaries you should consider the configuration parameters `cascadeRank` and `priority`. Dictionaries with a higher rank than others are not even considered if dictionaries with a lower rank produce good correction candidates. Only if the lower rank dictionaries cannot produce good correction candidates the higher rank dictionaries are considered. If multiple dictionaries of the same rank produce valid correction candidates the priority decides which correction from which dictionary is considered.

Using multiple dictionaries can improve the quality of the profiler's results. There is no general answer on how to combine multiple dictionaries, though. You have to experiment with the different dictionary configuration parameters in the configuration file in order to get the best results for your specific setting.

As a rule of thumb, you should always use at least one general purpose dictionary of rank 0. You can then start to add more specialized dictionaries with higher ranks. This ensures that error pattern on common words are recognized soon in the process and that the more specialized dictionaries are used just on some rare cases.

2.6 Profiling a file

If you have compiled a language model you can now use the language aware error profiler on your files. The profiler accepts input files in different formats and is able to write different kinds of output files.

You need the configuration file for your language model. Make sure that the paths in the configuration file can be found from the place where you execute the profiler. If in doubt use absolute paths in the configuration file or execute the profiler from the same directory where the configuration and the other files of the language model are located. Otherwise you will get errors while executing the profiler.

The profiler understands a number of different command line options. You need to specify at least these:

- `--config iniFile` sets the configuration file and therefore the language aware error profile the profiler uses on the input file.
- `--sourceFile inputFile` sets the input file.
- `--sourceFormat formatString` sets the format of the according input file. The `formatString` must be one of the following:
  - `DocXML` for files in the internal format used by the postcorrection tool backend and the profiler web service.
  - `AltoXML` for files that are encoded in the Alto XML format
  - `ABBYY_XML_DIR` specifies that the input file is not a file but a directory that contains valid files in the Abbyy XML format
  - `TXT` for simple text files
- at least one output option using one of these command line parameters:
  - `--out_xml outputFile` writes an XML file that contains a list of correction candidates and OCR errors for unknown words found in the input document.
  - `--out_html outputFile` writes a general HTML file that contains various pieces of information about the profiler's performance (this output format is very helpful if you want to examine different language settings).
  - `--out_doc outputFile` writes the original input document with correction candidates for each token as a `DocXML` formatted file.

You can additionally restrict the number of iterations using the `--iteratons n` command line option. This overwrites any settings in the configuration file.

If you want to profile a simple text file and see all the output files the profiler generates you could use the following command:

```
$ profiler --config lang.ini --sourceFile input.txt \
  --sourceFormat TXT --out_xml out.xml \
  --out_html out.html --out_doc outdoc.xml
```

2.7 Language resources

We provide some basic language resources. These files contain dictionaries, pattern rules and training files for different languages[7]. They also include simple `.ini` files for each language. You can use those language resources to start building your own custom language models.

You can obtain these resources by checking out the appropriate git repository:

```
$ git clone https://github.com/cisocrgroup/Resources
```

Afterwards change into the `Resources/lexica` directory. This directory contains the `.ini` files for the languages and some directories that contain the different files for the different languages. There is a Makefile that automates the generation of all needed files. In order to use it, you need some additional command line tools installed on your system:

- perl
- uconv

In order to build the resources change into the directory that contains the Makefile and simply type `make`. You can speed up the build using parallel processes with `make -j n` where `n` is the number of parallel processes `make` uses.

The Makefile assumes that the `compileFBDic` and the `trainFrequencyList` command line tools are installed under your home directory at `~/local/bin`. You can explicitly tell `make` where to find these programs using the variables `FBDIC` and `TRAIN` respectively.

```
$ FBDIC=/path/to/compileFBDic TRAIN=/path/to/trainFrequencyList make -j 4
```

After the build has finished you have all the required files for the profiler. You can use these resources for the profiler web service. Just copy the language directories along with their `.ini` configuration files to the language backend of the profiler web service.

2.8 Missing patterns file

You can use the profiler even if you do not have any patterns file. In order to use the profiler without historical patterns, you can specify the `--allModern` command line option for the `trainFrequencyList` tool. For the profiler itself there is currently no such command line switch. To use the profiler without any historical patterns you have to provide a patterns file that must at least contain one (fake) pattern.

[7] Currently there are language resources for German, Latin and Ancient Greek.

3 Profiler Web Service

The profiler web service supplies the profiling tools as a web service. It is written in Java and uses the profiler executable to profile documents. You can use this web service to provide different language models for your users. The postcorrection tool PoCoTo needs this web service in order to supply correction candidates and error patterns for the postcorrection of your OCR documents. For more information about the postcorrection of OCR documents in general and PoCoTo's connection to the profiler web service see the relevant PoCoTo manual.

The profiler web service depends on the Apache Tomcat web server and the Axis2 SOAP engine. You should have read the basic documentation of both tools if you plan to make the profiler web service publicly available.

3.1 Requirements

In order to deploy the profiler web service, you need to have some additional tools installed:

- The Apache Tomcat web server
- The Apache Axis2 SOAP engine
- A language backend that contains various language models, the profiler executable and the configuration files for the language models
- The Apache Ant build tool

3.2 Apache Tomcat web server

In order to deploy the profiler web service you need a running instance of the Apache Tomcat web server. If you do not already have a running instance of an Apache Tomcat server or if you want to test the profiler web service locally you can follow the instructions given in this chapter. If you plan to deploy the web service for a greater audience make sure to read the official documentation of the Apache web server.

You can download the latest stable version[8] of the Apache Tomcat web server from the Apache download page. Download the core distribution from the download page and extract the content of the archive to some convenient directory.

First of all you should edit the `conf/tomcat-users.xml` configuration file in the directory of the extracted archive. Read the hints in the comments of the file and configure your users and roles accordingly. Make sure that you have defined the correct roles if you want to access the GUI configuration interface of the tomcat web server. You can edit the `conf/server.xml` file to set further internal server settings like ports etc.

Note that none of these settings are mandatory. You can skip the whole configuration process and just use the default settings of the server.

After you have finished to setup the server, you can start the server. To start the server you have to execute the `bin/startup.sh` script. If you get an error, try to set the `JAVA_HOME` environment

[8] At the time of this writing the latest stable version was 8.0.24

variable to point to the Java installation of your system as mentioned in the chapter Building the profiler. To stop the server you can execute the `bin/shutdown.sh` script.

In order to test if the server is running, you can open the URL `localhost:8080`[9] in the web browser of your choice. You should see the Tomcat web server start page. If your browser cannot open this URL or if you do not see the start page, try to restart the server executing the `bin/startup.sh` script again. Make sure that the script finishes without any errors and try to open the URL again.

3.3 Apache Axis2

The profiler web service needs the Axis2 web service engine deployed on the tomcat web server. You can download the latest stable version[10] of the web archive (WAR) distribution from the axis2 download page.

The deployment of the axis2 engine is straightforward. Just extract the archive and copy the `axis2.war` web archive into the `webapps` directory of your tomcat installation. Restart the tomcat server afterwards[11].

You can test the axis2 web service now. Just open the following URL in your web browser: `http://localhost:8080/axis2/services/Version?wsdl`. You should get an XML file that describes the version of the running axis2 web service. If you get an error, make sure that you have copied the `axis2.war` file to the correct directory. You can check the log files in the `log` directory of your tomcat installation to see if any additional errors happened during the deployment.

3.4 Versions of Tomcat and Axis2

There are a lot of different versions of Tomcat and Axis2. Older versions of Axis2 will *not* work with newer version of Tomcat and vice versa. Make sure that you use recent versions of both Tomcat and Axis2.

In addition you should make sure that you always use *the same* versions when you deploy the profiler web service. You cannot compile the web service with one Tomcat and Axis2 version and deploy the web service to another Tomcat server of a different version that runs yet another version of the Axis2 service.

3.5 The language backend

The profiler web service needs the profiler executable and various language models that have been compiled as described in the previous chapter Generating a language model. This so called language backend is a directory that contains all the different language models that the profiler web service provides. For obvious reasons both the profiler executable and directory must be accessible by the Tomcat web server.

[9] Make sure that the port number of the address matches your settings in the `conf/server.xml` configuration file.
[10] At the time of this writing the latest stable version was 1.6.3
[11] To restart the server you have to execute the `bin/shutdown.sh` and `bin/startup.sh` scripts

The language backend consist of a set of directories. Each directory contains external resources of the profiler[12] for *one* language. For each supported language there is also an additional profiler configuration file needed. This configuration file should contain all the settings that have been described in chapter Creation of the `.ini` file. The variables in the configuration files should point to the respective language resources in the language directory.

The profiler web service uses those configuration files to detect the languages it should support. It expects these files to be named after the language it supports and the file extension `.ini`. It wont detect configuration files with different naming conventions[13] and therefore ignore all other files.

Consider the following layout of a language backend:

```
/path/to/language/backend
    |-German.ini
    |-German
    |-----patterns.txt
    |-----dictionary.fbdic
    |-----initialWeights.txt
    |-----frequencyList.binfreq
    |-----Geo.fbdic
    |-Latin.ini
    |-Latin
    |-----patterns.txt
    |-----dictionary.fbdic
    |-----initialWeights.txt
    |-----frequencyList.binfreq
    |-Greek.ini.disabled
    |-Greek
    |-----patterns.txt
    |-----dictionary.fbdic
    |-----initialWeights.txt
    |-----frequencyList.binfreq
```

In this example the profiler would only recognize 2 different language models – it would not recognize the Greek language model.

If you build the language backend for the profiler web service, you should try to follow these conventions. It is possible to construct the language backend in a different way – just make sure that all variables in the configuration files point to the right language resource files. E.g. you could just provide a directory that contains configuration files as language backend and make the variables in these configuration files to point to the language models in different directories.

If you have problems with the profiler web service, make sure that all variables and configuration files are valid and point to existing files on the server. You can check the `logs/catalina.out` logfile of the Tomcat server to check for any problems with the profiler.

[12] The pattern file, the initial pattern weights, the different dictionaries etc.

[13] You can exploit this behavior to temporary disable a language – just rename the extension of the according configuration file to something else.

3.6 Deployment of the profiler web service

The profiler web service is a small application that needs to be compiled against the Tomcat and Axis2 libraries you use. It uses one small configuration file to discover the profiler executable and the language backend. In order to deploy the web service, you have to download and compile the source code, create the configuration file and copy the compiled archive into the axis2 directory of the Tomcat server.

3.6.1 Downloading the source code

The source code of the profiler web service is maintained in a git repository. Clone the source code of the project using git:

```
$ git clone https://github.com/cisocrgroup/ProfilerWebService.git
```

3.6.2 Downloading the Axis2 libraries

To compile the profiler web service you need the binary distribution of the Axis2 web service. You need to download it from the Axis2 download page and extract the archive. Make sure that the version of the binary distribution matches the version of the WAR archive you used for the Tomcat server.

3.6.3 Building the profiler web service

To build the web service, first change into the pws[14] directory. You need to set the two two environment variables `JAVA_HOME` and `AXIS2_HOME`. `JAVA_HOME` should point to your system's java installation; `AXIS2_HOME` should point to the directory where you put the binray distribution of Axis2. If these variabes have been set, just compile the web service using `make`.

```
$ export JAVA_HOME=/usr/lib/jvm/openjdk
$ export AXIS2_HOME=~/Downloads/axis2-1.6.3
$ make
```

You can also use this shorter version:

```
$ make JAVA_HOME=/usr/lib/jvm/openjdk \
  AXIS2_HOME=~/Downloads/axis2-1.6.3
```

After `make` finishes the profiler web service archive `ProfilerWebService.aar` can be found in the `build/lib/` directory.

3.6.4 Creating the configuration file

The profiler web service needs a configuration file that points to the language backend and to the profiler executable. The configuration file is a simple `ini` type file that sets the two variables `profiler`

[14] This means the directory that you have cloned using git

and backend. The variable profiler specifies the (absolute) path tho the profiler executable and backend points to the language backend. It should be the directory that contains the various language configuration files. The configuration file of the web service must be named profiler.ini – otherwise the web service will not find the file. A simple version of this file looks like this:

```
#
# profiler.ini -- configuration file for the profiler web service
#
backend = /path/to/the/language/backend/directory
profiler = /path/to/the/profiler/executable
```

3.6.5 Deploying the web service

The deployment of the web service is simple. You have to copy the ProfilerWebService.aar archive and the profiler.ini configuration file to the right places in the axis2 directory of the Tomcat server:

```
$ cp build/lib/ProfilerWebService.aar /path/to/tomcat/webapps/axis2/web
$ cp profiler.ini  /path/to/tomcat/webapps/axis2/web/conf
```

Restart the tomcat server (or just restart the axis2 service from the GUI management console) and the profiler web service should be running.

3.7 Testing the profiler web service

The simplest way to test if the profiler web service is running is to use the curl utility program to access the service remotely. If you do not have the utility available you can use your web browser as well. Just copy and paste the appropriate URLS into the address bar of your browser.

First of all you have to check if the Service is up and if Tomcat and Axis2 have successfully registered the new service. In order to test this issue the following on the console:

```
$ curl localhost:8080/axis2/services/ProfilerWebService?wsdl
```

This command should output a large XML file that describes the services that the profiler web service provides. If you get an error message, make sure that you have copied the ProfilerWebService.aar file to the right location in the Tomcat installation and the server is up and running. You should also check the appropriate log files in the logs directory for any errors.

If the previous command finished successfully you should check if the language backend is installed properly and if the web service can find all the required files. Issue the following command:

```
$ curl localhost:8080/axis2/services/ProfilerWebService/getConfigurations
```

You should get a simple XML file that lists all available languages in the backend. If you do not get the list of languages, make sure that the profiler.ini configuration file is at the right place, that the backend variable points to a valid language backend and that the language backend can be accessed from within the server. Also make sure that all languages you want to provide have a configuration file with the file extension .ini.

Unfortunately there is no easy way to test the profiler and profile a document. You have to use PoCoTo to test if the profiling works. See the PoCoTo manual for further information.

3.8 How the profiler web service works

This part just explains some internals of the profiling web service. It could help you to debug the service. If you are curious you could check the implementation of the service in the `src/cis/profiler/web` directory.

The profiling web service offers two main operations[15]:

1. `getConfigurations`: sends a list of language configurations in the configured language backend.
2. `getProfile`: profiles a document for a given language configuration and sends back the results.

3.8.1 The operation `getConfigurations`

The `getConfigurations` operation is very simple:

1. It iterates over all files and directories in the language backend.
2. Each file it finds whose name ends with `.ini` is considered to be a language configuration.
3. It sends back a list of these file names without the `.ini` extension.

3.8.2 The operation `getProfile`

The `getProfile` operation is more complex. It needs additional information from the user to do its job:

1. The name of a (valid) language configuration in the backend.
2. The format of the file that should be profiled[16].
3. The file to profile. This file should be zipped and encoded in a `base64` encoding.

It follows these five steps to profile the document:

1. The web service checks if the according language configuration file exists in the backend. It simply adds `.ini` to the name of the language and checks if this file exists in the language backend – if not it returns an error to the caller.
2. If the respective language file exists, it extracts the file contents of the request, unzips it and writes it to a temporary file in the `temp` directory of the tomcat installation.
3. It creates two more temporary file names as output files for the profiler in the `temp` directory.
4. It issues a system command to call the profiler executable with the following arguments:
   - The configuration file (command line option: `--config`)
   - The format of the file (command line option: `--sourceFormat`)
   - The temporary input file (command line option: `--sourceFile`)

[15] In actual fact it offers some more operations – they are considered deprecated at this point.

[16] The format is usually `OCRCXML` but it could also be `TXT` for plain text files.

- The two temporary output files (command line options: `--out_xml` and `--out_doc`) Then it waits for the profiler to finish.

5. It zips the two output files of the profiler together, encodes them as `base64`, builds up the answer XML file and sends it back to the caller.

4 References

Mihov, Stoyan, and Klaus U. Schulz. 2004. "Fast Approximate Search in Large Dictionaries." Computational Linguistics 30 (4). MIT Press: 451–77.

Reffle, Ulrich. 2011. Algorithmen und Methoden zur dokumentenspezifischen Analyse historischer und OCR-erfasster Texte. Verlag Dr. Hut.

Reffle, Ulrich, and Christoph Ringlstetter. 2013. "Unsupervised Profiling of OCRed Historical Documents." Pattern Recognition 46 (5): 1346–57. doi:http://doi.org/http://dx.doi.org/10.1016/j.patcog.2012.10.002.

Schulz, Klaus U., and Stoyan Mihov. 2002. "Fast String Correction with Levenshtein Automata." International Journal on Document Analysis and Recognition 5 (1). Springer: 67–85.